第六z章 动态规划法
Dynamic Programming

湖南大学信息科学与工程学院

提纲

7.1 兔子的故事

一个故事：

一对兔子饲养在围墙中，如果它们每个月生一对兔子，且新生的兔子在第二个月后的每个月生一对兔子。假设所有兔子都不会死去，能够一直活下去，那么n个月以后这对兔子可以繁殖多少对兔子呢？（不包括开始的那对兔子）

如何建立数学模型呢？

7.1 兔子的故事

数学模型

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

斐波那契(Fibonacci)序列

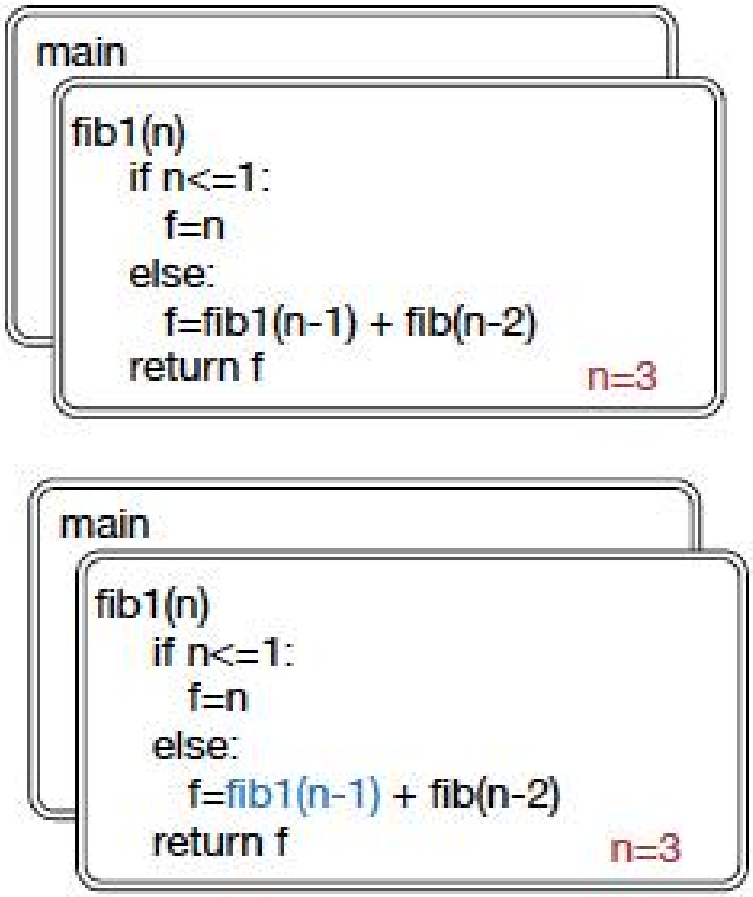
7.1 兔子的故事

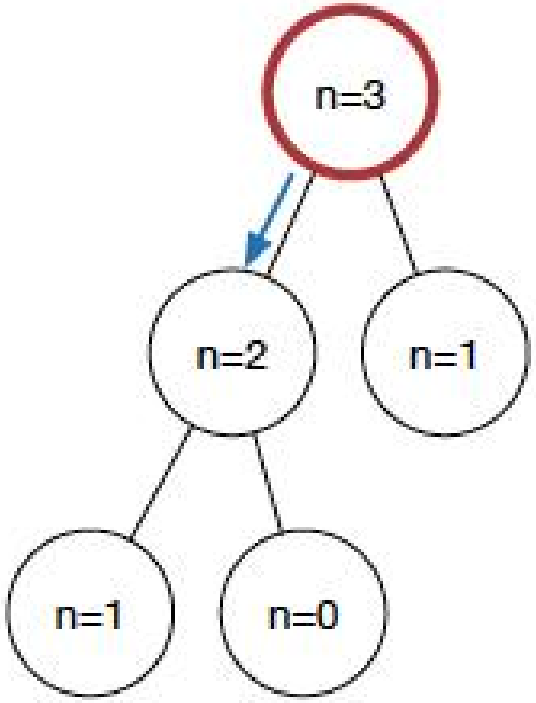
程序代码

```
def fib_rec(n):
    if n <= 1:
        f=n
    else:
        f=fib_rec(n-1)+fib_rec(n-2)
    return f


def main():
    num = 5
    print('{0:5}==>{1:10d}'.format('fib('+str(num)+')', fib_rec(num)))


if __name__ == "__main__":
    main()
```

