北京航空航天大学
BEIHANG UNIVERSITY

# 大学计算机基础
## （理科类）

## 第9讲 动态规划与贪心法

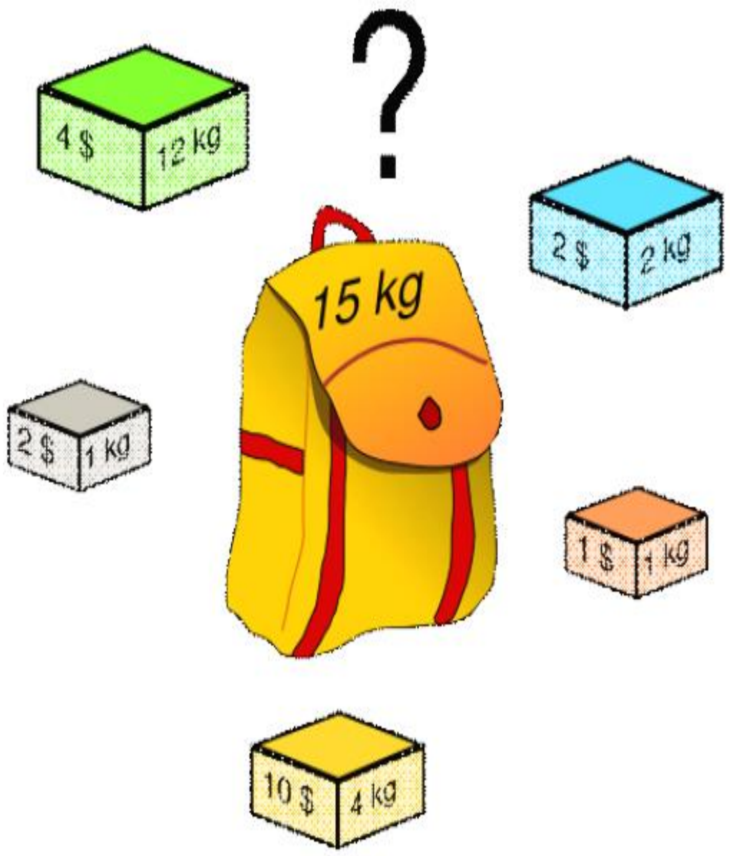
北京航空航天大学

# 目　录

# 9.1 动态规划

北京航空航天大学

# 0/1背包问题




**多阶段决策问题**

- 一个旅行者有一个**承重**最大为**C**公斤的背包，现在有n件物品，物品i的重量是$w_i$，其价值为$v_i$
- 旅行者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**是指选择装入背包的物品时，每种物品只能有两种选择："**不装入背包**"或"**装入背包**"（非此即彼），不能将物品装入背包多次且不能只装入部分物品

# 【讨论1】

- 你能想到怎么来装入吗?
  - 我们学过的算法，可以用什么算法?如何实现?
  - 还有什么方法?

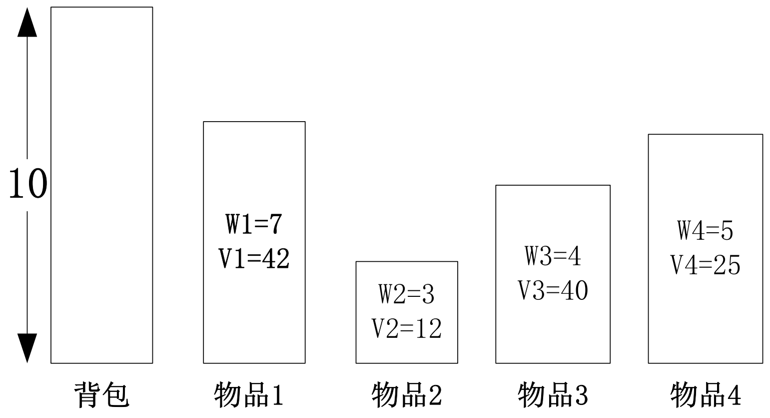
# 【例9.1】枚举法解背包问题

## 枚举法

- 考虑给定n个物品集合的所有子集，找出**所有可能的子集**（总重量不超过背包重量的子集），计算每个子集的**总重量**和**总价值**，然后在它们中找到**价值最大**的子集

**【例9.1】**有**4**个物品和一个承重为**10kg**的背包

- 4件物品各自的重量w=[7,3,4,5]
- 4件物品各自的价值v=[12,12,40,25]
- 如何装包，能获得最大价值?



# 思考：约束条件和目标函数是什么？

- **思考：约束条件和目标函数是什么?**

- **约束条件：放入物品的重量之和<=10kg**
- **目标函数：放入物品的总价值最大。**

# 问题分析

| 序号 | 子集 | 总重量 | 总价值 | 序号 | 子集 | 总重量 | 总价值 |
|---|---|---|---|---|---|---|---|
| 1 | 空集 | 0 | 0 | 9 | {2，3} | 7 | 52 |
| 2 | {1} | 7 | 42 | 10 | {2，4} | 8 | 37 |
| 3 | {2} | 3 | 12 | 11 | {3，4} | 9 | 65 |
| 4 | {3} | 4 | 40 | 12 | {1,2,3} | 14 | 不可行 |
| 5 | {4} | 5 | 25 | 13 | {1,2,4} | 15 | 不可行 |
| 6 | {1，2} | 10 | 54 | 14 | {1,3,4} | 16 | 不可行 |
| 7 | {1，3} | 11 | 不可行 | 15 | {2,3,4} | 12 | 不可行 |
| 8 | {1，4} | 12 | 不可行 | 16 | {1,2,3,4} | 19 | 不可行 |

- **枚举**：一共有**$2^4$=16**种组合
- **列出其所有可能的解**：所有可能的子集有**10**个
- **找出符合条件的解**：价值最大的子集为**{3,4}**，其总价值为**65**

# 求解思路

- **关键：如何找出物品装包的所有可能组合？用什么数据结构来存储?**
  - 先将4种物品的信息存入一个**嵌套列表goods**，每个子列表分别存储物品**序号、重量、价值**

    goods=[[1,7,12],[2,3,12],[3,4,40],[4,5,25]]

    还可以用什么数据结构?
  - 因为有4种物品，每种物品有选择和不选择两种可能，可以用1和0来标识，则4种物品的选择可以用**4位二进制数**来表示。二进制数某一位为'**1**'，表示**选择**对应的物品装包；为'**0**'则**不选择**
  - 4位二进制数的16种组合，对应16种装包情况，存入一个**嵌套列表full_set**，每个子列表是装包的一个物品组合子集

