第三章 动态规划—— 自底向上

- 1. 动态规划概述
- 2. 矩阵连乘问题
- 3. 动态规划的基本要素
- 4. 字符串匹配
- 5. 图像压缩
- 6. 流水作业调度
- 7. 0-1背包问题
- 8. 最优二叉树

第三章 动态规划—— 自底向上

- **1. 动态规划概述**
- 2. 矩阵连乘问题
- 3. 动态规划的基本要素
- 4. 字符串匹配
- 5. 图像压缩
- 6. 流水作业调度
- 7. 0-1背包问题
- 8. 最优二叉树

3.1 动态规划概述

- dynamic programming：20 世纪50年代美国数学家 Richard Bellman发明的。
- programming——planning
- 应用数学中的重要工具，解决特定类型的最优问题。
- 实现某些分治算法的有效途径。

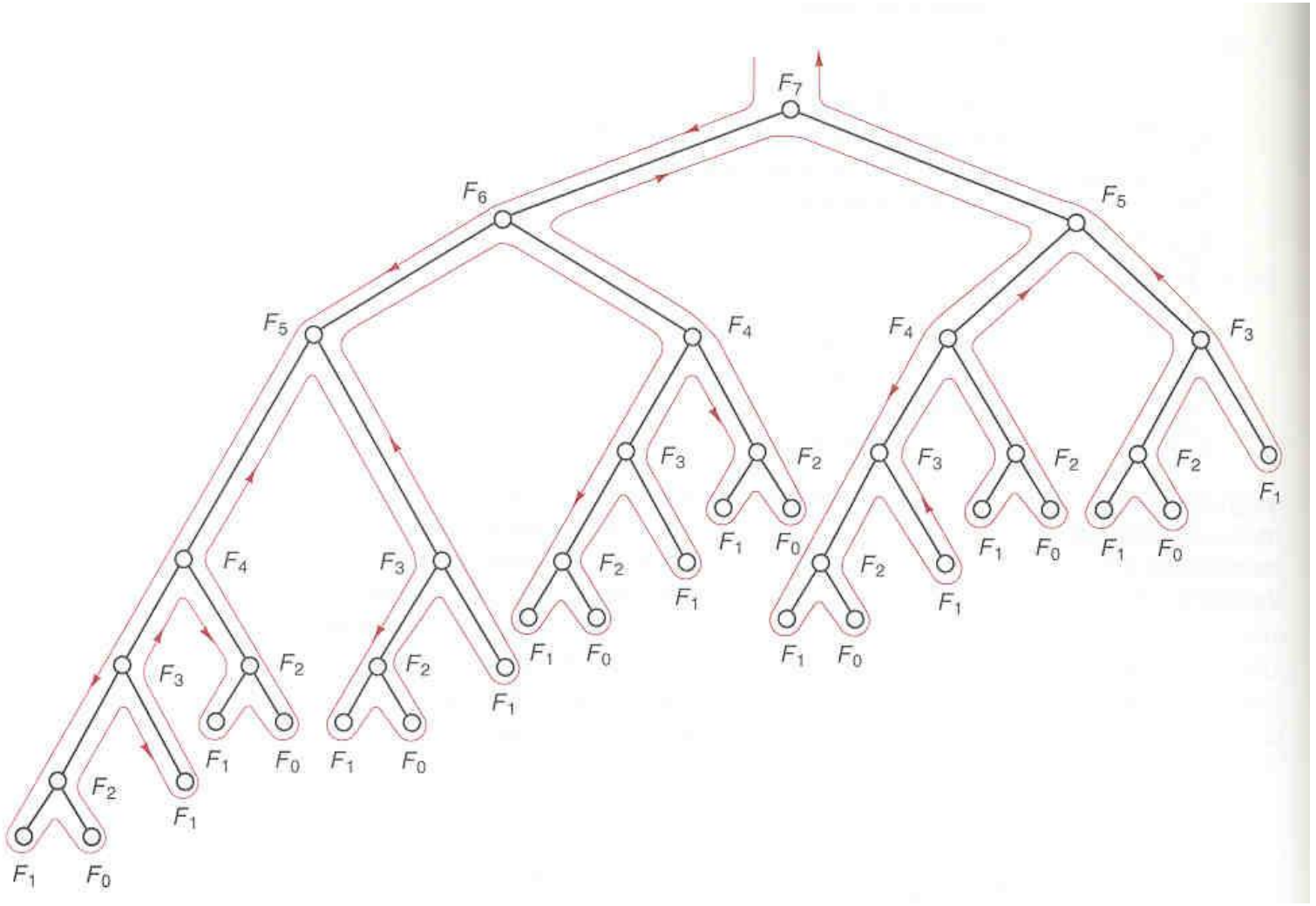
分治问题： 例1: Fibonacci numbers

- 分治法效率低.
- **Example**: Computing the Fibonacci numbers:
  - F(0) = 1;
  - F(1)=1;
  - F(n)=F(n-1)+F(n-2);
- The first few Fibonacci numbers:
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, …

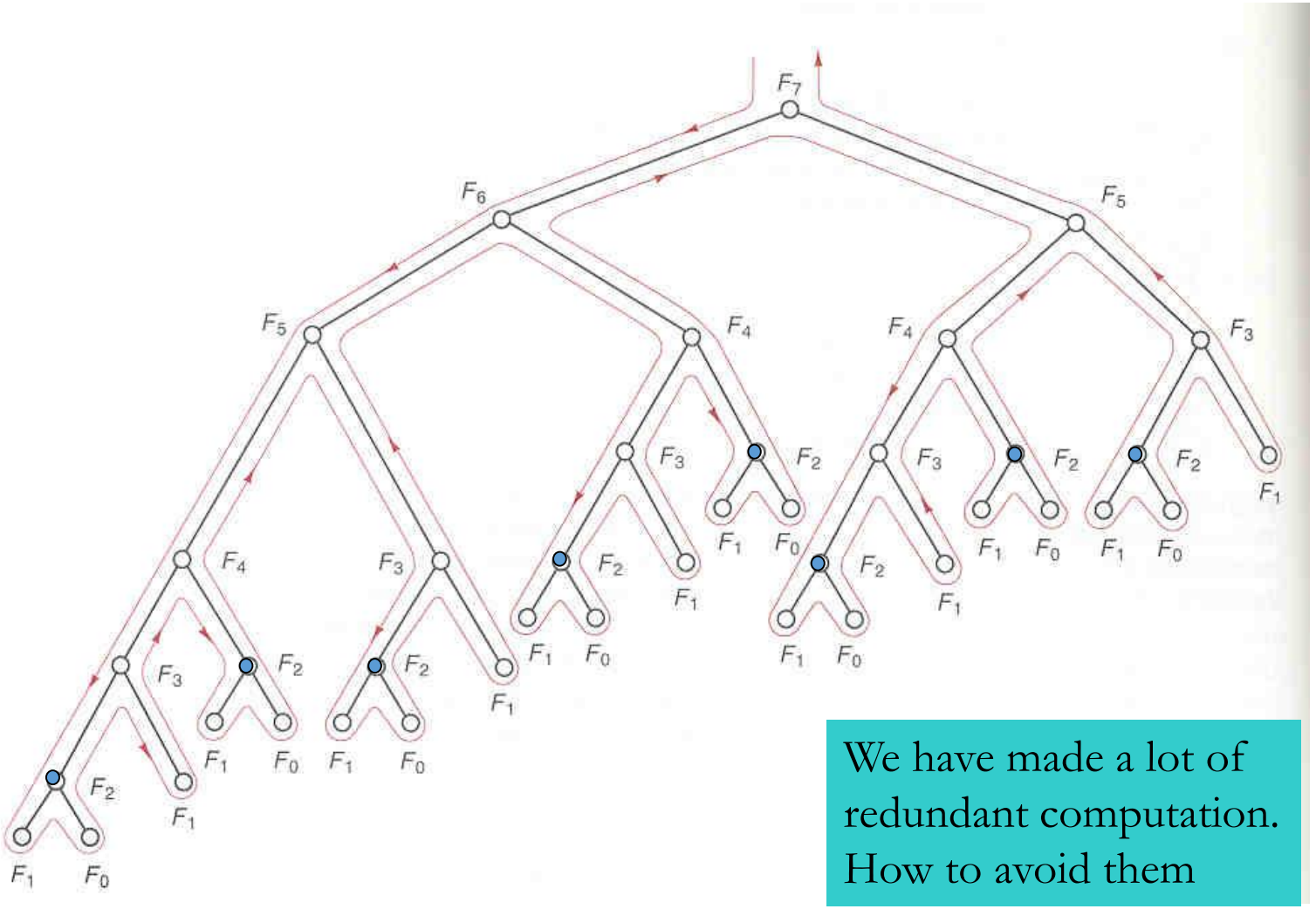
```
int F(int n)
{ int m1,m2;
  if (n < 2) return 1;
  m1 = F(n-1);
  m2 = F(n-2);
  return (m1+m2);
}
```

It takes a long time
to compute F(n) even
for small n, such as n=50.
Why?

Eg. 求解 F(7)


F_7
F_6
F_5
F_5
F_4
F_4
F_3
F_4
F_3
F_3
F_2
F_3
F_2
F_2
F_2
F_1
F_1
F_0
F_1
F_0
F_1
F_0
F_1
F_0
F_2
F_2
F_2
F_2
F_1
F_1
F_1
F_1
F_1
F_0
F_1
F_0
F_1
F_0
F_2
F_1
F_1
F_0

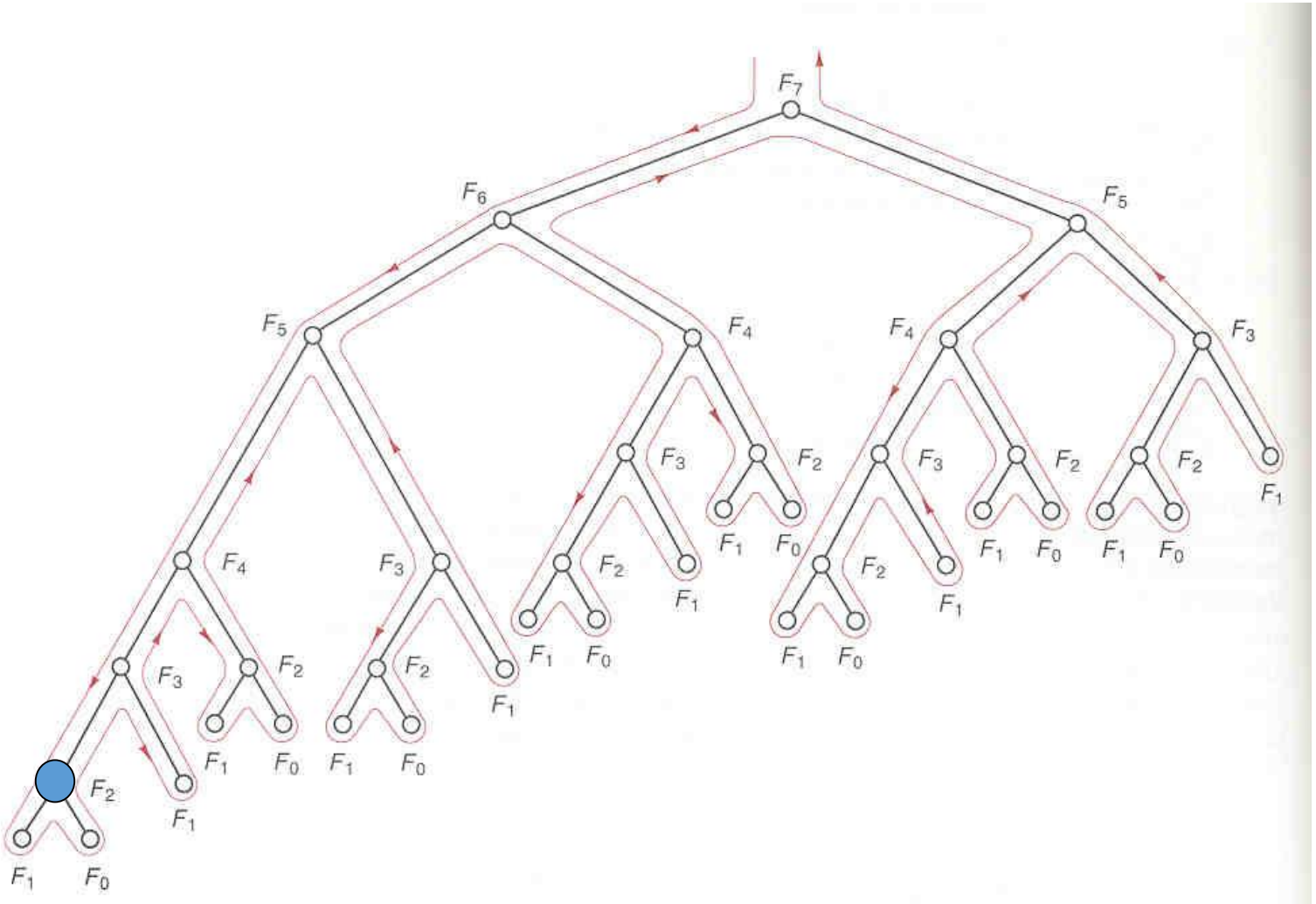
Eg. 求解 F(7)



v[0]	1
v[1]	1
v[2]	-1
v[3]	-1
v[4]	-1
v[5]	-1
v[6]	-1
v[7]	-1

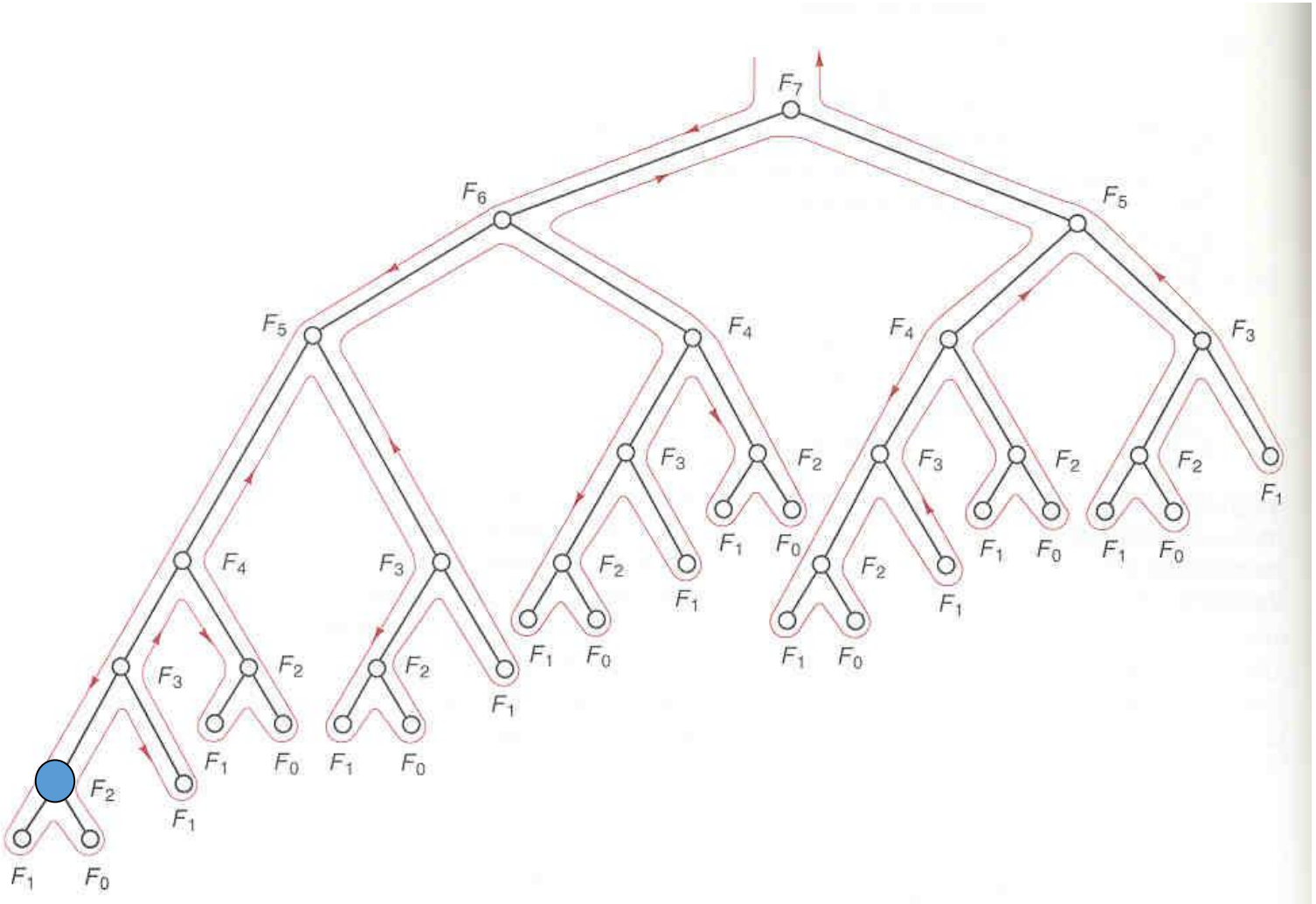
We have made a lot of redundant computation. How to avoid them

Eg. 求解 F(7)



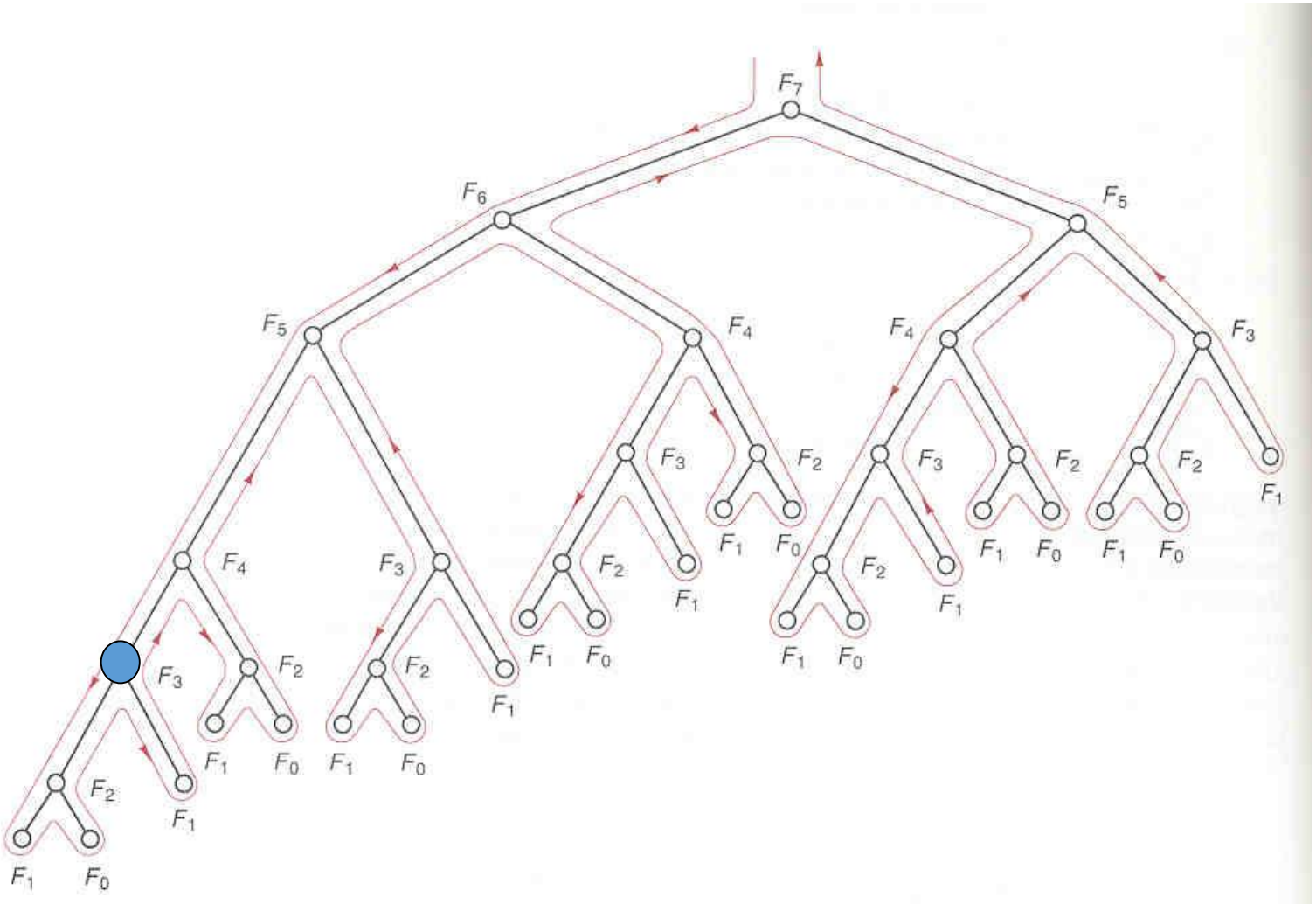
v[0]	1
v[1]	1
v[2]	-1
v[3]	-1
v[4]	-1
v[5]	-1
v[6]	-1
v[7]	-1

Eg. 求解 F(7)



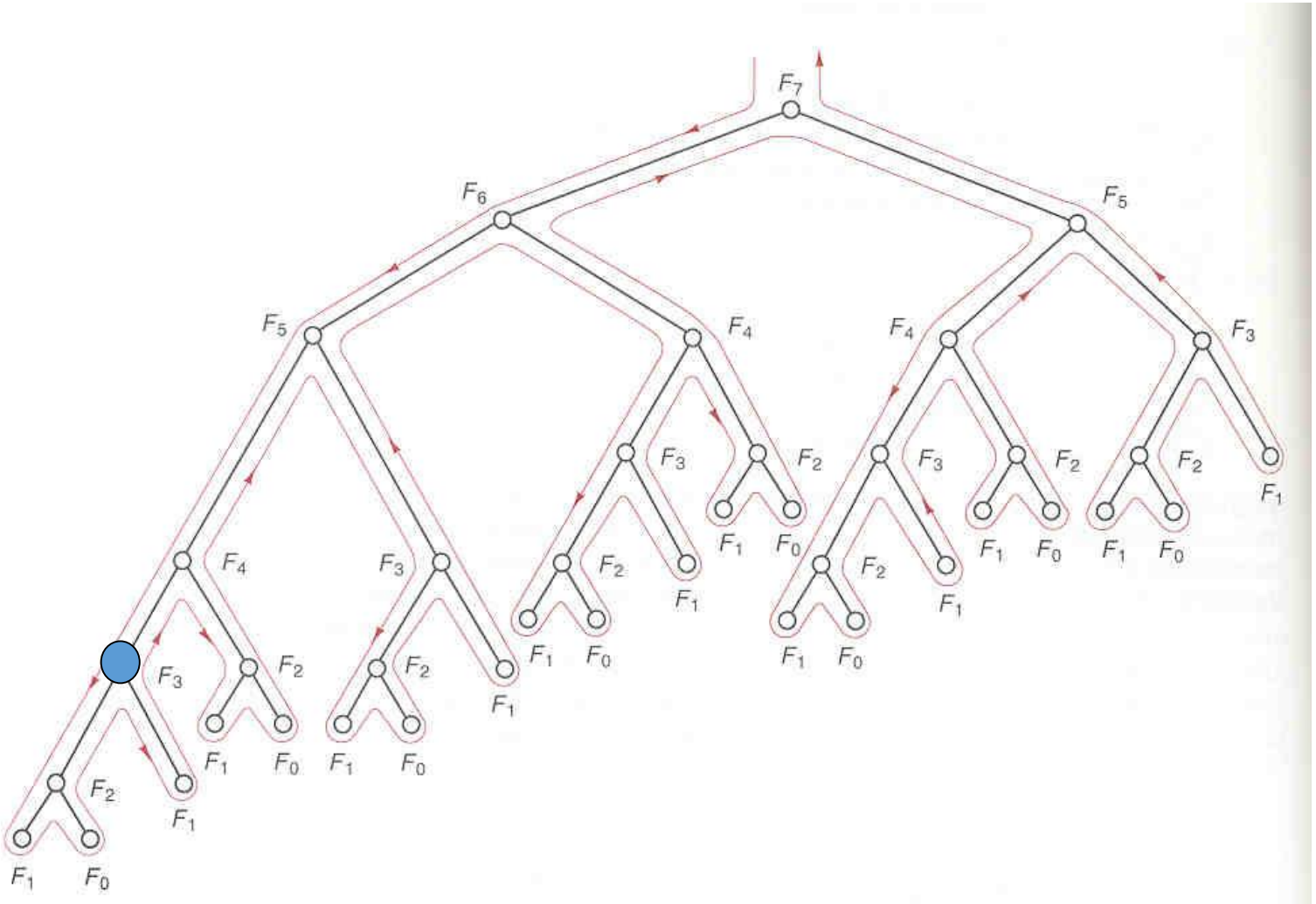
v[0]	1
v[1]	1
v[2]	2
v[3]	-1
v[4]	-1
v[5]	-1
v[6]	-1
v[7]	-1

Eg. 求解 F(7)



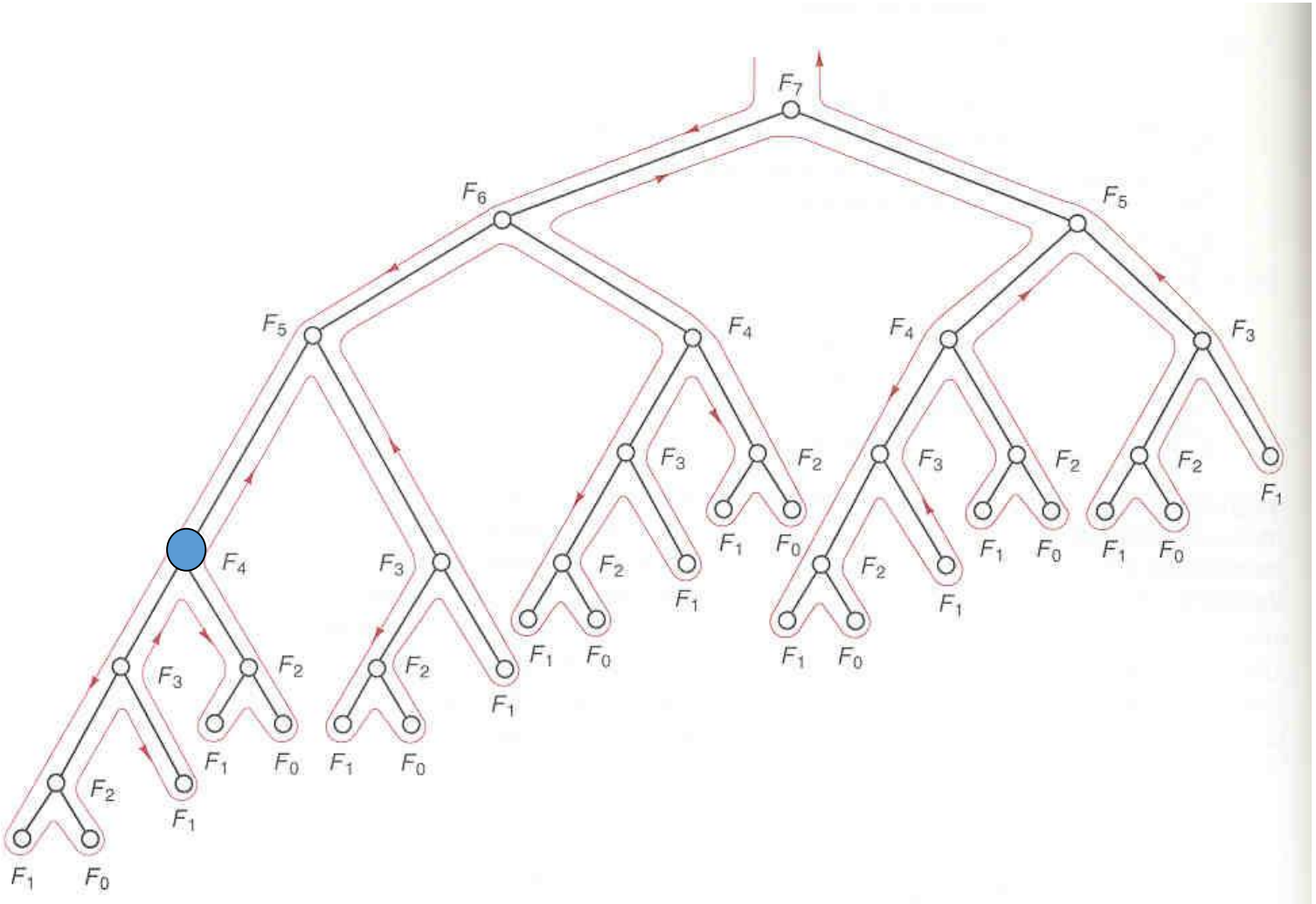
v[0]	1
v[1]	1
v[2]	2
v[3]	-1
v[4]	-1
v[5]	-1
v[6]	-1
v[7]	-1

Eg. 求解 F(7)