7.1 兔子的故事

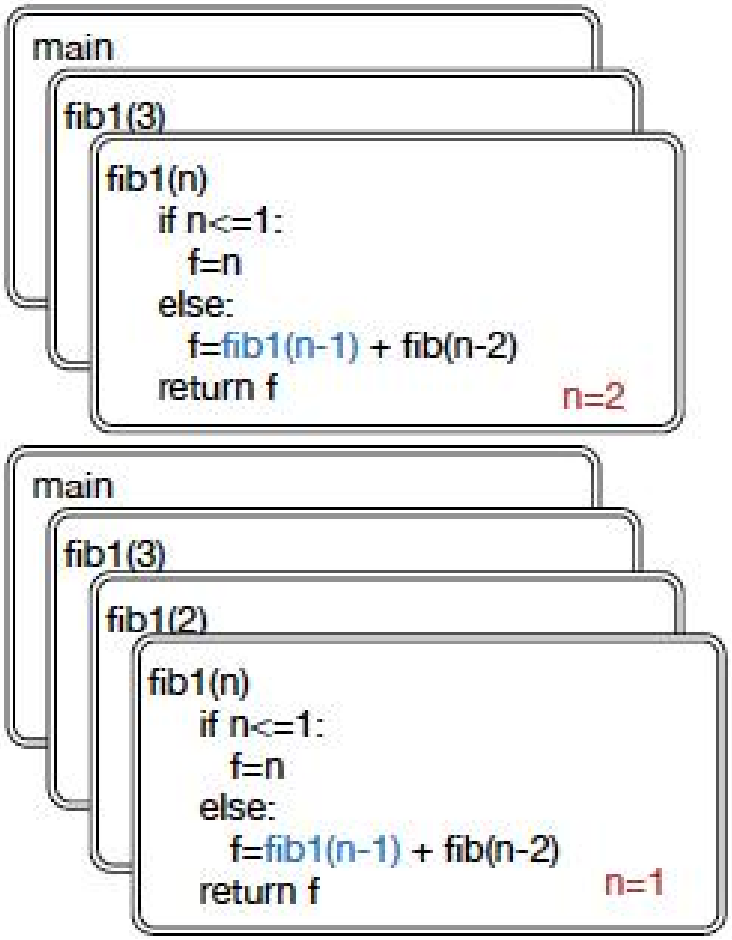
跟踪递归函数的执行过程

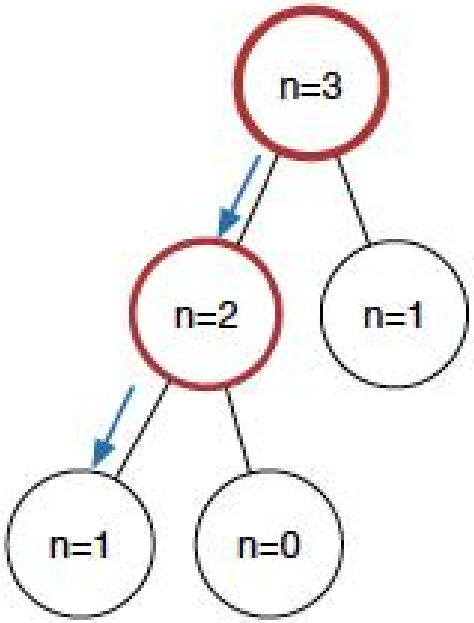




7.1 兔子的故事

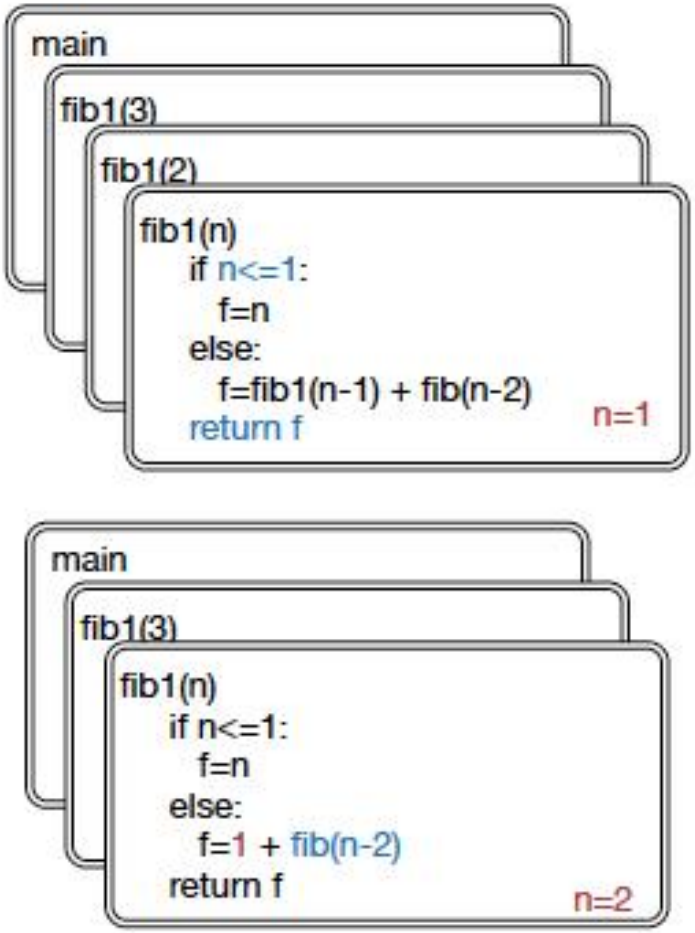
跟踪递归函数的执行过程

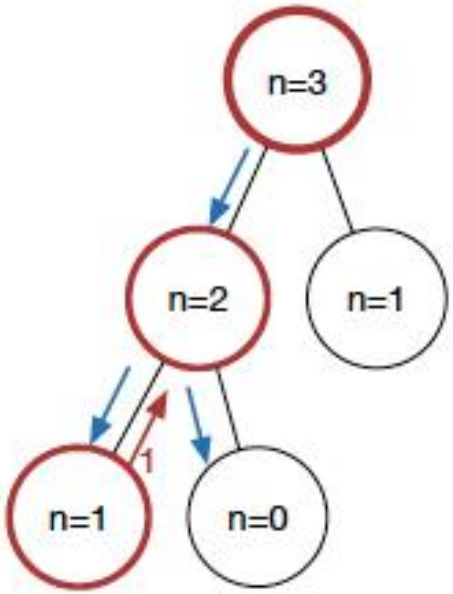




7.1 兔子的故事

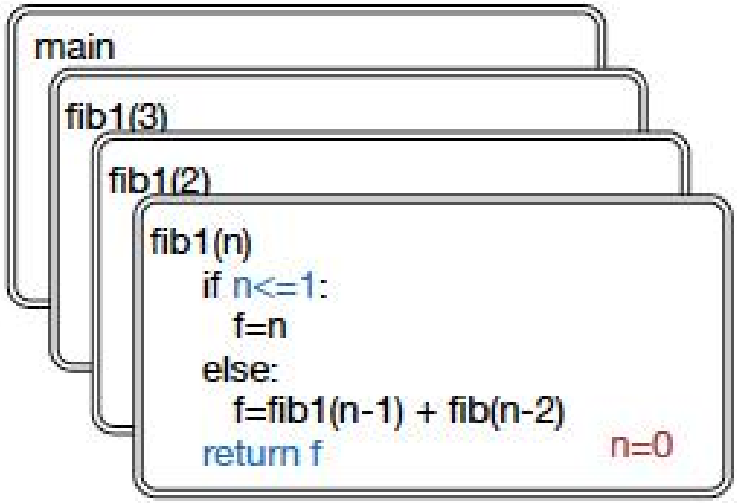
跟踪递归函数的执行过程

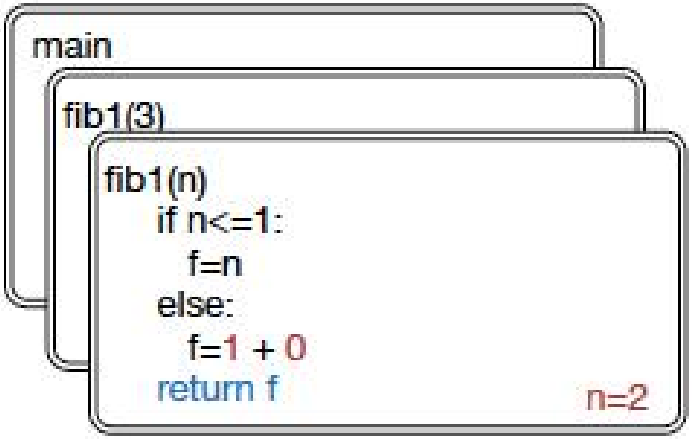


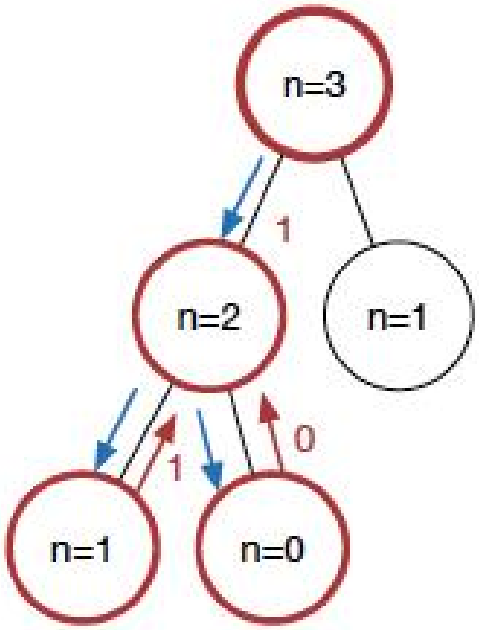


7.1 兔子的故事

跟踪递归函数的执行过程

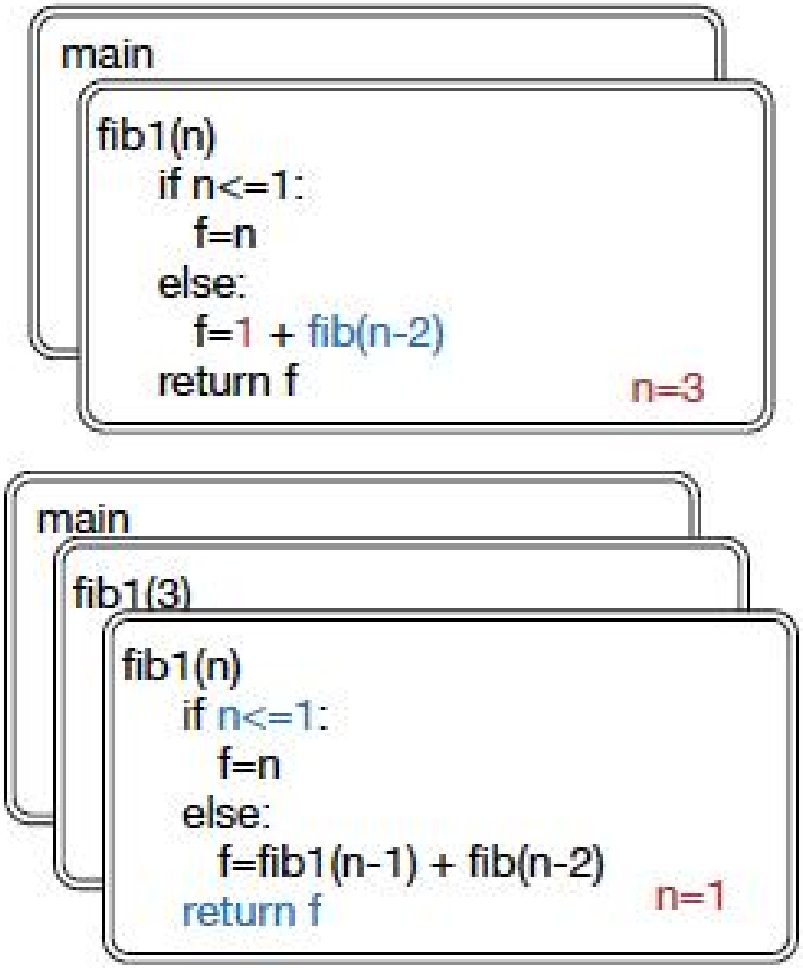


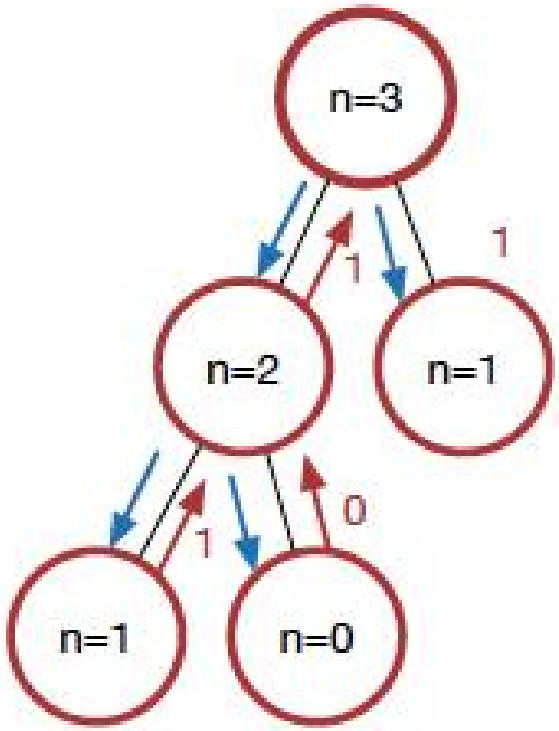




7.1 兔子的故事

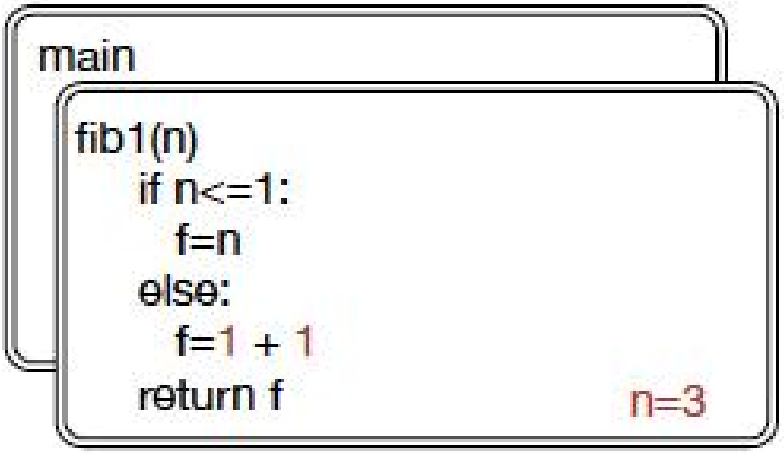
跟踪递归函数的执行过程

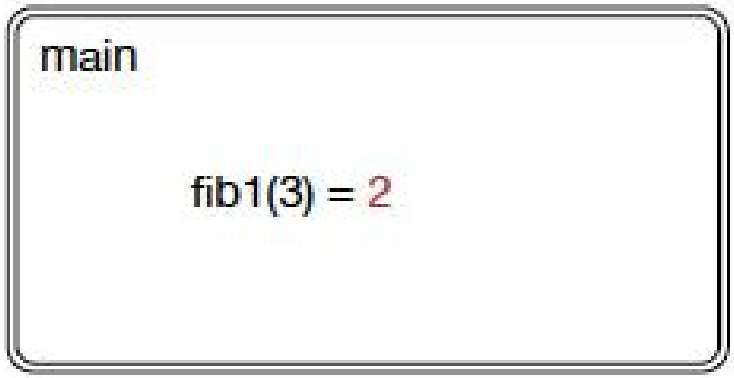


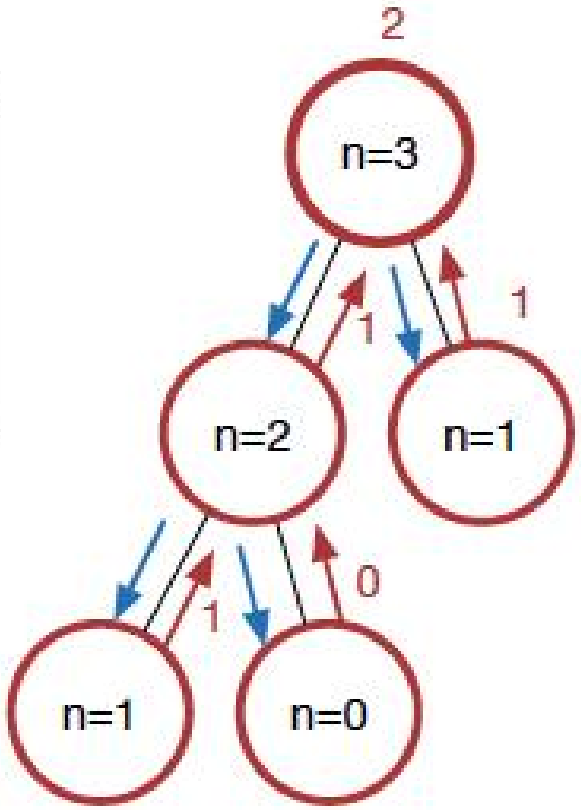


7.1 兔子的故事

跟踪递归函数的执行过程

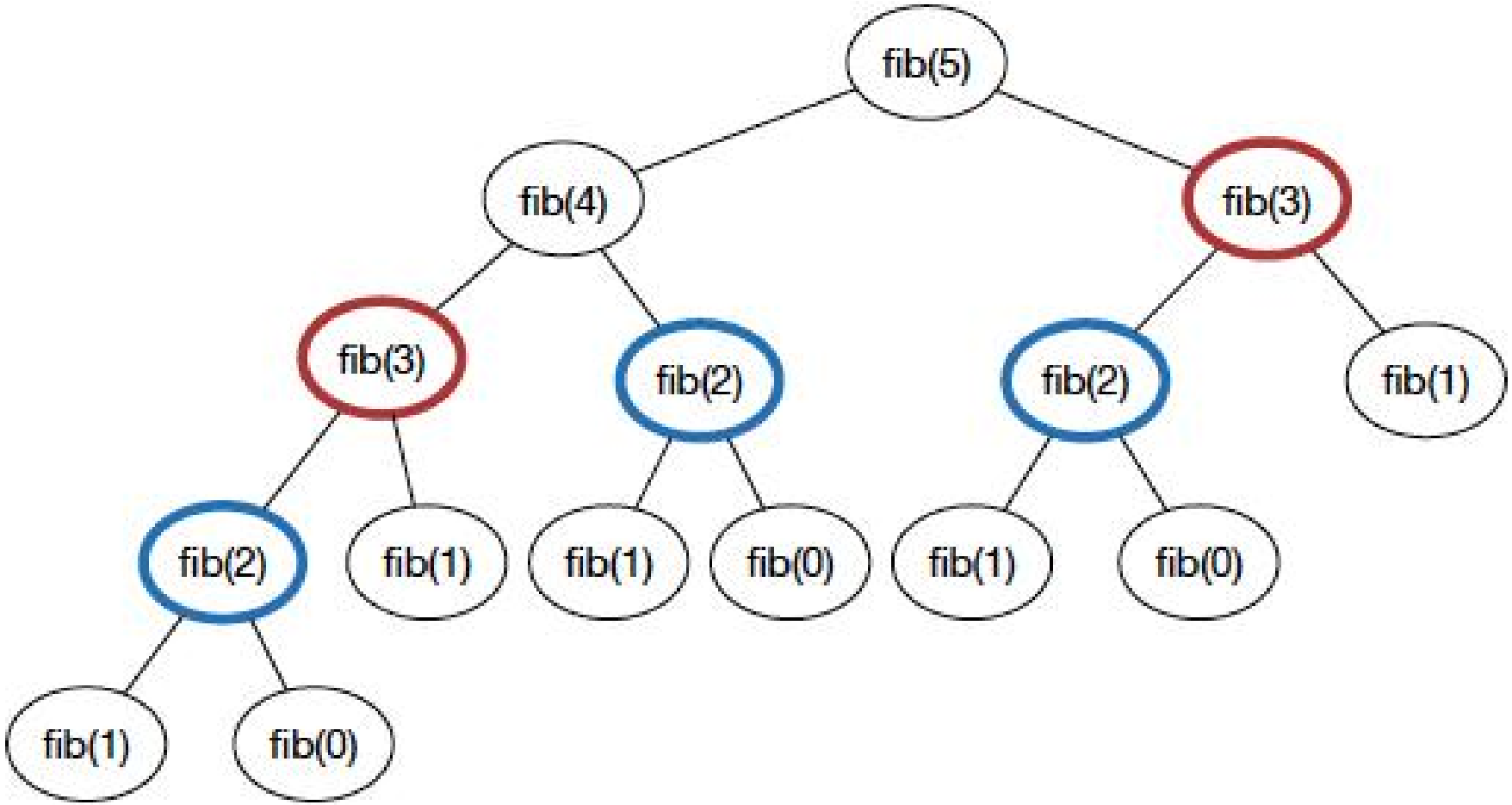






7.1 兔子的故事

求斐波那契序列算法的时间复杂度：$O(n^2)$



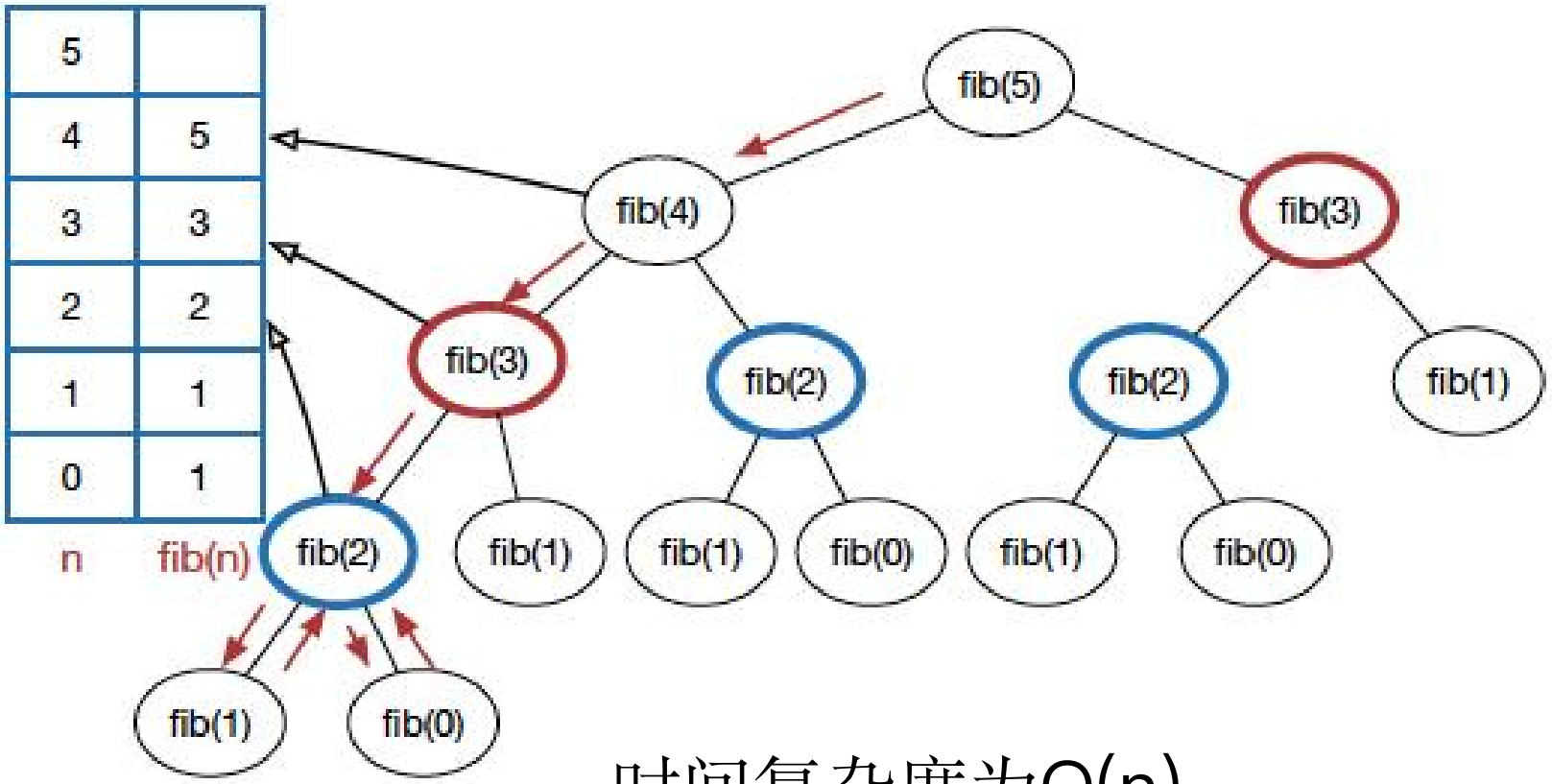
7.1 兔子的故事

采用“记忆”减少递归调用

0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	1	2	3	5	8	13	21	34	55	89	144