# 程序框架

## 1、构建二进制序列flag，标识$2^n$种组合

flag=[0000, 0001, 0010, 0011, ……1111]

## 2、枚举物品装包的所有可能组合

for i=0~($2^{n-1}$)：  16种组合

for j=0~(n-1)：  4种物品

物品1 物品2 物品3 物品4

goods=[[1,7,12],[2,3,12],[3,4,40],[4,5,25]]

if flag[i][j]='1"，则将goods[j]存入一个列表**sub_set**

子集sub_set存入一个嵌套列表**full_set**

**full_set:**

flag[3]=**0011**

[[], [[4, 5, 25]], [[3, 4, 40]], **[[3, 4, 40], [4, 5, 25]]**, [[2, 3, 12]], [[2, 3, 12], [4, 5, 25]], [[2, 3, 12], [3, 4, 40]], [[2, 3, 12], [3, 4, 40], [4, 5, 25]], [[1, 7, 12], ……]

# 程序框架（续）

## 3、选择最优装包组合

**遍历full_set**：

**遍历每个子集**，计算每个子集包含物品的**重量之和**、**价值之和**

若某个子集的重量之和<=m 且价值之和>best_v：

更新最大价值best_v

更新最优组合best_set

## 4、输出最优装包方案和最大价值

# 【例9.1】Python程序

**例9.1-背包问题【枚举法】.py**

```
#1、原始数据
m=10                    #背包总承重
w=[7,3,4,5]             #4件物品各自的重量
p=[12,12,40,25]         #4件物品各自的价值

#预处理
n=len(w)                #物品件数
goods=[[1,7,12],[2,3,12],[3,4,40],[4,5,25]] #4种物品，每个子列表分别存储物品序号、重量、价值
print('goods=',goods)

#2、构建二进制序列，标识2^n种组合
flag=[]
for i in range(2**n):
    s=bin(i)[2:]        #将十进制数i转换为二进制数。从第2位开始截取，去掉“0b”
    s='0'*(n-len(s))+s  #若s的长度<n，则在前面补齐0，使其长度变为n
    flag.append(s)
print('flag=',flag)
```

生成二进制序列的方法，**巧妙**！

# 【例9.1】Python程序（续1）

枚举

```
#3、枚举物品装包的所有可能组合
full_set=[]                              #所有可能的子集
for i in range(2**n):                    #遍历物品所有组合
    sub_set = []                         #存储某个子集
    for j in range(n):                   #遍历4种物品
        if flag[i][j] == '1':            #如果flag第i个元素的第j位为'1'
            sub_set.append(goods[j])     #则将goods[j]添加到子集中
    full_set.append(sub_set)

print('full_set=',full_set)
```

一个子集中可能包含几个列表

# 【例9.1】Python程序（续2）

**#4、选择最优装包组合**

```
best_v = 0                          #最大价值，初值为0
best_set = None                     #最优组合
for items in full_set:              #遍历所有子集
    total_w = 0                     #每个子集的重量之和
    total_v = 0                     #每个子集的价值之和
    for item in items:              #遍历每个子集，计算每个子集包含物品的重量之和、价值之和
        total_w += item[1]
        total_v += item[2]
        if total_w > m:             #若某个子集重量之和大于背包总承重
            break                   #则不必继续计算，终止本层循环
    if total_w <= m and total_v > best_v:#若某个子集的重量之和<=m 且价值之和>best_v
        best_v = total_v            #则更新best_v
        best_set = items            #则更新best_set
```

优化程序，提高效率

**#5、输出最优装包方案和最大价值**

```
print('装入物品：',best_set)
print('装入物品的总价值为：',best_v)
```

# 【例9.1】程序运行结果

```
goods= [[1, 7, 12], [2, 3, 12], [3, 4, 40], [4, 5, 25]]
flag= ['0000', '0001', '0010', '0011', '0100', '0101', '0110', '0111', '1000',
'1001', '1010', '1011', '1100', '1101', '1110', '1111']
full_set= [[], [[4, 5, 25]], [[3, 4, 40]], [[3, 4, 40], [4, 5, 25]], [[2, 3, 12
]], [[2, 3, 12], [4, 5, 25]], [[2, 3, 12], [3, 4, 40]], [[2, 3, 12], [3, 4, 40]
, [4, 5, 25]], [[1, 7, 12]], [[1, 7, 12], [4, 5, 25]], [[1, 7, 12], [3, 4, 40]]
, [[1, 7, 12], [3, 4, 40], [4, 5, 25]], [[1, 7, 12], [2, 3, 12]], [[1, 7, 12],
[2, 3, 12], [4, 5, 25]], [[1, 7, 12], [2, 3, 12], [3, 4, 40]], [[1, 7, 12], [2,
3, 12], [3, 4, 40], [4, 5, 25]]]
装入物品： [[3, 4, 40], [4, 5, 25]]
装入物品的总价值为： 65
```

# 多阶段决策问题

- 如果一类**活动过程**，可以分为若干个互相联系的阶段，在**每一个阶段**都需要作出**决策**（采取措施），从而使整个过程达到最好的活动效果。一个阶段的决策确定以后，常常影响到下一个阶段的决策。当各个阶段决策都确定后，就完全确定了一个过程的活动路线。则称这种类型的活动过程为**多阶段决策过程**，这类问题称为**多阶段决策问题**
  - **例如**：**最短路线、库存管理、资源分配、设备更新、排序、装载**等问题

# 动态规划

- 20世纪50年代初，美国数学家**R.E.Bellman**等人提出了著名的**最优化原理**（principle of optimality），把多阶段过程转化为一系列单阶段问题，利用各阶段之间的关系，逐个求解，同时创立了解决多阶段决策过程的最优化问题的新方法——**动态规划**（Dynamic Programming，DP）
- **动态规划**是**运筹学**的一个分支，是求解**决策过程**（decision process）**最优化**的数学方法
  - 是解最优化问题的一种途径、一种方法

# 动态规划的基本思想

## 动态的基本思想

- 将待求解的问题分解成若干**相互联系**的子问题，先求解子问题，然后根据子问题之间的关系来得到原问题的解
- 在计算过程中**每个子问题只求解一次**，将其结果保存在一张表（**最优决策表**）中，以便在需要时直接使用
- 避免每次遇到某个子问题时的重复计算，**大大提高了**算法的**运行效率**