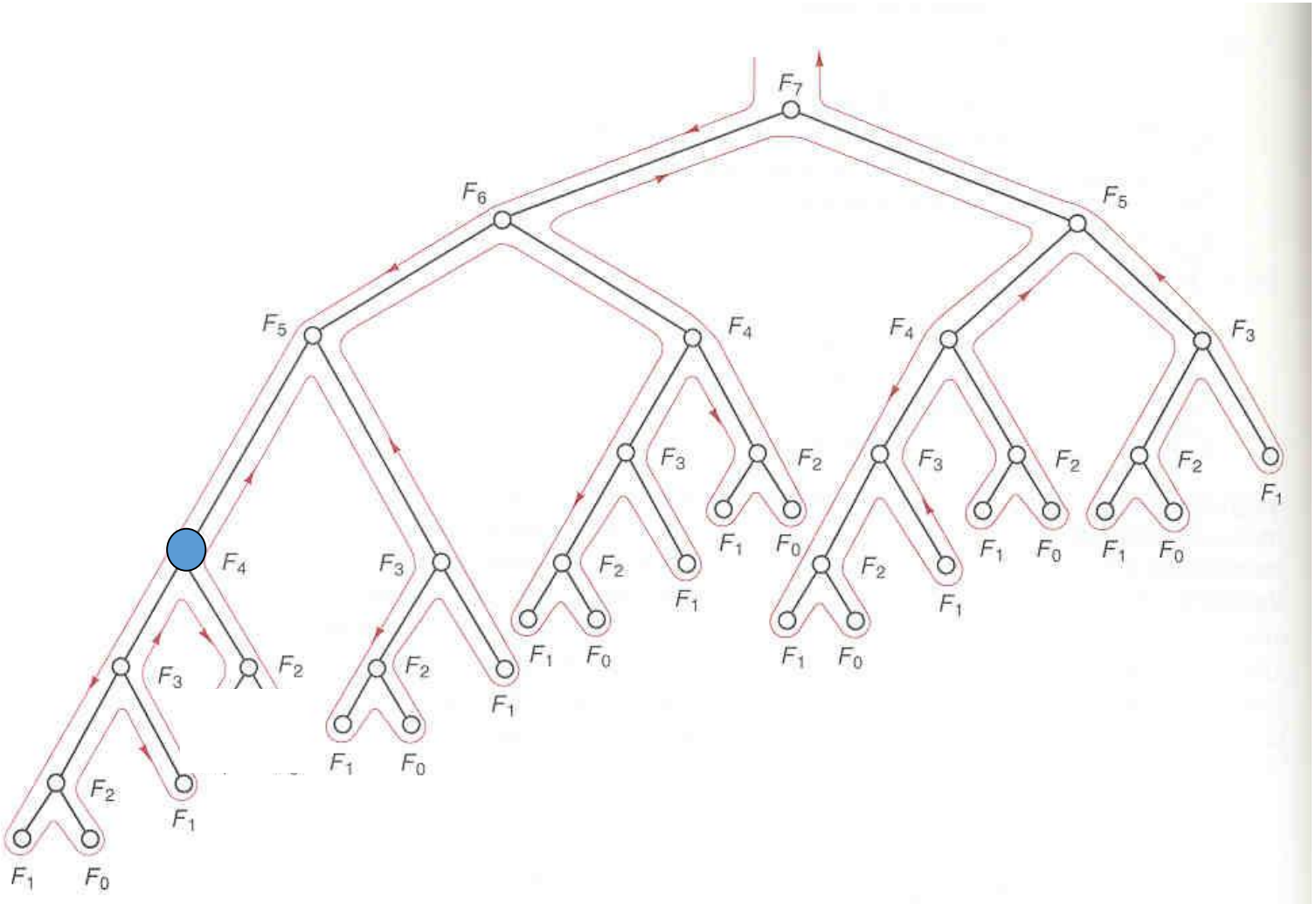
v[0]	1
v[1]	1
v[2]	2
v[3]	3
v[4]	-1
v[5]	-1
v[6]	-1
v[7]	-1

Eg. 求解 F(7)



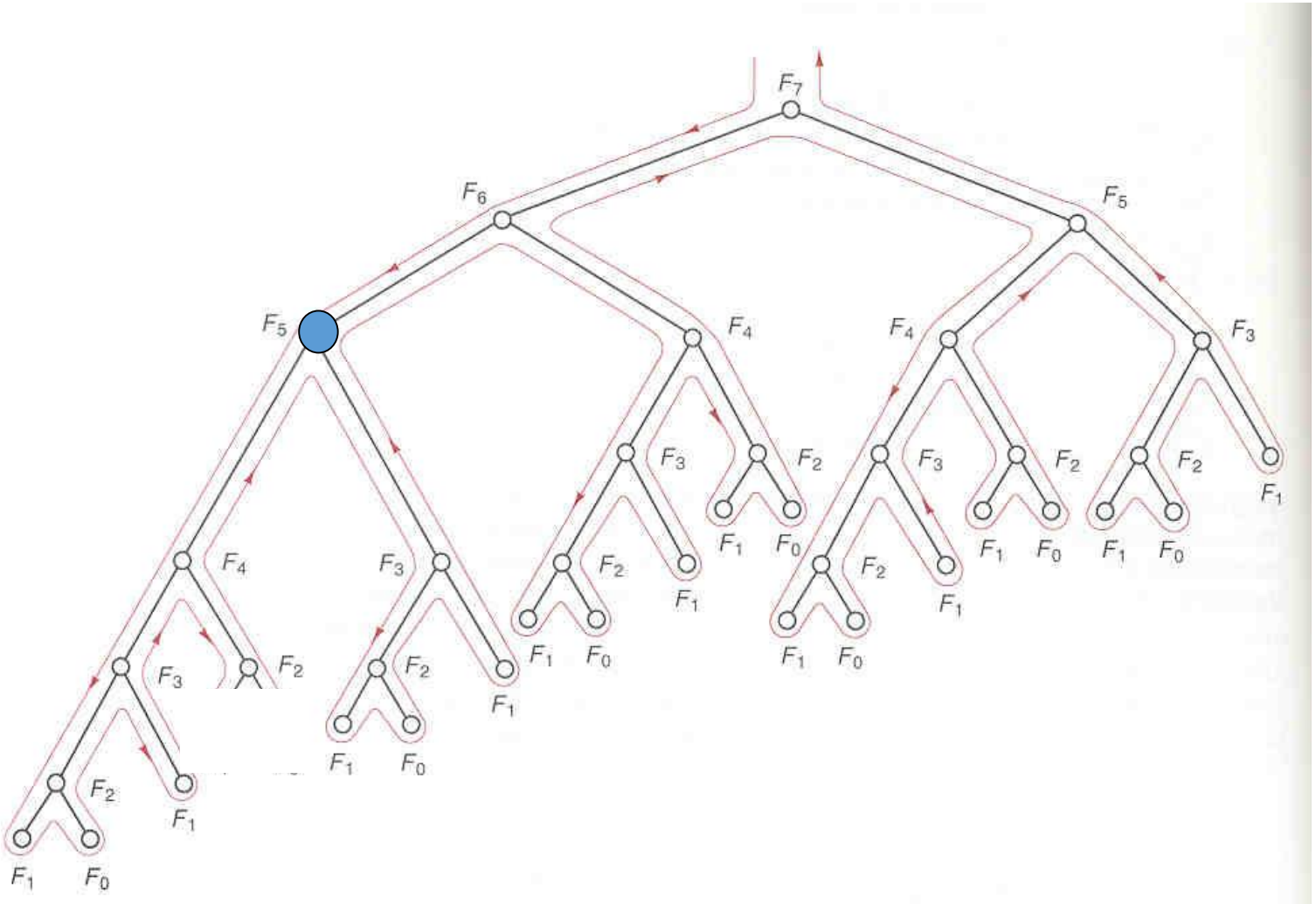
v[0]	1
v[1]	1
v[2]	2
v[3]	3
v[4]	-1
v[5]	-1
v[6]	-1
v[7]	-1

Eg. 求解 F(7)



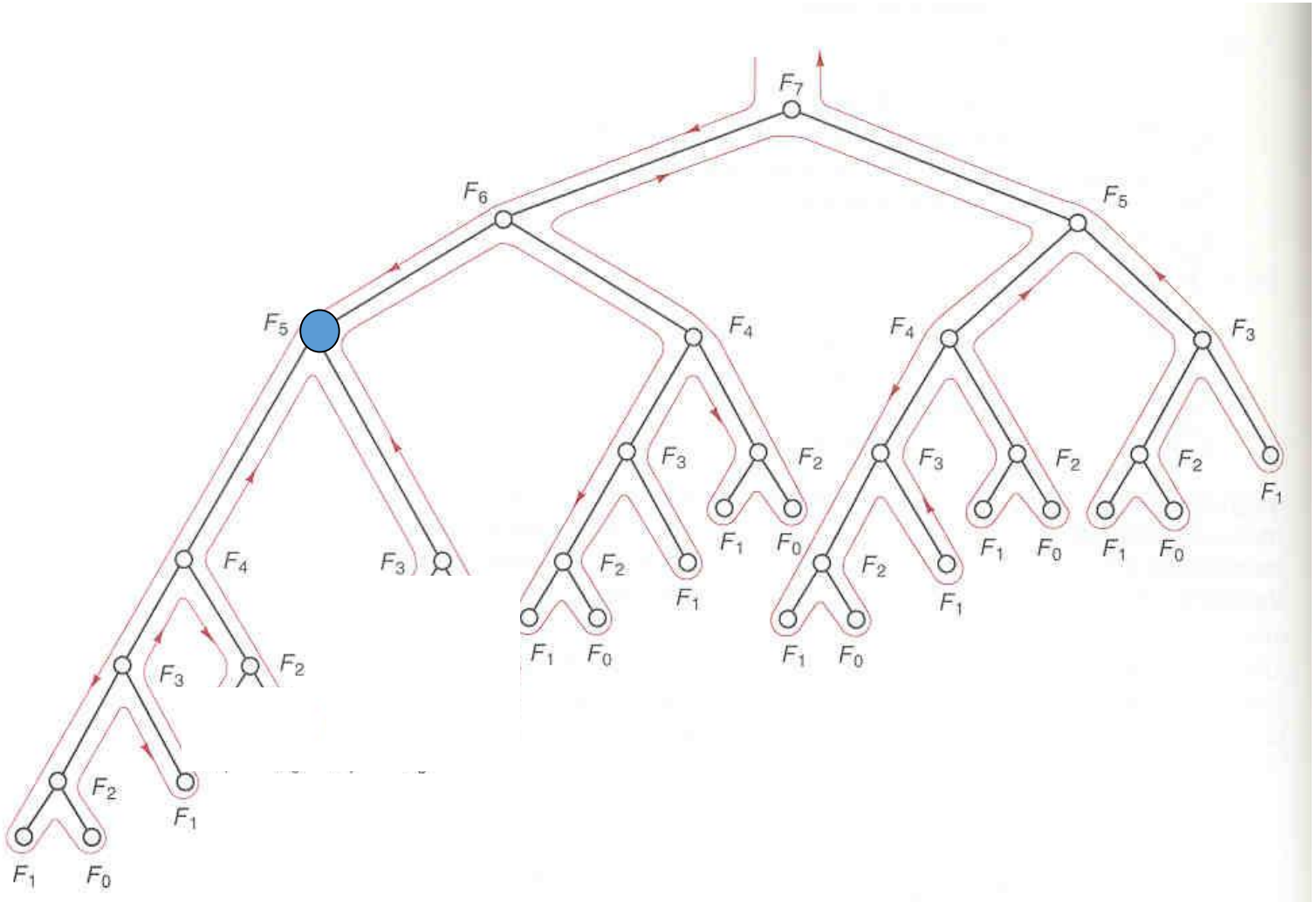
v[0]	1
v[1]	1
v[2]	2
v[3]	3
v[4]	5
v[5]	-1
v[6]	-1
v[7]	-1

Eg. 求解 F(7)



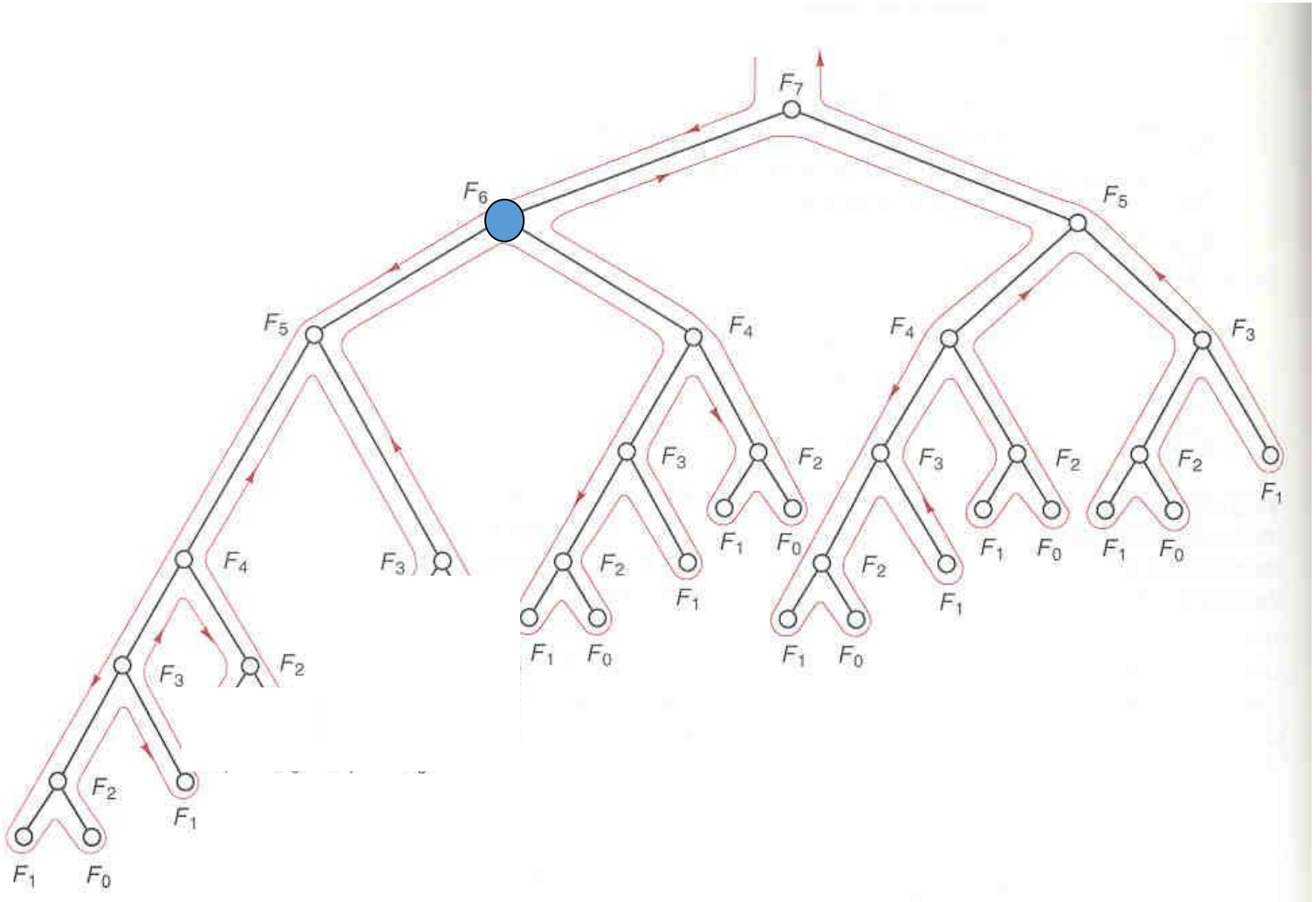
v[0]	1
v[1]	1
v[2]	2
v[3]	3
v[4]	5
v[5]	-1
v[6]	-1
v[7]	-1

Eg. 求解 F(7)



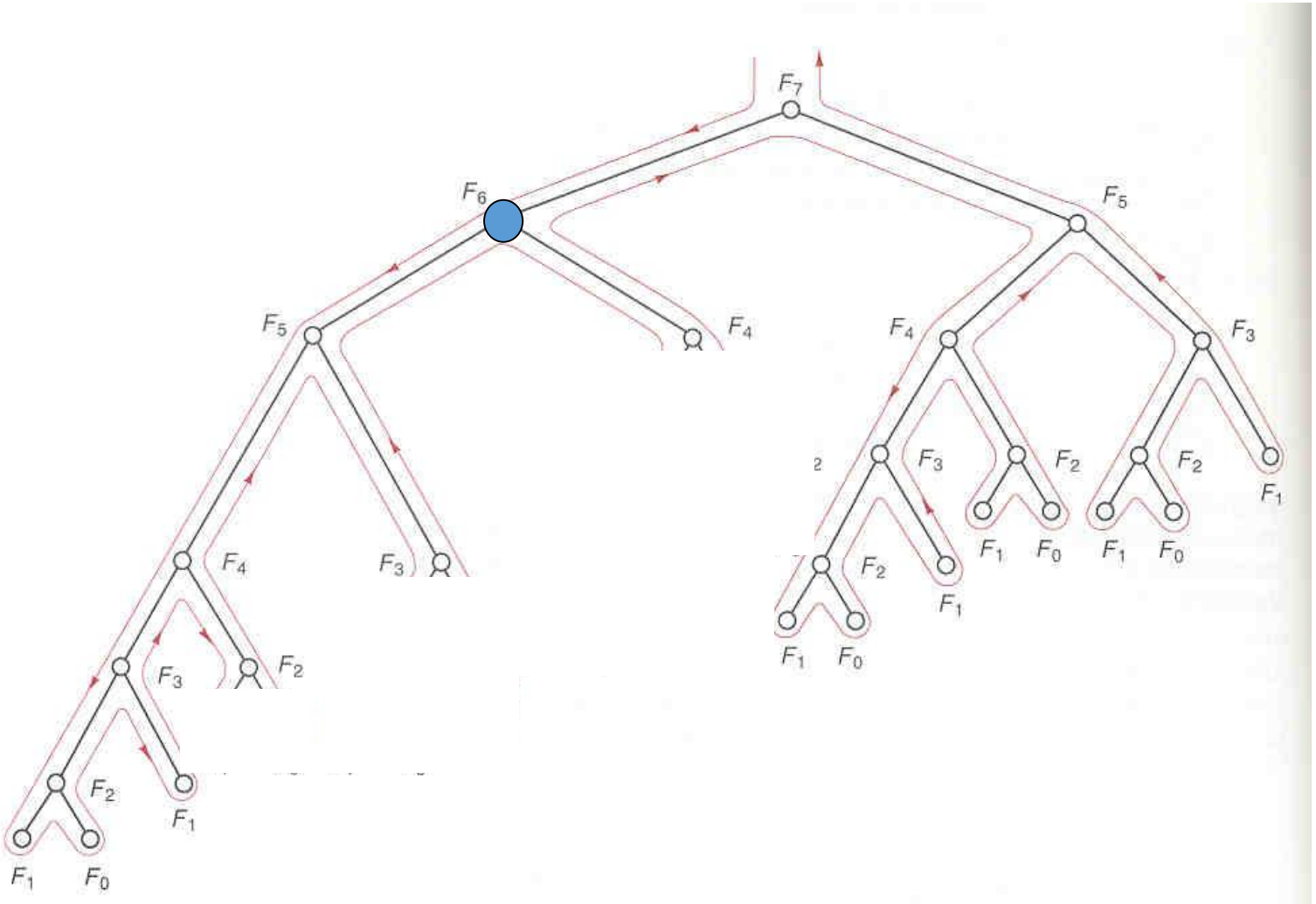
v[0]	1
v[1]	1
v[2]	2
v[3]	3
v[4]	5
v[5]	8
v[6]	-1
v[7]	-1

Eg. 求解 F(7)



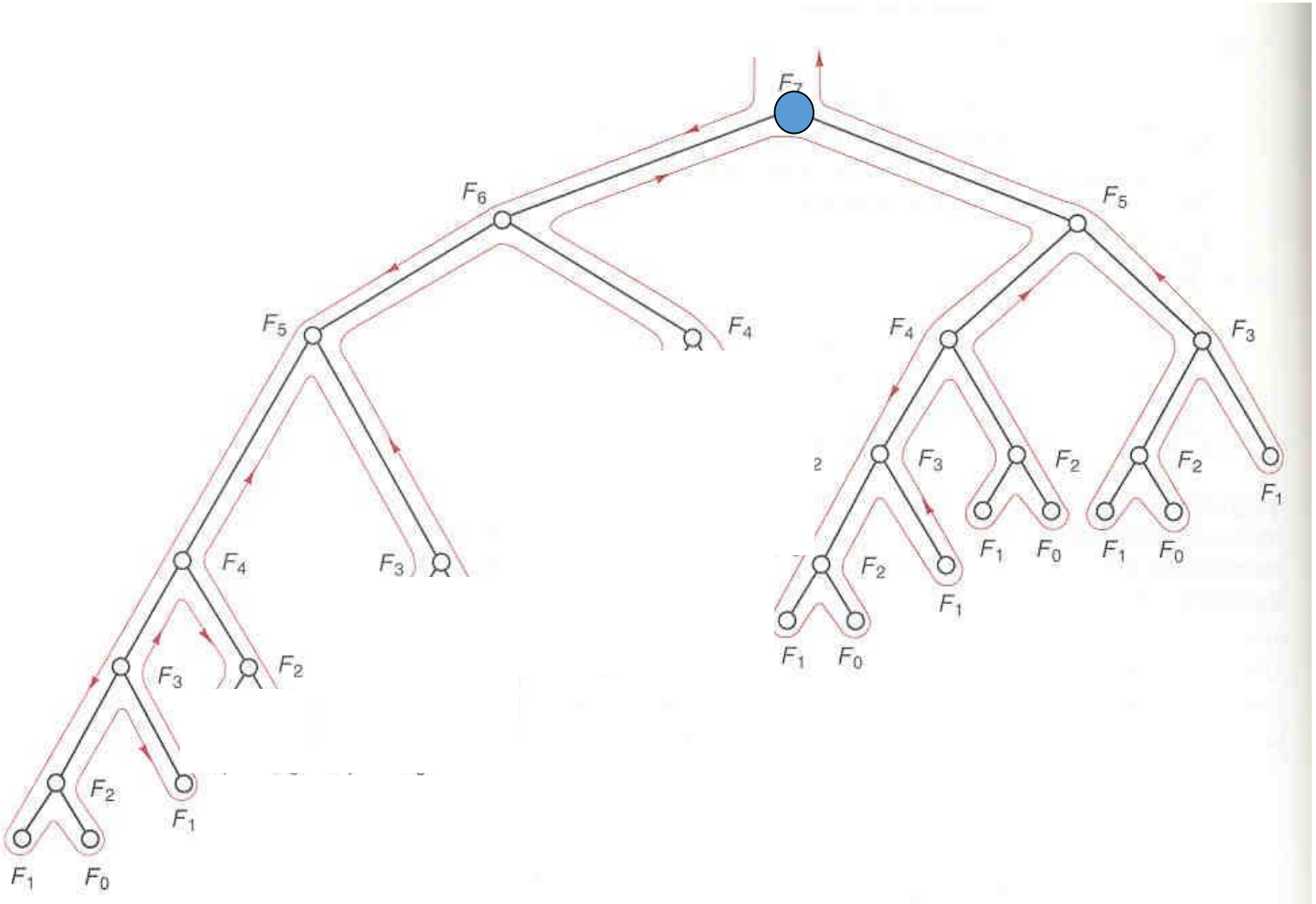
v[0]	1
v[1]	1
v[2]	2
v[3]	3
v[4]	5
v[5]	8
v[6]	-1
v[7]	-1

Eg. 求解 F(7)



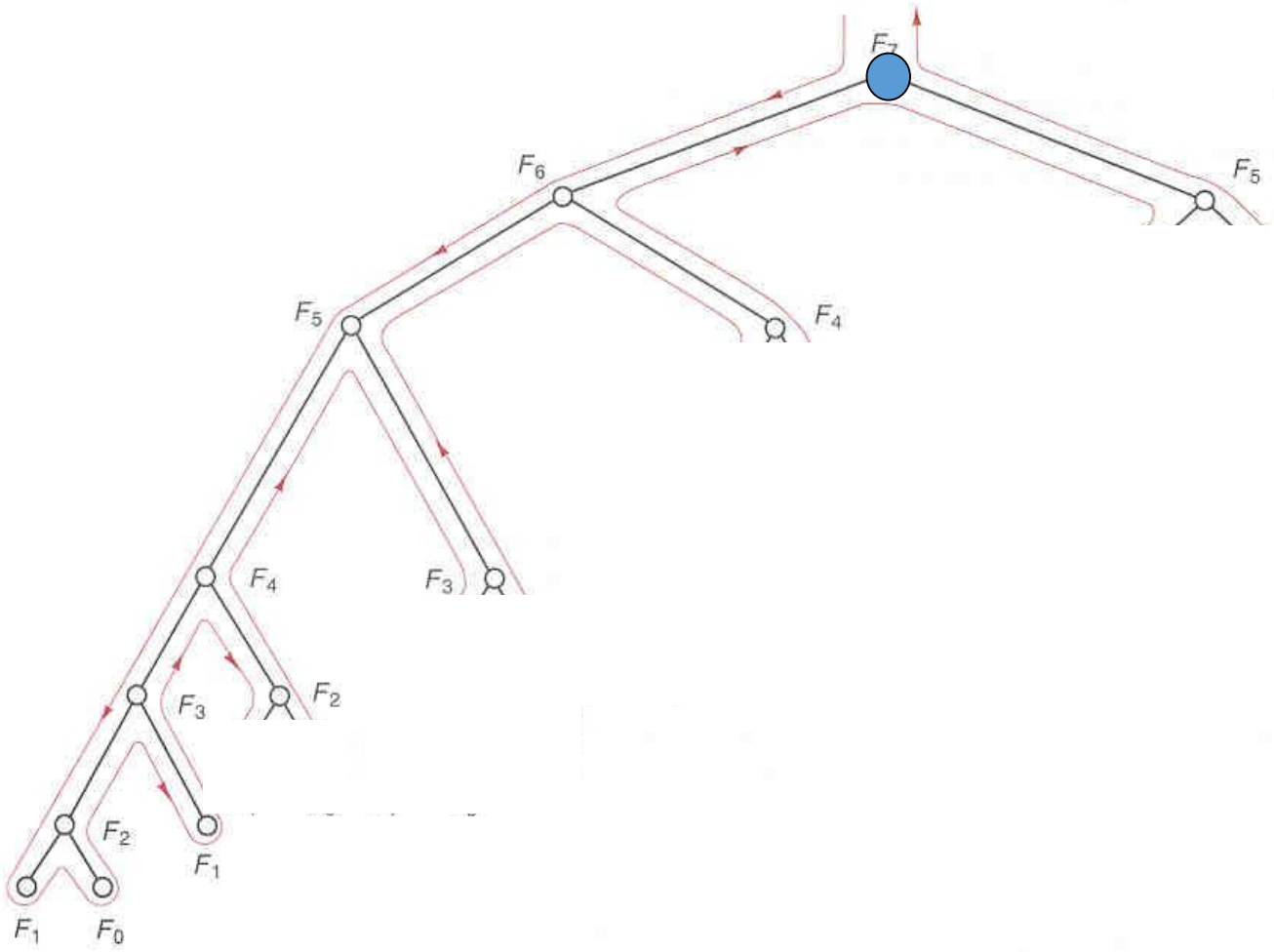
v[0]	1
v[1]	1
v[2]	2
v[3]	3
v[4]	5
v[5]	8
v[6]	13
v[7]	-1

Eg. 求解 F(7)



v[0]	1
v[1]	1
v[2]	2
v[3]	3
v[4]	5
v[5]	8
v[6]	13
v[7]	-1

Eg. 求解 F(7)



v[0]	1
v[1]	1
v[2]	2
v[3]	3
v[4]	5
v[5]	8
v[6]	13
v[7]	21

改进

- 原因
  - 函数调用次数过多： 参数传递， 动态连接……
  - 计算 F(n), 需要计算 F(n-1), F(n-2)
    一般化为计算 F(i), 需要计算 F(i-1) 和 F(i-2)

- 措施
  - 自底向上计算， 即 已知 F(0)=F(1)=1, 计算 F(2),
    然后计算 F(3)， F(4)…

- 避免重复计算

```
  int F(int n)
{   int i, A[1000];
     A[0]=A[1]=1;
     for (i=2; i <=n; i++)
           A[i]=A[i-1]+A[i-2];
}
```

方法总结:

- 采用递归式描述问题的解
  - F(n) = F(n-1) + F(n-2).
- 建立索引，以便用表格存储和检索子问题
- 以自底向上的方式填表：从最小子问题开始，保证解题过程中每个子问题都有更小子问题的解可利用
- Dynamic Programming: In the late 40's (when computers were rare), programming refers to the "tabular method".