7.1 兔子的故事

采用“记忆”减少递归调用



时间复杂度为O(n)

7.1 兔子的故事

采用“记忆”减少递归调用

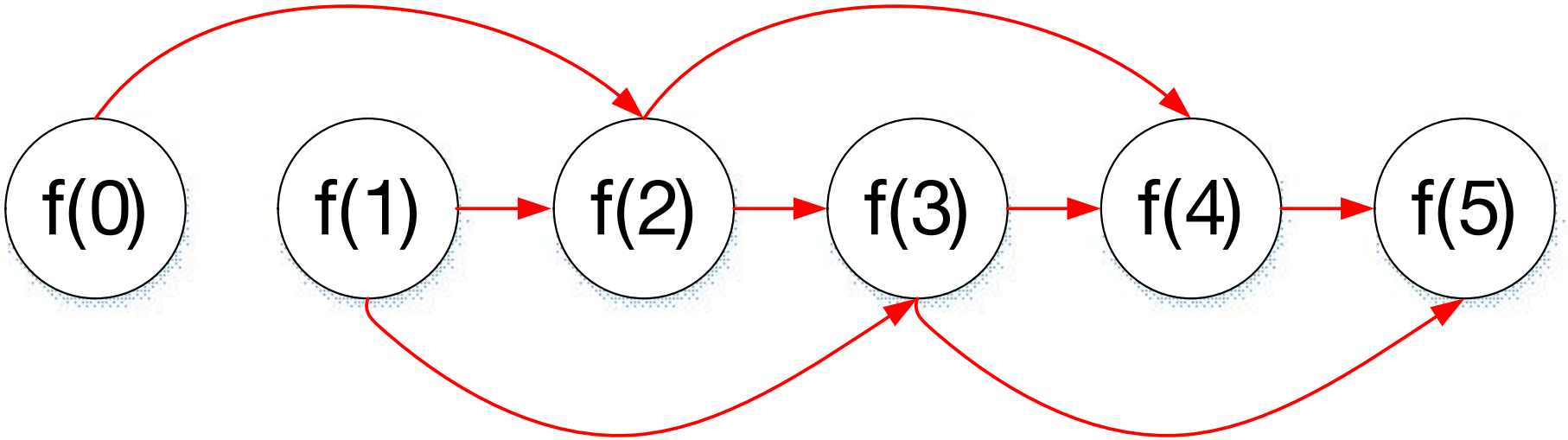
自上而下的实现

```
memo = {}
def fib2(n):
    if n in memo:                          # 查表
        return memo[n]
    else:
        if n <= 1:                         # 边界条件
            f = n
        else:
            f = fib2(n-1) + fib2(n-2)      # 递归调用
        memo[n] = f                        # 将结果存储于表中
        return f
```

7.1 兔子的故事

采用“记忆”传递信息

自底向上的实现



时间复杂度为O(n)

7.1 兔子的故事

采用“记忆”传递信息

自底向上的实现

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

时间复杂度为O(n)

7.1 兔子的故事

求解斐波那契数列问题可以转变为求解矩阵幂运算问题：

$$\begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \times \begin{bmatrix} F(1) \\ F(0) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \times \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

快速幂下，可以将复杂度将为O(log n)