## 动态的含义

- 在系统发展的不同**时刻**（或阶段），根据系统所处的**状态**，做出决策

# 动态规划问题中的术语：阶段

## 阶段

- 把所给求解问题的过程恰当地分成若干个相互联系的**阶段**，以便于求解。过程不同，阶段数就可能不同
- **描述阶段的变量**称为**阶段变量**。多数情况下，阶段变量是**离散**的，一般用**k**表示
- 若过程可以在任何时刻作出决策，且在任意两个不同的时刻之间允许有无穷多个决策时，阶段变量就是**连续**的

✓ 例如，**背包问题**考虑装入第1件、前2件、前3件、前n件物品，将整个过程划分为n个阶段。阶段变量：**物品件数i**

# 动态规划问题中的术语：状态

## 状态

- **状态表示每个阶段开始面临的自然状况或客观条件**，它不以人们的主观意志为转移，也称为**不可控因素**
- 过程的状态通常可以用一个或一组数来描述，**描述状态的变量**称为**状态变量**
- 一般，状态是**离散**的，但有时为了方便也将状态取成**连续**的

> ✓ 例如，背包问题的**状态**——包的**容量**：
>
> 0、1、2……、C公斤

# 动态规划问题中的术语：决策

## 决策

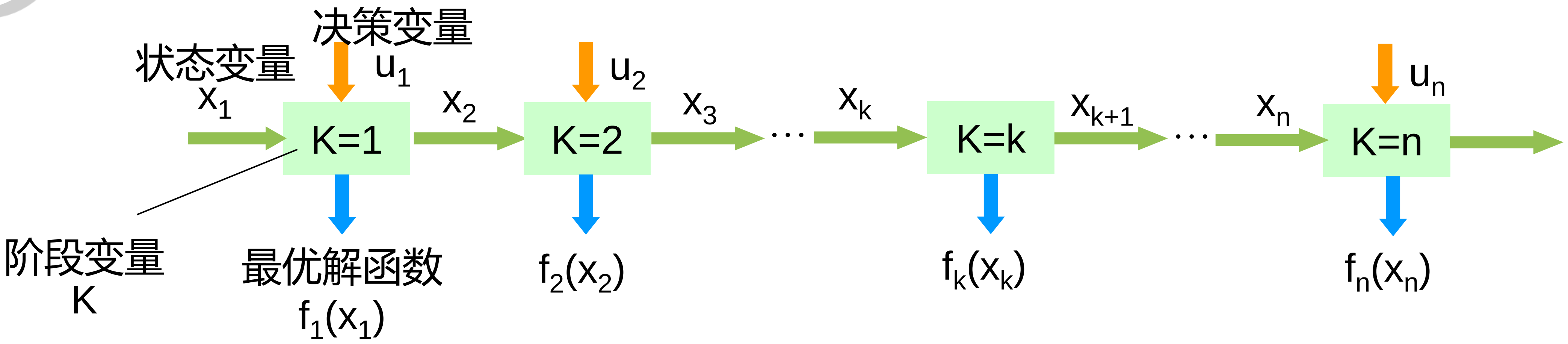
- **一个阶段的状态给定以后，从该状态演变到下一阶段某个状态的一种选择（行动）**称为**决策**。在最优控制中，也称为**控制**
- 在许多问题中，决策可以自然而然地表示为一个数或一组数。不同的决策对应着不同的数值
- **描述决策的变量**称**决策变量**

> - **思考：背包问题的决策是什么？是什么因素决定一件物品是否装入？**
> - $v_i$表示第i个物品的价值，作为**决策变量**

# 动态规划问题中的术语：状态转移方程

- **状态转移方程**
  - <u>**从k阶段到k+1阶段的状态转移规律**</u>，称为**状态转移方程**
  - 用**x(k+1)=Tk(x(k),u(k))**表示
  - 给定k阶段状态变量x(k)的值后，如果这一阶段的**决策变量u(k)**一经确定，第k+1阶段的状态变量x(k+1)也就完全确定，即x(k+1)的值随x(k)和第k阶段的决策u(k)的值变化而变化

# 动态规划的三个要素




- **要素：阶段，状态，状态转移方程**
  - **（1）阶段**： 一个问题可被划分为若干个阶段求解
  - **（2）状态**：每个阶段开始面临的自然状况或客观条件
  - **（3）状态转移方程**：$x_{k+1} = T_k(x_k, u_k)$

    形式化描述一个状态到另一个状态的演变过程

## 数学模型建立

划分阶段 → 确定状态 → 定义决策 → 创建状态转移方程

- 首先，根据问题时、空特征，将其划分为若干个有序**阶段**（**阶段变量k**）
- 其次，用不同**状态**表示问题发展到各个阶段时所处于的客观情况（**状态变量$x_k$**）
- 再次，定义从一个阶段某状态演变为下一阶段某状态的**决策**选择（**决策变量$u_k$**）
- 最后，创建**状态转移方程**：$x_{k+1} = T_k(x_k, u_k)$