分治问题：例2: 计算二项式系数

$(a+b)^n=C(n,0)a^n+\cdots+C(n,i)a^{n-i}b^i+\cdots+C(n,n)b^n$

当n>k>0时，　C(n,k)= C(n-1,k-1)+ C(n-1,k)

C(n,0)= C(n,n)=1

	0	1	2	…	k-1	k
0	1					
1	1	1				
2	1	2	1			
…						
k	1					1
…						
n-2				**C(n-2,k-2)**	**C(n-2,k-1)**	**C(n-1,k)**
n-1	1				**C(n-1,k-1)**	**C(n-1,k)**
n	1					C(n,k)

例2: 计算二项式系数

- 性能分析:
  - 记A(n,k)为该算法在计算C(n,k)时所作的加法总次数。

$$A(n,k)=\sum_{i=1}^{k}\sum_{j=1}^{i-1}1+\sum_{i=1+1}^{n}\sum_{j=1}^{k}1=\sum_{i=1}^{k}(i-1)+\sum_{i=k+1}^{n}k$$

$$=\frac{(k-1)k}{2}+k(n-k)\in\Theta(nk)$$

动态规划 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

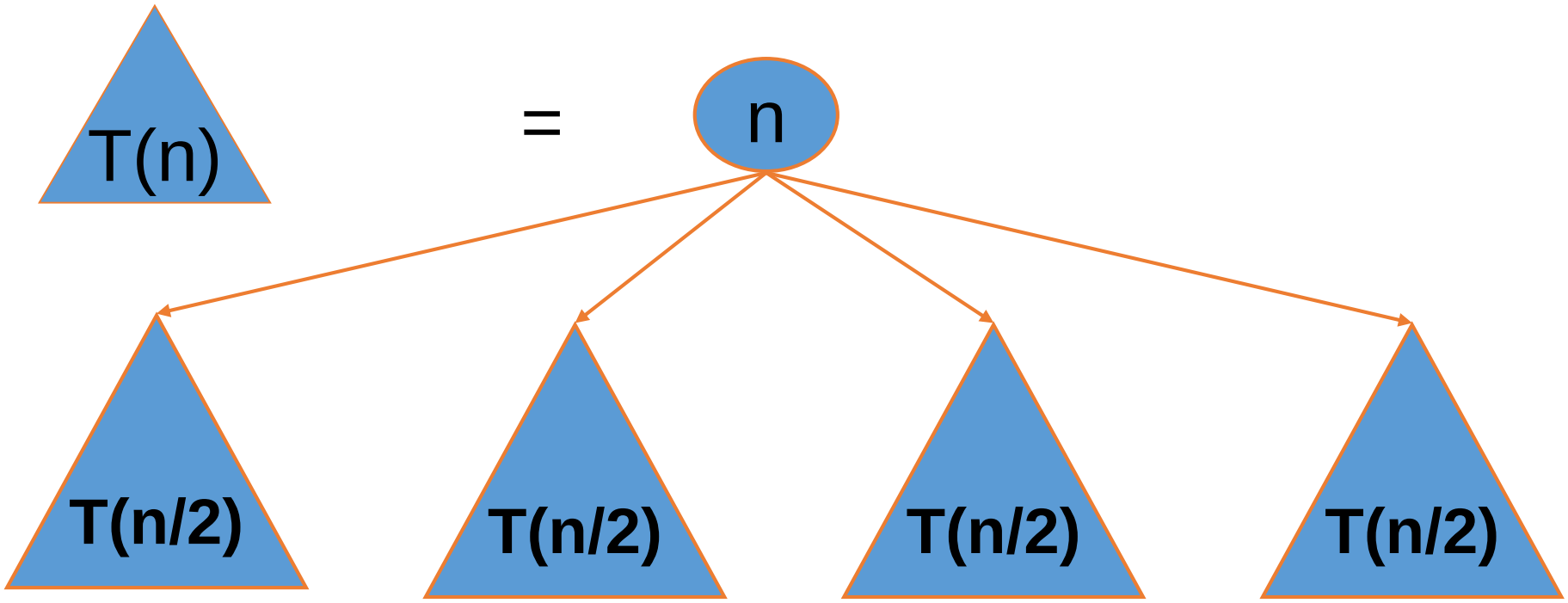
- 掌握设计动态规划算法的步骤
  (1) 找出最优解的性质，并刻划其结构特征。
  (2) 递归地定义最优值。
  (3) 以自底向上的方式计算出最优值。
  (4) 根据计算最优值时得到的信息，构造最优解。

动态规划 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) 字符串匹配
  - (3) 图像压缩
  - (4) 流水作业调度
  - (5) 背包问题
  - (6) 最优二叉搜索树

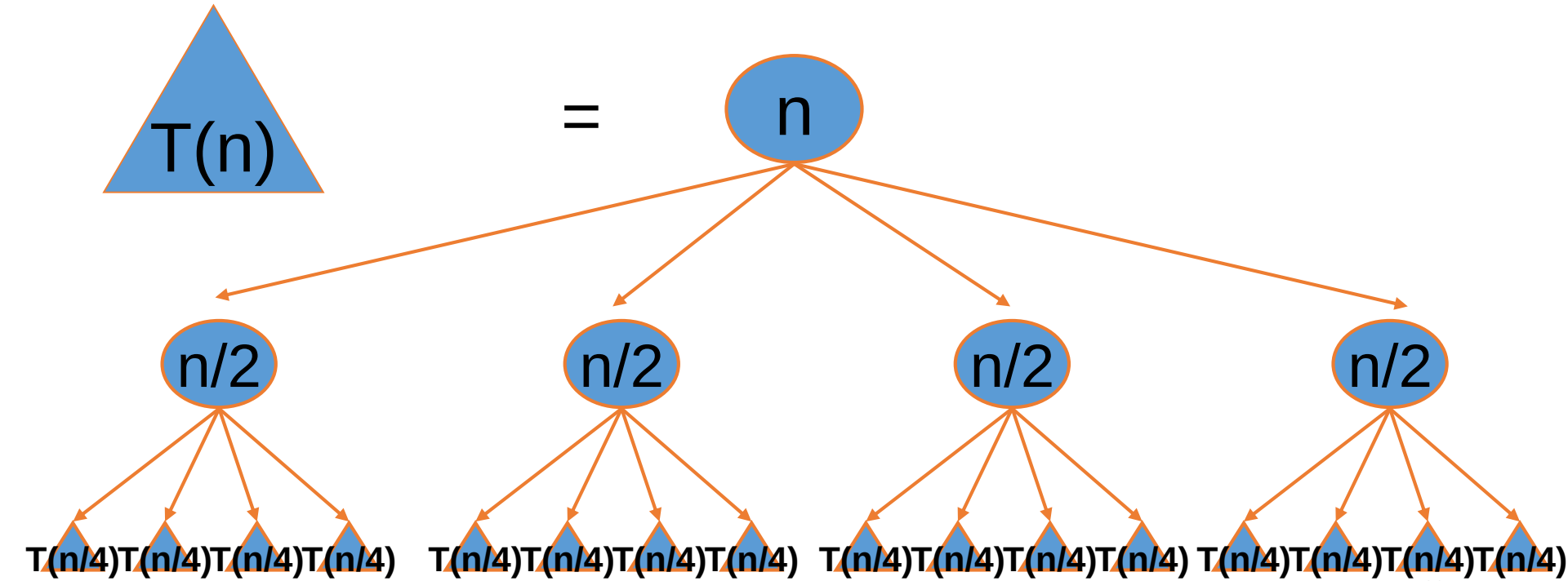
算法总体思想

- 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。



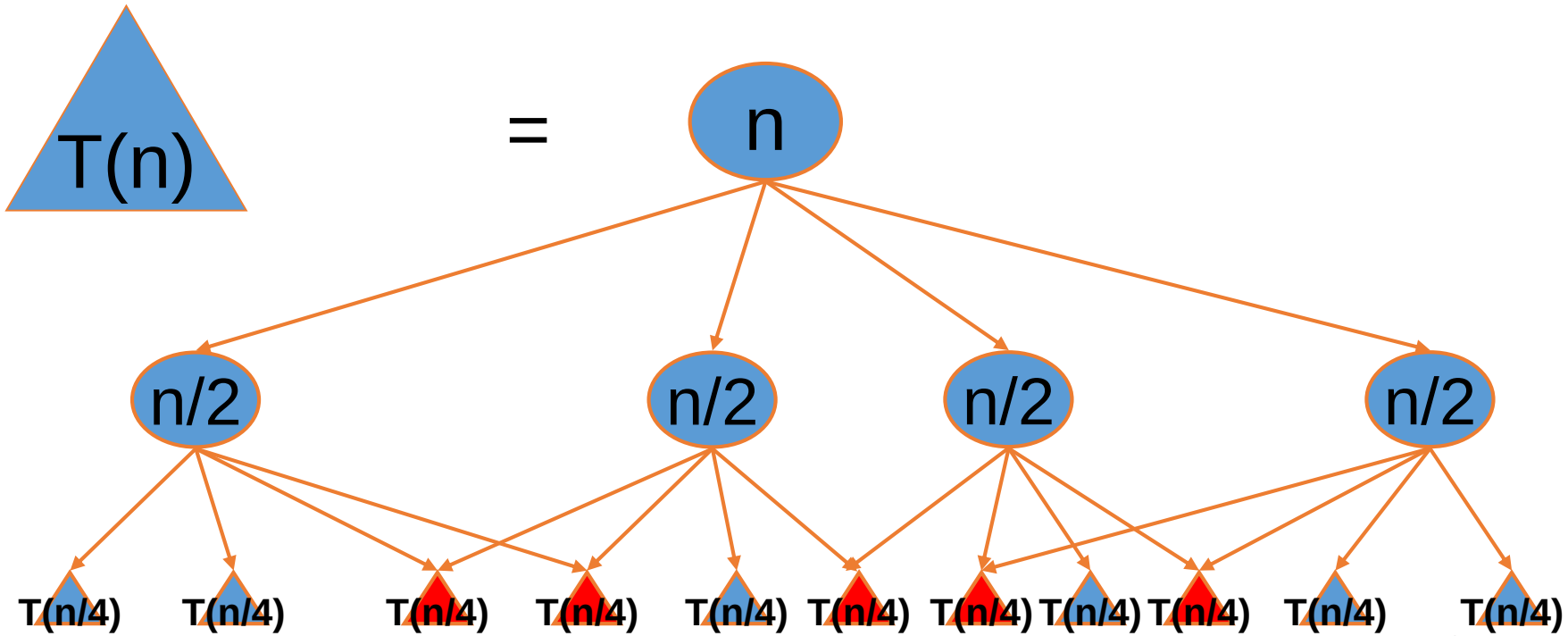
算法总体思想

- 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。



算法总体思想

- 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。



Those who cannot remember the past are doomed to repeat it.

-----George Santayana,

The life of Reason,

Book I: Introduction

and Reason in Common Sense

(1905)

动态规划基本步骤

- 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。

3.2 矩阵连乘问题

- 两个矩阵相乘

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \\ 9 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 3 & 2 & 1 \\ 7 & 4 & 3 \\ 11 & 6 & 5 \\ 15 & 8 & 7 \\ 9 & 0 & 9 \end{pmatrix}$$

5 x2　　2x3　　5 x3

```
Multiply(A[p,r], B[r,s])
{
   for (i=1; i<=p; i++)
    for (j=1; j<=s; j++)
    { C[i,j]=0;
      for (k=1; k<=r; k++)
        C[i,j]=C[i,j]+A[i,k]*B[k,j];
    }
    return C;
}
```

约束：
一个 pxq 矩阵乘以 rxs 矩阵，要求 q=r.
得到矩阵的维度 pxs.

Time complexity = O(prs).

第三章 动态规划—— 自底向上

- 1. 动态规划概述
- **2. 矩阵连乘问题**
- 3. 动态规划的基本要素
- 4. 字符串匹配
- 5. 图像压缩
- 6. 流水作业调度
- 7. 0-1背包问题
- 8. 最优二叉树

3.2 矩阵连乘问题

- **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）

- **结合律：**(B*C)*D=B*(C*D)

例：两个3维图象的匹配问题

- 一个图象需旋转、平移、缩放 n 次才能逼近另一副图象。
  迭代向量 T= Σ A(x,y,z)*B(x,y,z)*C(x,y,z)，迭代100次
  A=12 × 3，B=3 × 3，C=3 × 1
  设图象含256*256*256个向量

- 计算量：$100*256^3*T$　$1.7*10^9$ T
  A*B*C =12*3*3+12*3*1=144　$2.4*10^{11}$　40分钟
  A*(B*C)=3*3*1+12*3*1=45　$7.5*10^{10}$　12.5分钟

例：4 矩阵连乘

- 设有四个矩阵 A，B，C，D，它们的维数分别是：

A=50 × 10，B=10 × 40，C=40 × 30，D=30 × 5

- 总共有五种完全加括号的方式：

((A(BC))D)，**(A(B(CD)))**，((AB)(CD))，
(((AB)C)D)，(A((BC)D))

- 乘法次数：

34500 = 10*40*30+ 50*10*30+ 50*30*5

10500 = 40*30*5+10*40*5+50*10*5

36000 =50*10*40+40*30*5+50*40*5

87500 =50*10*40+50*40*30+50*30*5

21000 =10*40*30+10*30*5+50*30*5

矩阵连乘问题

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

$$A_1A_2\cdots A_n$$

- 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

给定n个矩阵 $\{A_1,A_2,\cdots,A_n\}$，其中A_i与A_{i+1}是可乘的，i=1，2…，n-1。如何确定计算矩阵连乘积的计算次序，使得依此次序计算矩阵连乘积需要的数乘次数最少。

- **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。

算法复杂度分析：

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：$(A_1\cdots A_k)(A_{k+1}\cdots A_n)$可以得到关于P(n)的递推式如下：

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

矩阵连乘问题

- **穷举法 ×**
- **动态规划**

- 将矩阵连乘积 $A_i A_{i+1} \ldots A_j$ 简记为A[i:j]，这里i≤j
- 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≤k<j，则其相应完全加括号方式为$(A_iA_{i+1}\cdots A_k)(A_{k+1}A_{k+2}\cdots A_j)$
- 计算量：

A[i:k]的计算量 + A[k+1:j]的计算量 + A[i:k]与A[k+1:j]相乘的计算量

分析最优解的结构

- 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。

- 矩阵连乘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。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。

建立递归关系

- 设计算A[i:j]，1≤i≤j≤n，所需要的最少乘法次数为m[i,j]，则原问题的最优值为m[1,n]
- 当 i=j 时， A[i:j]=A_i，
  因此，m[i,i]=0，i=1,2,…,n
- 当 i<j 时， m[i,j]=m[i,k]+m[k+1,j]+$p_{i-1}p_kp_j$，
  这里A_i的维数为$p_{i-1}\times p_i$
- 递归地定义m[i,j]为：

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

k 的位置只有 $j-i$ 种可能

计算最优值

- 对于1≤i≤j≤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。

- 用动态规划算法解此问题，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。

```
public  static void MatrixChain(int[] p,  int[][] m,  int[][] s)
{
      int n=p.length-1
      for (int i = 1; i <= n; i++) m[i][i] = 0;   // i=j,  A[i:j]=Ai
      for (int r = 2; r <= n; r++)                    //矩阵链长度r
        for (int i = 1; i <= n - r+1; i++) {       //矩阵链起始 i
          int j=i+r-1;                                  //矩阵链结束 j
          m[i][j] = m[i+1][j]+ p[i-1]*p[i]*p[j];
          s[i][j] = i;
          for (int k = i+1; k < j; k++) {          //断开点 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; }     //记载断开点
          }
       }
}
```

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

p[ ] = { 30, 35, 15, 5, 10, 20, 25 }

m[1][1] = m[2][2] = m[3][3]

= m[4][4] = m[5][5] = m[6][6] = 0

m[1][2]=m[2][2]+p[0]*p[1]*p[2]=0+30*35*15=15750

m[2][3]=m[3][3]+p[1]*p[2]*p[3]=0+35*15*5=2625

m[3][4]=m[4][4]+p[2]*p[3]*p[4]=0+15*5*10=750

m[4][5]=m[5][5]+p[3]*p[4]*p[5]=0+5*10*20=1000

m[5][6]=m[5][5]+p[4]*p[5]*p[6]=0+10*20*25=5000

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

m[1][3]=m[2][3]+p[0]*p[1]*p[3]=2625+30*35*5=7875

s[1][3]=1

m[1][3] =
m[1][2]+m[3][3]+p[0]*p[2]*p[3]=15750+1350=16100X

m[2][4]=m[3][4]+p[1]*p[2]*p[4]=750+35*15*10=6000X

s[2][4]= 2

m[2][4]=
m[2][3]+m[4][4]+p[1]*p[3]*p[4]=2625+0+35*5*10=4375

s[2][4]=3

m[3][5]=m[4][5]+p[2]*p[3]*p[5]=1000+15*5*20=2500

= m[3][4]+m[5][5]+p[2]*p[4]*p[5]=750+15*10*20=3750X

s[3][5]=3

m[4][6]=m[5][6]+p[3]*p[4]*p[6]=5000+5*10*25=6250X

=m[4][5]+m[6][6]+p[3]*p[5]*p[6]=1000+0+5*20*25=3500

s[4][6]=5

m[1][4]=m[2][4]+p[0]*p[1]*p[4]=4375+30*35*10=14875X

=m[1][3]+m[4][4]+p[0]*p[3]*p[4]=7875+0+30*5*10=9375

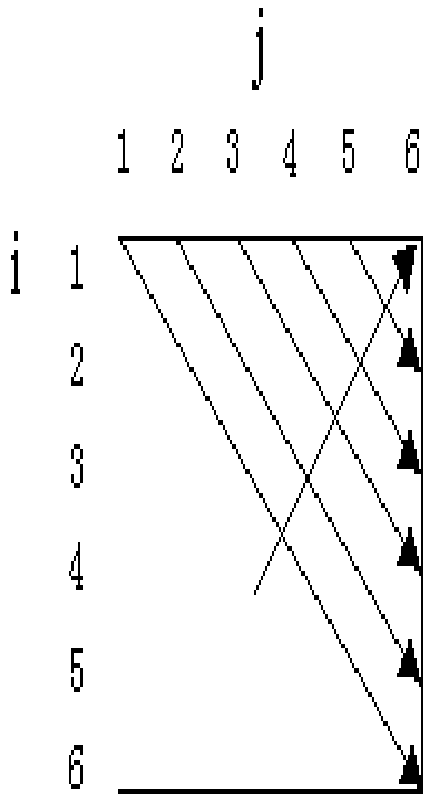
s[1][4]=3

m[2][5]=m[3][5]+p[1][2][5]=2500+35*15*20=13000X

=m[2][3]+m[4][5]+p[1][3][5]=2625+1000+35*5*20=7125

k=3, s[2][5]=3

$$m[2][5]=\min\begin{cases} m[2][2]+m[3][5]+p_1p_2p_5=0+2500+35\times15\times20 \\ \qquad =13000 \quad X \\ m[2][3]+m[4][5]+p_1p_3p_5=2625+1000+35\times5\times20 \\ \qquad =7125 \quad \surd \\ m[2][4]+m[5][5]+p_1p_4p_5=4375+0+35\times10\times20 \\ \qquad =11375 \quad X \end{cases}$$



(a) 计算次序

j

i	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

(b) m[i][j]

j

i	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

(c) s[i][j]

用动态规划法求最优解

s[i][j]中的k表明最佳位置： A_k和A_{k+1}之间断开

A[1:n]的最优加括号方式:

(A[1: s[1][n]]) (A [s[1][n]+1:n])

A[1: s[1][n]]的最优加括号方式:

(A[1: s[1][s[1][n]]]) (A[s[1][s[1][n]]+1: s[1][n]])

A[s[1][n]+1 : n]的最优加括号方式在s[s[[1][n]+1][n]]处断开

用动态规划法求最优解

```
public  static void traceback(int[][] s，  int i, int j)
{
        if ( i = =j) return;
        traceback( s, i, s[ i ][j] );
        traceback( s, i, s[ i ][j] +1, j);
       System.out.println(“Multiply A” + i + “,”+ s[i][j]+
                                “and A”+(s[i][j]+1)+ “,”+j);
}
```

traceback(s,1,6)

s[1][6]=3

traceback(s,1,3) $(A_1A_2A_3)(A_4A_5A_6)$

s[1][3]=1

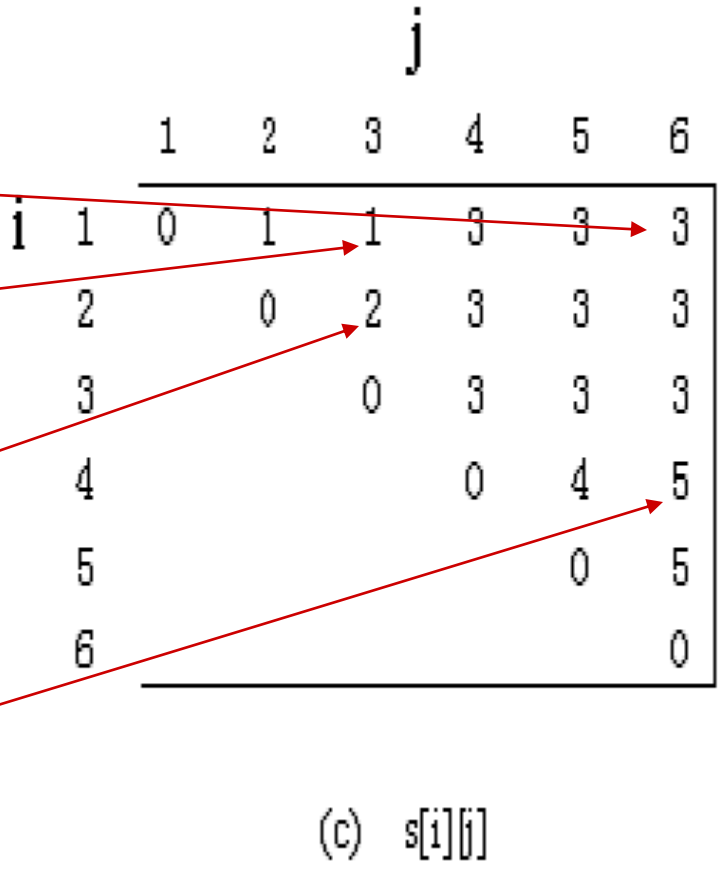
traceback(s,1,1) $(A_1(A_2A_3))(A_4A_5A_6)$

traceback(s,2,3)

s[2][3]=2

traceback(s,4,6) $(A_1(A_2A_3))((A_4A_5)A_6)$

s[4][6]=5



矩阵连乘问题小结

- **算法复杂度分析：**
- 空间：
  - p[ ], m[ ][ ], s[ ][ ], Ai
  - 算法所占用的空间为$O(n^2)$。
- 时间：
  - 算法**matrixChain**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$O(n^3)$。

第三章 动态规划—— 自底向上

- 1. 动态规划概述
- 2. 矩阵连乘问题
- **3. 动态规划的基本要素**
- 4. 序列匹配
- 5. 图像压缩
- 6. 流水作业调度
- 7. 0-1背包问题
- 8. 最优二叉树

动态规划算法的基本要素

一、最优子结构

- 矩阵连乘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。
- 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。
- 利用问题的最优子结构性质，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。最优子结构是问题能用动态规划算法求解的前提。

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）

寻找最优子结构的共同模式：

1. 问题的一个解可以是做一个选择，做这种选择会得到一个或多个有待解决的子问题。
2. 假设对一个给定的问题，已知的是一个可以导致最优解的选择。
3. 在已知这个选择后，确定哪些子问题会随之发生，以及如何最好地描述所得到的子问题空间。
4. 利用“cut-and-paste”技术，来证明在问题的一个最优解中，使用的子问题的解本身也必须是最优的。

最优子结构在问题域中以两种方式变化:

(1) 原问题的一个最优解中使用了多少个子问题。

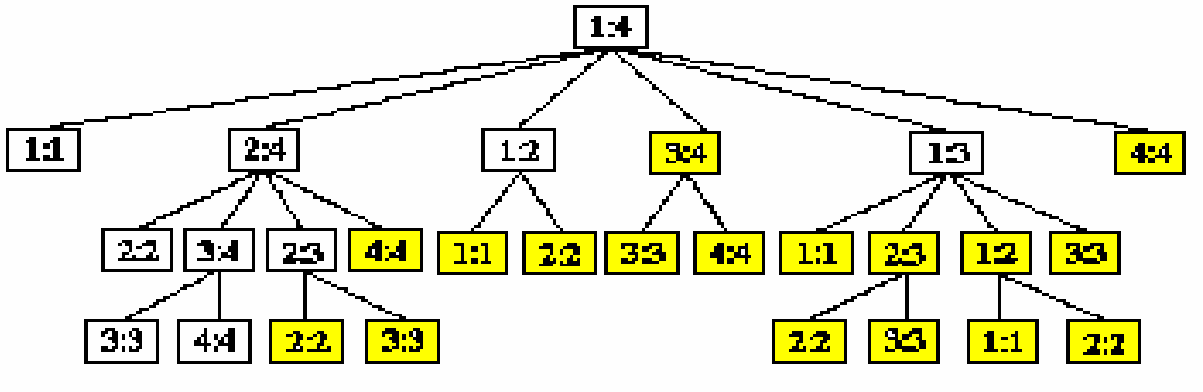
(2) 在决定一个最优解中使用哪些子问题时有多少个选择。

非正式地，一个动态规划算法的运行时间依赖于两个因素的乘积：子问题的总个数和每个子问题中有多少种选择。

动态规划算法的基本要素

二、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

三、备忘录方法

- 备忘录方法的控制结构与直接递归方法的控制结构相同，区别在于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。

```
public static int  memoizedMatrixChain(int n)
{
        for (int i = 1; i < n; i++)
              for (int j = i; j < n; j++)
                    m[i][j] = 0;
      return   LookupChain(1,n);
}
```

```
private static int  LookupChain(int i，  int j)
{
      if (m[i][j] > 0) return m[i][j]; //该子问题已求解
      if (i == j) return 0;
      int u = LookupChain(i+1，  j) + p[i-1]*p[i]*p[j];
      s[i][j] = i;
      for (int k = i+1; k < j; k++) {
        int t = LookupChain(i,k) + LookupChain(k+1,j)
                    + p[i-1]*p[k]*p[j];
        if (t < u) { u = t; s[i][j] = k;}
        }
      m[i][j] = u;
      return u;
}
```

动态规划算法与备忘录方法:

相同:　用表格保存已解决子问题的答案

用数组m[ ][ ]记录子问题的最优值

耗时$O(n^3)$

不同:　动态规划算法是自底向上递归

面向问题的所有子问题都至少要解一次

时间复杂度好一个常数因子

备忘录方法是自顶向下递归

面向问题的部分子问题可不必求解。

第三章 动态规划—— 自底向上

- 1. 动态规划概述
- 2. 矩阵连乘问题
- 3. 动态规划的基本要素
- **4. 字符串匹配**
- 5. 图像压缩
- 6. 流水作业调度
- 7. 0-1背包问题
- 8. 最优二叉树

字符串匹配的动态规划算法

- 最长公共子序列
- 序列比对
  - 全局算法
  - 局部算法

实际应用：Web 检索

ocurrance
occurrence

o-currance
occurrence

o-curr-ance
occurre-nce

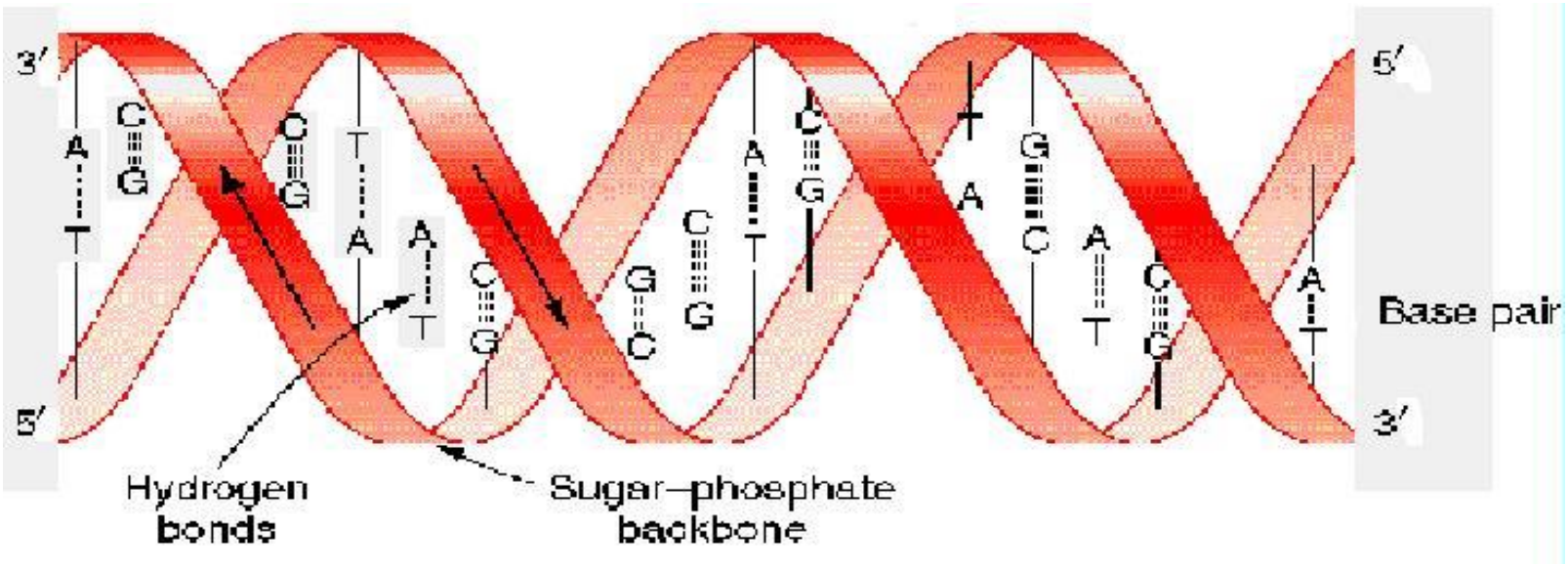
- 在2015-10-16日召开的首都国际癌症论坛上，中国抗癌协会秘书长王瑛说，据2012年癌症有关报告显示，我国每年新发癌症病例约337万，死亡约211万。“癌症已成为我国死亡第一大原因，死亡人数占全球癌症死亡人数四分之一。”

- 2012年世界癌症报告显示，每年癌症新发病例约1400万，死亡约800万，我国新发病例占全球新发病例22%，死亡人数占26%，超过全球癌症死亡人数的四分之一。男性当中肺癌发病率最高，女性是乳腺癌。