提纲

7.1 兔子的故事

7.2 动态规划法的设计思想

7.3 子问题重叠的例子：三角数塔问题

7.4 最大子段和（最大子数组）

7.5 适用动态规划法的场景

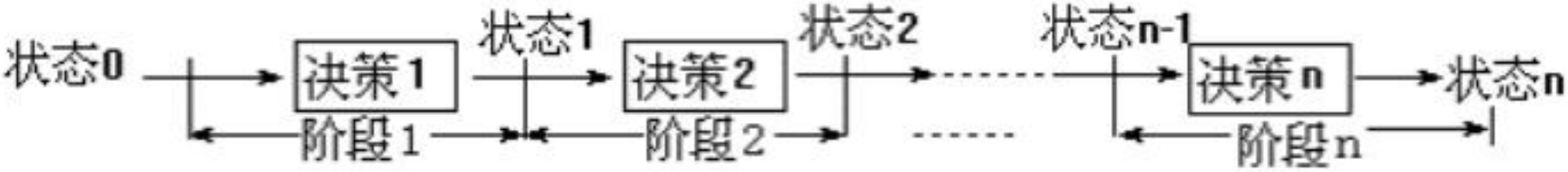
7.6 最长公共子序列

7.7 0–1背包问题

7.8矩阵乘法

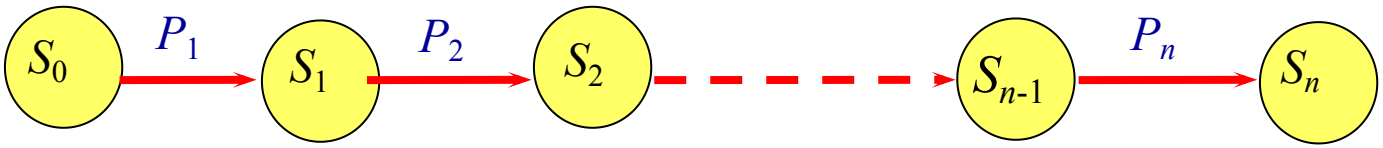
7.2 动态规划法的设计思想

7.2.2 最优化问题求解：多阶段决策过程



具有n个输入的最优化问题，其求解过程划分为若干个阶段，每一阶段的决策仅依赖于前一阶段的状态，由决策所采取的动作使状态发生转移，成为下一阶段决策的依据。

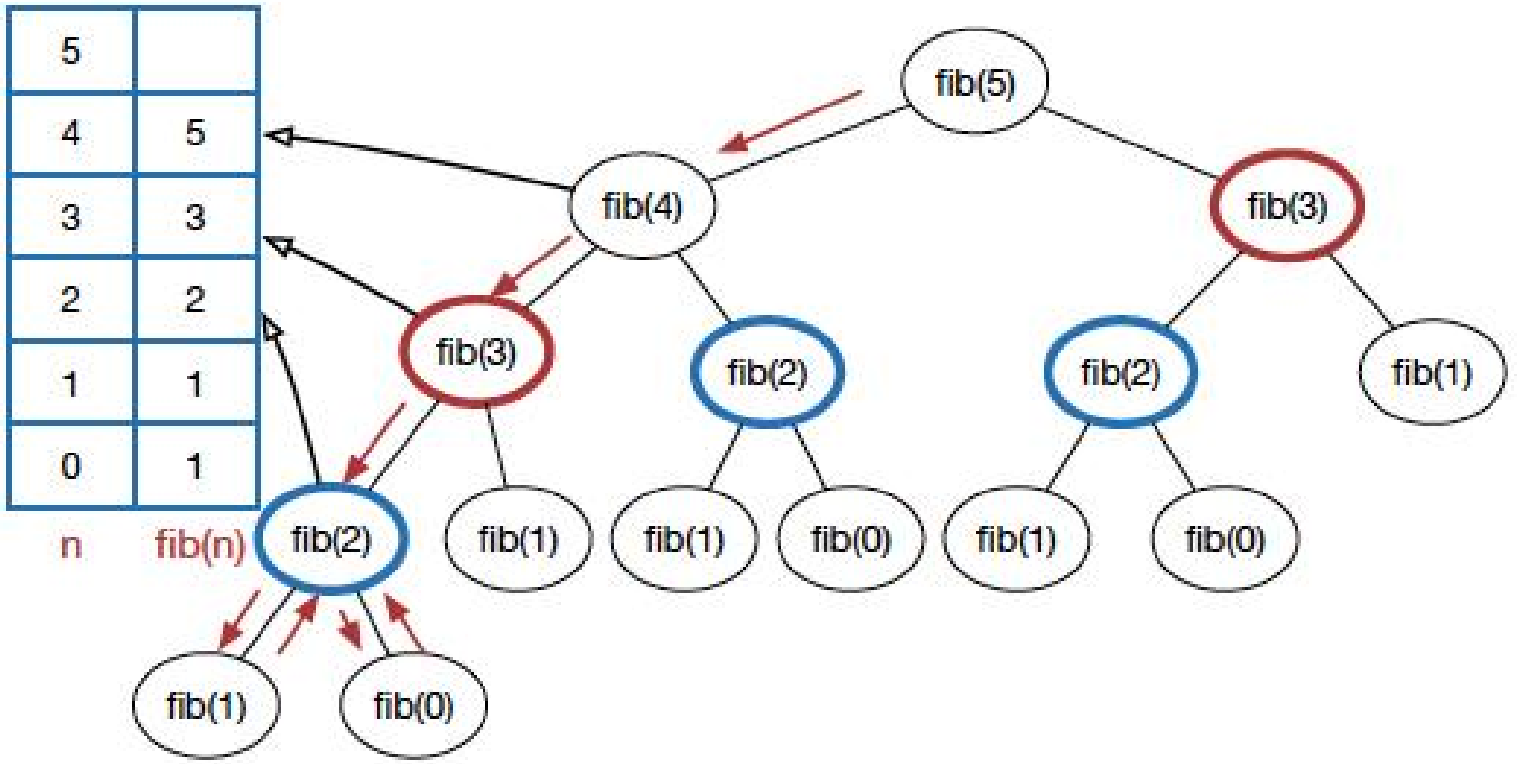
一个决策序列在不断变化的状态中产生。这个决策序列产生的过程称为多阶段决策过程。



7.2 动态规划法的设计思想

7.2.1 Dynamic Programming

Programming如何翻译?



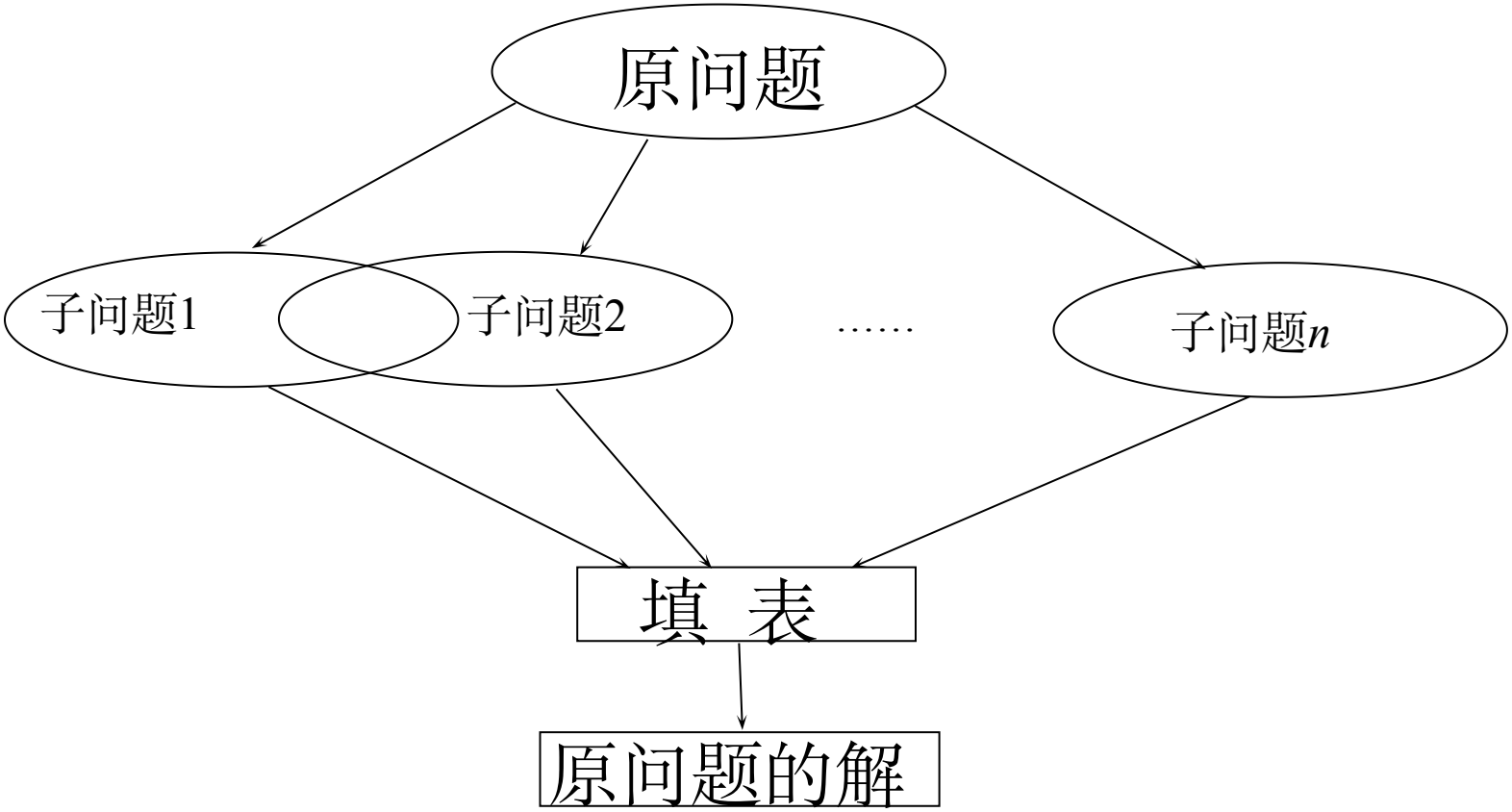
7.2 动态规划法的设计思想

7.2.2 动态规划法概念

动态规划是运筹学的一个分支，20世纪50年代初美国数学家Bellman等人在研究多阶段决策过程的优化问题时，提出了著名的最优性原理，创立了解决这类过程优化问题的新方法——动态规划法。

7.2 动态规划法的设计思想

7.2.2 动态规划法概念



7.2 动态规划法的设计思想

7.2.2 动态规划法概念

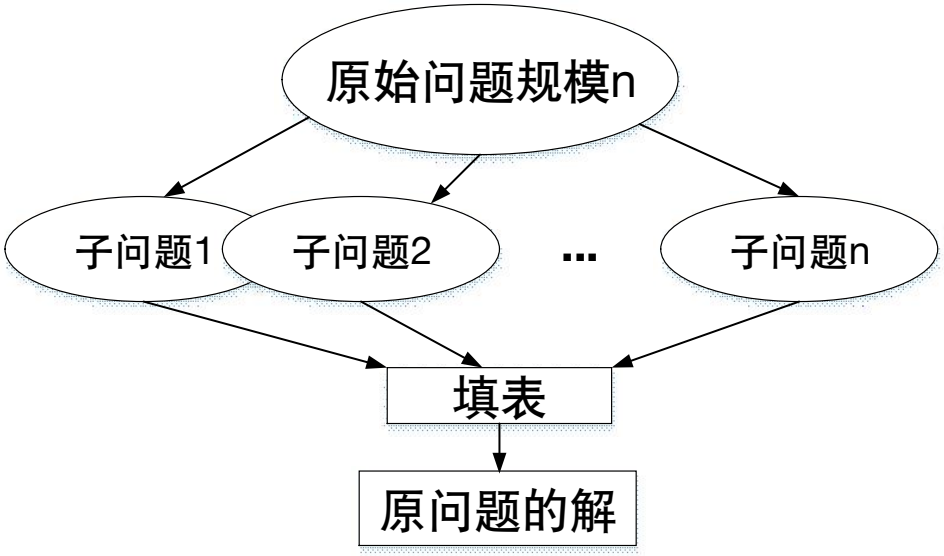
$$F(n)=\begin{cases}0 & n=0\\ 1 & n=1\\ F(n-1)+F(n-2) & n\geq 2\end{cases}$$

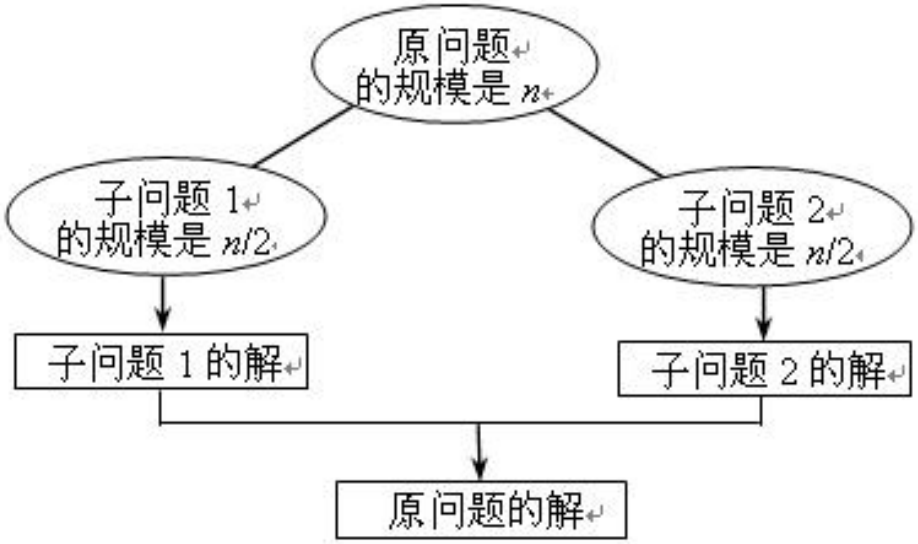
斐波那契(Fibonacci)序列

0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	1	2	3	5	8	13	21	34	55	89	144