# 动态规划算法的设计技巧

（1）状态转移方程具有**递归**特征（调用自身）。利用**递归思想**分析问题，但实现时常用**多重循环**方式以提高效率

（2）状态转移方程具有**最优解**特征（min、max），会在递归表达式中出现min或max函数

**背包问题：**

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

（3）**编程**：应用递推求解每一阶段**最优值**（即最大值、最小值等），保存计算过的子问题结果；**构造最优解**

# 动态规划与分治法的区别

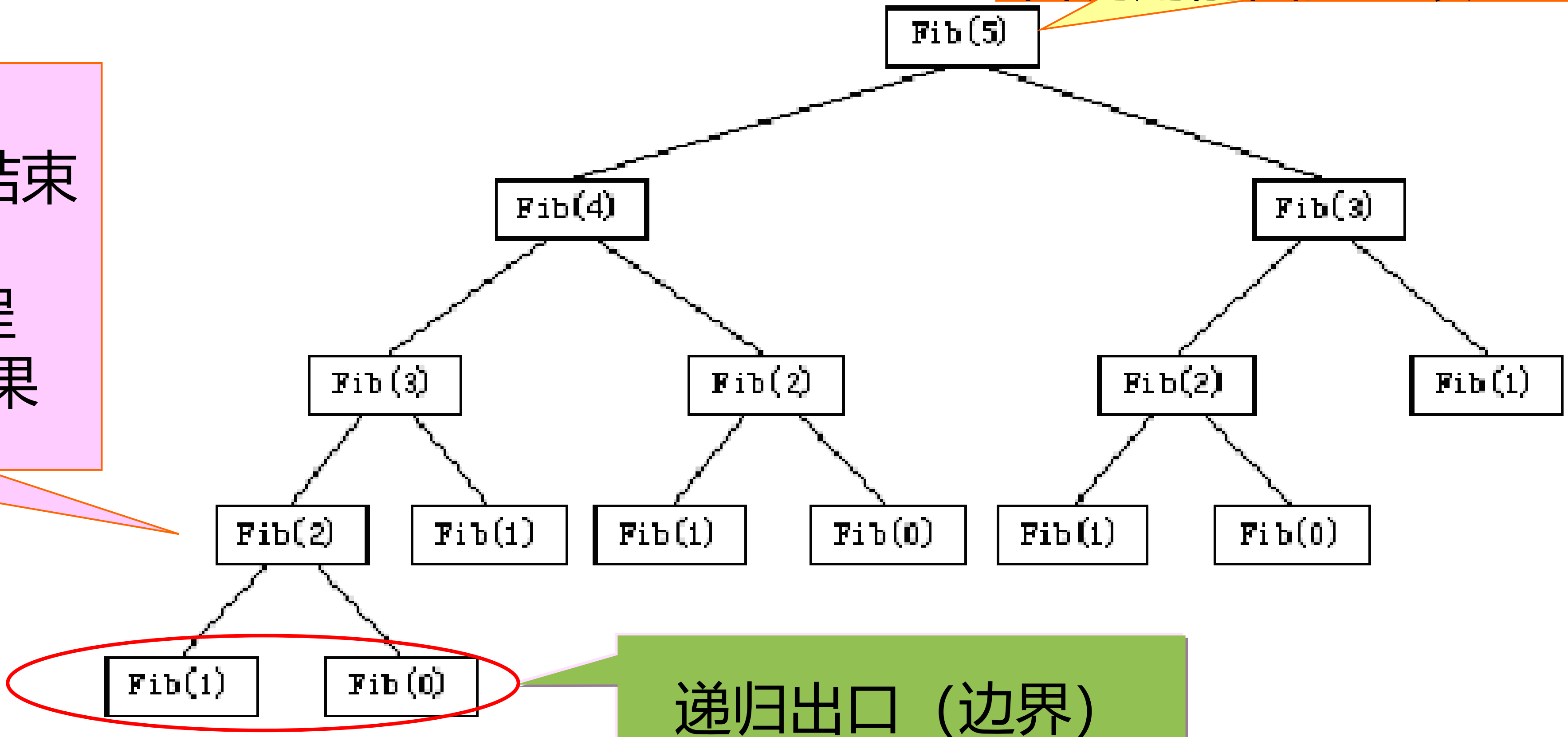
- **动态规划**与**分治法**十分相似，都是将复杂问题分解为相似子问题，并通过组合子问题的解来求解
- **分治法**将问题分解成若干个**相互独立**的子问题
  - 常采用**递归算法**求解
  - **缺点**：有些子问题会被重复计算多次，运行**效率低下**
- **动态规划**经分解得到的子问题往往**不是互相独立**的，它们可能共享更小的子问题，即**重叠子问题**（overlapping subproblem）
  - 常采用**多重循环**求解
  - **优点**：与其它方法相比，能够**提高时间效率**和**空间效率**
  - **缺点**：算法设计过程较复杂

# 斐波那契递归执行过程

- 斐波那契递归执行过程

（1）从最后一项开始自调用前面的几项

（2）到达递归出口（Fib(1)、Fib(0)）才结束自调用过程
（3）按最后调用的过程（Fib(2)）最先返回结果值的次序返回值




递归出口（边界）

- **需要15个临时存储单元。Fib(2)被计算了3次，Fib(3)被计算了两次**

# 动态规划的适用场合

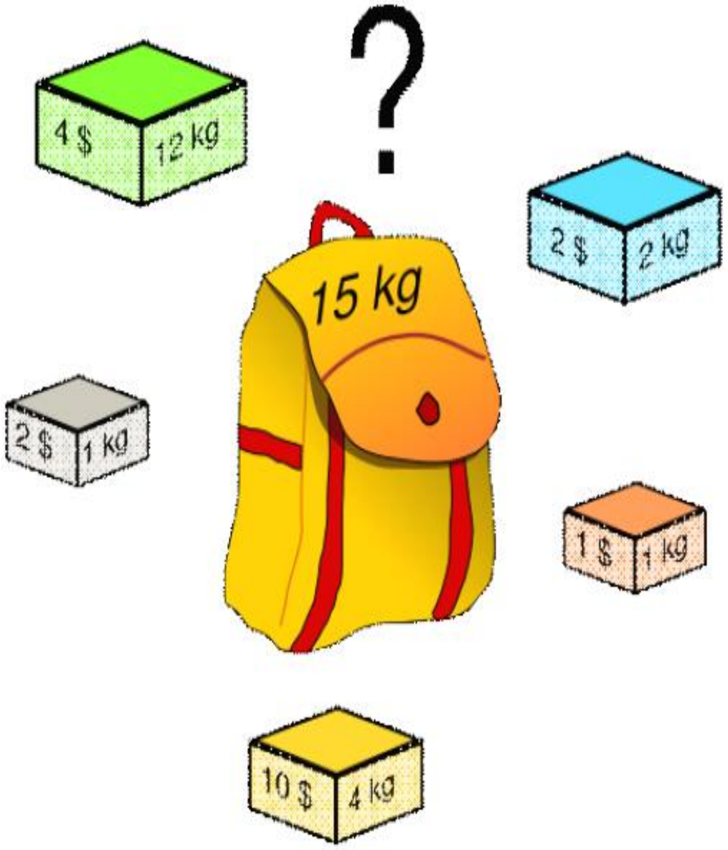
- **适用范围**：最优化问题中的**多阶段决策**问题
  - ✓ 解决**最值**（最优，最大，最小，最长……）**问题**
- **适于用动态规划法求解的问题具有以下特点**

**（1）重叠子问题**：子问题间不相互独立，包含**公共**子问题

**（2）最优子结构**：问题的最优解包含其**子问题的最优解**，一个最优化策略的子策略总是最优的

**（3）无后效性**：某阶段状态一旦确定，就不受这个状态以后决策的影响。即某状态以后的过程不会影响以前的状态，只与当前状态有关

# 0/1背包问题




- 一个旅行者有一个承重最大为C公斤的背包，现在有n件物品，物品i的重量是$w_i$，其价值为$v_i$
- 旅行者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?