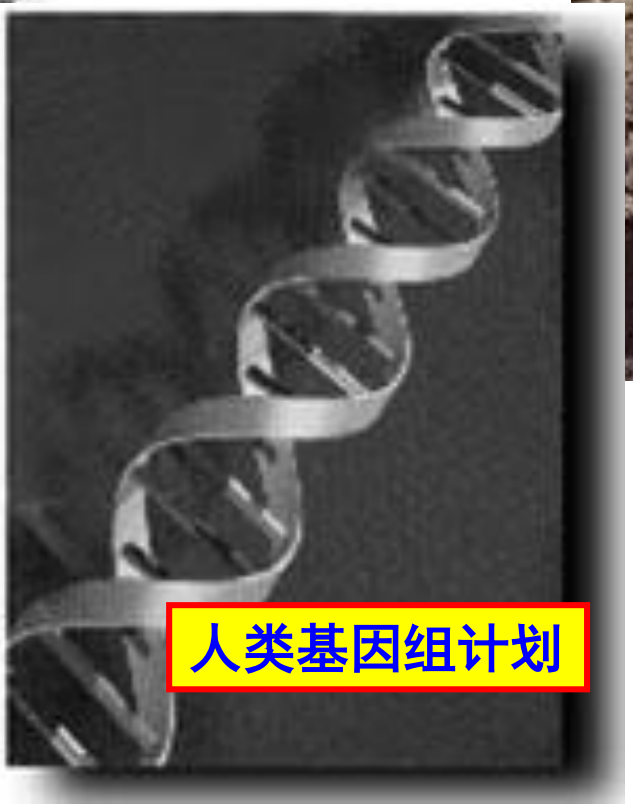
实际应用：DNA序列匹配

- DNA的双螺旋结构

  碱基对之间的互补能力

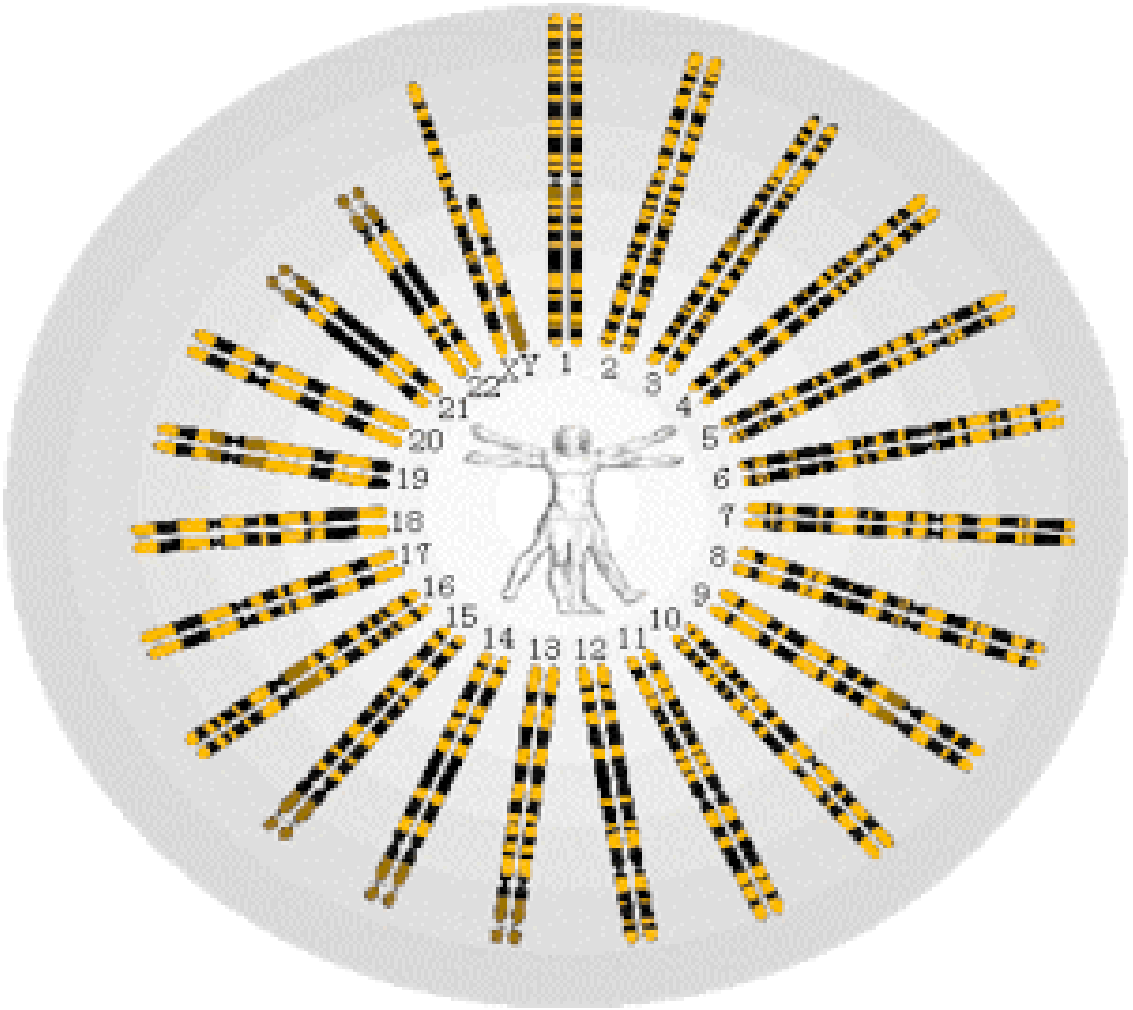


从人类基因组计划 (HGP)说起



- **基因组(Genome)：** 包含细胞或生物体全套的遗传信息的全部遗传物质。原核生物(细菌、病毒等)
  真核生物(真菌、植物、动物等)

- **人类基因组：**
  3.2×10^9 bp



HGP的历史回顾

1984.12 犹他州阿尔塔组织会议，初步研讨测定人类整个基因组DNA序列的意义。

1985 Dulbecco在《Science》撰文 “肿瘤研究的转折点: 人类基因组的测序”，美国能源部(DOE)提出“人类基因组计划”草案。

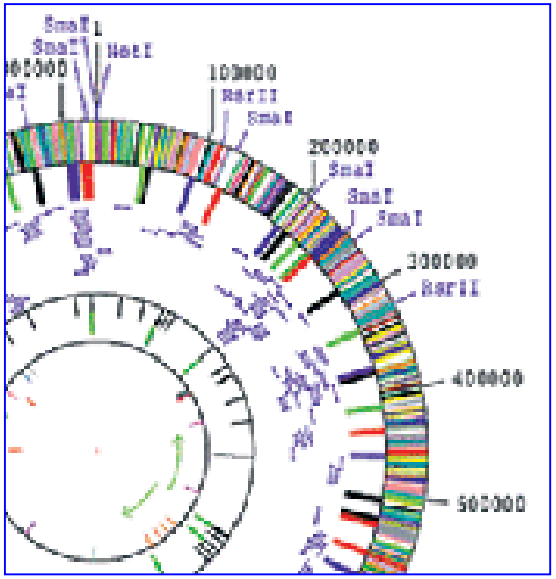
1987 美国能源部和国家卫生研究院（NIH）联合为“人类基因组计划”下拨启动经费约550万美元。

1989 美国成立“国家人类基因组研究中心”，Watson担任第一任主任。

1990.10 经美国国会批准，人类基因组计划正式启动。

1995 第一个自由生物体流感嗜血菌(*H.inf*)的全基因组测序完成。

1996 完成人类基因组计划的遗传作图，启动模式生物基因组计划。



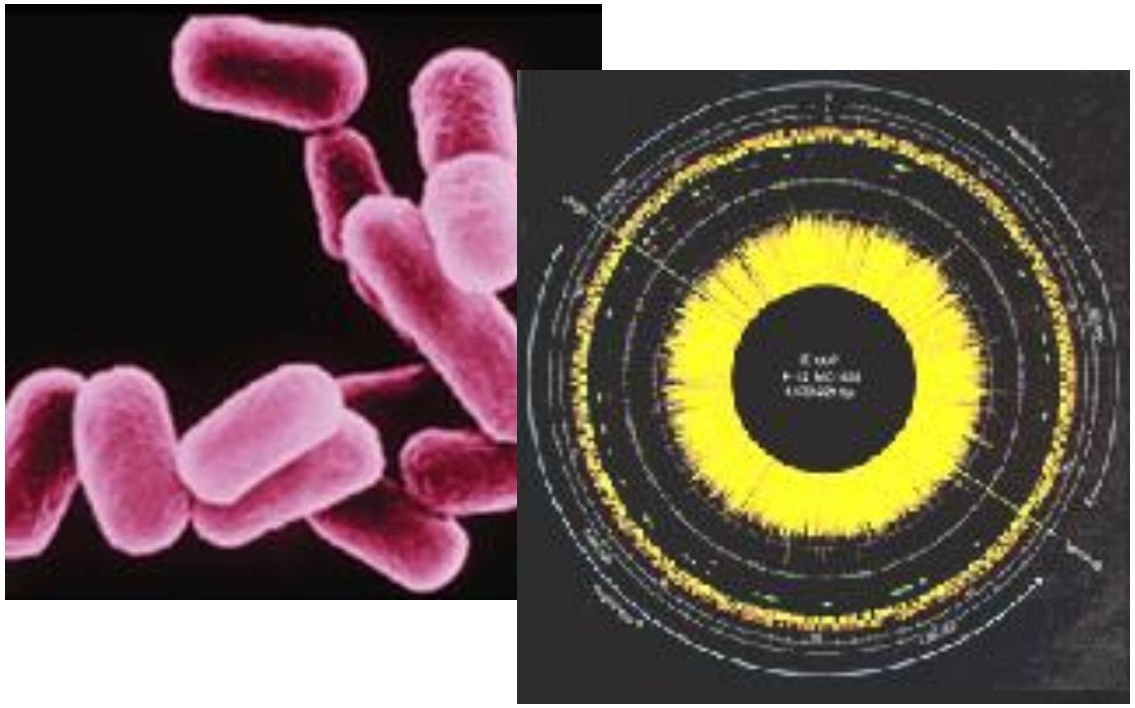
***H.inf*全基因组**

Saccharomyces cerevisiae
酿酒酵母

Caenorhabditis elegans
秀丽线虫

1997 大肠杆菌(*E.coli*)全基因组测序完成。

1998 完成人类基因组计划的物理作图，开始人类基因组的大规模测序，Celera公司加入，与公共领域竞争，启动水稻基因组计划。



大肠杆菌及其全基因组

水稻基因组计划

1999.7 第5届国际公共领域人类基因组测序会议，加快测序速度。

2000 Celera公司宣布完成果蝇基因组测序，国际公共领域宣布完成第一个植物基因组——拟南芥全基因组的测序工作。

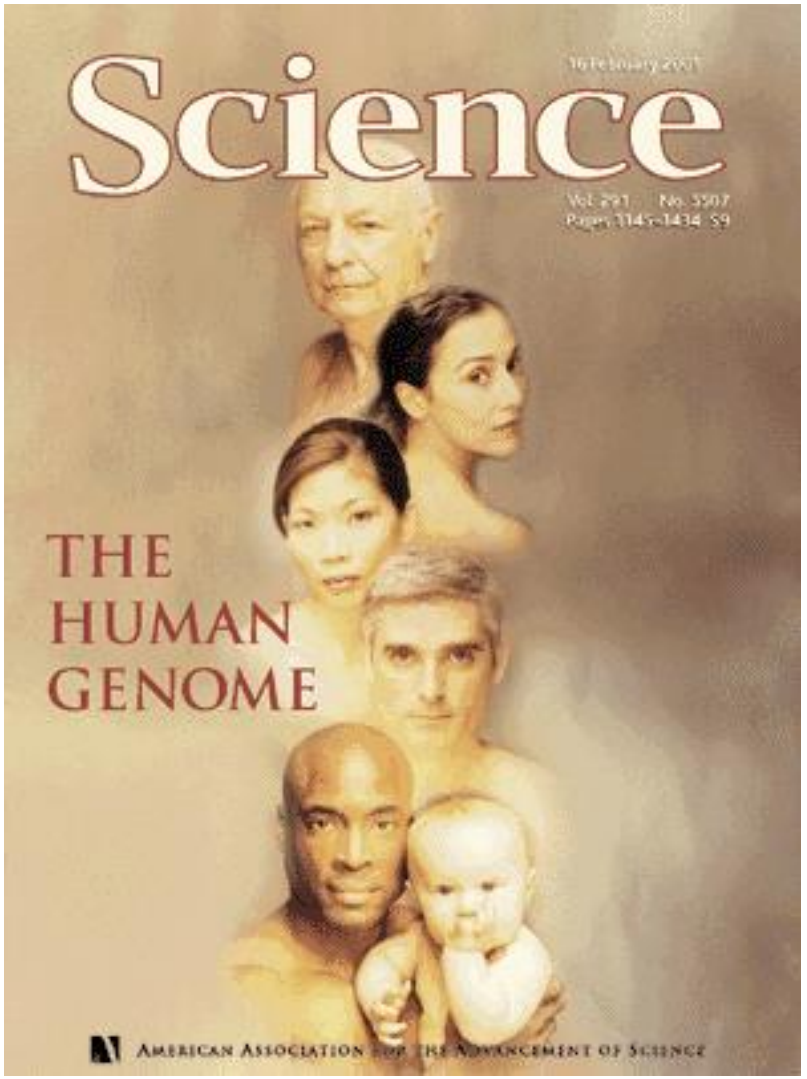
Drosophila melanogaster
果蝇

Arabidopsis thaliana
拟南芥

2000.6.26 公共领域和Celera公司同时宣布完成人类基因组工作草图
2001.2.15 《Nature》刊文发表国际公共领域结果
2001.2.16 《Science》刊文发表Celera公司及其合作者结果

2001年2月15日《Nature》封面

2001年2月16日《Science》封面



How many characters are in the “Heaven Book”?

$3*10^9$ 10,000 books

1 book 100 pages

1 page 3,000 characters

CCGGTCTCCCCGCCCGCGCGCGAAGTAAAGGCCCAGCGCAGCCCGCGCTCCTGCCCTGG
GGCCTCGTCTTTCTCCAGGAAAACGTGGACCGCTCTCCGCCGACAGTCTCTTCCACAGAC
CCCTGTCGCCTTCGCCCCCCGGTCTCTTCCGGTTCTGTCTTTTCGCTGGCTCGATACGAAC
AAGGAAGTCGCCCCCAGCGAGCCCCGGCTCCCCCAGGCAGAGGCGGCCCCGGGGGCGG
AGTCAACGGCGGAGGCACGCCCTCTGTGAAAGGGCGGGGCATGCAAATTCGAAATGAAA
GCCCGGGAACGCCGAAGAAGCACGGGTGTAAGATTTCCCTTTTCAAAGGCGGGAGAATAA
GAAATCAGCCCGAGAGTGTAAGGGCGTCAATAGCGCTGTGGACGAGACAGAGGGAATGG
GGCAAGGAGCGAGGCTGGGGCTCTCACCGCGACTTGAATGTGGATGAGAGTGGGACGGT
GACGGCGGGCGCGAAGGCGAGCGCATCGCTTCTCGGCCTTTTGGCTAAGATCAAGTGTA
GTATCTGTTCTTATCAGTTTAATATCTGATACGTCCTCTATCCGAGGACAATATATTAAATGG
ATTGATCAATCCGCTTCAGCCTCCCGAGTAGCTGGGACTACAGACGGTGCCATCACGCCC
AGCTCATTGTTGATTCCCGCCCCCTTGGTAGAGACGGGATTCCGCTATATTGCCTGGGCTG
GTGTCGAACTCATAGAACAAAGGATCCTCCCTCCTGGGCCTGGGCGTGGGCTCGCAAAAC
GCTGGGATTCCCGGATTACAGGCGGGCGCACCACACCAGGAGCAAACACTTCCGGTTTTA
AAAATTCAGTTTGTGATTGGCTGTCATTCAGTATTATGCTAATTAAGCATGCCCGGTTTTAAA
CCTCTTAAAACAACTTTTAAAATTACCTTTCCACCTAAAACGTTAAAATTTGTCAAGTGATAAT
ATTCGACAAGCTGTTATTGCCAAACTATTTTCCTATTTGTTTCCTAATGGCATCGGAACTAG
CGAAAGTTTCTCGCCATCAGTTAAAAGTTTGCGGCAGATGTAGACCTAGCAGAGGTGTGCG
AGGAGGCCGTTAAGACTATACTTTCAGGGATCATTTCTATAGTGTGTTACTAGAGAAGTTTC
TCTGAACGTGTAGAGCACCGAAAACCACGAGGAAGAGAGGTAGCGTTTTCATCGGGTTAC
CTAAGTGCAGTGTCCCCCCTGGCGCGCAATTGGGAACCCCACACGCGGTGTAGAAATATA
TTTTAAGGGCGCG

Where is the gene ?

(1250 characters)

DNA数据

- 美国的核酸数据库GenBank〖Banson,D.A. et al. (1998) Nucleic Acids Res. 26, 1-7〗从1979年开始建设，1982年正式运行；
- 欧洲分子生物学实验室的EMBL数据库也于1982年开始服务；
- 日本于1984年开始建立国家级的核酸数据库DDBJ，并于1987年正式服务。
- 从此，DNA序列的数据已经从80年代初期的百把条序列，几十万碱基上升至现在的110亿碱基！即在短短的约18年间，数据量增长了近十万倍。

基因?

- DNA上具有特定功能的一个片断，负责一种特定性状的表达。

- 在生物学研究中，将未知序列同已知序列进行比较分析已经成为一种强有力的研究手段，生物序列相似性比较中绝大部分的问题在计算机科学领域中主要体现为:

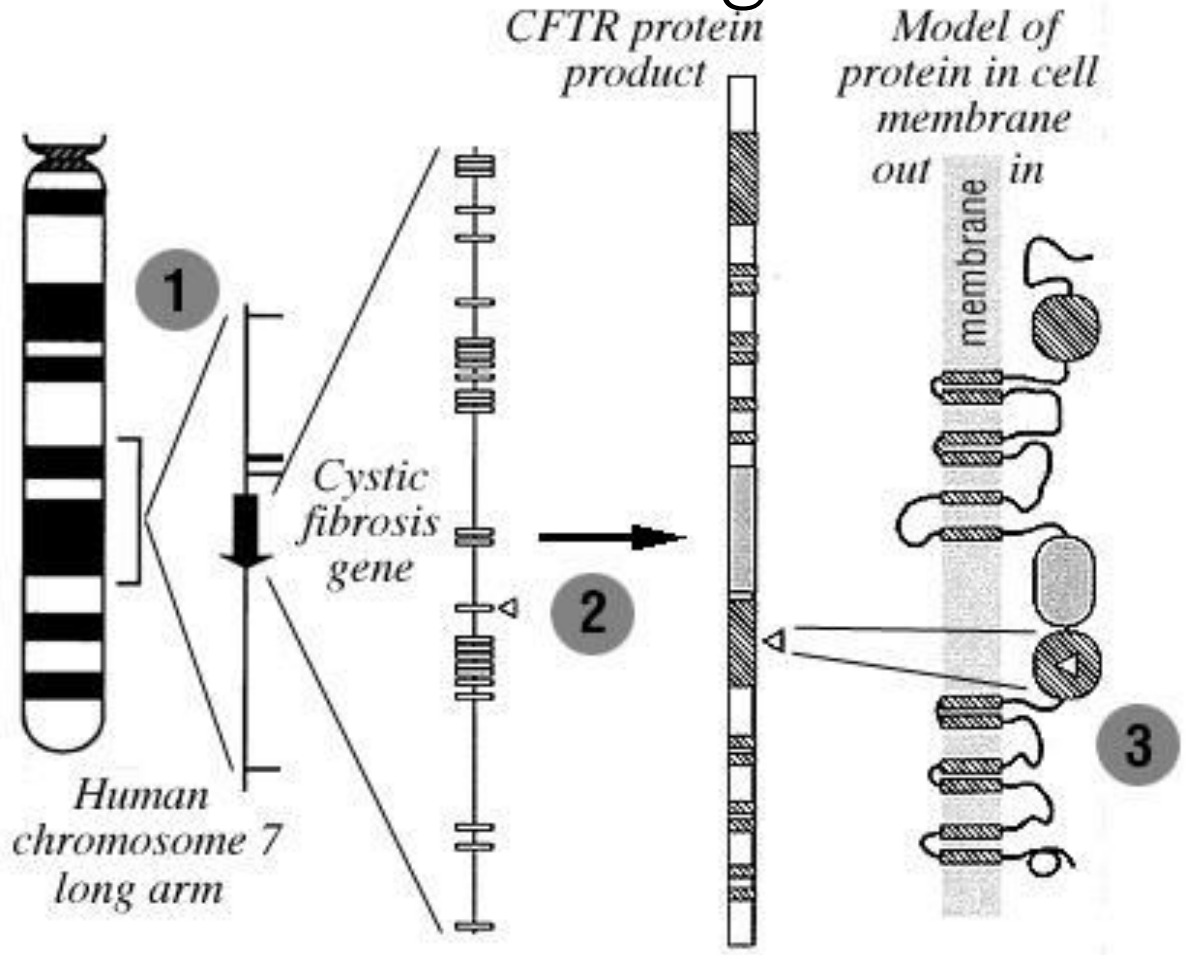
字符串的匹配和查找！

DNA 序列比较的成功案例

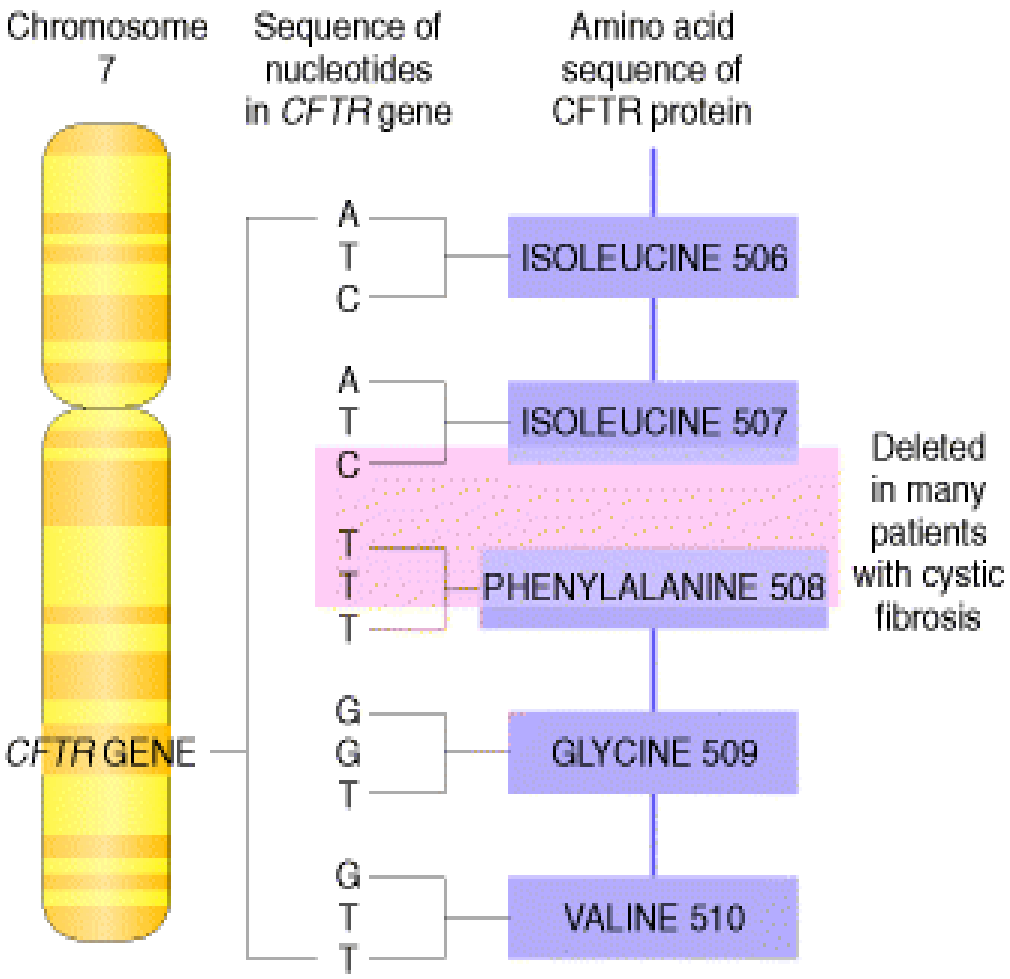
- 1984 Russell Doolittle团队发现致癌基因和 正常生长因子 (PDGF) 的相似性。

- **Cystic fibrosis (CF) 囊性纤维症**，遗传疾病，腺体黏液水平高于正常值，影响儿童的呼吸系统。
  - 1980s 早期，**Francis Collins** 假设 CF 是由基因变异引起的常染色体隐性失常所导致的疾病
  - 1989 确认.

Cystic Fibrosis: Finding the Gene

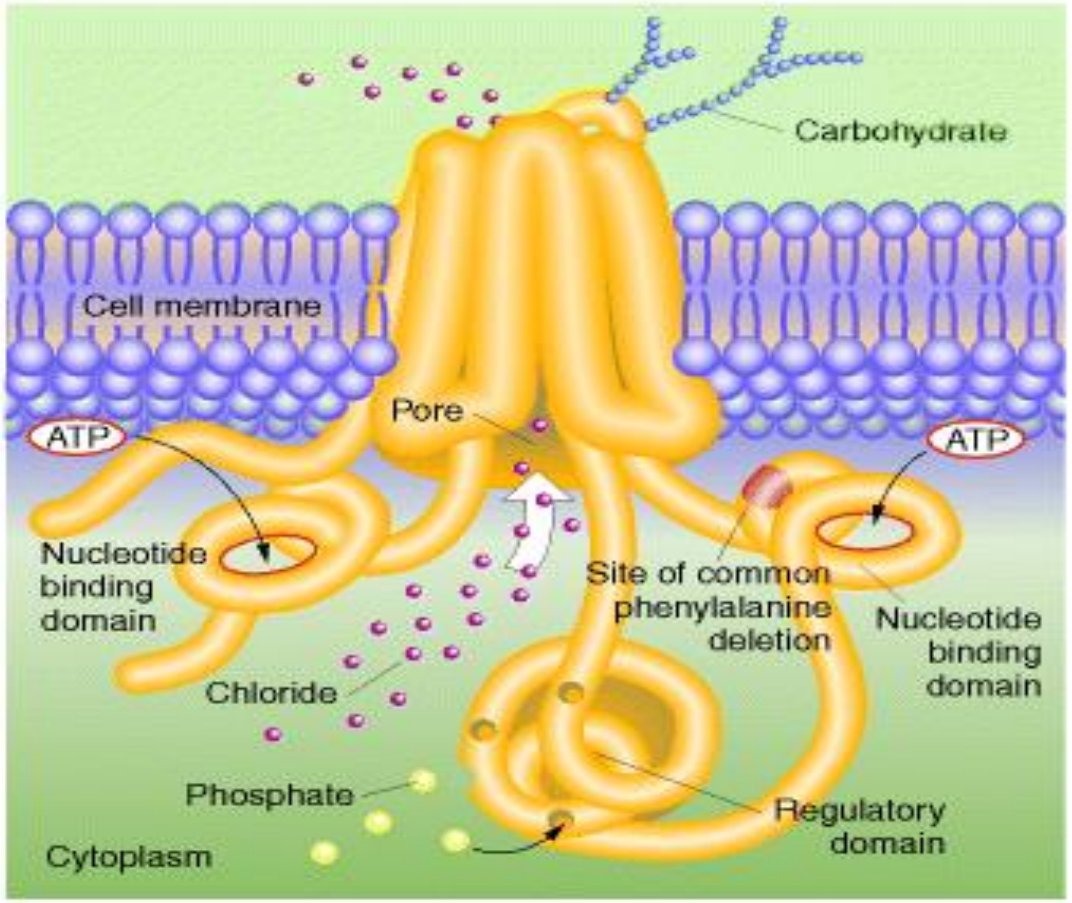


Cystic Fibrosis and CFTR Gene :



Cystic Fibrosis and the CFTR Protein

- **CFTR (Cystic Fibrosis Transmembrane conductance Regulator)** protein is acting in the cell membrane of epithelial cells that secrete mucus.
- These cells line the airways of the nose, lungs, the stomach wall, etc.