7.2 动态规划法的设计思想

7.2.3 动态规划法vs 分治法





7.2 动态规划法的设计思想

7.2.4 动态规划求解问题的5个步骤

1. 定义子问题

2. 建立各个子问题之间的递归关系（动态规划函数，数学模型）

3. 自底向上的求解递归式（或自顶向下）

4. 组合所有子问题的解从而获得原问题的解

7.2 动态规划法的设计思想

7.2.5 动态规划算法时间复杂度分析

子问题数 × 每个子问题的单次求解时间

提纲

7.1 兔子的故事

7.2 动态规划法的设计思想

7.3 子问题重叠的例子：三角数塔问题

7.4 最大子段和（最大子数组）

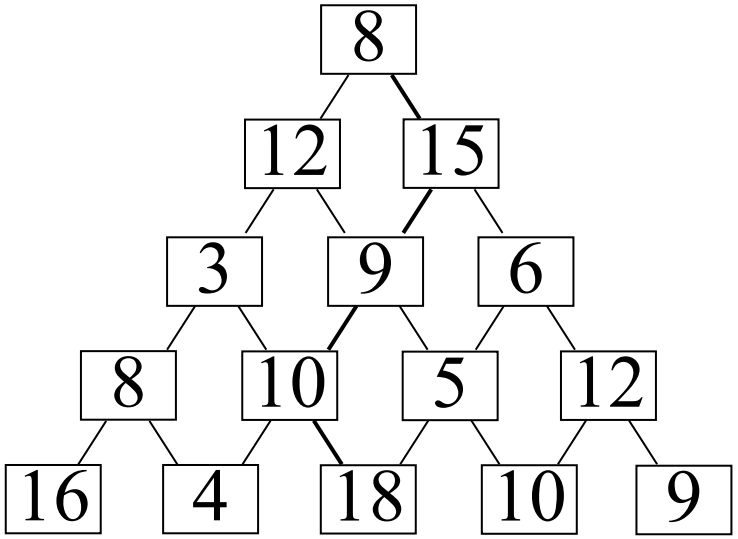
7.5 适用动态规划法的场景

7.6 最长公共子序列

7.7 0–1背包问题

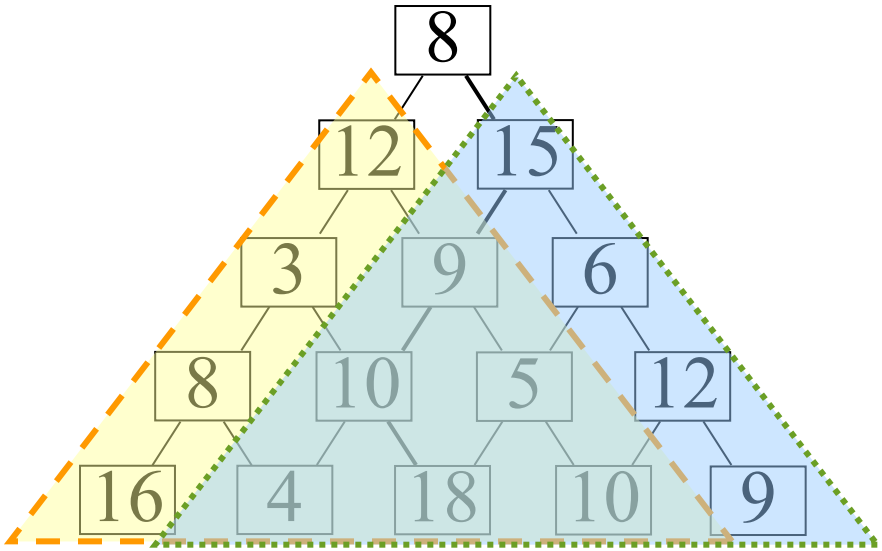
7.8矩阵乘法

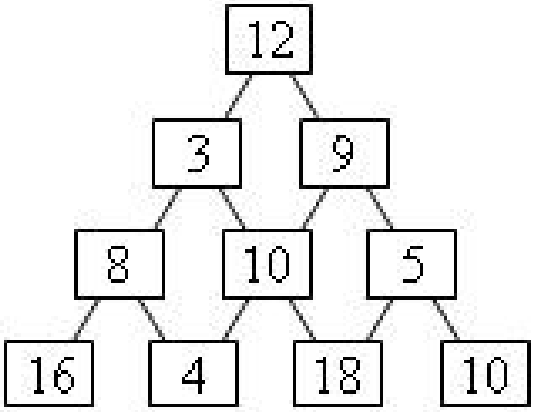
7.3子问题重叠的例子：三角数塔问题

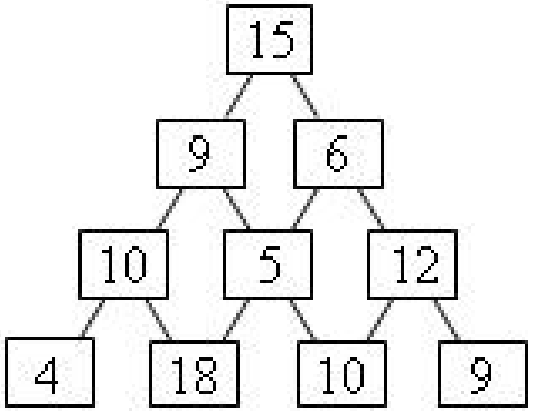


7.3子问题重叠的例子：三角数塔问题

想法





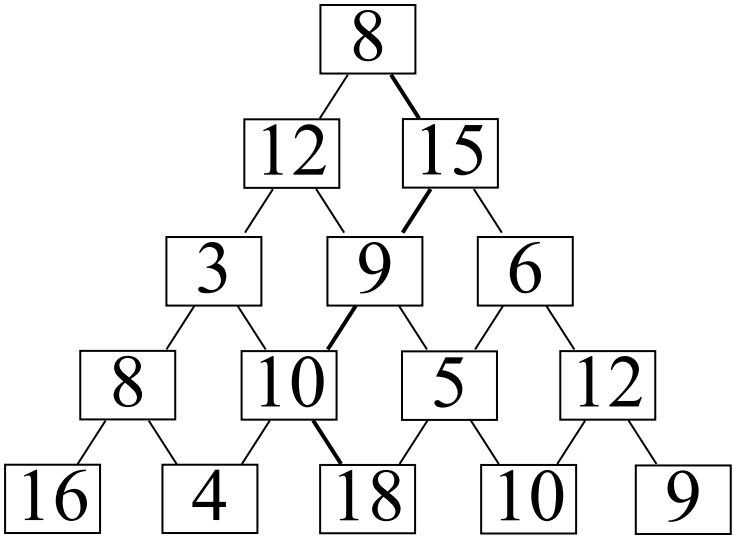


7.3子问题重叠的例子：三角数塔问题

三角数塔决策过程

第 1 层的决策	8+max{49,52} = 60				
第 2 层的决策	12+max{31,37} = 49	15+max{37,29} = 52			
第 3 层的决策	3+max{24, 28} = 31	9+max{28,23} = 37	6+max{23,22} = 29		
第 4 层的决策	8+max{16,4} = 24	10+max{4,18} = 28	5+max{18,10} = 23	12+ max{10,9} = 22	
初始化	16	4	18	10	9

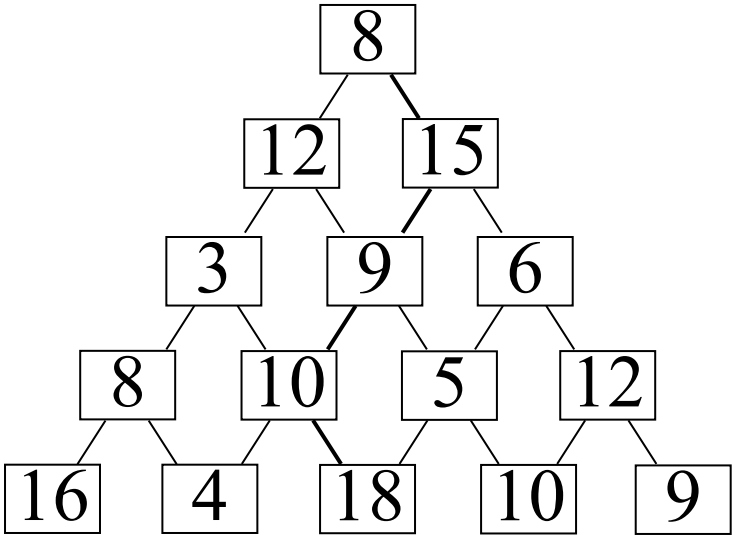
自底向上填写



7.3子问题重叠的例子：数塔问题

数据结构设计

$$data[5][5]=\begin{bmatrix} 8 & 0 & 0 & 0 & 0 \\ 12 & 15 & 0 & 0 & 0 \\ 3 & 9 & 6 & 0 & 0 \\ 8 & 10 & 5 & 12 & 0 \\ 16 & 4 & 18 & 10 & 9 \end{bmatrix}$$



7.3子问题重叠的例子：数塔问题

数据结构设计

$$\text{maxAdd}[n][n] = \begin{bmatrix} \mathbf{0} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 16 & 4 & 18 & 10 & 9 \end{bmatrix}$$

7.3子问题重叠的例子：数塔问题

动态规划函数：

$$\begin{cases} \text{maxAdd}[n-1][j] = \text{data}[n-1][j] \ \ (0 \le j \le n-1) \\ \text{maxAdd}[i][j] = \text{data}[i][j] + \max(\text{maxAdd}[i+1][j], \text{maxAdd}[i+1][j+1]) \\ (0 \le i \le n-2, 0 \le j \le i) \end{cases}$$

$$\begin{cases} \text{path}[i][j] = j \quad \text{maxAdd}[i+1][j] > \text{maxAdd}[i+1][j+1] \\ \ (0 \le i \le n-2, 0 \le j \le i) \\ \text{path}[i][j] = j+1 \quad \text{maxAdd}[i+1][j] \le \text{maxAdd}[i+1][j+1] \\ (0 \le i \le n-2, 0 \le j \le i) \end{cases}$$

7.3子问题重叠的例子：数塔问题

伪代码

1. 初始化数组maxAdd的最后一行为数塔的底层数据：

```
for (j = 0; j < n; j++)
    maxAdd[n-1][j] = d[n-1][j];
```

2. 从第n-1层开始直到第1层对下三角元素maxAdd[i][j]执行下述操作：

2.1 maxAdd[i][j]= d[i][j] + max{maxAdd[i+1][j], maxAdd[i+1][j+1]};

2.2 如果选择下标j的元素，则path[i][j] = j,

否则path[i][j] = j+1；

3. 输出最大数值和maxAdd[0][0]；

4. 根据path数组确定每一层决策的列下标，输出路径信息。

7.3子问题重叠的例子：数塔问题

时间复杂度分析：$O(n^2)$

7.3子问题重叠的例子：数塔问题

代码实现

```
import numpy as np
def DataTower1(data):
    maxAdd = np.zeros(np.array(data).shape) #初始化
    path = np.zeros(np.array(data).shape) #初始化
    n,m = np.array(data).shape #n为行数，在这里相当于塔数
    #下面这个循环相当于把data的最后一行付给了maxAdd的最后一行，就是从最后一行向上递归
    for i in range(n):
        maxAdd[n-1][i] = data[n-1][i]
    print(maxAdd)
    for i in range(n-2,-1,-1):    #进行第i层的决策
        #填写addMax[i][j],只填写下三角
        for j in range(i+1):
            if maxAdd[i+1][j] > maxAdd[i+1][j+1] :
                maxAdd[i][j] = data[i][j] + maxAdd[i+1][j]
                path[i][j] = j  #本次决策选择下标j的元素
            else :
                maxAdd[i][j] = data[i][j] + maxAdd[i+1][j+1]
                path[i][j] = j + 1   #本次决策选择下标j+1的元素
    print('path:%d'%data[0][0])  #输出顶层数字
    j=int(path[0][0])  #顶层决策试选择下一层列下标伪path[0][0]的元素
    for i in range(1,n):
        print('-> %d'%data[i][j])
        j=int(path[i][j])   #本层决策试选择下一层列下标伪path[i][j]的元素
    return maxAdd[0][0]
```

提纲

7.4 最大子段和（最大子数组）

最大字段和问题或最大子数组问题

给定由n个整数（可能有负整数）组成的序列$(a_1, a_2, \ldots, a_n)$，最大子段和问题要求该序列形如 的最大值（$1 \leqslant i \leqslant j \leqslant n$），当序列中所有整数均为负整数时，其最大子段和为0。例如，序列(−20, 11, −4, 13, −5, −2)的最大子段和为 $\sum_{k=2}^{4} a_k = 20$