**如何用动态规划求解?**

# 问题的抽象与建模：抽象

## ■ 第一步：问题的抽象

- 属于**组合优化**问题，是运筹学（operations research）中的一个重要分支，其目标是寻找离散事件的**最优编排**、**分组**、**次序**或**筛选**
- **组合优化问题**在现实生活中有着广泛的应用：**资源分配**、**投资决策**、**装载设计**、**公交车调度**

- **典型的组合优化问题**
  - ✓ 0-1背包问题，旅行商问题，有约束的机器调度问题，装箱问题，车间作业调度问题，图的顶点着色问题

# 问题的抽象与建模：建模

## 第二步：模型的建立

- **序列$X$**，对其中的任意一个变量$x_i$ $(i=1,2,\dots n)$进行判断，当 $x_i$=1时，表示物品$i$被**装入**背包（表示被选中）；当$x_i$=0时，表示物品$i$**没有被装入**背包（表示未被选中）
- **序列$V$**为**价值**序列， $v_i$表示对应$x_i$的价值
- **序列$W$**为**重量**序列， $w_i$表示对应$x_i$的重量

# 问题的抽象与建模：建模（续）

- **模型的建立（同时存在约束条件，背包的容量）**
  - 在0/1背包问题中，物品$i$或者被装入背包，或者不被装入背包，设$x_i$表示物品$i$装入背包的情况，物品$i$的重量是$w_i$，其价值为$v_i$。
  - 有如下**约束条件**和**目标函数**

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

约束条件 （1）

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

（2） 目标函数

**问题归结为寻找一个满足约束条件式（1），并使目标函数式（2）达到最大的解向量$X=(x_1, x_2, \ldots, x_n)$**

# 组合优化问题的求解方法

- 最直接的方法是**枚举法**
  - 但当问题的计算量超出计算机在有效时间内的计算能力时，问题的求解会变得困难
- 通常采用**动态规划**求解，或者**回溯算法**、**贪心法**
  - **贪心法**：在求解问题时，总是做出在当前看来是最好的选择（局部最优解）

# 【例9.2】 0/1背包问题

**【例9.2】** 0/1背包问题

- 一个背包的承重量为10kg（C=10），有5（n=5）件物品，其重量（单位为kg）分别是{2, 2, 6, 5, 4}，价值（单位为$）分别为{6, 3, 5, 4, 6}。
- 要求挑选几件物品放入背包，不能超重，装入背包的物品的价值最大且不能分割物品。
- 试采用**动态规划**求解**需放入哪几件物品，取得的最大价值**为多少?

**动态规划求解0/1背包问题：**

- **Step1：确定阶段变量、状态变量和决策变量**
  - 使用$i$表示**前$i$件**物品，作为**阶段变量**（i = 1,2,3,4,5）
  - 使用$j$表示背包的承重（$0 \le j \le 10$），作为**状态变量**
  - $v_i$表示第i个物品的价值，作为**决策变量**

# Step2：确定状态转移方程

## Step2：确定状态转移方程（动态规划函数）

- 0/1背包问题可以看作是决策一个序列$(x_1, x_2, \ldots, x_n)$，对任一变量$x_i$的决策是决定$x_i$=1还是$x_i$=0
- 在对$x_{i-1}$决策后，已确定了$(x_1, \ldots, x_{i-1})$，在决策$x_i$时，会有以下两种情况：
  - ✓（1）背包容量不足以装入物品$i$，则 $x_i$ =0，背包的价值不增加
  - ✓（2）背包的容量可以装下物品$i$，对于物品$i$有两种决策选择，**装入**（则$x_i$=1）或**不装入**（则$x_i$=0）

# 状态转移方程（递归的定义）

令$V(i, j)$表示在前$i(1 \le i \le n)$个物品中能够装入承重为$j$（$1 \le j \le C$）的背包中的物品的**最大价值**，则可以得到：

**状态转移方程**

$$V(i, 0) = V(0, j) = 0 \qquad (3)$$

$$V(i, j) = \begin{cases} V(i-1, j) & j < w_i \\ \max\{V(i-1, j),\ V(i-1, j-w_i) + v_i\} & j \ge w_i \end{cases} \qquad (4)$$

- 式（3）表明：把前面$i$个物品装入**承重为0**的背包或把**0个物品**装入承重为$j$的背包，得到的**价值均为0**

# 状态转移方程—含义

$$V(i,j)=\begin{cases}V(i-1,j) & j<w_i\\ \max\{V(i-1,j),\ V(i-1,j-w_i)+v_i\} & j\ge w_i\end{cases}\quad (4)$$

- 式（4）的第一个式子表明：如果**背包当前的容量*j*小于第*i*个物品的重量**，则物品*i*不能装入背包
- 则装入前*i*个物品得到的最大价值和装入前*i*-1个物品得到的最大价值是相同的

# 状态转移方程—含义（续）

$$V(i,j)=\begin{cases}V(i-1,j) & j<w_i\\ \max\{V(i-1,j),\ \ V(i-1,j-w_i)+v_i\} & j\ge w_i\end{cases}\qquad（4）$$

- 式（4）的第二个式子表明，如果**背包当前的容量$j$大于或等于第$i$个物品的重量**，该物品**有可能**被装入。存在以下两种情况：

① 如果第$i$个物品没有装入背包，则背包中物品的价值就等于把前$i$-1个物品装入容量为$j$的背包中所取得的价值

② 如果把第$i$个物品装入背包，则背包中物品的价值等于把**前$i$-1**个物品装入容量为$j$-$w_i$的背包中的价值，加上**第$i$个**物品的价值$v_i$

**取二者中价值较大者作为把前$i$个物品装入容量为$j$的背包中的最优解**

# 最优决策表

- **最优决策表：直观描述动态规划的求解过程**
  - 一个**二维表**，**行**表示决策的**阶段**，**列**表示问题**状态**
  - 每个**单元格**中数据：在某个阶段某个状态下的**最优值**（如最短路径，最长公共子序列，最大价值等）
  - 根据递推关系，从第0行第0列开始，以**行**或者**列**优先的顺序，依次填写表格
  - 根据整个表格的数据，通过简单的取舍或者运算，**倒推**求得问题的**最优解**