大规模基因组测序中的信息分析

- 测序之前全是生物学问题。
- 测序之后就是计算机问题。
- 天然的形式化：ATCG
- 核心问题：
  - 序列拼接
  - 填补序列间隙
  - 原因是测试仪限制，海量数据和高度重复序列

大规模基因组测序中的信息分析

- 测序之前全是生物学问题。
- 测序之后就是计算机问题。
- 天然的形式化：ATCG
- 核心问题：
  - **序列拼接**
  - **填补序列间隙**
  - **原因是测试仪限制，海量数据和高度重复序列**

3.3 最长公共子序列LCS

- **完整基因组的比较研究**
- 序列相似性比较（比对）。完成这一工作需要使用两两序列比较算法。常用的程序包有BLAST、FASTA等。
- 两个(或多个)不同有机体的DNA链比较

  {A-腺嘌呤，C-胞核嘧啶，G -鸟嘌呤，T-胸腺嘧啶}

S1=ACCGGTCGAGTGCGCGGAAGCCGGCCGAA

S2=GTCGTTCGGAATGCCGTTGCTCTGTAAA

3.3 最长公共子序列LCS

- S1和S2是否相似？相似程度？

- S1=ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
- S2=GTCGTTCGGAATGCCGTTGCTCTGTAAA

- S3=GTCGTCGGAAGCCGGCCGAA

最长公共子序列LCS

- 例： S = “a c g c t g”和T = “c a t g t”

- 三种可能的最优比对序列:

S: a c g c t g -
T: - c – a t g t

S: a c g c t g -
T: - c a – t g t

S: - a c g c t g
T: c a t g - t -

最长公共子序列LCS

- 若给定两序列 $X=\{x_1,x_2,\cdots,x_m\}$,

  $Z=\{z_1,z_2,\cdots,z_k\}$,

  Z是X的子序列，指存在一个严格递增下标序列$\{i_1,i_2,\cdots,i_k\}$使得对于所有 $j=1,2,\cdots,k$， 有：$z_j = x_{i_j}$

例如：X={A，B，C，B，D，A，B}

Z={B，C，D，B},

Z是X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LCS

- 问题：给定两个序列 $X=\{x_1,x_2,\cdots,x_m\}$和 $Y=\{y_1,y_2,\cdots,y_n\}$，找出 X 和 Y 的最长公共子序列。

- **穷举法**：对X的所有子序列，检查其是否为Y的子序列，以确定X和Y的公共子序列，并在检查过程中记载最长公共子序列。
  - X的子序列数为2^m，指数时间。 ☹

- **动态规划法** ☺
- **步骤1：描述一个最长公共子序列**
- 给定一个序列 $X=<x_1,x_2,\cdots,x_m>$，对i=0,1,…,m,定义X的第i个前缀为$X_i=<x_1,x_2,\cdots,x_i>$，X_0为空序列。

最长公共子序列的结构

- 设序列 $X=\{x_1,x_2,\cdots,x_m\}$
  和 $Y=\{y_1,y_2,\cdots,y_n\}$ 的最长公共子序列为：
  $Z=\{z_1,z_2,\cdots,z_k\}$，则

(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}的最长公共子序列。

(2) 若 $x_m\neq y_n$ 且 $z_k\neq x_m$，则Z是X_{m-1}和Y的最长公共子序列。

(3) 若$x_m\neq y_n$ 且 $z_k\neq y_n$，则Z是X和Y_{n-1}的最长公共子序列。

由此可见，两个序列的最长公共子序列包含了这两个序列的前缀的最长公共子序列。因此，最长公共子序列问题具有**最优子结构性质**。

最长公共子序列LCS

- **步骤2：子问题的递归结构**
- 由最长公共子序列问题的最优子结构性质建立子问题最优值的递归关系。
- 用 c[i][j] 记录序列 X_i和 Y_j的最长公共子序列的长度。其中，$X_i=\{x_1,x_2,\cdots,x_i\}$；$Y_j=\{y_1,y_2,\cdots,y_j\}$。
- 当 i=0 或 j=0时，空序列是X_i和Y_j的最长公共子序列。故此时c[i][j]=0。
- 其它情况下，由最优子结构性质可建立递归关系如下：

$$c[i][j]=\begin{cases}0 & i=0,\ j=0\\ c[i-1][j-1]+1 & i,\ j>0;\ x_i=y_j\\ \max\{c[i][j-1],c[i-1][j]\} & i,\ j>0;\ x_i\neq y_j\end{cases}$$

最长公共子序列LCS

- **步骤3：计算最优值**
- 直接利用递归式计算 c[i][j] 的算法，计算时间随输入长度指数增长。
- 由于在所考虑的子问题空间中，总共有$\Theta(mn)$个不同的子问题，因此，用动态规划算法自底向上地计算最优值能提高算法的效率。

```
public static int  LCSLength(char[]  x,   char[] y,   int[][] b)
{
      int m=x.length,   n=y.length;
      int [][] c= new int [m][n];
      for (int i = 1; i <= m; i++) c[i][0] = 0;
      for (i = 1; i <= n; i++) c[0][i] = 0;
      for (i = 1; i <= m; i++)
         for (j = 1; j <= n; j++) {
            if (x[i]==y[j]) {
                 c[i][j] = c[i-1][j-1] +1;  b[i][j]=1; }
            else if (c[i-1][j] >= c[i][j-1]) {  //Xi-1与Yj 的最长公共子序列
                 c[i][j] = c[i-1][j]; b[i][j]=2; }
                   else { c[i][j]=c[i][j-1]; b[i][j]=3; } //Xi与Yj -1的最长公共子序
            }
      return  c[m][n];
}
```

最长公共子序列LCS

- **步骤4：构造最长公共子序列**

```
public static void LCS(int i,  int j,  char[] x,  int[][] b)
{
    if (i ==0 || j==0) return;
    if (b[i][j]== 1){
    LCS(i-1,  j-1,  x,  b);
    System.out.print(x[i]);
    }
    else if (b[i][j]==2) LCS(i-1,  j,  x,  b);
            else LCS(i,  j-1,  x,  b);
}
```

c / b

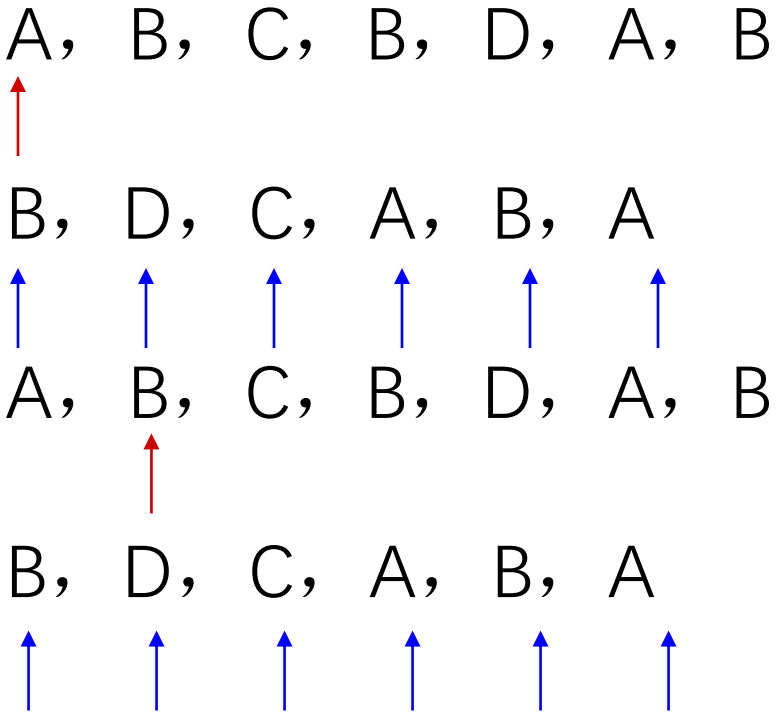
	y_j	B	D	C	A	B	A
	0	1	2	3	4	5	6
x_i 0	0	0	0	0	0	0	0
A 1	0	0/2	0/2	0/2	1/1	1/3	1/1
B 2	0	1/1	1/3	1/3	1/2	2/1	2/3
C 3	0	1/2	1/2	2/1	2/3	2/2	2/2
B 4	0	1/1	1/2	2/2	2/2	3/1	3/3
D 5	0	1/2	2/1	2/2	2/2	3/2	3/2
A 6	0	1/2	2/2	2/2	3/1	3/2	4/1
B 7	0	1/1	2/2	2/2	3/2	4/1	4/2

实例

X={A，B，C，B，D，A，B}

Y={B，D，C，A，B，A}

LCSLength(x，y，b)



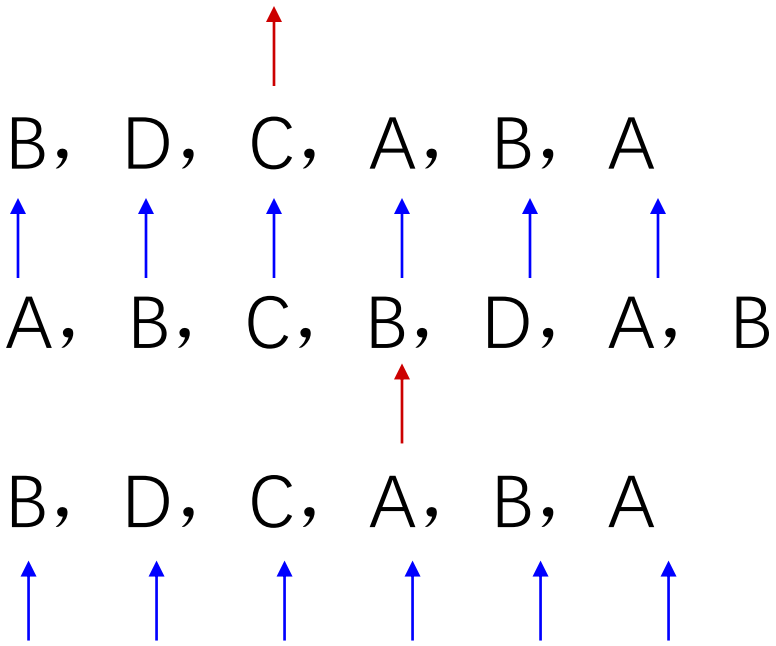
c / b

i \ j	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0/2	0/2	0/2	1/1	1/3	1/1
2	0	1/1	1/3	1/3	1/2	2/1	2/3
3	0	1/2	1/2	2/1	2/3	2/2	2/2
4	0	1/1	1/2	2/2	2/2	3/1	3/3
5	0	1/2	2/1	2/2	2/2	3/2	3/2
6	0	1/2	2/2	2/2	3/1	3/2	4/1
7	0	1/1	2/2	2/2	3/2	4/1	4/2

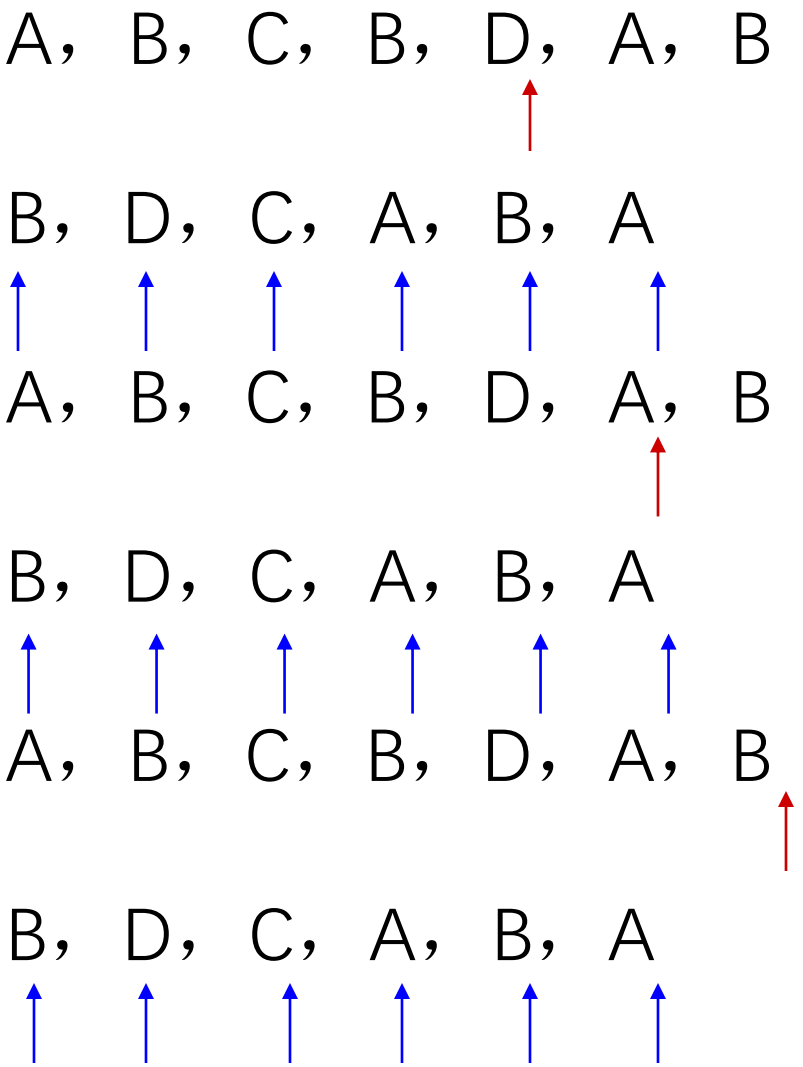
X={A, B, C, B, D, A, B}

Y={B, D, C, A, B, A}

LCSLength(x, y, b)



i \ c / b, j	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0/2	0/2	0/2	1/1	1/3	1/1
2	0	1/1	1/3	1/3	1/2	2/1	2/3
3	0	1/2	1/2	2/1	2/3	2/2	2/2
4	0	1/1	1/2	2/2	2/2	3/1	3/3
5	0	1/2	2/1	2/2	2/2	3/2	3/2
6	0	1/2	2/2	2/2	3/1	3/2	4/1
7	0	1/1	2/2	2/2	3/2	4/1	4/2



c / b

j

i	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0/2	0/2	0/2	1/1	1/3	1/1
2	0	1/1	1/3	1/3	1/2	2/1	2/3
3	0	1/2	1/2	2/1	2/3	2/2	2/2
4	0	1/1	½	2/2	2/2	3/1	3/3
5	0	1/2	2/1	2/2	2/2	3/2	3/2
6	0	1/2	2/2	2/2	3/1	3/2	4/1
7	0	1/1	2/2	2/2	3/2	4/1	4/2

X={A，B，C，B，D，A，B}

LCS(7，6，x，b)

b[7][6]=2

LCS(7-1，6，x，b)

b[6][6]=1

LCS(6-1，6-1，x，b)

b[5][5]=2

LCS(5-1，5，x，b)

b[4][5]=1

LCS(4-1，5-1，x，b)

b[3][4]=3

LCS(3，4-1，x，b)

b[3][3]=1

LCS(3-1，3-1，x，b)

b[2][2]=3

LCS(2，2-1，x，b)

b[2][1]=1

LCS(2-1，1-1，x，b) return

Z={B，　　} ……

Z={B，C，B，A}　i=2，3，4，6

c / b

		y_j	B	D	C	A	B	A
		0	1	2	3	4	5	6
x_i	0	0	0	0	0	0	0	0
A	1	0	0/2	0/2	0/2	1/1	1/3	1/1
B	2	0	1/1	1/3	1/3	1/2	2/1	2/3
C	3	0	1/2	1/2	2/1	2/3	2/2	2/2
B	4	0	1/1	1/2	2/2	2/2	3/1	3/3
D	5	0	1/2	2/1	2/2	2/2	3/2	3/2
A	6	0	1/2	2/2	2/2	3/1	3/2	4/1
B	7	0	1/1	2/2	2/2	3/2	4/1	4/2

算法复杂度分析

- 空间：
  - X[m], Y[n], c[m][n], b[m][n]
  - 算法所占用的空间为O(mn)。
- 时间：
  - 算法LCSLength的主要计算量取决于算法中对 I 和 j 的两重循环。循环体内的计算量为O(1)，而两重循环的总次数为O(mn)。因此算法的计算时间上界为O(mn)。

算法空间复杂度的改进

- 在算法lcsLength和lcs中，可进一步将数组b省去。事实上，数组元素c[i][j]的值仅由c[i-1][j-1]，c[i-1][j]和c[i][j-1]这3个数组元素的值所确定。对于给定的数组元素 c[i][j]，可以不借助于数组 b 而仅借助于 c 本身在O(1)时间内确定 c[i][j]的值是由 c[i-1][j-1]，c[i-1][j]和c[i][j-1]中哪一个值所确定的。

- 如果只需要计算最长公共子序列的长度，则可减少算法的空间需求。事实上，在计算c[i][j]时，只用到数组c的第i行和第 i-1行。因此，用两行的数组空间就可以计算出最长公共子序列的长度。进一步的分析还可将空间需求减至O(min(m,n))。

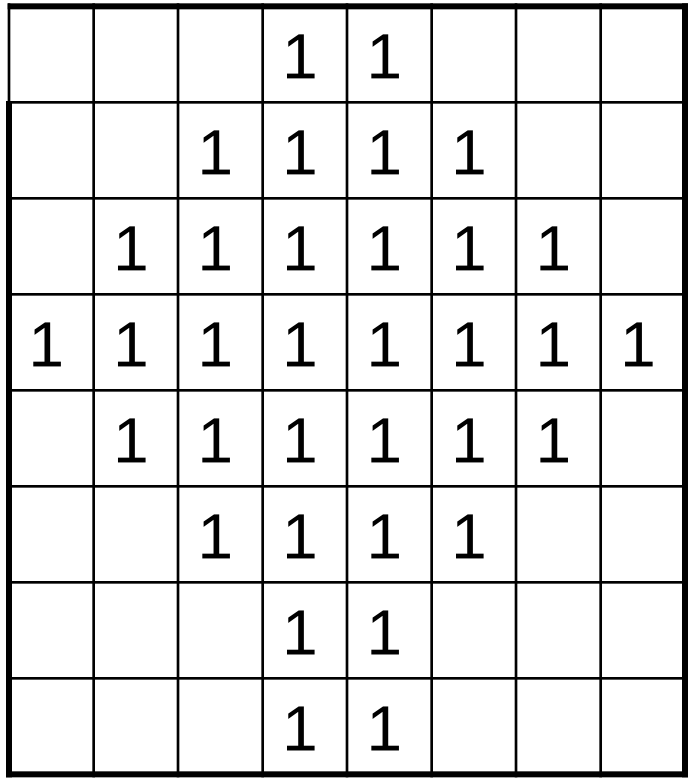
第三章 动态规划—— 自底向上

- 1. 动态规划概述
- 2. 矩阵连乘问题
- 3. 动态规划的基本要素
- 4. 字符串匹配
- **5. 图像压缩**
- 6. 流水作业调度
- 7. 0-1背包问题
- 8. 最优二叉树

3.5 图像压缩

- 数字化图象：
  - 黑白（单色0，1）图象

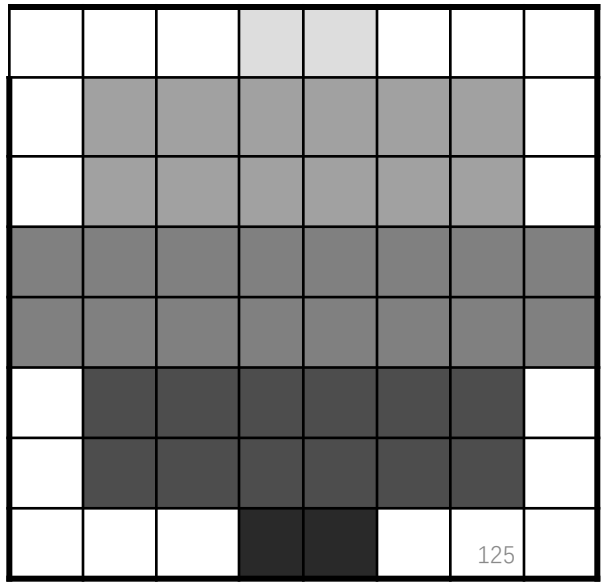
    m x m像素阵列，存储空间m^2



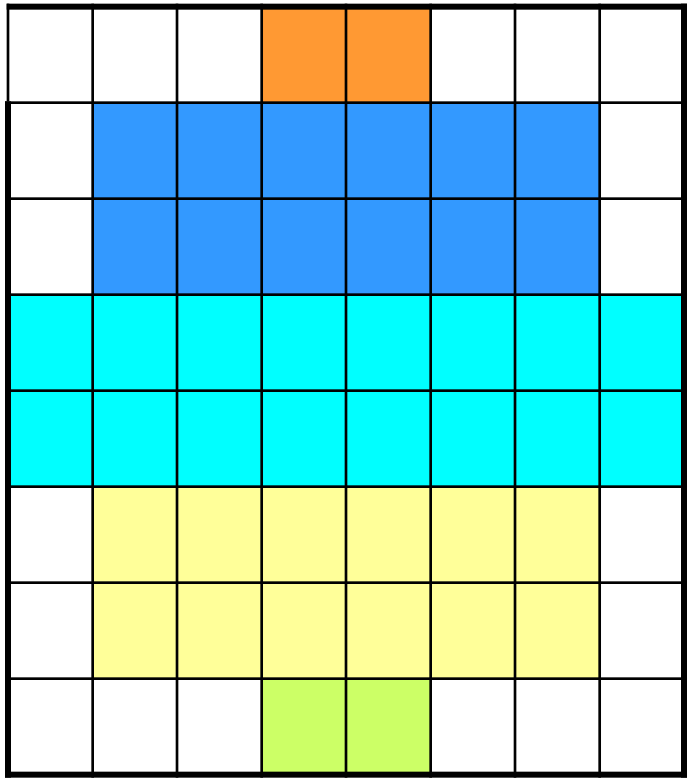
  - 灰度图象(0~255灰度值)

    8 x m x m像素阵列，存储空间$8m^2$

    颜色数为$2^8 = 256$种



3.5 图像压缩



- 彩色图象：
  - 一幅彩色图像的每个像素用R，G，B 三个分量表示，若每个分量用8位，那么一个像素共用24位表示。
  - 颜色数为2^{24} ＝16 777 216种
  - 存储空间24m^2
  - 用RGB 5：5：5表示 的彩色图像颜色数为2^{15} = 32K
  - 存储空间15m^2

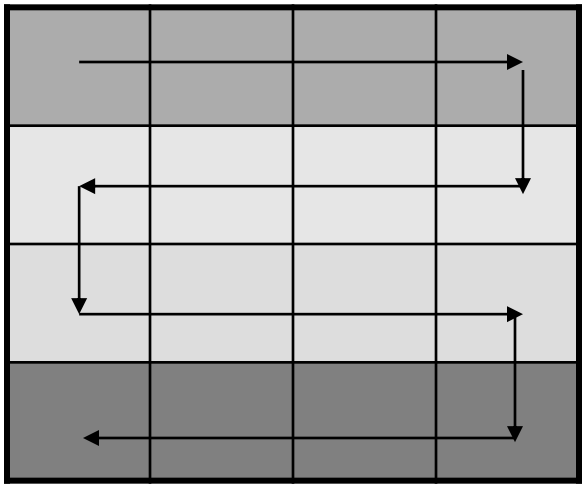
图像压缩

- 减少存储空间，采用变长模式，不同像素用不同位数来存储。

像素值	0,1	2,3	4,5,6,7	8~15	9~31	32~63	64~127	128~255
所需位数	1	2	3	4	5	6	7	8

使用变长模式的步骤:

- 1. **图象线性化**: m x m→1 x m^2



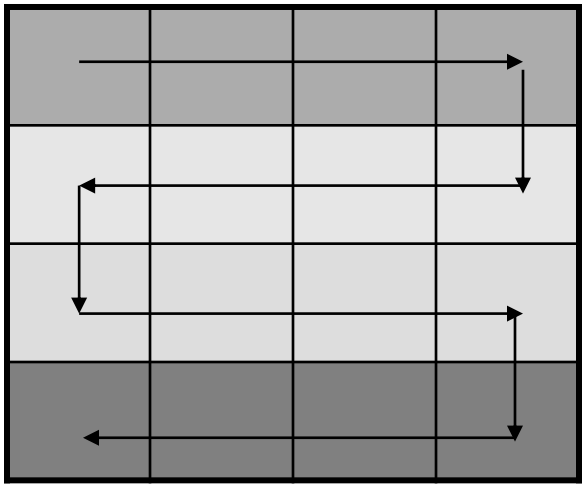
10	9	12	40
12	15	35	50
8	10	9	15
240	160	130	11

使用变长模式的步骤:

- 2. **分段**，分段规则:
  - 每段中的像素位数相同;
  - 每个段是相邻像素的集合;
  - 每个段最多含256个像素，若相邻的相同位数的像素超过256个，则用2个以上的段表示。

使用变长模式的步骤:

- 3. **创建文件**，创建3个文件:
  - SegmentLength: 包含步骤2中所建的段的长度(减1)，文件中各项均为8位长。
  - BitsPerPixels: 包括各段中每个像素的存储位数(减1)，文件中各项均为3位长。
  - Pixels: 以变长格式存储的像素的二进制串。
- 4. **压缩文件**，压缩步骤3所建立的文件，以节约空间。



10	9	12	40
12	15	35	50
8	10	9	15
240	160	130	11

- 例子： 图中4x4图象
  - 灰度值:

    {10,9,12,40,50,35,15,12,8,10,9,15,11,130,160,240}
  - 所需位数:

    { 4,4,4, 6,6,6, 4,4,4,4,4,4,4, 8,8,8 }
  - 分段:

[10,9,12], [40,50,35], [15,12,8,10,9,15,11], [130,160,240]

像素值	0,1	2,3	4,5,6,7	8~15	9~31	32~63	64~127	128~255
所需位数	1	2	3	4	5	6	7	8

例子：图中4x4图象

- 分段：[10,9,12]，[40,50,35]，[15,12,8,10,9,15,11]，[130,160,240]
- 所需位数：{ 4,4,4, 6,6,6, 4,4,4,4,4,4,4, 8,8,8 }
- 压缩文件：
  - SegmentLength: 2, 2, 6, 2； 共需8*4=32位
  - BitsPerPixels: 3, 5, 3, 7； 共需3*4=12位
  - Pixels: 共需4*3+6*3+4*7+8*3=82位

  （存储每段长度和该段中每个像素占用的位数需8+3=11）

合计126位 VS. 8*16=128

节省2位

- 改进：
  - 第1段与第2段合并：
  - SegmentLength: 5, 6, 2； 共需8*3=24位
  - BitsPerPixels: 5, 3, 7； 共需3*3=9位
  - Pixels: 共需6*6+4*7+8*3=88位

合计121位 VS. 8*16=128

节省8位

图像压缩

- 目标：设计一种算法，使得在产生 n 个段后，能对相邻段进行合并，以便产生一个具有最小空间需求的新的段集合。
- 令:Sq为前q个段的最优合并所需要的空间(位数)；
  l_i表示第i段的段长;
  b_i表示该段每个像素的长度。
- 定义：$S_0 = 0$
- 假设在最优合并C中，第i段与i-1，i-2，…i-r+1段合并，而不包括第i-r段。合并C所需要的空间消耗等于:
- 第1段到第i-r段所需要的空间 + lsum(i-r+1,i)*bmax(i-r+1,i) + 11
- 其中：$lsum(a,b) = \sum l_j$, j=a…b
  $bmax(a,b) = max\{b_a, \cdots\cdots, b_b\}$

1 最优子结构

- 设S[i], 1≤i≤n, 是像素序列{p1,…,pn}的最优分段所需的存储位数。
- $S_i = S_{i-r} + lsum(i-r+1,i) * bmax(i-r+1,i) + 11$
- 其中: r=1,…i; lsum<256
- 寻找r, 产生最小的空间需求。
- 令 kay_i 存储取得最小值时k的值。最优子结构如下:

$$s[i] = \min_{\substack{1\le k\le i \\ lsum(i-k+1,i)\le 256}} \{s[i-k] + lsum(i-k+1,i) * \mathrm{b}\max(i-k+1,i)\} + 11$$

例子:

- 一副图象的像素组分为5段,
  长度为 [6, 3, 10, 2, 3]
  像素位数为[1, 2, 3, 2, 1]

- $S_0=0$
- $S_1=S_0+l_1*b_1+11=0+6*1+11=17$ $kay_1=1$
- $S_2=\min\{S_1+l_2*b_2\ , S_0+(l_1+\ l_2)*\max\{b_1, b_1\}\}+11$

  $=\min\{17+6, 0+9*2\}+11=29$ $kay_2=2$

  ……
- S[1:5]=[17, 29, 67, 73, 82]
- kay[1:5]=[1, 2, 2, 3, 4]

2 迭代计算最优值

```
public static void compress(int[ ] l, int[ ] b, int[ ] s,int[ ] kay)
{
        int n=b.length-1;
        s[0] =0;
        for (int  i=1;i<=n; i++ ){
            int lsum=l[i], bmax=b[i];  //k=1时计算最小值
            s[i]=s[i-1]+lsum*bmax;
            kay[i]=1;
            //lmax=256，对其余的 k 计算最小值并更新
            for (int  k=2;k<=i && lsum+l[i-k+1]<=lmax; k++ ) {
                lsum+=l[i-k+1];
                if( bmax<b[i-k+1] ) bmax=b[i-k+1];
                if( s[i]>s[i-k]+lsum*bmax ) {
                    s[i]=s[i-k]+lsum*bmax;
                    kay[i]=k;   }
                }
                s[i]+=header; //header=11
        }
}
```

例子：一副图象的像素组分为5段，
长度为[6,3,10,2,3]，像素位数为[1,2,3,2,1]

l[i]

i	0	1	2	3	4	5
	0	6	3	10	2	3

b[i]

i	0	1	2	3	2	1

s[i]

i	0	1	2	3	4	5
	0	0	0	0	0	0

kay[i]

	0	0	0	0	0	0

例子：长度为[6, 3, 10, 2, 3]，像素位数为[1, 2, 3, 2, 1]

s[i]

i	0	1	2	3	4	5
	0	17	29	67	73	82

kay[i]

	0	1	2	2	3	4

s[1]=s[0]+lsum*bmax=0+6*1=6

kay[1]=1　　s[1]=s[1]+11=17

s[2]=s[1]+lsum*bmax=17+3*2=23 ×

s[2]=s[0]+lsum*bmax=0+ (3+6)*2 =18

kay[1]=2　　s[2]=18+11=29

s[i]

i	0	1	2	3	4	5
	0	17	29	67	73	82

kay[i]

	0	1	2	2	3	4

k=1

s[3]=s[2]+lsum*bmax=29+10*3=59 ×

kay[3]=1

k=2

s[3]=s[1]+lsum*bmax=17+(10+3)*3=56

kay[3]=2

k=3

s[3]=s[0]+lsum*bmax=0 + (10+3+6)*3=57 ×

s[3]=56+11=67

s[i]

i	0	1	2	3	4	5
	0	17	29	67	73	82

kay[i]

	0	1	2	2	3	4

k=1

s[4]=s[3]+lsum*bmax=67+2*2=71 ×

kay[4]=1

k=2

s[4]=s[2]+lsum*bmax=29+(2+10)*3=65 ×

kay[4]=2

k=3

s[4]=s[1]+lsum*bmax=17+(2+10+3)*3=62

kay[4]=3

k=4

s[4]=s[0]+lsum*bmax=0 + (2+10+3+6)*3=63 ×

s[4]=62+11=73

s[i]

i	0	1	2	3	4	5
	0	17	29	67	73	82

kay[i]

	0	1	2	2	3	4

k=1 s[5]=s[4]+lsum*bmax=73+3*1=76 ×

kay[5]=1

k=2 s[5]=s[3]+lsum*bmax=67+(3+2)*2=77 ×

k=3 s[5]=s[2]+lsum*bmax=29+(3+2+10)*3=74 ×

kay[5]=3

k=4 s[5]=s[1]+lsum*bmax=17 + (3+2+10+3)*3=71

kay[5]=4

k=5 s[5]=s[0]+lsum*bmax=0 + (3+2+10+3+6)*3=72 ×

s[5]=71+11=82

3 构造最优解

s[i]和kay[i]记录了最优分段的信息。
s[i]存储了最优分段的像素位数;
kay[i]存储了最优分段的最后一段的段长度。

```
public static void traceback(int n,int [] s, int [] kay)
        {
                if (n==0)return;
                traceback(n-kay[n],s,kay);
      System.out.println("new segment begins at "+(n-kay[n]+1));
        }
```

traceback(5,s,kay); kay[5]=4

traceback((5-kay[5]), s, kay); kay[1]=1

traceback((1- kay[1]), s, kay);

```
new segment begins at 1
new segment begins at 2
```

Segment: [6], [3,10,2,3]

BitsPerPixels: (1, 3)

4 计算复杂性分析

空间：

l[n], b[n], s[n], kay[n],　O(n)

时间：

由于算法**compress**中对k的循环次数不超这256，故对每一个确定的i，可在时间O(1)内完成的计算。因此整个算法所需的计算时间为O(n)。

第三章 动态规划—— 自底向上

- 1. 动态规划概述
- 2. 矩阵连乘问题
- 3. 动态规划的基本要素
- 4. 序列匹配
- 5. 图像压缩
- **6. 流水作业调度**
- 7. 0-1背包问题
- 8. 最优二叉树