7.4 最大子段和（最大子数组）

蛮力法：时间复杂度

- $O(n^2)$

分治算法：时间复杂度

- $O(n \log n)$

7.4 最大子段和（最大子数组）

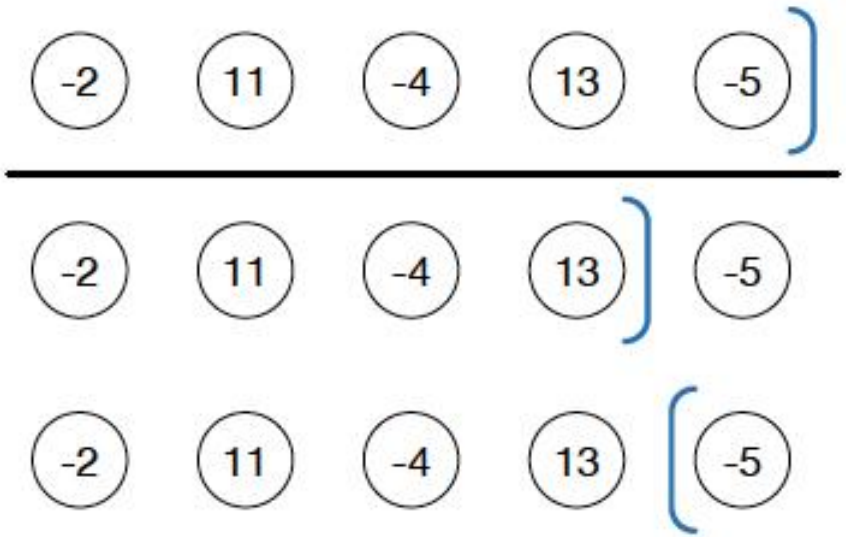
动态规划: Step1 定义子问题

- 该问题输入的是一个有n 个元素的序列A，不妨设P(i) 为直到第i 个元素的最大和
  - 将输入序列的前缀作为子问题

7.4 最大子段和（最大子数组）

动态规划: Step2 猜测部分解

- 子问题P(i) 的解包括第i 个元素，此时有P(i)=P(i –1)+A[i]
- 子问题P(i) 的解不包括第i 个元素，意味着需要从第i 个元素开始重新计算值，即P(i)=A[i]



7.4 最大子段和（最大子数组）

动态规划： Step3 建立各个子问题之间的动态规划函数

$$P(i) = \max\{P(i-1) + A[i], A[i]\}$$

初始条件为

$$P(0) = 0$$

7.4 最大子段和（最大子数组）

动态规划: Step4 自底向上的求解递归式

```
3  def bottom_up_cont_subseq(alist):
4      table = [None] * (len(alist) + 1)
5      table[0] = 0
6      for i in range(1, len(alist)+1):
7          table[i] = max(table[i-1] + alist[i-1], alist[i-1])
8      return table
```

A=[–2, 11, –4, 13, –5, 2]

i	0	1	2	3	4	5	6
table[i]	0	-2	11	7	20	15	17

时间复杂度: O(n)

7.4 最大子段和（最大子数组）

动态规划: Step4 自底向上的求解递归式

```
def track_back_subseq(alist, table):
    select = []
    ind_max = np.argmax(table)  # 得到table中最大值索引
    print('ind_max=',ind_max)
    while ind_max >= 1:
        if table[ind_max] == alist[ind_max-1]+table[ind_max-1]:
            print('alist:',alist[ind_max-1])
            select.append(alist[ind_max-1])
            ind_max -= 1
        else:
            select.append(alist[ind_max-1])
            print('alist-break:',alist[ind_max-1])
            break
    return select
```

```
[-2, 11, -4, 13, -5, 2]
[None, None, None, None, None, None, None]
[0, -2, 11, 7, 20, 15, 17]
ind_max= 4
alist: 13
alist: -4
alist-break: 11
[13, -4, 11]
```

i	0	1	2	3	4	5	6
A[i]	-2	11	-4	13	-5	2	
table[i]	0	-2	11	7	20	15	17

提纲

7.5 适用动态规划法的场景

什么情况下可以采用动态规划方法求解最优问题呢?

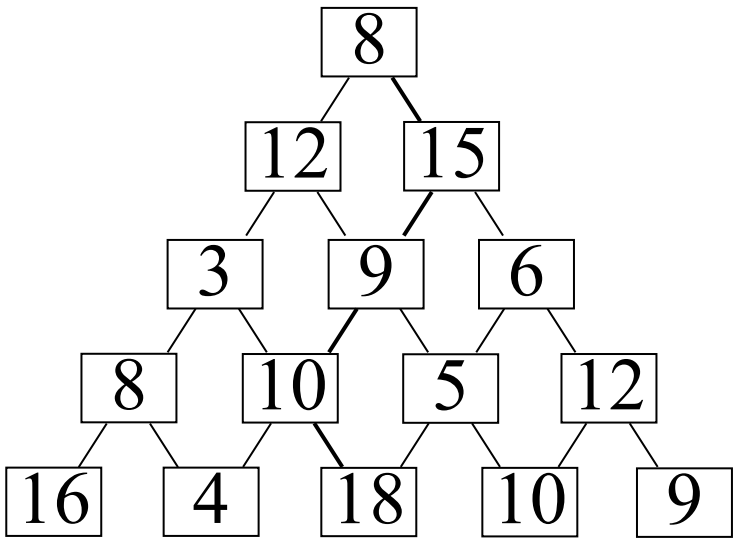
具备两个要素:

(1) 最优子结构

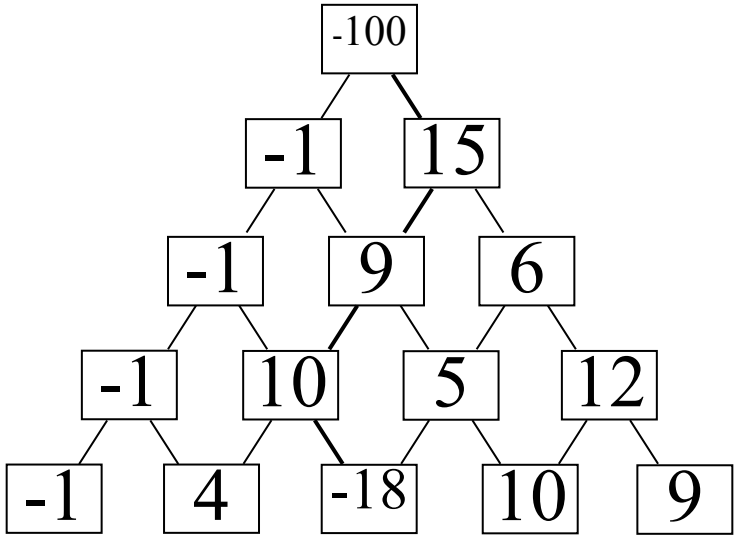
(2) 子问题重叠

7.5 适用动态规划法的场景

最优子结构的反例：数塔问题



路径上的数值和最大



路径上的数值和的绝对值最大

7.5 适用动态规划法的场景

最优子结构的反例：数塔问题

$$\begin{cases} \text{maxAdd}[n-1][j] = \text{data}[n-1][j] \ \ (0 \le j \le n-1) \\ \text{maxAdd}[i][j] = \text{data}[i][j] + \max(\text{maxAdd}[i+1][j], \text{maxAdd}[i+1][j+1]) \\ (0 \le i \le n-2, 0 \le j \le i) \end{cases}$$

maxAdd[n–1][i] = abs(data[n–1][i])
maxAdd[i][j]=abs(data[i][j])+max(maxAdd[i+1][j],
maxAdd[i+1][j+1])

7.5 适用动态规划法的场景

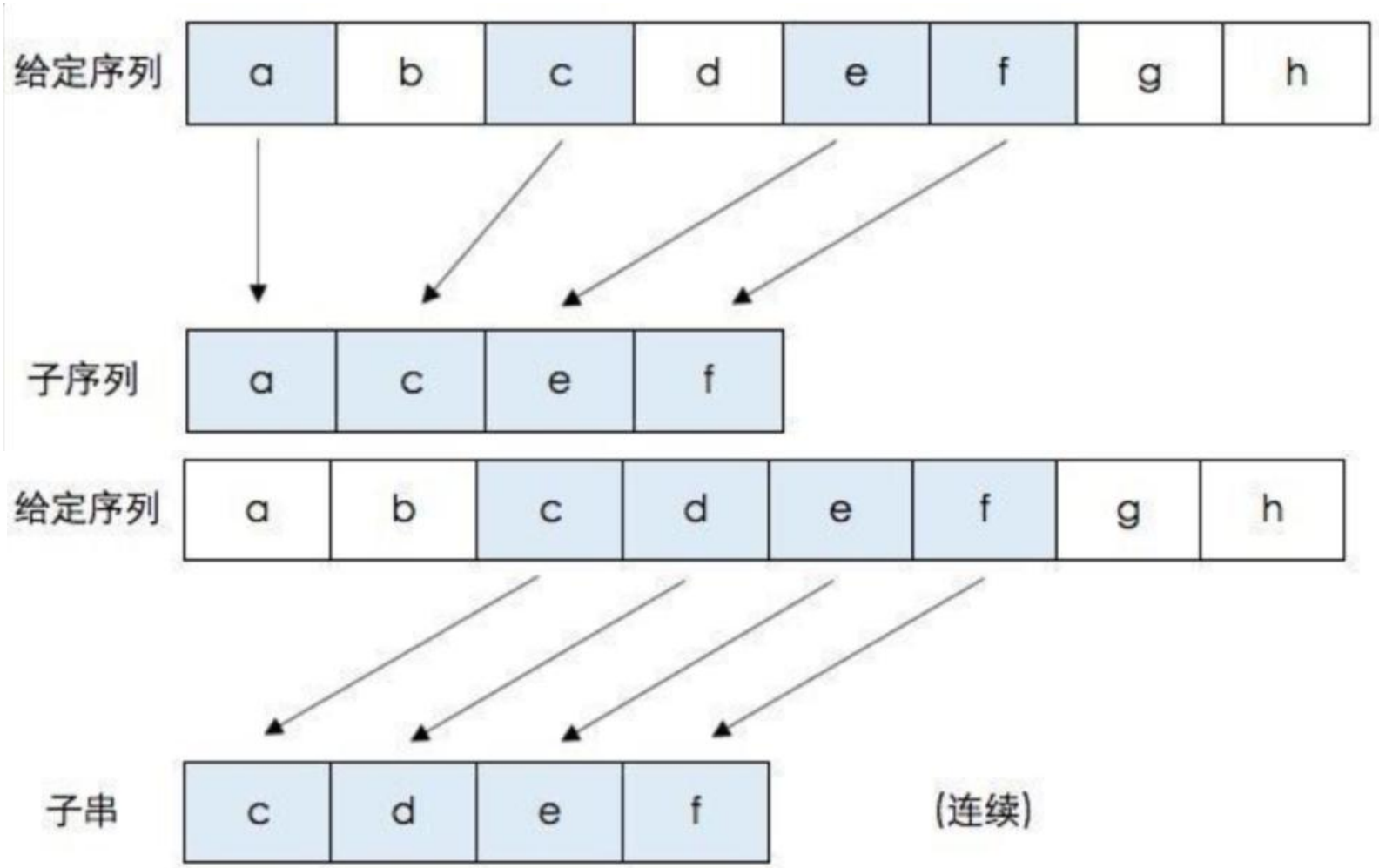
动态规划求解问题的5个步骤

1. 定义子问题
2. 猜测部分解（刻画问题的最优子结构）
3. 建立各个子问题之间的递归关系（动态规划函数，数学模型）
4. 自底向上的求解递归式（或自顶向下）
5. 组合所有子问题的解从而获得原问题的解

提纲

7.6 最长公共子序列

子序列与子串



7.6 最长公共子序列

问题描述:

对给定序列$X=(x_1, x_2,\ldots, x_m)$和序列$Z=(z_1, z_2,\ldots, z_k)$，Z是X的子序列当且仅当存在一个递增下标序列$(i_1, i_2,\ldots, i_k)$，使得对于所有$j=1, 2, \ldots, k$，有$x_{i_j} = z_j$ $(1\le i_j\le m)$ 。

给定两个序列X和Y，当序列Z既是X的子序列又是Y的子序列时，称Z是序列X和Y的公共子序列。最长公共子序列问题就是在序列X和Y中查找最长的公共子序列。 **Longest-common-subsequence:LCS**