# Step3：填写最优决策表

- **Step3：填写最优决策表（(n+1)×(C+1)）**
  - V[i][j]表示把**前i个物品**装入**容量为j**的背包中获得的最大价值
  - **状态转移方程**如下

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

- **行**表示决策的**阶段**，第0行i=0，不选择任何一件物品，第1行i=1，表示可以考虑装入第1件物品；第2行i=2，表示可以考虑继续装入第2件物品
- **列**表示问题**状态**（不同的承重），**每个格**需要填写的数据为在某个阶段某个状态下的**最优值V**

# 填写最优决策表

## 填写最优决策表

5件物品，重量分别是{2, 2, 6, 5, 4}，价值分别为{6, 3, 5, 4, 6}

| | i \ j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $w_1=2, v_1=6$ | 1 | 0 | 0 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| $w_2=2, v_2=3$ | 2 | 0 | 0 | 6 | 6 | 9 | 9 | 9 | 9 | 9 | 9 | 9 |
| $w_3=6, v_3=5$ | 3 | 0 | 0 | 6 | 6 | 9 | 9 | 9 | 9 | 11 | 11 | 14 |
| $w_4=5, v_4=4$ | 4 | 0 | 0 | 6 | 6 | 9 | 9 | 9 | 10 | 11 | 13 | 14 |
| $w_5=4, v_5=6$ | 5 | 0 | 0 | 6 | 6 | 9 | 9 | 12 | 12 | 15 | 15 | 15 |

**行：决策阶段**
**列：背包承重**

**装入第1、3件物品**

**装入第1、2、4件物品**

- ✓ 第0行没有选择任何一件物品，故每个格中价值为0
- ✓ 第0列表示背包当前承重为0，则每个格中价值也为0
- ✓ 第2行表示**只装入前2件**物品时，背包在不同承重下分别能够得到的最大价值（当承重为2、3时，只能装入第1件物品，故背包价值为6；当承重为4、5、……、10时，可以装入前2件物品，故背包价值为9）

# 【举手发言】

| | i \ j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $w_1=2, v_1=6$ | 1 | 0 | 0 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| $w_2=2, v_2=3$ | 2 | 0 | 0 | 6 | 6 | 9 | 9 | 9 | 9 | 9 | 9 | 9 |
| $w_3=6, v_3=5$ | 3 | 0 | 0 | 6 | 6 | 9 | 9 | 9 | 9 | 11 | | |
| $w_4=5, v_4=4$ | 4 | 0 | 0 | | | | | | | | | |
| $w_5=4, v_5=6$ | 5 | 0 | 0 | | | | | | | | | |

装入第1、3件物品

- 当i=3、承重j=8时，为什么装入的是物品1和3呢?

## Step4：求最优解

- 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
- 若$V(i,j)=V(i-1,j)$，说明第$i$件物品未被装入背包，$x_i=0$
- 每次根据$V(i,j)>V(i-1,j)$进行判断，若成立，则第$i$件物品被装入背包，$x_i=1$；同时背包的承重$j$减掉该物品的重量，即$j=j-w_i$；
- 再往前推，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：

$$x_i=\begin{cases}0 & V(i,j)=V(i-1,j)\\ 1,\quad j=j-w_i & V(i,j)>V(i-1,j)\end{cases} \qquad (5)$$

- **最优解——$X=(x_1, x_2, x_5)$**

# 根据最优决策表求最优解

| | i \ j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $w_1=2, v_1=6$ | 1 | 0 | 0 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| $w_2=2, v_2=3$ | 2 | 0 | 0 | 6 | 6 | 9 | 9 | 9 | 9 | 9 | 9 | 9 |
| $w_3=6, v_3=5$ | 3 | 0 | 0 | 6 | 6 | 9 | 9 | 9 | 9 | 11 | 11 | 14 |
| $w_4=5, v_4=4$ | 4 | 0 | 0 | 6 | 6 | 9 | 9 | 9 | 10 | 11 | 13 | 14 |
| $w_5=4, v_5=6$ | 5 | 0 | 0 | 6 | 6 | 9 | 9 | 12 | 12 | 15 | 15 | 15 |

**最优解**

$x_1=1$

$x_2=1$

$x_3=0$

$x_4=0$

$x_5=1$

$V(n,C)=15$

**从$V(n,C)$往回推**

- ✓ $V(5,10)>V(4,10)$，说明第5个物品被装入背包，$x_5=1$；前4个物品被装入容量为$C-w_5$（=10-4=6）的背包中
- ✓ $V(4,6)=V(3,6)=9$，说明第4个物品未被装入背包，$x_4=0$
- ✓ $V(3,6)= V(2,6)=9$， 第3个物品未被装入背包，$x_3=0$
- ✓ $V(2,6)=9> V(1,6)=6$，说明第2个物品被装入背包，$x_2=1$，则背包还剩容量为6-w2（=6-2=4）；$V(1,4)=6> V(0,4)=0$，说明第1个物品被装入背包，$x_1=1$

# 【例9.2】程序设计

- 首先定义一个用动态规划解背包问题的函数

def zeroOneknapsack(w,p,m,x):

#***w***为5件物品的重量，***p***为5件物品的价值，***m***为背包总承重，***x***列表用于存储最优解（即哪几件物品被装包）

- 该函数包括两部分

**（1）采用两重for循环，计算Load[i][j]，即V(i,j)**

**（2）递推装入背包的物品是什么**

# 【例9.2】程序设计（续1）

## （1）采用两重for循环，计算Load[i][j]，即V(i,j)

```
v=0    #初始化总价值
#初始化Load[i][j]前i个物品中装入承重为j的背包的物品总价值
Load=[[0 for col in range(m+1)] for raw in range(n+1)]
#两重for循环计算Load[i][j]
for i in range(1,n+1):
    for j in range(1,m+1):
        #首先将装入第i个物品的价值赋值为装入第i-1个物品的价值
        Load[i][j]=Load[i-1][j]
        #如果背包的容量比第i个物品的质量大（可以装入）
        #并且装入第i个物品后总价值会上升，则将第i个物品装入
        if(j>=w[i])and(Load[i-1][j-w[i]]+p[i]>Load[i-1][j]):
            Load[i][j]=Load[i-1][j-w[i]]+p[i]
```

# 【例9.2】程序设计（续2）

## （2）递推装入背包的物品是什么

根据下式最优解的公式来描述

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

```
j=m                                #j初值为总承重
for i in range(n,0,-1):            #i=n,n-1,n-2,……,1
    if Load[i][j]>Load[i-1][j]:    #表明第i个物品可以被装入承重为j的背包中
        x[i]="load"
        j=j-w[i]                   #则从j中减掉装入第i个物品的重量
v=Load[n][m]                       #在容量为m的背包中装入n个物品时取得的最大价值
```