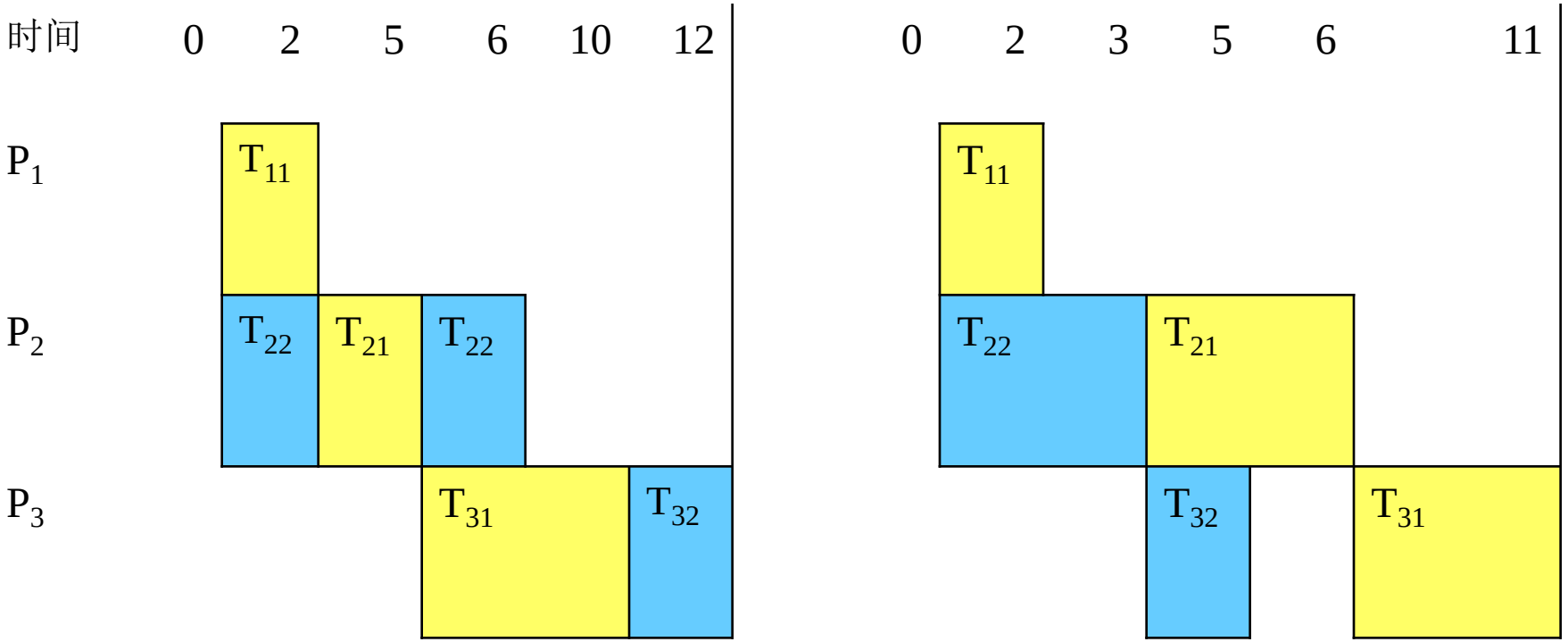
3.6 流水作业调度

- n 个作业 {1，2，…，n} 要在由 m 台机器 $P_1,\cdots,P_m$组成的流水线上完成加工。
- 每个作业加工的顺序都是先在 P_1 上加工，然后在 P_2上加工。
- $P_1,\cdots,P_m$ 加工作业 i 所需的时间分别为 $t_1,\cdots,t_m$。

- 流水作业调度问题要求确定这 n 个作业的最优加工顺序，使得从第一个作业在机器 P_1 上开始加工，到最后一个作业在机器 P_m 上加工完成所需的时间最少。

例子：在 3 台机器上调度 2 个作业，每个作业包含 3 个任务，任务完成时间分别为：

$$j = \begin{bmatrix} 2 & 0 \\ 3 & 3 \\ 5 & 2 \end{bmatrix}$$



抢先调度　　非抢先调度

流水作业调度

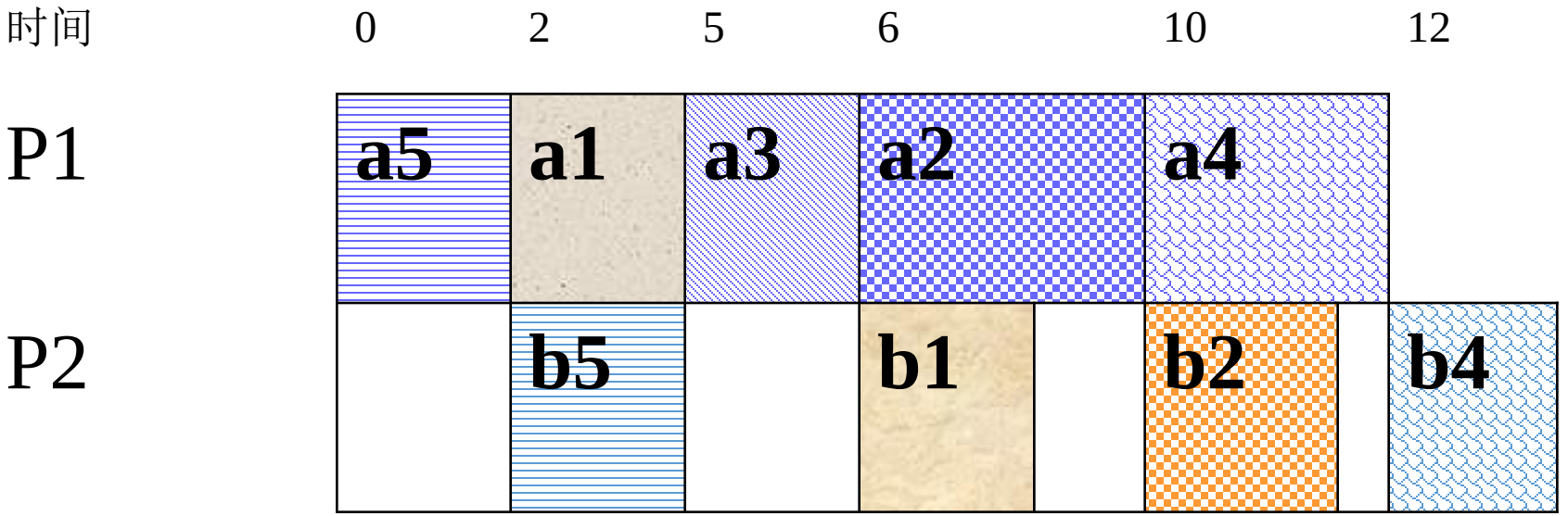
- **fi(S)**：在调度方案S下完成作业 i 的所有任务的时刻。
  - 抢先调度：　f1(S) =10　　f2(S) =12
  - 非抢先调度：f1(S) =11　　f2(S) = 5

- 调度S的完成时间F(S)：$$F(S) = \sum_{1 \le i \le n} \{f_i(S)\}$$

- 平均流动时间MFT (S)：$$MFT(S) = \frac{1}{n} \sum_{1 \le i \le n} \{f_i(S)\}$$

流水作业调度

- OFT(S)：最优完成时间调度，指在所有非抢先调度S中，F(S) 取值最小的调度。
- POFT(S)：抢先最优完成时间调度。
- OMFT(S)：最优平均完成时间调度。
- POMFT(S)：抢先最优完成时间调度。
- m>2时，OFT和POFT调度方案难以得到。



调度方案的好坏完全取决于作业在每台设备上被处理的排列次序。

$a_i \neq 0$，$1 \leq i \leq n$

$a_i = 0$，首先对于所有$a_i \neq 0$的作业求出一种最优调度的排列，然后把所有的$a_i = 0$的作业以任意次序加到这一排列的前面。

最优调度的排列具有下述性质:

- 给出了这个排列的第一个作业后，剩下的排列相对于这两台设备在完成第一个作业时所处的状态而言是最优的。
- 设作业1，2，…, k 的一种调度排列为σ_1，σ_2，…, σ_k，对于这种调度，
- 设 h_1 表示在设备P_1 上完成任务 (T_{11},T_{12}, …, T_{1K}) 的时间.
- 设 h_2 表示在设备P_2 上完成任务 (T_{21},T_{22}, …, T_{2K}) 的时间.

3.6 流水作业调度

- 设 t = h2-h1

- t 表示如果要在设备 P_1 和 P_2 上处理其他作业，则必须在这两台设备同时处理前 k 个作业的不同任务后，设备 P_2还要用大小为 t 的时间段处理前 k 个作业中未处理完的任务 。

- 设 S 为其他作业的集合。

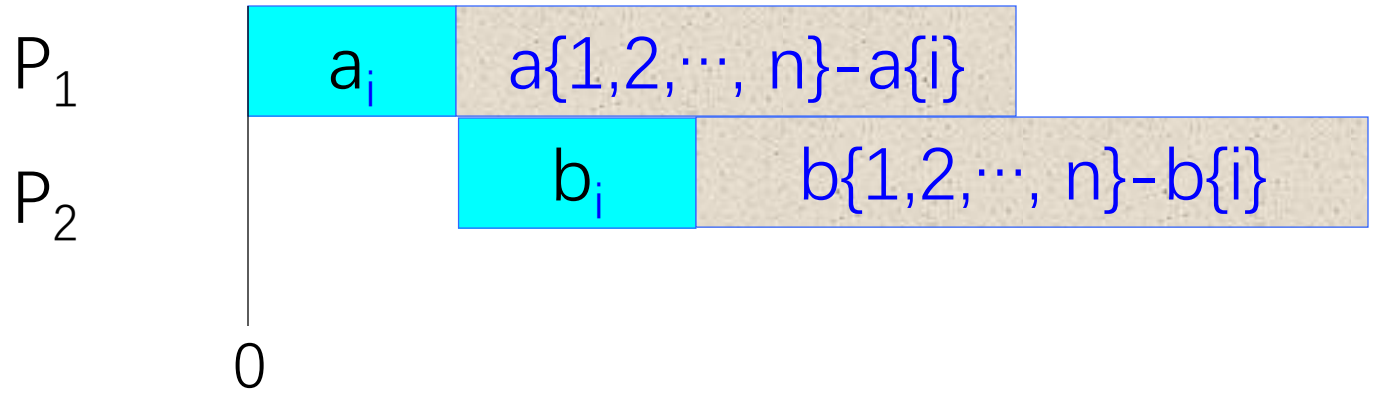
  g(S，t)是上述 t 下 S 的最优调度长度。

- 则作业集 {1，2，…,n} 的最优长度是 g({1,2,…, n}，0)

3.6 流水作业调度

由最优调度排列的性质，可得：

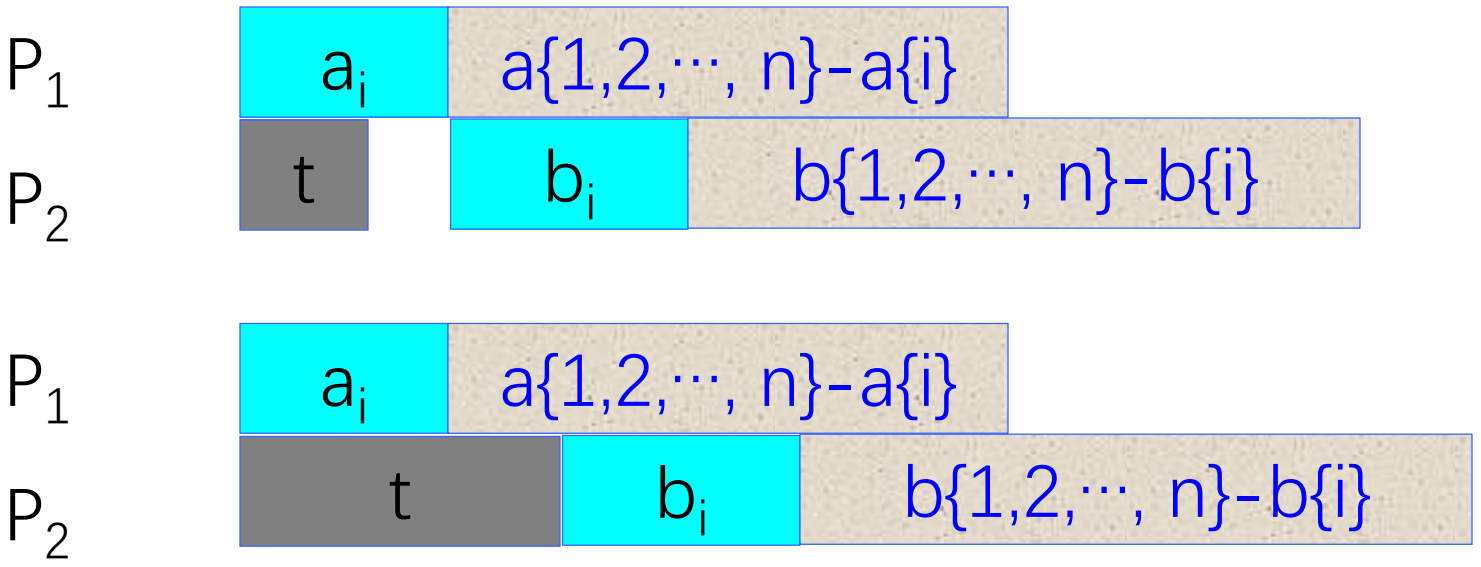
$$g(\{1,2,\cdots,n\},0)=\min_{1\le i\le n}\{a_i+g(\{1,2,\cdots,n\}-\{i\},b_i)\}$$



3.6 流水作业调度

推而广之，对于任意的 S 和 i ,在条件 $g(\Phi, t)=\max\{t,0\}$, 且$a_i \neq 0$，$1 \le i \le n$ 下：

$$g(S, t) = \min_{i \in S}\{ a_i + g(S-\{i\}, b_i+\max\{t-a_i, 0\}) \} \quad (1)$$



max { t-a_i，0}:

由于任务 T_{2i} 在 max{a_i，t } 这段时间及以前不能使用设备P_2，因此，

$h_2-h_1 = b_i + \max\{ a_i,t \} - a_i = b_i + \max \{t-a_i,0\}$

g({1,2,…, n}，0)的时间复杂度$O(2^n)$ ☹效率太低

考虑S的任意一种调度R，假设直到 t 这段时间P_2都不能用来处理 S 中的作业，如果 i 和 j 是这种调度中排在前面的两个作业，则由公式 (1) 可得：

g(S，t)

$= a_i + g(S-\{i\}，\ b_i+\max\{ t-a_i，\ 0\})$

$= a_i + a_j + g(S-\{i,j\}，\ b_j+\max\{b_i+\max\{t-a_i，\ 0\}-a_j,0\})$

令 $t_{ij} = b_j + \max\{ b_i+\max\{t-a_i, 0\}-a_j, 0 \}$

$= b_j + b_i - a_j + \max\{ \max\{t-a_i, 0\}, a_j-b_i \}$

$= b_j + b_i - a_j + \max\{ t-a_i, a_j-b_i, 0 \}$

$= b_j + b_i - a_j - a_i + \max\{ t, a_i+a_j-b_i, a_i \}$

$g(S, t) = a_i + a_j + g(S-\{i,j\}, t_{ij})$

如果作业i和j在R中互相易位，则完成时间：

$g'(S, t) = a_i + a_j + g(S-\{i,j\}, t_{ji})$

其中 $t_{ji} = b_j + b_i - a_j - a_i + \max\{ t, a_i+a_j-b_j, a_j \}$

比较$g(S, t)$ 和$g'(S, t)$，

如果： $\max\{ t, a_i+a_j-b_i, a_i \} \le \max\{ t, a_i+a_j-b_j, a_j \}$

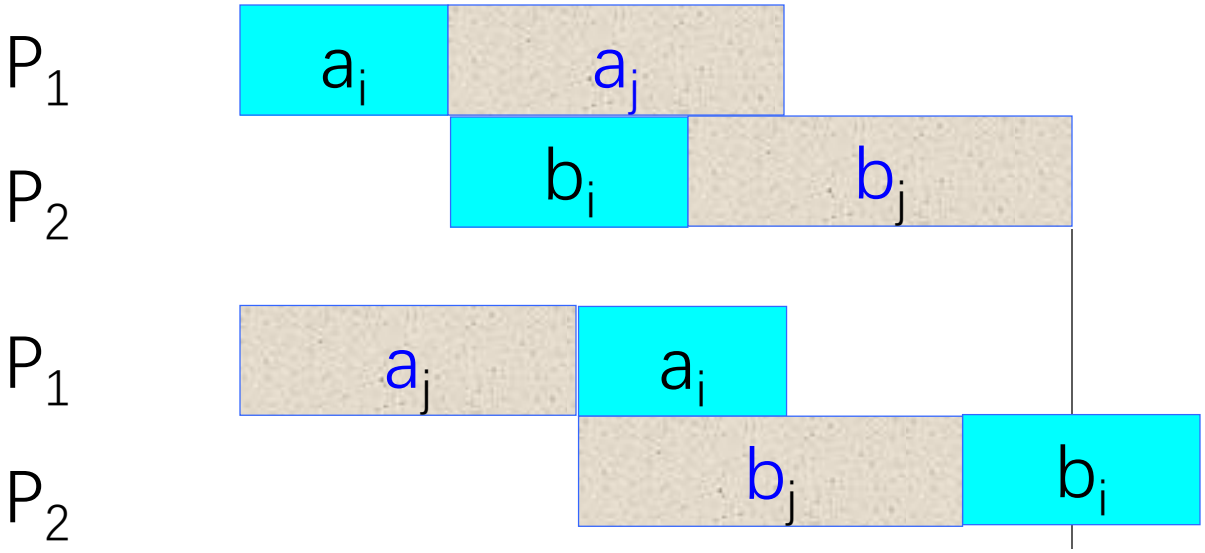
则： $g(S, t) \le g'(S, t)$

为使其对 t 的所有取值都成立，则需要：

$\max\{ a_i+a_j-b_i, a_i \} \le \max\{ a_i+a_j-b_j, a_j \}$

$a_i+a_j+\max\{-b_i, -a_j\} \le a_i+a_j+\max\{-b_j, -a_i\}$

或 $\min\{ b_i, a_j \} \ge \min\{ b_j, a_i \}$



如果作业i和j满足$\min\{b_i, a_j\} \geqslant \min\{b_j, a_i\}$，则称作业i和j满足Johnson不等式。

当作业 i 和作业 j 不满足Johnson不等式时，交换它们的加工顺序后，不增加加工时间。对于流水作业调度问题，必存在最优调度σ，使得作业 σ(i) 和 σ(i+1) 满足Johnson不等式。进一步还可以证明，调度满足 Johnson 法则当且仅当对任意 i<j 有：

$$\min\{b_i, a_j\} \geqslant \min\{b_j, a_i\}$$

由此可知，**所有满足Johnson法则的调度均为最优调度。**

算法描述

流水作业调度问题的Johnson算法

(1) 将全部 a_i 和 b_i 按非减序排序；

(2) 如果 $\min\{a_1,a_2,\cdots,a_n,b_1,b_2,\cdots,b_n\}$ 是 a_i，
则作业 i 是最优调度中的第一个作业。

(3) 如果 $\min\{a_1,a_2,\cdots,a_n,b_1,b_2,\cdots,b_n\}$ 是 b_i，
则作业 i 是最优调度中的最后一个作业。

(4) 再对其余的n-1个作业，决定另一作业的位置。

例子：设n=4，$(a_1,a_2,a_3,a_4) = (3,4,8,10)$

$(b_1,b_2,b_3,b_4) = (6,2,9,15)$

设σ_1，σ_2，σ_3，σ_4是最优调度。

排序 $(b_2, a_1, a_2, b_1, a_3, b_3, a_4, b_4)$

$=(2, 3, 4, 6, 8, 9, 10, 15)$

$(b_2, a_1, a_2, b_1, a_3, b_3, a_4, b_4)$

σ_1，σ_2，σ_3，σ_4

1 3 4 2

```
public static int flowShop(int[] a, int[] b, int[]c)
{                   int n=a.length;
                    Element[] d=new Element[n];
                    for(int i=0; i<n; i++) {
                              int key=a[i]>b[i]? b[i]:a[i];
                              boolean job= a[i]<=b[i];
                              d[i]=new Element(key,i,job);     }
          Sorts.bubbleSort(d,0,(n-1));    //排序
            int j=0,k=n-1;
            for(int i=0;i<n;i++){                  //排出调度序列
                      if(d[i].job)c[j++]=d[i].index;
                      else c[k--]=d[i].index;                    }
            j=a[c[0]];        //P1处理时间
            k=j+b[c[0]];    //P2处理时间
        for(int i=0;i<n;i++){
                    j+= a[c[i]] ;
                    k=j<k? k+b[c[i]] : j+b[c[i]];                }
            return k;        //完成全部任务的结束时间
}
```

算法复杂度分析

空间：
O(n)
时间：
算法的主要计算时间花在对作业集的排序。因此，在最坏情况下算法所需的计算时间为O(nlogn)。

第三章 动态规划—— 自底向上

- 1. 动态规划概述
- 2. 矩阵连乘问题
- 3. 动态规划的基本要素
- 4. 字符串匹配
- 5. 图像压缩
- 6. 流水作业调度
- **7. 0-1背包问题**
- 8. 最优二叉树

0-1背包问题

- 问题的描述
  - 已知： n 种物品具有重量 $(w_1,w_2,\cdots,w_n)$

    效益值 $(p_1,p_2,\cdots,p_n)$

    及一个可容纳 M 重量的背包;

  设当物品 i 全部或一部分放入背包将得到 $p_i x_i$ 的效益，这里，$0 \leqslant x_i \leqslant 1$，$p_i > 0$。

  - 问题：采用怎样的装包方法才能使装入背包的物品的总效益最大?

0-1背包问题

- **分析：**

  ① 装入背包的总重量不能超过 M

  ② 如果所有物品的总重量不超过 M，即：$\sum_{1\le i\le n} w_i x_i \leqslant M$，

  则把所有的物品都装入背包中，获得最大可能的效益值

  ③ 如果物品的总重量超过了M，则将有物品不能（全部）装入背包中。由于$0 \leqslant x_i \leqslant 1$，故可以把物品的一部分装入背包，所以最终背包中可刚好装入重量为 M 的若干物品（整个或一部分）

- **目标：**使装入背包的物品的总效益达到**最大**。

问题的形式描述

- 目标函数: $\sum_{1\le i\le n} p_i x_i$
- 约束条件: $\sum_{1\le i\le n} w_i x_i \le M$

  $0 \le x_i \le 1, p_i > 0, w_i > 0, 1 \le i \le n$
- 可 行 解:
  - 满足上述约束条件的任一集合$(x_1,x_2,\cdots,x_n)$都是问题的一个可行解——可行解可能为多个。
  - $(x_1,x_2,\cdots,x_n)$称为问题的一个解向量
- 最 优 解:
  - 能够使目标函数取最大值的可行解是问题的最优解——最优解也可能为多个。

例 背包问题的实例

- 假设，n=3，M=20，

  $(p_1,p_2,p_3) = (25,24,15)$，$(w_1,w_2,w_3) = (18,15,10)$。

- 可能的可行解如下：

(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$	
① (1/2,1/3,1/4)	16.5	24.25	//没有放满背包//
② (1, 2/15, 0)	20	28.2	
③ (0, 2/3, 1)	20	31	
④ (0, 1, 1/2)	20	31.5	

3.7 0-1背包问题

- **条件**：给定 n 种物品和一个背包。物品i的重量是w_i，其价值为v_i，背包的容量为C。
- **问题**：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。

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

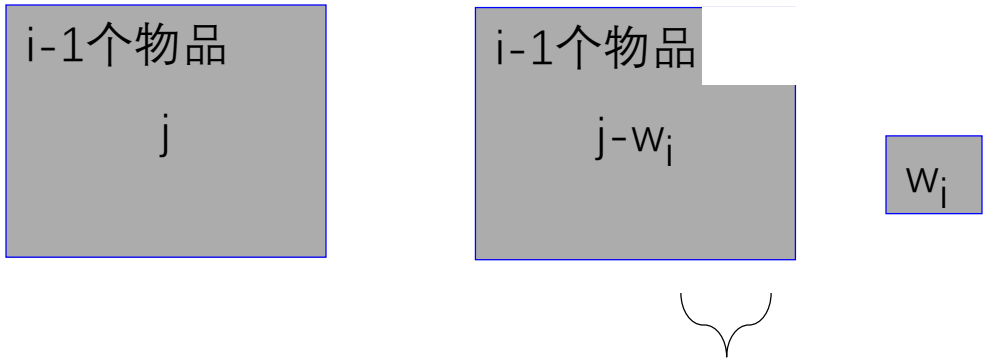
$$\begin{cases} \sum_{i=1}^{n} w_i x_i \le C \\ x_i \in \{0,1\}, 1 \le i \le n \end{cases}$$

实例： C=10， W={ 7, 3, 4, 5 }, V={ 42, 12, 40, 25 }

子集	总重量	总价值
ϕ	0	$0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$36
{1, 3}	11	不可行
{1, 4}	12	不可行
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	9	**$65**
{1, 2, 3}	14	不可行
{1, 2, 3}	15	不可行
{1, 2, 3}	16	不可行
{1, 2, 3}	12	不可行
{1, 2, 3, 4}	19	不可行

递推关系式

考虑前 i 个物品
重量：$w_1,w_2,\cdots\cdots,w_i$
价值：$v_1,v_2,\cdots\cdots,v_i$
背包的承重量：j
设 m[i,j] 为该实例的
最优解的物品总价值。



m[i-1,j]　　m[i-1，j-w_i] + v_i

分析：

1 在不包括第 i 个物品的子集中，最优子集的价值为m[i-1,j]
2 在包括第 i 个物品的子集中，最优子集由该物品和前 i-1个物品中能够放进承重量为j-w_i的背包的最优子集组成，总价值为 v_i+ m[i-1，j-w_i]

建立计算m(i，j)的递归式如下:

递推关系式(向后)

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

初始条件:

当j≥0时，m[0][j]=0;

当i≥0时，m[i][0]=0;

求m[n,W]

递推关系式

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

		0	$j-w_i$	j	W
	0	0	0	0	0
w_i,v_i	i-1	0	$v[i-1,j-w_i]$	v[i-1,j]	
	i	0		v[i,j]	
	n	0			目标

实例:

物品	重量	价值	承重量
1	2	$12	W=5
2	1	$10	
3	3	$20	
4	2	$15	

w_i	v_i	i \ j	0	1	2	3	4	5
		0	0	0	0	0	0	0
2	12	1	0	0	0	0	0	0
1	10	2	0	0	0	0	0	0
3	20	3	0	0	0	0	0	0
2	15	4	0	0	0	0	0	0

w_i	v_i	i \ j	0	1	2	3	4	5
		0	0	0	0	0	0	0
2	12	1	0	0	12	12	12	12
1	10	2	0	0	0	0	0	0
3	20	3	0	0	0	0	0	0
2	15	4	0	0	0	0	0	0

m[1][0] ~ m[1][1] = 0, m[1][2] ~ m[1][5] = 12

w_i	v_i	i \ j	0	1	2	3	4	5
		0	0	0	0	0	0	0
2	12	1	0	0	12	12	12	12
1	10	2	0	10	12	22	22	22
3	20	3	0	0	0	0	0	0
2	15	4	0	0	0	0	0	0

m[2][0] = 0, m[2][1] = 10,

m[2][2] =12, m[2][3] ~m[2][5]=22

w_i	v_i	i \ j	0	1	2	3	4	5
		0	0	0	0	0	0	0
2	12	1	0	0	12	12	12	12
1	10	2	0	10	12	22	22	22
3	20	3	0	10	12	22	30	32
2	15	4	0	0	0	0	0	0

m[3][0] = 0, m[3][1] = 10, m[3][2]=12, m[3][3]=22

m[3][4] = 30, m[3][5] = 32

w_i	v_i	i \ j	0	1	2	3	4	5
		0	0	0	0	0	0	0
2	12	1	0	0	12	12	12	12
1	10	2	0	10	12	22	22	22
3	20	3	0	10	12	22	30	32
2	15	4	0	10	15	25	30	37

m[4][0] = 0, m[4][1] = 10, m[4][2] =15, m[4][3] =25

m[4][4] = 30, m[4][5] =37

最优子集的组成元素：

m[4,5] ≠ m[3,5]，v4在最优解中。j=5-2

m[3,3] = m[2,3]

m[2,3] ≠ m[1,3]，v2在最优解中。j=3-1

m[1,2] ≠ m[0,2]，v1在最优解中。j=2-2

w_i	v_i	i \ j	0	1	2	3	4	5
		0	0	0	0	0	0	0
2	12	1	0	0	12	12	12	12
1	10	2	0	10	12	22	22	22
3	20	3	0	10	12	22	30	32
2	15	4	0	10	15	25	30	37

备忘录方法:

```
MFKnapsack(i,j)
if m[i][j]<0
    if  j<w[i]
        value=MFKnapsack(i-1,j)
    else
        value=max{MFKnapsack(i-1,j),v[i]+MFKnapsack(i-1,j-w[i])
    v[i][j]=value
return v[i][j]
```

w_i	v_i	i \ j	0	1	2	3	4	5
		0	0	0	0	0	0	0
2	12	1	0	-1	-1	-1	-1	-1
1	10	2	0	-1	-1	-1	-1	-1
3	20	3	0	-1	-1	-1	-1	-1
2	15	4	0	-1	-1	-1	-1	-1

备忘录方法:

```
MFKnapsack(i,j)
if m[i][j]<0
    if  j<w[i]
        value=MFKnapsack(i-1,j)
    else
        value=max{MFKnapsack(i-1,j),v[i]+MFKnapsack(i-1,j-w[i])
    v[i][j]=value
return v[i][j]
```

w_i	v_i	i \ j	0	1	2	3	4	5
		0	0	0	0	0	0	0
2	12	1	0	0	12	-1	12	12
1	10	2	0	-1	12	22	-1	22
3	20	3	0	-1	-1	22	-1	32
2	15	4	0	-1	-1	-1	-1	37

递推关系式(向前)

设所给0-1背包问题的子问题

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

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

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。

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

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

例子:

物品	重量	价值	承重量
1	2	$12	W=5
2	1	$10	
3	3	$20	
4	2	$15	

w_i	v_i	i \ j	0	1	2	3	4	5
		0	0	0	0	0	0	0
2	12	1	0	0	0	0	0	0
1	10	2	0	0	0	0	0	0
3	20	3	0	0	0	0	0	0
2	15	4	0	0	0	0	0	0

w_i	v_i	i \ j	0	1	2	3	4	5
		0	0	0	0	0	0	0
2	12	1	0	0	0	0	0	0
1	10	2	0	0	0	0	0	0
3	20	3	0	0	0	0	0	0
2	15	4	0	0	15	15	15	15

m[4][0] ~ m[4][1] = 0, m[4][2] ~ m[4][5] =15

w_i	v_i	i \ j	0	1	2	3	4	5
		0	0	0	0	0	0	0
2	12	1	0	0	0	0	0	0
1	10	2	0	0	0	0	0	0
3	20	3	0	0	15	20	20	35
2	15	4	0	0	15	15	15	15

m[3][0] ~ m[3][1] = 0, m[3][2] =15,

m[3][3] ~ m[3][4] =20

m[3][5] = 35

w_i	v_i	i \ j	0	1	2	3	4	5
		0	0	0	0	0	0	0
2	12	1	0	0	0	0	0	0
1	10	2	0	10	15	25	30	30
3	20	3	0	0	15	20	20	35
2	15	4	0	0	15	15	15	15

m[2][0] =0, m[2][1] = 10, m[2][2] =15, m[2][3] =25

m[2][4] ~ m[2][5] = 30

w_i	v_i	i \ j	0	1	2	3	4	5
		0	0	0	0	0	0	0
2	12	1	0	10	15	25	27	37
1	10	2	0	10	15	25	30	30
3	20	3	0	0	15	20	20	35
2	15	4	0	0	15	15	15	15

m[1][0]= 0, m[1][1]= 10, m[1][2]= 15, m[1][3]= 25

m[1][4]= 27, m[1][5] = 37

最优子集的组成元素：

m[1,5] ≠ m[2,5]，v1在最优解中。j=5-2

m[2,3] ≠ m[3,3]，v2在最优解中。j=3-1

m[3,2] = m[4,2]

m[4,2] >0，　　v4在最优解中。j=2-2

w_i	v_i	j / i	0	1	2	3	4	5
		0	0	0	0	0	0	0
2	12	1	0	10	15	25	27	37
1	10	2	0	10	15	25	30	30
3	20	3	0	0	15	20	20	35
2	15	4	0	0	15	15	15	15