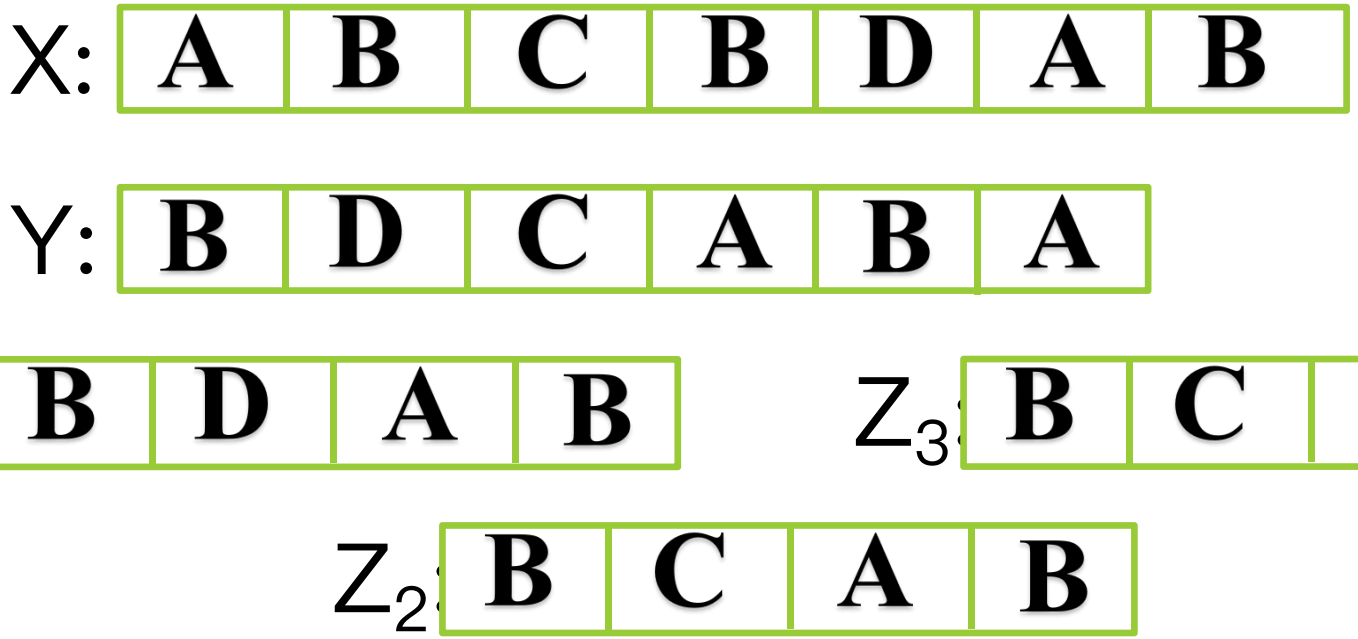
7.6 最长公共子序列

应用场景:

- 基因工程DNA
- 软件版本管理
- 软件自动测试
- 检查论文抄袭率
- 程序代码量相似度度量
- 视频段匹配

7.6 最长公共子序列

例子



7.6 最长公共子序列

暴力搜索方法求解LCS问题

对于此问题，可以采用暴力求解的方式来比对，即穷举出X的所有子序列，用每个子序列与y做一 一比较。假如X序列共有m个元素，对每个元素可以决定选或不选，则X的子序列个数共有2^m个，可见与长度m呈指数阶，这种方法效率会很低。

7.6 最长公共子序列

问题步骤

第一步、给定序列$X =< x_1, x_2, \ldots, x_m >$和$Y =< y_1, y_2, \ldots, y_n >$，求X 和Y 的最长公共子序列**长度**

第二步、给定序列$X =< x_1, x_2, \ldots, x_m >$和$Y =< y_1, y_2, \ldots, y_n >$，求X 和Y 的最长公共子序列

7.6 最长公共子序列

动态规划: Step1 定义子问题

给定序列$X =< x_1, x_2, \ldots, x_m >$和$Y =< y_1, y_2, \ldots, y_n >$，考虑X和Y的前缀，定义C[i, j]为$X_i =< x_1, x_2, \ldots, x_i >$与$Y_j =< y_1, y_2, \ldots, y_j >$的最长公共子序列的长度

于是，C[m, n]就是X和Y的最长公共子序列的长度

7.6 最长公共子序列

Step2:刻画最长公共子序列问题的最优子结构（猜测部分解）

定理15.1（LCS的最优子结构）

设$X=x_1x_2\ldots x_m$和$Y=y_1y_2\ldots y_n$是两个序列，$Z_k=z_1z_2\ldots z_k$是这两个序列的一个最长公共子序列。

1.如果$x_m=y_n$，那么$z_k=x_m=y_n$，且Z_{k-1}是X_{m-1}，Y_{n-1}的一个最长公共子序列；

2.如果$x_m\neq y_n$，那么Z_k是X_{m-1}，Y_n的一个最长公共子序列或者Z_k是X_m，Y_{n-1}的一个最长公共子序列。

从上面三种情况可以看出，两个序列的LCS包含两个序列的前缀的LCS。因此，LCS问题具有最优子结构特征。

7.6 最长公共子序列

Step2:刻画最长公共子序列问题的最优子结构（猜测部分解）

定理15.1（LCS的最优子结构）

1.对 X = "AGTGATG",Y = "GTTAG",则Z = "GTAG"，此时m = 7，n = 5，k = 4（从1开始）。
比较X和Y最后一位，x7 = y5，同时删去，Xm–1 = "AGTGAT"，Yn–1 = "GTTA"，Zk–1 = "GTA"。我们可以发现，Zk–1就是Xm–1和Yn–1的一个LCS。
2.对 X = "AGTGAT",Y = "GTTA",则Z = "GTA"，此时m = 6，n = 4，k = 3（从1开始）。
比较最后一位，x6 ≠ y4，z3 ≠ x6 ，Xm–1 = "AGTGA"，Z是Xm–1和Yn的一个LCS。
3.将2的X和Y互换即可
对 Y = "AGTGAT",X = "GTTA",则Z = "GTA"，此时n = 6，m = 4，k = 3（从1开始）。
比较最后一位，x4 ≠ y6，z3 ≠ x4 ，Yn–1 = "AGTGA"，Z是Xm和Yn–1的一个LCS。
此 定理说明两个序列的一个LCS也包含两个序列的前缀的一个LCS，即LCS问题具有最优子结构性质。

7.6 最长公共子序列

动态规划： Step3 建立各个子问题之间的动态规划函数

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

7.6 最长公共子序列

动态规划: Step4 自底向上的求解递归式(伪代码，计算LCS长度)

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

		j	0	1	2	3	4	5	6
i			y_j	B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

7.6 最长公共子序列

动态规划: Step4 自底向上的求解递归式(伪代码，计算LCS长度)

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1				
D	0	0	1	1				
C	0	0	1	2				
A	0							
B	0							
A	0							

7.6 最长公共子序列

动态规划: Step4 自底向上的求解递归式(伪代码，计算LCS长度)

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

7.6 最长公共子序列

代码实现

```
def LCS_LENGTH2(X,Y):
    m = len(X)
    n = len(Y)
    c = [[0 for i in range(n+1)] for j in range(m+1)]
    #print(c)
    b = [[0 for i in range(n+1)] for j in range(m+1)]
    for i in range(1,m+1):
        for j in range(1,n+1):
            if X[i-1]==Y[j-1]:
                c[i][j]=c[i-1][j-1]+1
                b[i][j]='x'
            elif c[i-1][j]>=c[i][j-1]:
                c[i][j]=c[i-1][j]
                b[i][j]='u'
            else:
                c[i][j]=c[i][j-1]
                b[i][j]='l'
    return c,b
```

7.6 最长公共子序列

动态规划: Step4 自底向上的求解递归式(伪代码，构造LCS)

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2       return
3  if b[i, j] == "↖"
4       PRINT-LCS(b, X, i − 1, j − 1)
5       print x_i
6  elseif b[i, j] == "↑"
7       PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

i		j	0	1	2	3	4	5	6
		y_j		B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

7.6 最长公共子序列

代码实现

```
def PRINT_LCS2(b,X,i,j):
    if i == 0 or j == 0:
        return
    if b[i][j] == 'x':
        PRINT_LCS2(b,X,i-1,j-1)
        print(X[i-1],end='')
    elif b[i][j] == 'u':
        PRINT_LCS2(b,X,i-1,j)
    else:
        PRINT_LCS2(b,X,i,j-1)
```

提纲

7.7 0–1背包问题

- 假设有n 个物品，每个物品i 的价值为vi，大小为si。包的容量为S，要求从这n 个物品中挑选若干物品装进包中，在所有装进包中物品的大小小于等于包容量S 的前提下，包中物品总价值最大
- 例 (s0 = 3; v0 = 4); (s1 = 4; v1 = 5); (s2 = 5; v2 = 6)
  - 包的容量S = 10。那么，应该选择物品(s1 = 4; v1 = 5); (s2 = 5; v2 =6)
  - s1 +s2 = 9 ⩽ S = 10，且这种选择的情况下总价值v1 +v2 = 11 最大

7.7 0–1背包问题

贪心解法

- 迭代地选取v_i/s_i最大的物品
- 这个算法无法得到最优解
- 反例：$s_1 = 1$，$v_1 = 2$；$s_2 = S$，$v_2 = S$

7.7 0–1背包问题

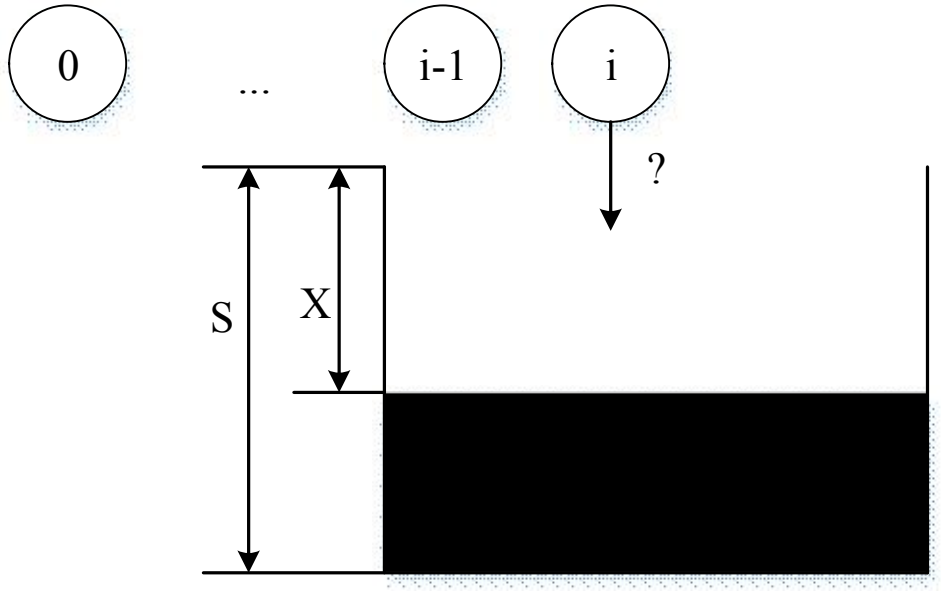
step1. 定义子问题

- 可以定义子问题从物品0; …… ; i – 1; i 中选取若干物品置于包中，这些物品的重量(容量)为X，且获取了最佳收益。

- 假设该子问题可以经由策略Knapsack 来求解，该策略此时的输入参数为Knapsack(i;X)。

7.7 0–1背包问题

step2. 刻画最优子结构



- 物品 i 不放入包中。那么 Knapsack(i, X) 的解等于从物品 $0, \cdots, i-1$ 选取容量为 X 的物品价值，也就是 Knapsack(i, X)=Knapsack$(i-1, X)$;
- 物品 i 放入包中。那么 Knapsack(i, X) 的解应该等于剩余物品 $0, \cdots, i-1$ 放入容量为 $X - s_i$ 的包中物品价值再加上物品 i 的价值 v_i，也就是 Knapsack(i, X)=Knapsack$(i-1, X-s_i)+v_i$。