# 【例9.2】Python程序

**例9.2- knapsack【动态规划】.py**

```
#1、定义用动态规划法解背包问题的函数
def zeroOneknapsack(w,p,m,x):
    #w、p为5件物品的重量、价值，m为背包总承重，x列表存储最优解
    v=0                              #最大总价值
    #初始化Load[i][j]，把前i个物品装入容量为j的背包中获得的最大价值
    Load=[[0 for col in range(0,m+1)] for row in range(0,n+1)]

    #（1）计算Load[i][j]
    for i in range(1,n+1):          #i：阶段变量，前j件物品
        for j in range(1,m+1):      #j：状态变量，背包的各种承重
            if(j>=w[i])and (Load[i-1][j-w[i]]+p[i]>Load[i-1][j]):
            #如果背包的容量j大于等于第i个物品的重量，且把第i个物品装包价值更大
                Load[i][j]=Load[i-1][j-w[i]]+p[i] #则将第i个物品装入背包
            else:
                Load[i][j]=Load[i-1][j] #否则不装入
            print ("Load[i][j]=", Load[i][j])
```

# 【例9.2】Python程序（续1）

```
# （2）递推装入背包的物品
j=m                                 #j初值为总承重
for i in range(n,0,-1):                 #i在n~1之间，不包括0。倒推
    #如果V(i,j)>V(i-1,j)，表明第i个物品被装入承重为j的背包中
    if Load[i][j]>Load[i-1][j]:
        x[i]="load"
        j=j-w[i]                        #从j中减掉装入第i个物品的重量
v=Load[n][m]                            #在容量为m的背包中装入n个物品时取得的最大价值
return v                                #返回最大总价值
```

# 【例9.2】Python程序（续2）

```
#2、原始数据
m=10                                #背包总承重
w=[0,2,2,6,5,4]                     #5件物品各自的重量
p=[0,6,3,5,4,6]                     #5件物品各自的价值
n=len(w)-1                          #物品件数
x=["unload"]*(n+1)                  #x列表用于存储最优解（即哪几件物品被装包）
#x=["unload" for k in range(n+1)]   #上面的语句也可以这样写

#3、调用函数解背包问题
totalV=zeroOneknapsack(w,p,m,x)
print ("装入背包中物品的总价值：",totalV)
print ("最优解x[1:n+1] is ", x[1:n+1])   #最优解为x[1]~x[n]
```

```
装入背包中物品的总价值： 15
最优解x[1:n+1] is  ['load', 'load', 'unload', 'unload', 'load']
```

# 9.2 贪心法

北京航空航天大学

# 【例9.3】找零问题

**【例9.3】找零问题。**

- 超市的自动柜员机（POS机）要找给顾客钞票张数最少的现金。假设有面值为1元、3元和5元的硬币若干枚，如何用最少的硬币凑够11元找零？试采用不同的算法进行问题求解。

- **解法**
  - **思路**：先选择最大面值，即**5元**；则问题变为凑6元找零，使硬币数最少；再选择最大面值**5元**；则问题变为凑1元找零，使硬币数最少；再选择**1元**
  - **每次都做出当前最好的选择——贪心法**

- **想一想，还有什么解法？**

# 贪心法的基本思想

- **贪心法（Greedy Algorithm）的基本思想**
  - 将待求解的问题分解成若干个子问题进行**分步求解**，且**每一步**总是做出**当前最好**的选择**（局部最优解）**，以期得到问题最优解。
  - 贪心算法对每个子问题得到其局部最优解，再将各个局部最优解整合成问题的解。
  - **"眼下能拿到的就先拿到"的策略就是这个算法名称的由来**

# 贪心法的特点和适用场合

- **优点**：思维复杂度低，开发速度快, 代码量小，可以相对**快速**地获得一个**可行解**
- **缺点**：对于大部分的优化问题都能产生最优解，但不一定能获得全局最优解，通常可以获得**近似最优解**
- **算法适用**：**最优化问题**

## 特点

**无后效性**：某阶段状态一旦确定，就不受这个状态以后决策的影响。某个状态以后的过程不会影响以前的状态，只与当前状态有关

## 【讨论2】

- 你知道贪心法还可以用于求解哪些问题?

# 贪心法不能保证一定得到最优解

- **例：找零钱问题**
  - **贪心法并不能保证任何情况下都能得到最优解。**
  - 如硬币面值改为1分、5分、11分；现在须找零15分。
    - ✓ 若按贪心算法：11+1+1+1+1
    - ✓ 而实际最优解为：5+5+5
- 贪心法总是做出在当前看来是最优的选择，并不是从整体上加以考虑，它所做出的选择只是在某种意义上的局部最优解。

- **贪心法的优劣与问题的特殊性密切相关。因此，其具有一定的局限性**

# 贪心法与动态规划的区别

## 贪心法与动态规划的区别

- **贪心法**的每一次操作都对结果产生直接影响，对每个子问题的解决方案都做出选择，**不能回退**
- **动态规划**会根据以前的选择结果对当前进行选择，**有回退功能**

## 例如：0/1背包问题

- **i=3**表示前3件物品可能被装入。**j=7**时，选择装入**物品1、2**，有V[i][j]=v1+v2=6+3=9
- **j=8**时，选择装入**物品1、3**，**物品2不装入**，有V[i][j]=v1+v3=6+5=11

# 贪心法求解问题的基本思路

## 1）建立数学模型来描述问题

**新所凑钱数 = 原所凑钱数%当前最大面值**

**原所凑钱数 = 新所凑钱数**

## 2）把求解的问题分成若干个子问题

- 先选择最大面值，即5元；则问题变为凑6元找零，使硬币数最少
- 再选择最大面值5元；则问题变为凑1元找零，使硬币数最少
- 再选择1元

# 贪心法求解问题的基本思路（续）

3）对每一子问题求解，得到子问题的**局部最优解**

**原所凑钱数/当前最大面值**　　　　#当前最大面值的个数

4）把子问题的局部最优解合成原问题的一个解

**各种面值各需多少个**

# 【例9.3】模型建立

## ■ 模型建立

- ◆ **Step1：问题抽象**：用数量最少的1、3、5相加，和为11
- ◆ **Step2：数学建模**——最优化方程组

$$\begin{cases} 1*x+3*y+5*z = \ 11 \\ \min(x+y+z) \end{cases}$$

**约束条件**
**目标函数**

# 【例9.3】贪心法的算法描述

- 【例9.3】贪心法的算法描述

局部最优解

```
输入所需凑钱数和找零硬币种类list_coin
对找零硬币种类从大到小排序
for  i in list_coin       //每次从找零硬币种类中找当前可选择的最大值i
    i的个数 ← change/i取整
    新所需凑硬币钱数 ← 原所需凑硬币钱数 % i
    if 新所需凑硬币钱数 == 0
        break              //终止循环
    else
        原所需凑硬币钱数 ←新所需凑硬币钱数
```