0-1背包问题的算法复杂度分析：

空间：

w[n], v[n], m[n][c], O(nc)

时间：

由m(i，j)的递归式，算法需要O(nc)计算时间。

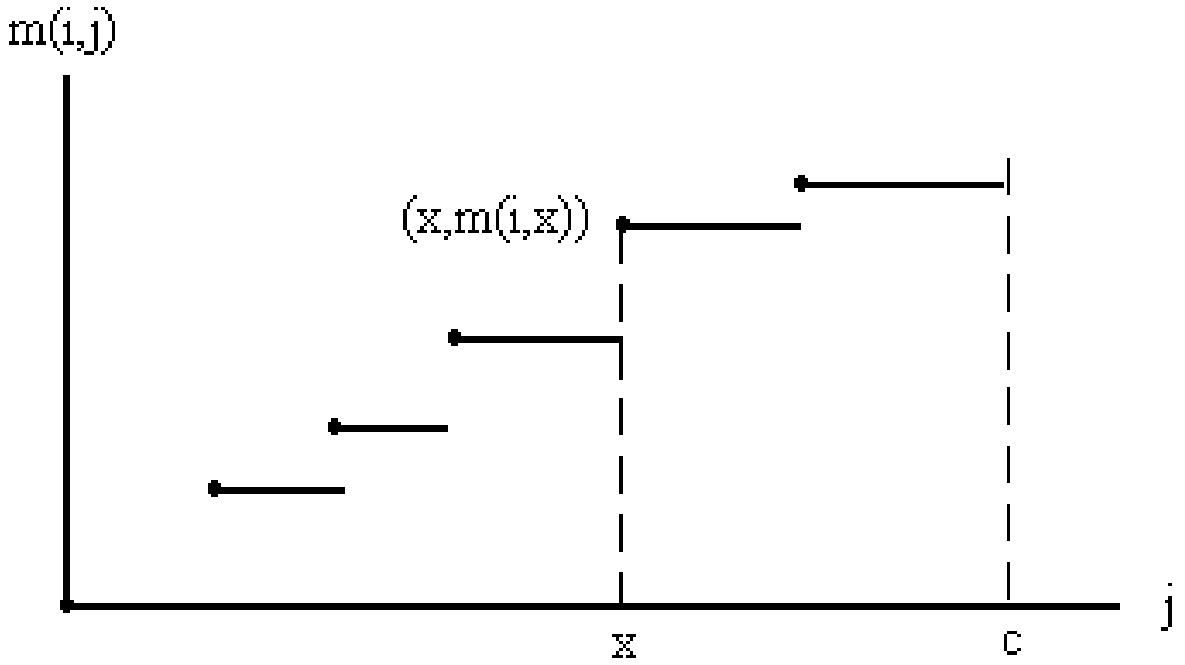
构造最优解时间： O(n)。

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。

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。



实例：n=5，c=10，w={2，2，6，5，4}，v={6，3，5，4，6}

i	j										
	0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	6	6	6	6	6	6	6
4	0	0	0	0	6	6	6	6	6	10	10
3	0	0	0	0	6	6	6	6	6	10	11
2	0	0	3	3	6	6	9	9	9	10	11
1	0	0	6	6	9	9	12	12	15	15	15

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

实例： 0/1背包问题

n=3, $(w_1,w_2,w_3)=(2,3,4)$, $(v_1,v_2,v_3)=(1,2,5)$, M=6

递推计算过程

$$f_0(X)=\begin{cases}-\infty & X<0\\ 0 & X\geqslant 0\end{cases}$$

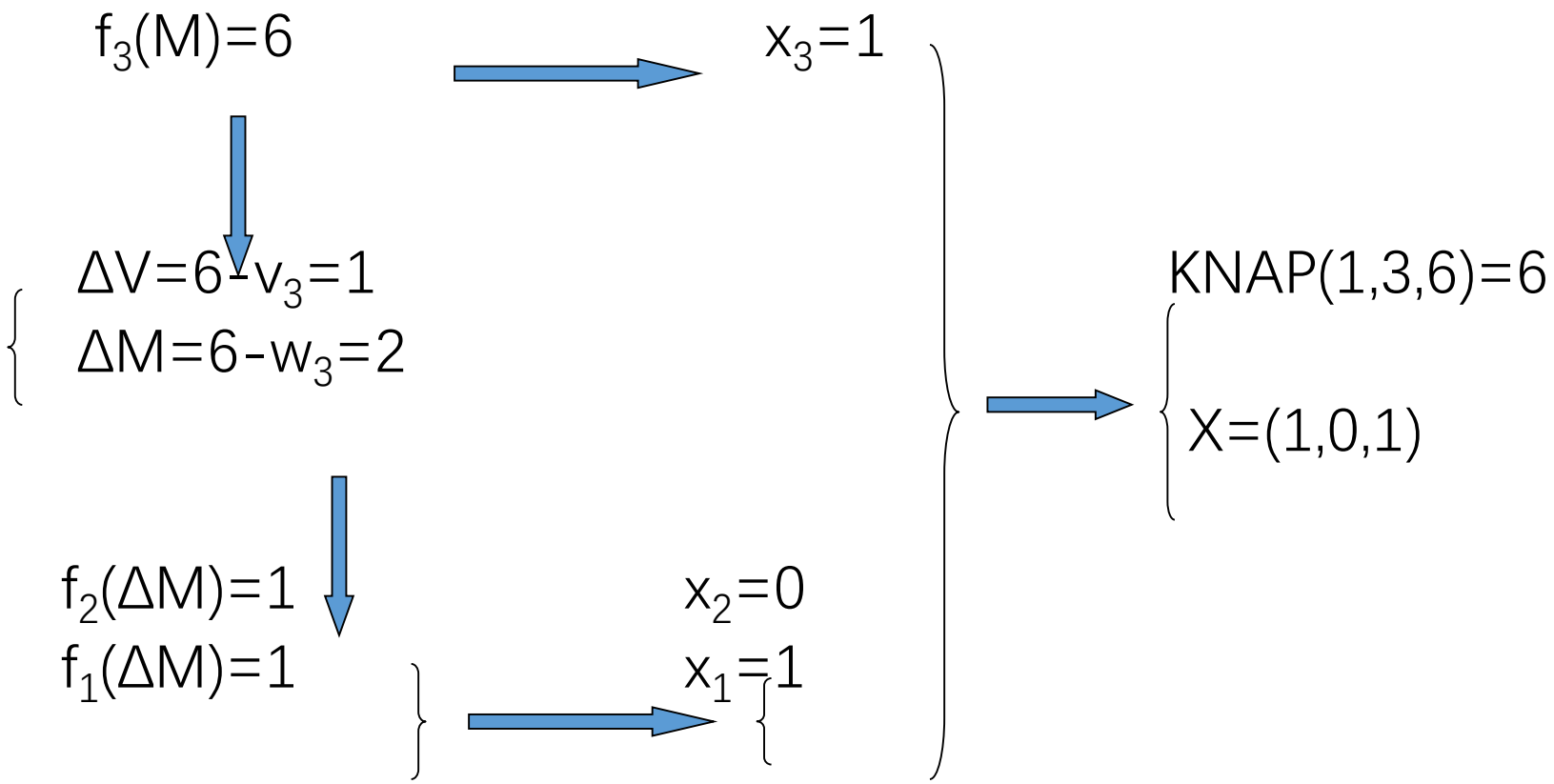
$$f_1(X)=\begin{cases}-\infty & X<0\\ \max\{0,-\infty+1\}=0 & 0\leqslant X<2\\ \max\{0,0+1\}=1 & X\geqslant 2\end{cases}$$

$$f_2(X)=\begin{cases}-\infty & X<0\\ 0 & 0\leqslant X<2\\ 1 & 2\leqslant X<3\\ \max\{1,0+2\}=2 & 3\leqslant X<5\\ \max\{1,1+2\}=3 & X\geqslant 5\end{cases}$$

$f_3(M) = \max\{3, 1+5\} = 6$

- $f_3(6) = \max\{f_2(6), f_2(6-4)+5\} = 6$
- $f_n(M) = \max\{f_{n-1}(M), f_{n-1}(M-w_n)+p_n\}$

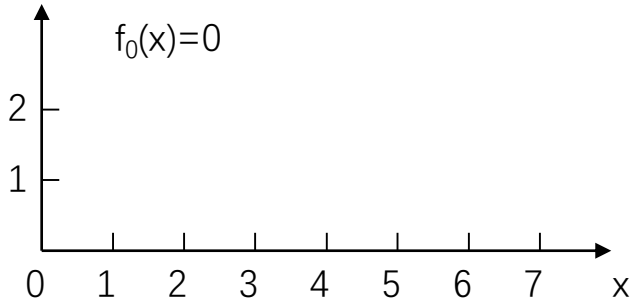
解向量的推导

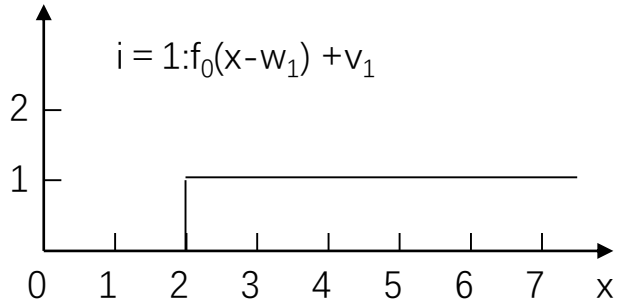

$f_3(M)=6$
$x_3=1$
$\Delta V=6-v_3=1$
$\Delta M=6-w_3=2$
KNAP(1,3,6)=6
X=(1,0,1)
$f_2(\Delta M)=1$
$f_1(\Delta M)=1$
$x_2=0$
$x_1=1$

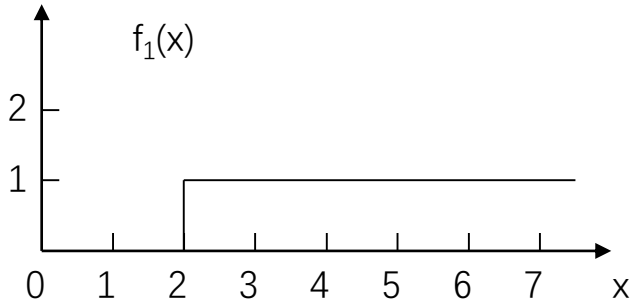
f_1, f_2, f_3计算过程（图解）

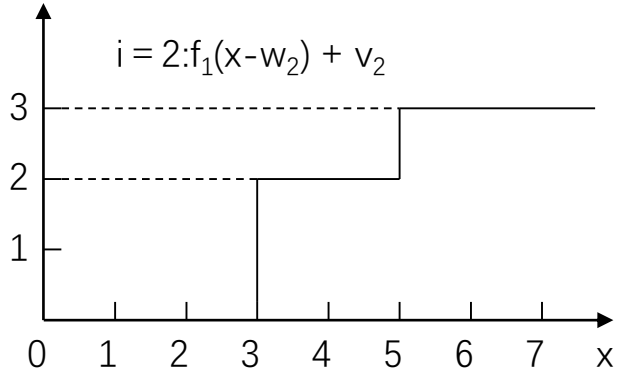
i:$f_{i-1}(x-w_i) + v_i$

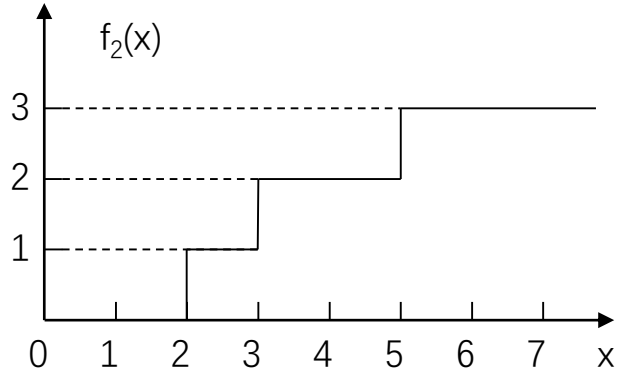
i=0:函数不存在

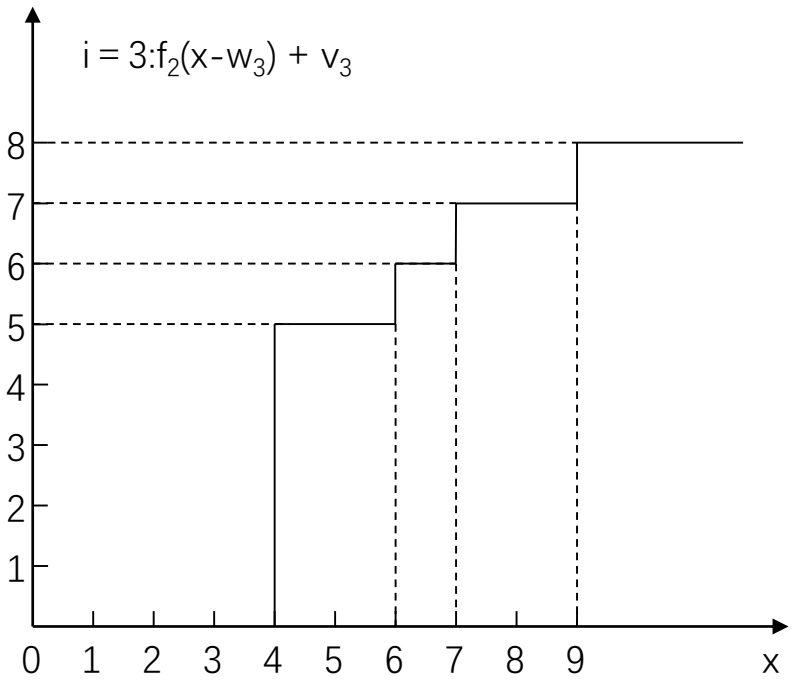


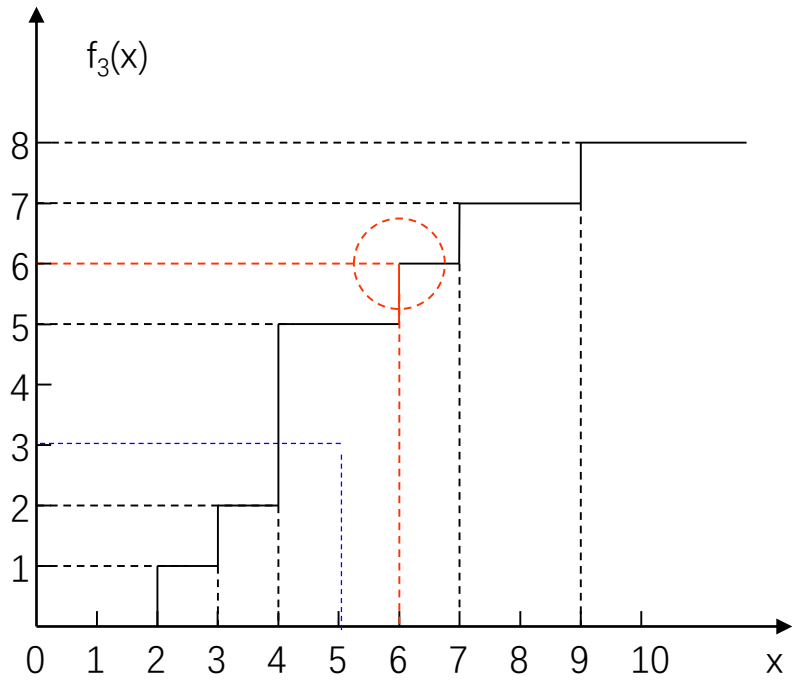












注：

- $f_{i-1}(X-w_i)+v_i$曲线的构造：将$f_{i-1}(X)$的曲线在X轴上右移w_i个单位，然后上移v_i个单位而得到；

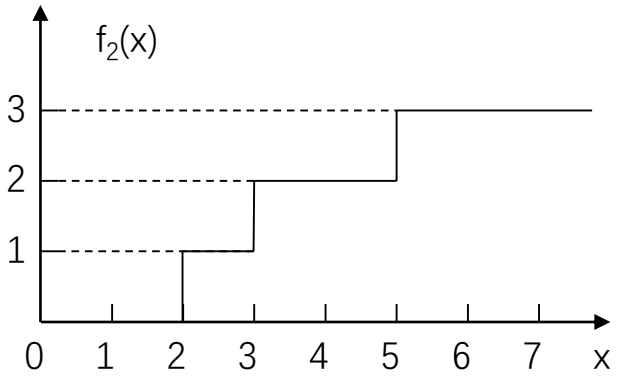
- $f_i(X)$曲线的构造：由$f_{i-1}(X)$ 和$f_{i-1}(X-w_i)+v_i$的曲线按X相同时取大值的规则归并而成

序偶表示

记 f_i的序偶集合为

S^i = { (P_j,W_j) | W_j 是 f_i 曲线中使得 f_i 产生一次阶跃的X值，$0 \leqslant j < r$, $P_j=f_i(W_j)$ }

- $(P_0,W_0)=(0,0)$
- 共有 r 个阶跃值，分别对应 r 个(P_j,W_j)序偶，$1 \leqslant j \leqslant r$
- 若 $W_j < W_{j+1}$, 则 $P_j < P_{j+1}$，$0 \leqslant j < r$
- 若 $W_j \leqslant X < W_{j+1}$，$f_i(X)=f_i(W_j)$
- 若 $X \geqslant W_r$，$f_i(X)=f_i(W_r)$



- S^i的构造

记 是S_1^i $f_{i-1}(X-w_j)+p_j$ 的所有序偶的集合，则

$$S_1^i = \{(P,W) \mid (P-p_i, W-w_i) \in S^{i-1}\}$$

其中，S^{i-1} 是 f_{i-1} 的所有序偶的集合

S^i的构造：由S^{i-1}和 按照S_1^i支配规则合并而成。

支配规则：如果S^{i-1}和 之S_1^i有序偶(P_j,W_j)，另一有(P_k,W_k)，且有 $W_j \geqslant W_k$，$P_j \leqslant P_k$，则序偶(P_j,W_j)将被舍弃。（即取最大值规则）。

注：

★ S^i 中的所有序偶是背包问题KNAP(1,i,X)在X各种取值下的最优解

例2　序偶计算

$(p_1,p_2,p_3)=(1,2,5)$,　$(w_1,w_2,w_3)=(2,3,4)$,　$M=6$

$S^0=\{(0,0)\}$　　$=\{(1,2)\}$　　S_1^1

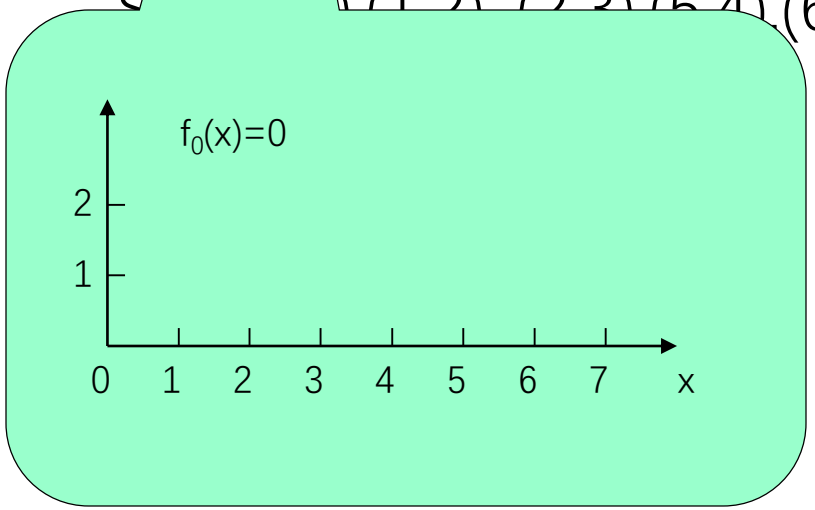
$S^1=\{(0,0),(1,2)\}$　　$=\{(2,3),(3,5)\}$　S_1^2

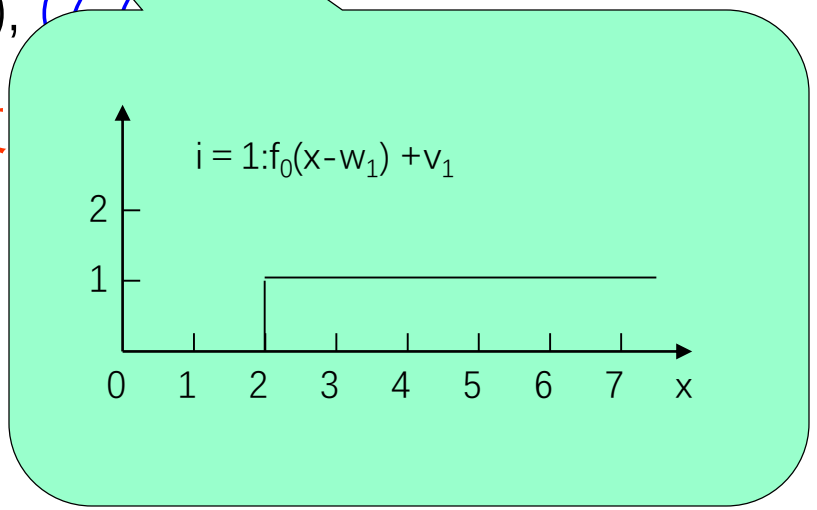
$S^2=\{(0,0),(1,2),(2,3),(3,5)\}$

S_1^3　　$=\{(5,4),(6,6),(7,7),(8,9)\}$

$S^3=\{(0,0),(1,2),(2,3),(5,4),(6,6),(7,7)$

“支





例2　序偶计算

$(p_1,p_2,p_3)=(1,2,5)$,　$(w_1,w_2,w_3)=(2,$

$S^0=\{(0,0)\}$　　$=\{(1,2)$

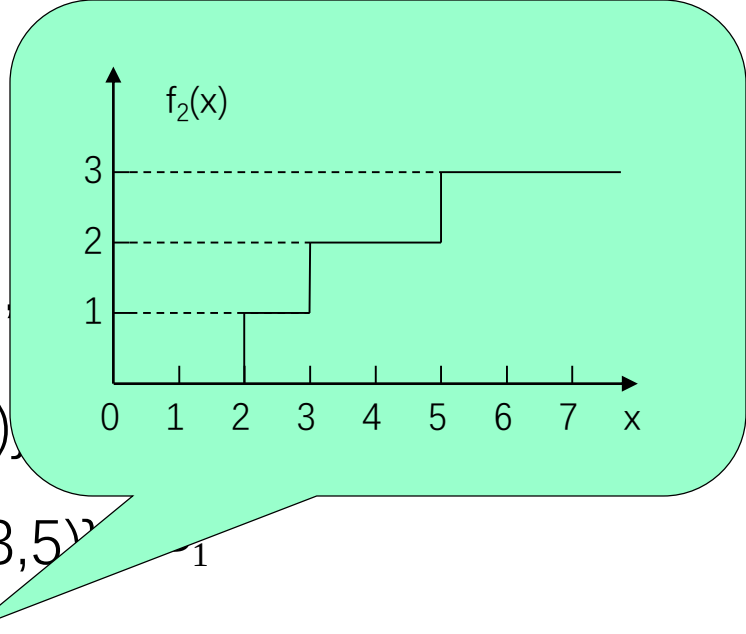
$S^1=\{(0,0),(1,2)\}$　　$=\{(2,3),(3,5)$

$S^2=\{(0,0),(1,2),(2,3),(3,5)\}$

S_1^3　　$=\{(5,4),(6,6),(7,7),(8,9)\}$

$S^3=\{(0,0),(1,2),(2,3),(5,4),(6,6),(7,7),(8,9)\}$

注：序偶 (3,5) 被 (5,4) “支配” 而删除



KNAP(1,n,M)问题的解

★ 计算S^n可以找到KNAP(1,n,X)在$0 \leqslant X \leqslant M$的所有解

★ KNAP(1,n,M)的最优解由S^n的最后一对序偶的P值给出。

x_i的推导

x_n: 设S^n的最后一对有效序偶是(P_1,W_1)，则(P_1,W_1)或者是S^{n-1}的最末一对序偶，或者是(P_j+p_n,W_j+w_n)，其中$(P_j,W_j) \in S^{n-1}$且W_j是S^{n-1}中满足$W_j+w_n \leqslant M$的最大值。

- 若$(P_1,W_1) \in S^{n-1}$，则$X_n = 0$；否则，
- $(P_1 - p_n,W_1-w_n) \in S^{n-1}$，$X_n = 1$

x_{n-1}: 若$x_n=0$，则判断$(P_1,W_1) \in S^{n-2}$？，以确定X_{n-1}的值

若$x_n=1$，则依据$(P_1 - p_n,W_1-w_n) \in S^{n-2}$？，以判断$X_{n-1}$的值

$x_{n-2},\cdots,x_1$将依次推导得出

实例

$S^0=\{(0,0)\}$

$S^1=\{(0,0),(1,2)\}$

$S^2=\{(0,0),(1,2),(2,3),(3,5)\}$

$S^3=\{(0,0),(1,2),(2,3),(5,4),(6,6),(7,7),(8,9)\}$

M=6，$f_3(6)$由S^3中的序偶(6,6)给出。

1) $\because (6,6) \notin S^2$

$\therefore x_3=1$

2) $\because (6-p_3, 6-w_3)=(1,2) \in S^2$且$(1,2) \in S^1$

$\therefore x_2=0$

3) $\because (1,2) \notin S^0$

$\therefore x_1=1$

算法1　非形式化的背包算法

procedure DKP(p,w,n,M)

$S^0 \leftarrow \{(0,0)\}$

for i←1 to n-1 do

$S_1^i \leftarrow \{(P_1,W_1) | (P_1-p_i, W_1-w_i) \in S^{i-1}$ and $W_1 \leqslant M\}$

$S^i \leftarrow$MERGE-PURGE(S^{i-1}, S_1^i)

endfor

(PX,WX)← S^{n-1}的最末一个有效序偶

(PY,WY)←(P1 + p_n,W1+w_n)，其中，W1是S^{n-1}中使得W + $w_n \leqslant$M的所有序偶中取最大值的W

//沿S^{n-1},…, S^1回溯确定x_n,x_{n-1},…,x_1的取值

if PX > PY then x_n←0　//PX是S^n的最末序偶

else　x_n←1　//PY是S^n的最末序偶

endif

回溯确定x_{n-1},…,x_1

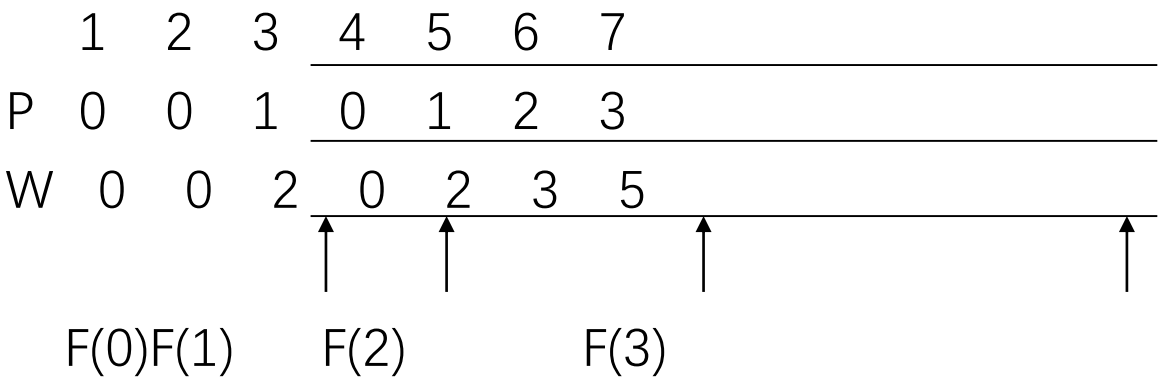
end DKP

3. DKP的实现

1)序偶集 S^i 的存储结构

使用两个一维数组P和W存放所有的序偶(P1,W1),其中P存放P1值，W存放W1值

- 序偶集$S^0, S^1,\cdots, S^{n-1}$顺序、连续存放于P和W中；
- 用指针F(i)表示S^i中第一个元素在数组(P,W)中的下标位置，$0 \leqslant i \leqslant n$;
- $F(n) = S^{n-1}$中最末元素位置 + 1



2) 序偶的生成与合并

★ S^i的序偶将按照P和W的递增次序生成

★ S_1^i中序偶的生成将与 S_1^i和S^{i-1}的合并同时进行

设 S_1^i生成的下一序偶是(PQ,WQ)；对所有的(PQ,WQ)，根据支配规则处理如下：

(1) 将S^{i-1}中所有W值小于WQ的序偶直接计入S^i中；

(2) 根据支配规则，若S^{i-1}中有W值小于WQ的序偶支配(PQ,WQ)，则(PQ,WQ)将被舍弃，否则(PQ,WQ)计入S^i中；并清除S^{i-1}中所有可被支配的序偶，这些序偶有W≥WQ且P≤PQ