7.7 0–1背包问题

step3. 建立各个子问题之间的递归关系（动态规划函数）

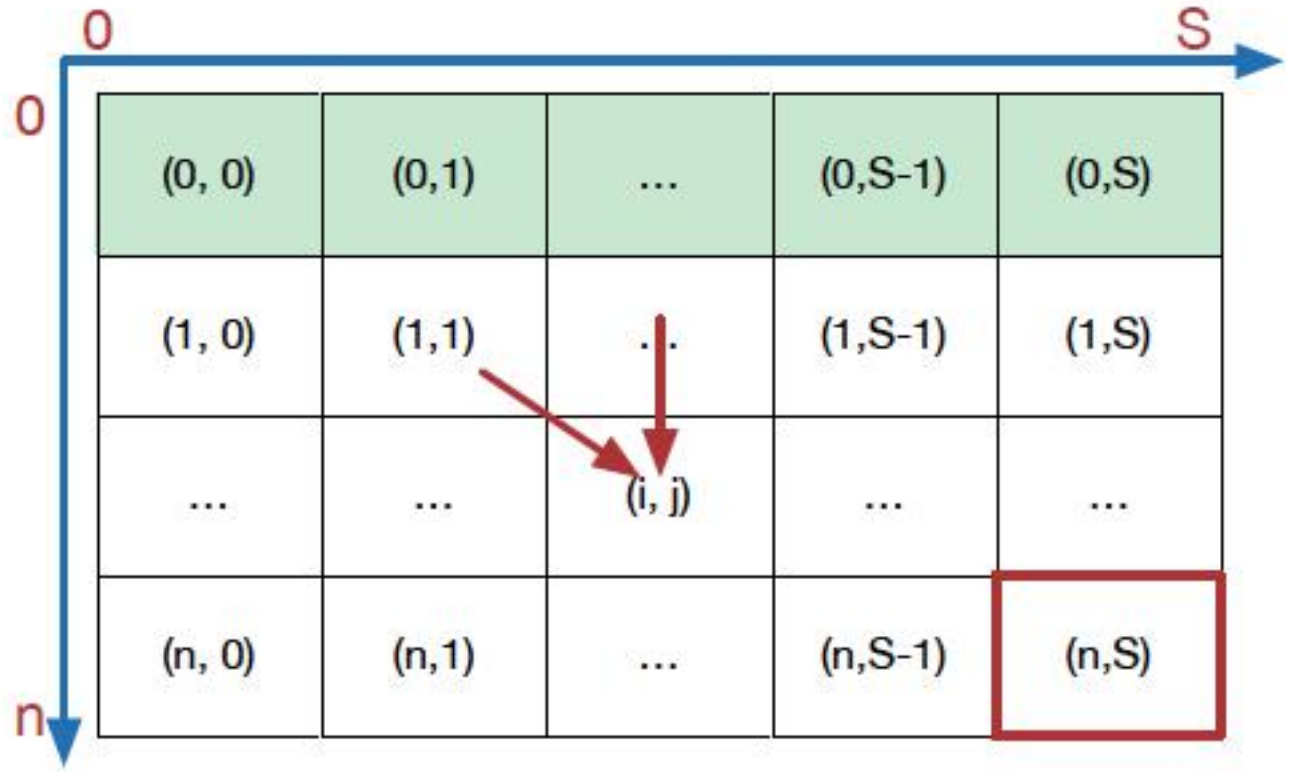
1. $X<s_i$时，
   $Knapsack(i, X)=Knapsack(i-1, X)$

2. $X>=s_i$时，
   $Knapsack(i, X)=max\{Knapsack(i-1, X-s_i)+v_i, Knapsack(i-1, X)\}$

https://www.cnblogs.com/Christal-R/p/Dynamic_programming.html

7.7 0–1背包问题

自底向上的求解递归式：填表



7.7 0–1背包问题

step3. 建立各个子问题之间的递归关系（动态规划函数）

程序代码中的符号

1) j(or X)<s(i)　　V(i,j)=V(i–1,j)

2) j(or X)>=s(i)　　V(i,j)=max { V(i–1,j), V(i–1,j–s(i))+v(i) }

V(i, 0)= V(0, j)=0

7.7 0–1背包问题

第一阶段，只装入前1个物品，确定在各种情况下的背包能够得到的最大价值；

第二阶段，只装入前2个物品，确定在各种情况下的背包能够得到的最大价值；依此类推，直到第n个阶段。

最后，$V(n,C)$便是在容量为C的背包中装入n个物品时取得的最大价值。为了确定装入背包的具体物品，从$V(n,C)$的值向前推，如果$V(n,C)>V(n-1,C)$，表明第n个物品被装入背包，前$n-1$个物品被装入容量为$C-w_n$的背包中；否则，第n个物品没有被装入背包，前$n-1$个物品被装入容量为C的背包中。依此类推，直到确定第1个物品是否被装入背包中为止。由此，得到如下函数：

$$x_i=\begin{cases}0 & V(i,j)=V(i-1,j)\\ 1, \quad j=j-w_i & V(i,j)>V(i-1,j)\end{cases}$$

7.7 0–1背包问题

例子

eg： number = 4， capacity = 8

i	1	2	3	4
s(大小)	2	3	4	5
v(价值)	3	4	5	6

7.7 0–1背包问题

自底向上的求解递归式：填表

i/j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0								
2	0								
3	0								
4	0								

i	1	2	3	4
s(大小)	2	3	4	5
v(价值)	3	4	5	6

number = 4
capacity = 8

7.7 0–1背包问题

自底向上的求解递归式：填表

i/j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3
2	0	0	3	4	4	7	7	7	7
3	0	0	3	4	5	7	8	9	9
4	0	0	3	4	5	7	8	9	10

i	1	2	3	4
s(大小)	2	3	4	5
v(价值)	3	4	5	6

number = 4
capacity = 8

7.7 0–1背包问题

代码实现

i/j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3
2	0	0	3	4	4	7	7	7	7
3	0	0	3	4	5	7	8	9	9
4	0	0	3	4	5	7	8	9	10

V(i,j)=V(i–1,j)时，说明没有选择第i 个商品，则回到V(i–1,j)；
V(i,j)=V(i–1,j–w(i))+v(i)时，说明装了第i个商品，该商品是最优解组成的一部分，随后我们得回到装该商品之前，即回到V(i–1,j–w(i))；一直遍历到i＝0结束为止，所有解的组成都会找到。

7.7 0–1背包问题

算法时间复杂度分析

时间代价$O(n \times S)$，空间代价$O(n \times S)$

注意：这个不是多项式时间，因为S是作为一个整数输入

设S的位数是$L = \log_2 S$，那么相当于时间代价和空间代价是$O(n2^L)$

练习：0–1背包问题的动态规划算法的时间复杂度是输入规模的多项式时间吗？

提纲

7.1 兔子的故事

7.2 动态规划法的设计思想

7.3 子问题重叠的例子：三角数塔问题

7.4 最大子段和（最大子数组）

7.5 适用动态规划法的场景

7.6 最长公共子序列

7.7 0–1背包问题

7.8矩阵乘法

矩阵乘法

设$A_1, A_2, \ldots, A_n$为矩阵序列，A_i为$P_{i-1} \times P_i$维矩阵，$i = 1,2,\ldots,n$，确定乘法顺序使得元素相乘的总次数最少

总的搜索空间大小为Catalan数，即$\frac{1}{n+1}\binom{2n}{n} = \Omega(4^n/n^{3/2})$

子问题定义

输入A=< A_0, A_1, …, A_n> , $A_{i..j}$ 表示乘积$A_iA_{i+1}…A_j$ 的结果，其最后一次相乘是$A_{i..j} = A_{i..k}\ A_{k+1..j}$

m[i,j] 表示得到$A_{i..j}$ 的最少的相乘次数，于是有如下等式

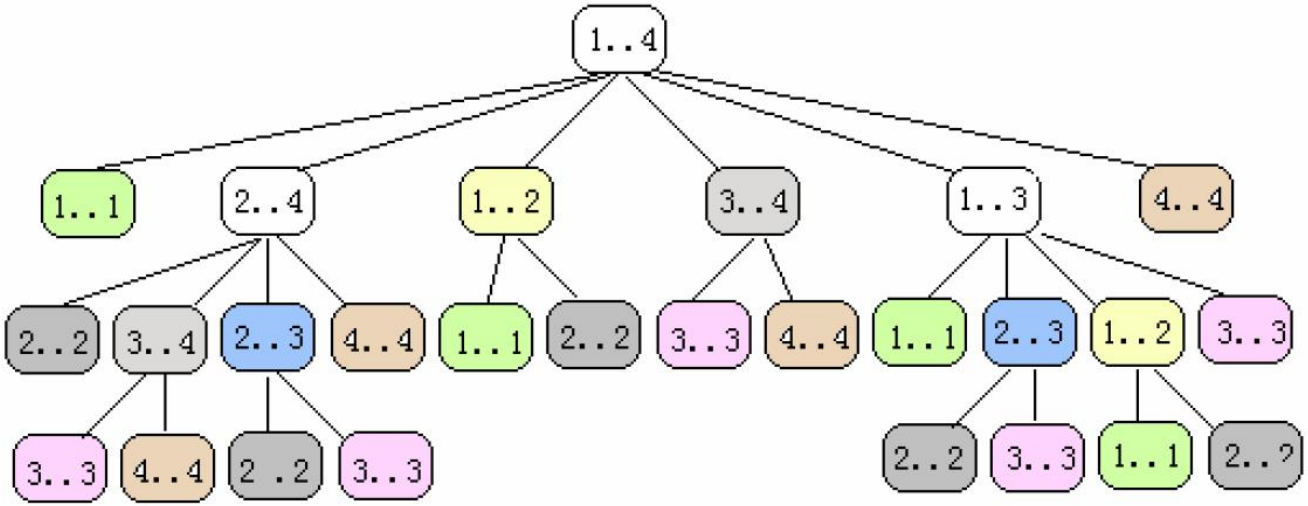
$$m[i,j] = \begin{cases} 0, & if\ \ i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + A_{i-1}A_kA_j\}, & if\ \ i < j \end{cases}$$

基本递归算法

```
RecurMatrixChain(P,i,j)
    m[i,j]=∞
    for k=i to j−1 do
        q= RecurMatrixChain(P,i,k)+ RecurMatrixChain(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]
```

时间复杂度$O(2^n)$

子问题重复程度高


1..4
1..1
2..4
1..2
3..4
1..3
4..4
2..2
3..4
2..3
4..4
1..1
2..2
3..3
4..4
1..1
2..3
1..2
3..3
3..3
4..4
2 .2
3..3
2..2
3..3
1..1
2..?

基于动态规划的矩阵乘法

MatrixChain(P,n)

for r=2 to n do // r为计算的矩阵链长度

for i =1 to n−r+1 do //n−r+1为最后r链的始位置

j =i+r−1 // 计算链i到j

$m[i,j] = m[i+1,j] + p_{i-1}*p_i*p_j$

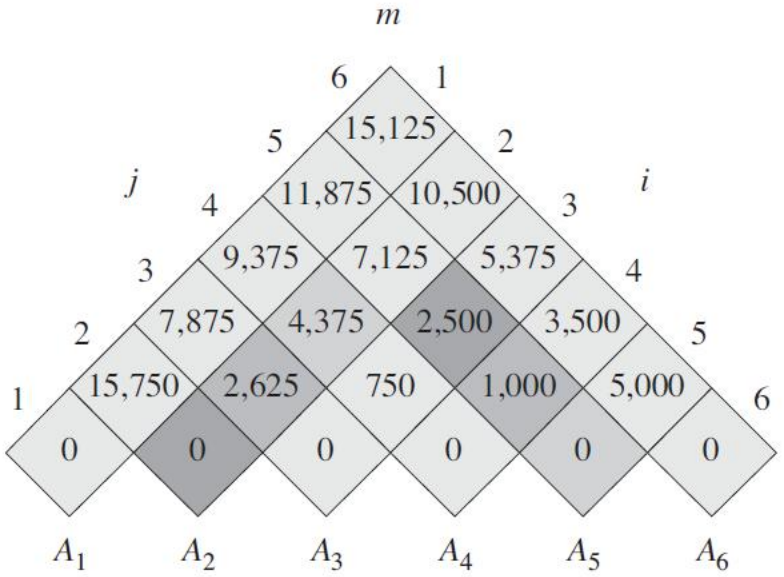
for k =i+1 to j−1 do

$t = m[i,k]+m[k+1,j]+ p_{i-1}*p_k*p_j$

if t<m[i,j] then m[i,j]=t

时间复杂度$O(n^3)$

示例



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