## 贪心法的设计技巧

（1）对贪心的可选对象进行**排序（从大到小）**

list_coin. sort(reverse = True)　　#倒序排列硬币面值列表

假如list_coin原始=[1,3,5]

（2）对贪心的求解目标进行**循环**

for i in list_coin:

change_dict[i] = int(change/i) #将change/i的商整数部分关联到change_dict字典的键i上，此即是求i的个数

newchange = change % i

# 贪心法的设计技巧（续）

**(3）在循环中内嵌条件判断**

```
if newchange == 0:              #若新所凑硬币钱数为0
    break                       #已凑够找零，则跳出循环
else:
    change = newchange          #否则，继续找零
```

# 【讨论3】

- **请问：以下哪种表述是错误的**？

  A：贪心法在短时间内能获得可行解，但不一定是全局最优解

  B：贪心法总能得到全部解

  C：枚举法总能得到全部解，但时间开销可能非常大

  D：动态规划可以获得全局最优解，时间开销比枚举法小

- **请选择一项**

# 【例9.3】程序

**例9.3-change_Greedy.py**

```
#解法二：贪心法（通用程序，可以由用户输入找零钱数和硬币种类）
#1、定义函数，利用贪心法求和为11的最优硬币组合，存入字典change_dict中
def change(coinValueList,change):
    #coinValueList为存储不同面值硬币的列表，change为需找给顾客的零钱
    if change == 0:                              #考虑边界条件，增强程序鲁棒性
        return 0
    elif change < 0:
        print('error')
        return 1
    else:
        list_coin = coinValueList
        list_coin.sort(reverse = True)     #倒序排列硬币面值列表
```

# 【例9.3】程序（续1）

```
for i in list_coin :  #遍历列表，从找零硬币种类中找当前可选择的最大值i
    print('硬币面值i为：',i)
    change_dict[i] = int(change/i) #创建字典change_dict，键i为最大面值；值为change/i（整数部分），此即是i的个数
    print ('最优硬币组合change_dict为：',change_dict)
    newchange = change % i        #新所需凑硬币钱数 ← 原所需凑硬币钱数 % i
    print ('新所需凑硬币钱数newchange为：',newchange)

    if newchange == 0:             #若新所凑硬币钱数为0，说明已凑够找零
        break
    else:
        change = newchange
```

最大面值的个数

# 【例9.3】程序（续2）

不写可以吗?

```
#2、检查所求得的解是否为可行解
def checkOk(change_res,oChange):
#change_res为存放问题求解结果的字典，键为某种硬币面值，值为该种硬币的个数；oChange
为需要找零的钱数
    csum = 0                                    #change_res中所有硬币钱数的和
    for key in change_res:                      #遍历结果字典
        csum = csum + key*change_res[key]       #将各种硬币面值与对应个数相乘，再相加
    if csum == oChange:                         #若所有硬币钱数的和等于需要找零的钱数
        return True
#3、定义函数，获得找零的硬币总个数
def coinNum(change_res):
    num = 0
    for key in change_res:                      #遍历结果字典
        num = num + change_res[key]
    return num
```

# 【例9.3】程序（续3）

```
#4、主函数
if __name__ == '__main__':
    clist = []                    #存放硬币种类列表
    change_dict = {}              #存放结果的字典，键为某种硬币面值，值为
该种硬币的个数
    #（1）输入找零数和硬币种类
    cvalue = int(input('请输入找零数(单位：元)：'))
    ctemp = int(input('请输入硬币种类，以非正数结束：\n'))
    while ctemp > 0:
        clist.append(ctemp)           #将输入的硬币种类添加到clist列表中
        ctemp = int(input())
```

多次输入硬币种类

# 【例9.3】程序（续4）

```
 #（2）调用change函数，求得和为11的最优硬币组合，得到字典change_dict
    change(clist,cvalue)

    #（3）调用checkOk函数，检查所求得的解是否为可行解
    if checkOk(change_dict, cvalue) == True:
        print ("找零结果：", change_dict)

        #  （4）调用coinNum函数，获得找零的硬币个数
        print ("找零个数：", coinNum(change_dict))
    else:
        print ("无解“)
```

# 【例9.3】程序运行结果

```
请输入找零数(单位：元)：11
请输入硬币种类, 以非正数结束：
1
3
5
-1
---------------------------------------------
最优硬币组合change_dict为： {5: 2}
新所凑硬币钱数newchange为： 1
---------------------------------------------
最优硬币组合change_dict为： {3: 0, 5: 2}
新所凑硬币钱数newchange为： 1
---------------------------------------------
最优硬币组合change_dict为： {1: 1, 3: 0, 5: 2}
新所凑硬币钱数newchange为： 0
---------------------------------------------
找零结果： {1: 1, 3: 0, 5: 2}
找零个数： 3
```

1元硬币1个，3元0个，5元2个

- **思考：还有最优解吗?**

**枚举法还有一个解：**

**3元2个，5元1个**

# 几种典型算法比较

| | |
|---|---|
| 枚举法 | • 总能得到全部解<br>• 时间开销可能非常大 |
| 动态规划 | • 可以获得全局最优解<br>• 时间开销比枚举法小 |
| 贪心法 | • 短时间内获得可行解<br>• 但不一定是全局最优解<br>• 不一定得到全部解 |

# 【课堂练习】

- 【**课堂练习**】已知有一个列表lis=[1,2,3,4,5,6,7,8,9,10]
  - 从lis中最后一个元素开始，截取**相邻4个**元素（如7、8、9、10），将其添加到列表diff中；然后指针向左移一位，再截取**相邻4个**元素（6、7、8、9）……，重复上述操作，直到截取完[1,2,3,4]为止
  - 打印diff
- 用Python编程实现