(3) 对所有的(PQ,WQ)重复上述处理；

(4) 将最后 S_1^i中剩余的序偶直接计入S^i中；

(5) 所有计入S^i中的新序偶依次存放到由F(i)指示的S^i的存放位置上。

注：不需要存放 的S_1^i专用空间

实例：　0/1背包问题

n=6, M=165

$(p_1,p_2,p_3,p_4,p_5,p_6)=(w_1,w_2,w_3,w_4,w_5,w_6)=(100,50,20,10,7,3)$

$S^0=\{0\}$,　$S_1^0= S^0 \oplus (p1,w1)=\{100\}$

$S^1= S^0 \cup S_1^0 =\{0,100\}$, $S_1^1= S^1 \oplus (p2,w2)=\{50,150\}$

$S^2=\{0,50,100,150\}$, $S_1^2= S^2 \oplus (p3,w3)=\{20,70,120\}$

$S^3=\{0,20,50,70,100,120,150\}$,

$S_1^3= S^3 \oplus (p4,w4)=\{10,30,60,80,110,130,160\}$

$S^4=\{0,10,20,30,50,60,70,80,100,110,120,130,150,160\}$

$S_1^4= S^4 \oplus (p5,w5)$

$=\{7,17,27,37,57,67,77,87,107,117,127,137,157\}$

$S^5=\{0,7,10,17,20,27,30,37,50,57,60,67,70,77,80,87,100,$
$107,110,117,120,127,130,137,150,157,160\}$

则，$f_6(165)=163$

注：每对序偶(P,W)仅用单一量P（或W）表示

过程DKNAP的复杂性分析

1) 空间复杂度

记S^i中的序偶数为：$|S^i|$

则有，$|S^i| \leqslant |S^{i-1}| + |S_1^i|$

又，$|S_1^i| \leqslant |S^{i-1}|$

所以，$|S^i| \leqslant 2|S^{i-1}|$

最坏情况下有（由S^{i-1}生成S_1^i时没有序偶被支配）：

$$\sum_{0 \leq i \leq n-1} |S^i| = \sum_{0 \leq i \leq n-1} 2^i = 2^n - 1$$

故，DKNAP所需的空间复杂度（P、W数组）为：$O(2^n)$

2) 时间复杂度

由S^{i-1}生成S^i的时间为：$\Theta(|S^{i-1}|)$，$0 \leqslant i \leqslant n-1$

合并S^i和S_1^i的时间也为：

故，DKNAP计算所有的S^i所需的时间为：$\sum_{0 \le i \le n-1} |S^{i-1}| = O(2^n)$

注：若每件物品的重量w_i和效益值p_i均为整数，则S^i中每个序偶(P,W)的P值和W值也是整数，且有 $P \le \sum_{0 \le j \le i} p_j$ $W \leqslant M$

又，在任一S^i中的所有序偶具有互异P值和W值，故有：

$$|S^i| \le 1 + \sum_{0 \le j \le i} |p_j| \quad \text{且} \quad |S^i| \le 1 + \min\{\sum_{0 \le j \le i} |w_j|, M\}$$

在所有w_j和p_j均为整数的情况下，DKNAP的时间和空间复杂度将为：

$$O(\min\{2^n, n\sum_{1\le i\le n}|p_i|, nM\})$$

第三章 动态规划—— 自底向上

- 1. 动态规划概述
- 2. 矩阵连乘问题
- 3. 动态规划的基本要素
- 4. 字符串匹配
- 5. 图像压缩
- 6. 流水作业调度
- 7. 0-1背包问题
- **8. 最优二叉树**

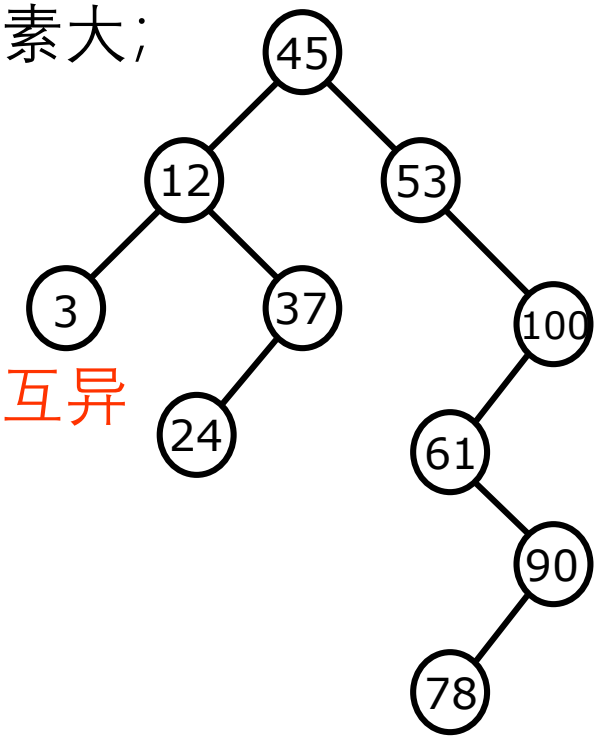
3.8 最优二叉搜索树

1. 问题的描述

1）二分检索树定义

二分检索树T是一棵二元树，它或者为空，或者其每个结点含有一个可以比较大小的数据元素，且有：

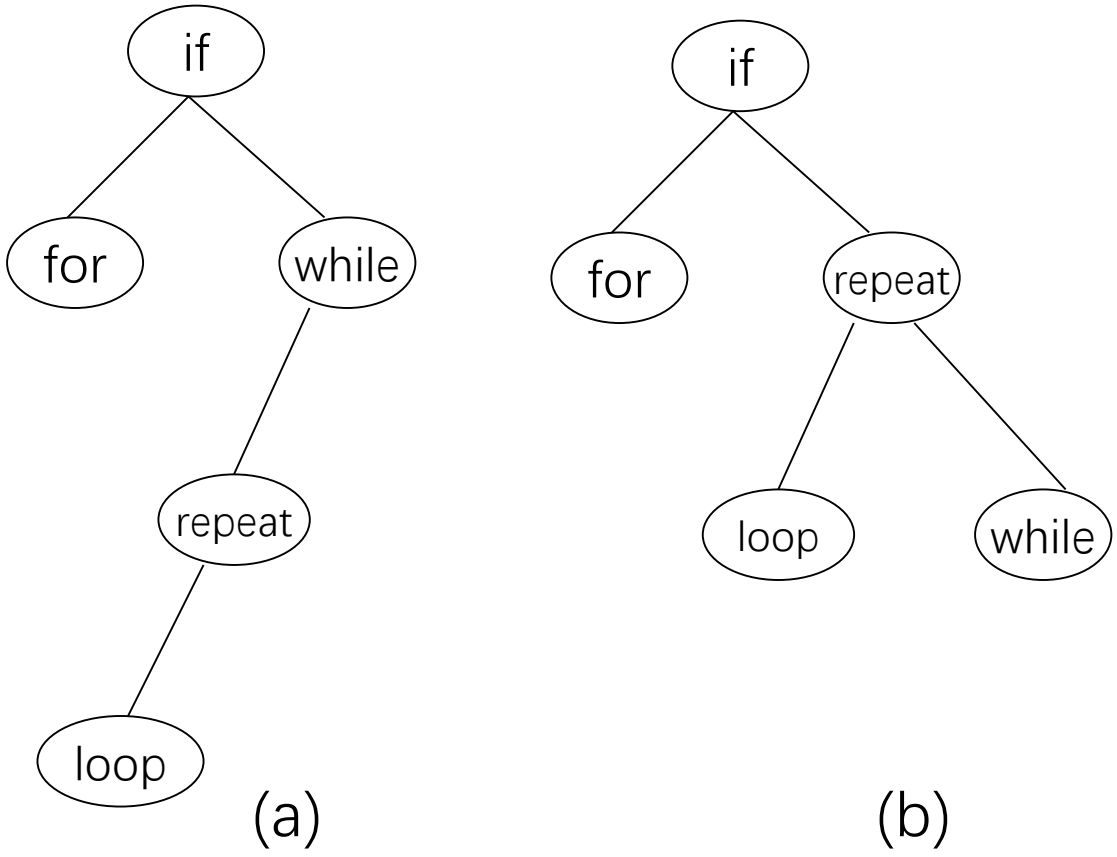
- T的左子树的所有元素比根结点中的元素小；
- T的右子树的所有元素比根结点中的元素大；
- T的左子树和右子树也是二分检索树。



注：

- 二分检索树要求树中所有结点的元素值互异

对于一个给定的标识符集合，可能有若干棵不同的二分检索树：



不同形态的二分检索树对标识符的检索性能是不同的。

2）二分检索树的检索

```
算法 SEARCH(T,X,i)
 /* 为X检索二分检索树T，如果X不在T中，则置i=0;
  否则i有IDENT(i)=X */
  i←T
  while i≠0 do
  case
   :X < IDENT(i):i←LCHILD(i)  //若X小于IDENT(i)，则在左子树中继续查找
   :X = IDENT(i):return       //X等于IDENT(i)，则返回
   :X > IDENT(i):i←RCHILD(i)  //若X大于IDENT(i)，则在右子树中继续查找
  endcase
 endwhile
end SEARCH
```

二分检索树的性能特征

① 例

图(a):

最坏情况下查找标识符loop需要做4次比较;

成功检索平均需要做12/5次比较;

图(b):

最坏情况下查找标识符loop/while需要做3次比较;

成功检索平均需要做11/5次比较

② 查找概率

可以期望不同标识符的检索频率是不同的。如

$$P_{for} > P_{while} > P_{loop}$$

③ 不成功检索

可以期望不成功检索出现的频率也是不同的。

3）最优二分检索树问题

设给定的标识符集合是 $\{a_1,a_2,\cdots,a_n\}$，并假定$a_1 < a_2 < \cdots < a_n$。

设，P(i)是对a_i检索的概率，

Q(i)是正被检索的标识符X恰好满足：$a_i < X < a_{i+1}$，$0 \leqslant i \leqslant n$（设$a_0=-\infty$，$a_{n+1}=+\infty$）时的概率，即一种不成功检索情况下的概率。

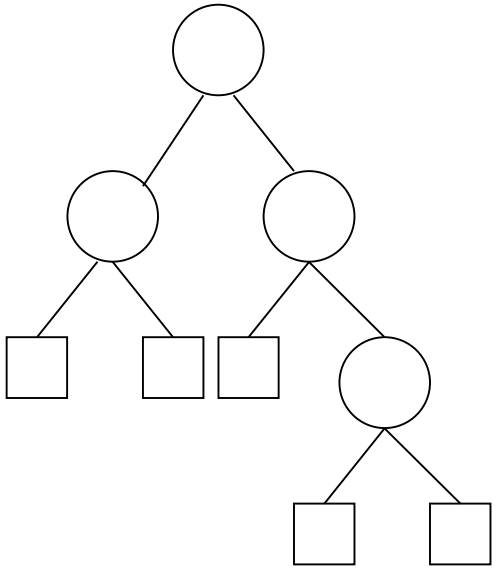
则 $\sum_{1\le i\le n} P(i)$ 表示所有成功检索的概率。

$\sum_{0\le i\le n} Q(i)$ 表示所有不成功检索的概率。

考虑所有可能的成功和不成功检索情况，共2n+1种可能的情况，有

$$\sum_{1\le i\le n} P(i) + \sum_{0\le i\le n} Q(i) = 1$$

二分检索树



内结点：代表成功检索情况，刚好有n个

外结点：代表不成功检索情况，刚好有n + 1个

关于$\{a_1,a_2,\cdots,a_n\}$的最优二分检索树含义如下：

二分检索树的预期成本

预期成本：算法对于所有可能的成功检索情况和不成功检索情况，平均所要做的比较次数。根据期望值计算方法，

平均检索成本 = $\sum$ 每种情况出现的概率×该情况下所需的比较次数

平均检索成本的构成：成功检索成分 + 不成功检索成分

- 成功检索：在 *l* 级内结点终止的成功检索，需要做 *l* 次比较运算（基于二分检索树结构）。该级上的任意结点a_i的成功检索的成本分担额为：

$$P(i) * level(a_i)\ ;$$

其中，$level(a_i)$ = 结点a_i 的级数 = *l*

● 不成功检索：

不成功检索的等价类：每种不成功检索情况定义了一个等价类，共有n+1个等价类 Ei (0≤i≤n)。其中，

$E0 = \{ X | X < a_0 \}$

$Ei = \{ X | a_i < X < a_{i+1} (1 \leq i < n) \}$

$En = \{ X | X > a_n \}$

若Ei处于/级，则算法需作 /-1次迭代。则， /级上的外部结点的不成功检索的成本分担额为：

$$Q(i)*(level(Ei)-1)$$

则预期总的成本公式表示如下：

$$\sum_{1\leq i\leq n} P(i) * level(a_i) + \sum_{0\leq i\leq n} Q(i) * (level(Ei) - 1)$$

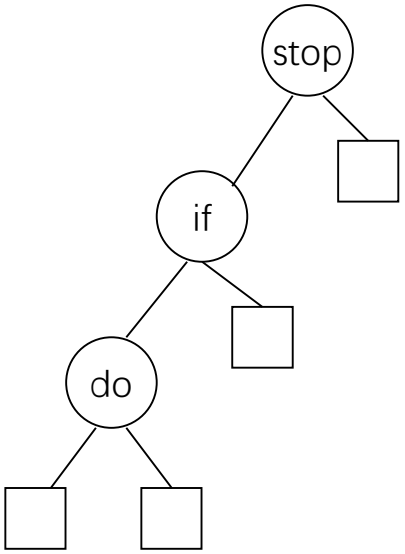
最优二分检索树问题：

求一棵使得预期成本最小的二分检索树

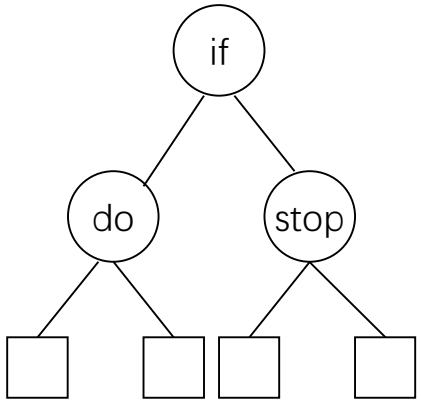
- 有 n 个节点的二叉树的个数为：$\Omega(4^n / n^{3/2})$
- 穷举搜索法的时间复杂度为指数级 ☹

例　标识符集合（a1,a2,a3）=（do,if,stop）。可能的二分检索树如下所示。

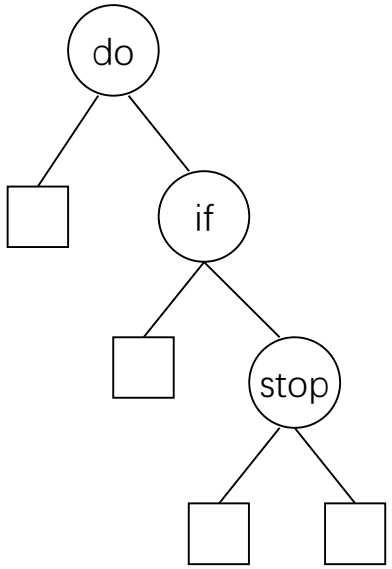
- 成 功 检 索：3种
- 不成功情况：4种



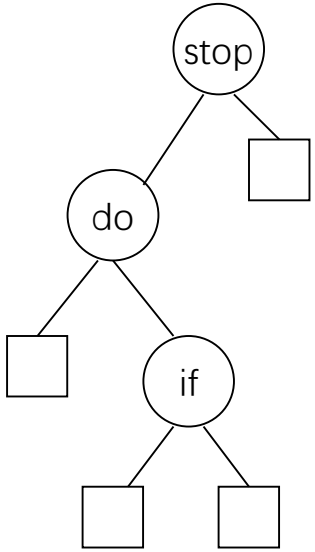
(a)



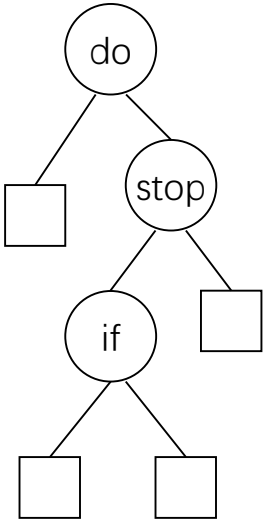
(b)



(c)



(d)



(e)

1) 等概率：P(i)=Q(i)=1/7

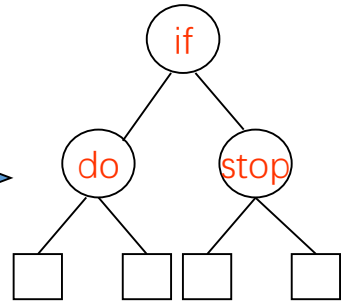
cost(a) = 15/7

cost(b) = 13/7 → 最优

cost(c) = 15/7

cost(d) = 15/7

cost(e) = 15/7



(b)

2）不等概率：

P(1)=0.5　P(2)=0.1　P(3)=0.05

Q(0)=0.15　Q(1)=0.1　Q(2)=0.05　Q(3)=0.05

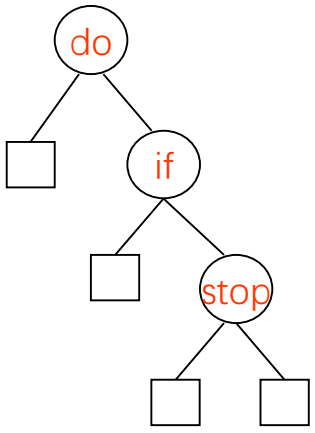
cost(a) = 2.65

cost(b) = 1.9

cost(c) = 1.5 → 最优

cost(d) = 2.15

cost(e) = 1.6

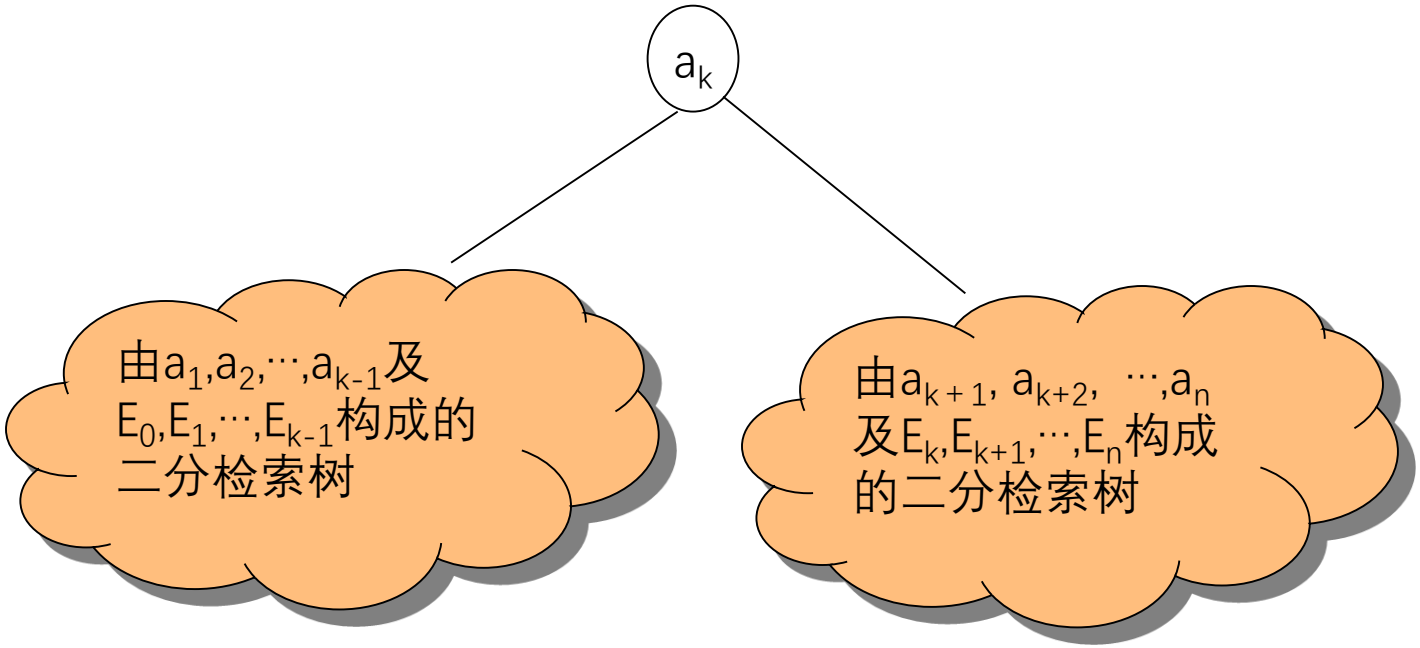


(c)

2. 多阶段决策过程

把构造二分检索树的过程看成一系列决策的结果。

决策的策略：决策树根，如果$\{a_1,a_2,\cdots,a_n\}$存在一棵二分检索树，a_k是该树的根，则内结点$a_1,a_2,\cdots,a_{k-1}$和外部结点$E_0,E_1,\cdots,E_{k-1}$将位于根a_k的左子树中，而其余的结点将位于右子树中。



定义:

$$COST(L)=\sum_{1\le i<k}P(i)*level(a_i)+\sum_{0\le i<k}Q(i)*(level(E_i)-1)$$

$$COST(R)=\sum_{k<i\le n}P(i)*level(a_i)+\sum_{k\le i\le n}Q(i)*(level(E_i)-1)$$

含义:

- 左、右子树的预期成本——左、右子树的根处于第一级
- 左、右子树中所有结点的级数相对于子树的根测定，而相对于原树的根少一。

记: $$w(i,\ j) = \mathrm{Q}(i) + \sum_{\mathrm{i}+1 \le l \le \mathrm{j}} (\mathrm{P}(l) + \mathrm{Q}(l))$$

则，原二分检索树的预期成本可表示为:

COST(T)=P(k)+COST(L)+COST(R)+W(0,k-1)+W(k,n)

最优二分检索树：COST(T)有最小值

最优性原理成立

若T最优二分检索树，则COST(L)和COST(R)将分别是包含$a_1,a_2,\cdots,a_{k-1}$和$E_0,E_1,\cdots,E_{k-1}$、及$a_{k+1}, a_{k+2}, \cdots,a_n$和$E_k,E_{k+1},\cdots,E_n$的最优二分检索(子)树。

记由 $a_{i+1},a_{i+2},\cdots,a_j$ 和 $E_i,E_{i+!},\cdots,E_j$ 构成的二分检索树的成本为C(i,j)，则对于最优二分检索树T有，

COST(L) = C(0,k-1)

COST(R) = C(k,n)

则,

$$C(0,n)=\min_{1\le k\le n}\{C(0,k-1)+C(k,n)+P(k)+W(0,k-1)+W(k,n)\}$$

对任何C(i,j)有,

$$C(i,j)=\min_{i<k\le j}\{C(i,k-1)+C(k,j)+P(k)+W(i,k-1)+W(k,j)\}$$

$$=\min_{i<k\le j}\{C(i,k-1)+C(k,j)\}+W(i,j)$$

向前递推过程:

★ 首先计算所有j-i=1的C(i,j)

★ 然后依次计算j-i=2,3,…,n的C(i,j)。

★ C(0,n)=最优二分检索树的成本。

初始值

$$\begin{cases} C(i,i)=0 \\ W(i,i)=Q(i), \quad 0\leqslant i\leqslant n \end{cases}$$

最优二分检索树的构造

- 在计算C(i,j)的过程中，记下使之取得最小值的k值，即树Tij的根，记为R(i,j)。
- 依据R(0,n)…，推导树的形态

实例:

设n=4，(a1,a2,a3,a4)=(do,if,read,while)。

设P(1:4) = (3,3,1,1)，Q(0:4) = (2,3,1,1,1)

(注：概率值“扩大”了16倍)

初始：W(i,i)=Q(i)

C(i,i)=0

R(i,i)=0

且有，W(i,j)=P(j)+Q(j)+W(i,j-1)

j-i=0：

0
- W(0,0) =Q(0) = 2
- C(0,0) = 0
- R(0,0) = 0

①
- W(1,1) =Q(1) = 3
- C(1,1) = 0
- R(1,1) = 0
- W(2,2) = Q(2) = 1

②
- C(2,2) = 0
- R(2,2) = 0
- W(3,3) = Q(3) = 1

③
- C(3,3) = 0
- R(3,3) = 0
- W(4,4) = Q(4) = 1

④
- C(4,4) = 0
- R(4,4) = 0

j-i=1：

- i=0,j=1 (do)
  - W(0,1)=P(1)+Q(1)+W(0,0) = 3+3+2 = 8
  - C(0,1)=W(0,1)+min{C(0,0)+C(1,1)} = 8
  - R(0,1) = 1
- i=1,j=2 (if)
  - W(1,2)=P(2)+Q(2)+W(1,1) = 7
  - C(1,2)=W(1,2)+min{C(1,1)+C(2,2)} = 7
  - R(1,2) = 2

j-i=1 :

- i=2,j=3 (read)
  - W(2,3)=P(3)+Q(3)+W(2,2) = 3
  - C(2,3)=W(2,3)+min{C(2,2)+C(3,3)} = 3
  - R(2,3) = 3
- i=3,j=4 (while)
  - W(3,4)=P(4)+Q(4)+W(3,3) = 3
  - C(3,4)=W(3,4)+min{C(3,3)+C(4,4)} = 3
  - R(3,4) = 4

j-i=2

- i=0,j=2 (do if)
  - W(0,2)=Q(0)+Q(1)+Q(2)+P(1)+P(2)=12
  - C(0,2)= W(0,2)+ min{c(0,0)+c(1,2),c(0,1)+C(2,2)}

    =12+ min{ 0+7, 8+0} =19
  - R(0,2)=1
- i=1,j=3 (if read)
  - W(1,3)= Q(1)+Q(2)+Q(3)+P(2)+P(3)= 9
  - C(1,3)=min{c(1,1)+c(2,3),c(1,2)+C(3,3)} +9=12
  - R(1,3)=2
- i=2,j=4 (read while)
  - W(2,4)= Q(2)+Q(3)+Q(4)+P(3)+P(4)= 5
  - C(2,4)=min{c(2,2)+c(3,4), C(3,3)+C(4,4)} +5=8
  - R(2,4)=3

j-i=3

- i=0,j=3 (do if read)
  - W(0,3)=Q(0)+Q(1)+Q(2)+Q(3)+P(1)+P(2)+P(3)=14
  - C(0,3)=min{c(0,0)+c(1,3),c(0,1)+C(2,3),C(0,2)+C(3,3)} +14
    =min{0+12,8+3,19+0}+14=25
  - R(0,3)=2
- i=1,j=4 (if read while)
  - W(1,4)= Q(1)+Q(2)+Q(3)+Q(4)+P(2)+P(3)+P(4)= 11
  - C(1,4)=min{c(1,1)+c(2,4),c(1,2)+C(3,4),C(1,3)+C(4,4)}+11
    =min{0+8,7+3,12+0}+11=19
  - R(1,4)=2

j-i=4

- i=0,j=4 (do if read while)
  - W(0,4)=…=16
  - C(0,4)=min{c(0,0)+c(1,4), c(0,1)+C(2,4), C(0,2)+C(3,4), C(0,3)+C(4,4)} +16=32
  - R(0,4)=2

计算结果

j-i \ i	0	1	2	3	4
0	2,0,0	3,0,0	1,0,0	1,0,0	1,0,0
1	8,8,1	7,7,2	3,3,3	3,3,4	
2	12,19,1	9,12,2	5,8,3		
3	14,25,2	11,19,2			
4	16,32,2			W(j,j+i),C(j,j+i),R(j,j+i)	

则，C(0,4)=32

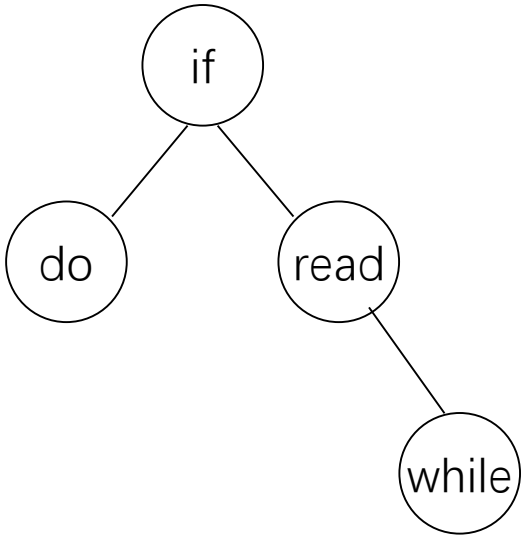
二分检索树:

T_{04}=2 =>T_{01}(左子树)，T_{24}(右子树)

T_{01}=1 =>T_{00}(左子树)，T_{11}(右子树)

T_{24}=3 =>T_{22}(左子树)，T_{34}(右子树)

- 树的形态



3.计算时间

- 当j-i=m时，有n-m+1个C(i,j)要计算
- C(i,j)的计算：O(m)

每个C(i,j)要求找出m个量中的最小值。

则，n-m+1个C(i,j)的计算时间：

$O(nm-m^2)$

对所有可能的m，总计算时间为：

$$\sum_{1\le m\le n}(nm-m^2)=O(n^3)$$

一种改进措施（D. E. Knuth）：

最优的k∈[R(i,j-1),R(i+1,j)]

4.算法描述

```
procedure OBST(P,Q,n)
/*给定n个标识符a1 < a2 < … < an。已知成功检索的概率P(i),不成功检索概率Q(i),
  0≤i≤n。算法对于标识符ai+1,…,aj计算最优二分检索树Tij的成本C(i,j)、根
  R(i,j)、权W(i,j) */
  real P(1:n),Q(1:n),C(0:n,0:n),W(0:n,0:n);integer R(0:n,0:n)
  for i←0 to n-1 do
      (W(i,i), R(i,i), C(i,i))←(Q(i),0,0)  //置初值
      (W(i,i+1), R(i,i+1), C(i,i+1))←
      (Q(i)+Q(i+1)+P(i+1),i+1,(i)+Q(i+1)+P(i+1)) //含有一个结点的最优树
  endfor
  (W(n,n), R(n,n), C(n,n))←(Q(n),0,0)
  for m←2 to n do //找有m个结点的最优树
      for i←0 to n-m do
          j←i+m
          W(i,j) ←W(i,j-1)+P(j)+Q(j)
          k←区间[R(i,j-1), R(i+1,j)]中使{C(i,l-1)+C(l,j)}取最小值的l值 //Knuth
          C(i,j) ←W(i,j)+C(i,k-1)+C(k,j)
          R(i,j)←k
      endfor
  endfor
end OBST
```

二分检索树

- OBST算法的计算时间:$O(n^2)$

动态规划 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。

动态规划 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) 字符串匹配
  - (3) 图像压缩
  - (4) 流水作业调度
  - (5) 背包问题
  - (6) 最优二叉搜索树

作业

- 算法分析题 3-5