

# 第7章 贪心法

## □ 概述

### □ 基本思想

### □ 解决方法

## □ 典型应用

### □ 活动安排问题★

### □ 背包问题★

### □ 多机调度问题★

### □ 搬桌子问题★

## □ 总结

# 找零钱问题

- 问题描述:

用面额为 $d_1 > d_2 > \dots > d_m$ 的最少数量的硬币找出金额为 $n$ 的零钱。

- 例：美国广泛使用的硬币面额为 $d_1=25$ ,  $d_2=10$ ,  $d_3=5$ ,  $d_4=1$ 。如何用这些面额给出48美分的找零？
- 零钱种类：1个25美分，2个10美分，3个1美分
- 基于“贪心”的想法可以将剩余的硬币数量降为最低：

25  $\rightarrow$  25+10+10(45)  $\rightarrow$  25+10+10+1+1+1(48)

# “贪心” 算法不可行的例子

- 例：假设有面值单位分别为1、4和6的硬币。如果需要找出金额为8的零钱。若采用贪心算法，得到的解决方案如何？
  - 先选取面值最大的，然后再选小面值的：  
 $6 \rightarrow 6+1+1$
  - 更优的解：2枚面值为4的硬币
- 贪心算法总是做出在当前看来最好的选择。
- 贪心算法并不能总是找出最优解！它做出的选择只是在某种意义上的局部最优解。
- 虽然不能对所有问题都得到整体最优解，但是对许多问题它能产生整体最优解。一些情况下，即使贪心算法不能得到整体最优解，其最终结果却是最优解的很好近似。

## 4.1 贪心法概述

### 什么是贪心法

贪心法 (Greedy algorithms) 在解决问题的策略上目光短浅，每一步对目前构造的部分解做一个扩展，直到获得问题的完整解为止。所做的每一步选择都必须满足以下条件：

- ✓ 可行：即它必须满足问题的约束。
- ✓ 局部最优：它是当前步骤中所有可行选择中最佳的局部选择。
- ✓ 不可取消：选择一旦做出，在算法的后面步骤中就无法改变了。

这种局部最优选择并不总能获得整体最优解 (Optimal Solution)，但通常能获得近似最优解 (Near-Optimal Solution)。



## 1. 贪心选择性质

所谓贪心选择性质是指所求问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来达到。

也就是说，贪心法仅在当前状态下做出最好选择，即局部最优选择，然后再去求解做出这个选择后产生的相应子问题的解。

要确定一个具体问题是否具有贪心选择性质，必须证明每一步所做的贪心选择最终导致问题的整体最优解。证明方法一般使用数学归纳法。

**局部最优：**当前所可能的选择中最佳的局部选择。

## 2. 最优子结构性质

一个问题的最优解包含其子问题的最优解，则称此问题具有最优子结构性质。

问题的最优子结构性质是该问题可用动态规划算法或贪心法求解的关键特征。

**动态规划算法具有最优子结构特性和子问题重叠特性，通常以自底向上的方式解各子问题；而贪心算法则以迭代的方式作出相继的贪心选择，每作一次贪心选择就将所求问题简化为规模更小的子问题。**

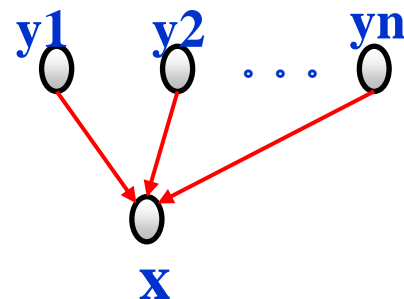
# 贪心算法与动态规划算法的差异

## □ 动态规划的递推式

□  $\text{opt}(x) = \max\{\text{opt}(y) + \text{payoff}(y, x)\}$

□  $y$  是  $x$  能达到的所有可能的状态

□  $x$  是“固定”的， $y$  是变量取决于  $x$



## □ 贪心算法的递推式

□  $\text{opt}(x) = \text{opt}(y) + \max\{\text{payoff}(y, x)\}$

□  $y$  是“固定”的，选择收益最大的，变成状态  $x$

□  $x$  取决于  $y$



是目前  
最优的

# 例子：背包问题和0-1背包问题

- 背包问题：

- 0/1背包问题：物品不可分
- 一般背包问题：物品是可分割的

与0-1背包问题类似，所不同的是在选择物品 $i$ 装入背包时，**可以选择物品 $i$ 的一部分**，而不一定要全部装入背包， $1 \leq i \leq n$ 。

这2类问题都具有**最优子结构性质**，极为相似，但背包问题可以用贪心算法求解，而0-1背包问题却不能用贪心算法求解。

**贪心法不能保证总能得到最优解**

**一系列的局部最优选择不能保证最后得到整体最优解**



**例：**  $n=3, C=50$

物品：	$u_1$	$u_2$	$u_3$
重量 $w_i$ ：	10	20	30
价值 $v_i$ ：	60	100	120
$v_i / w_i$ ：	6	5	4

0-1背包问题，  
正确的装入策略  
是：  $u_2 + u_3 = 220$

不同选择函数贪心选择结果：

选择剩余物品中价值最大的：  $1 * u_3 + 1 * u_2 = 220$

选择剩余物品中体积最小的：  $1 * u_1 + 1 * u_2 + 0.6 * u_3 = 232$

选择单位重量上价值最高的：  $1 * u_1 + 1 * u_2 + 0.6 * u_3 = 232$

贪心选择策略：按单位价值从大到小的顺序  
选择物品装入背包。

对于0-1背包问题使用“单位重量上价值最高”策略：

装入结果：  $u_1 + u_2 = 160$

显然不是最优解！

对于**0-1背包问题**，贪心选择之所以不能得到最优解是因为在这种情况下，它无法保证最终能将背包装满，部分闲置的背包空间使每公斤背包空间的价值降低了。

事实上，在考虑0-1背包问题时，应比较(**选择**该物品)和(**不选择**该物品)所导致的最终方案，然后再作出最好选择。由此就导出许多互相重叠的子问题。这正是该问题可用**动态规划算法**求解的另一重要特征。

实际上也是如此，动态规划算法的确可以有效地解0-1背包问题。

- 贪心算法的关键：贪心选择策略的确定。
- 贪心算法的特点：贪心算法往往效率高，一般时间复杂性为多项式阶。
- 贪心算法一般较简单，
- 其关键和难点在于：
  - 贪心选择策略的确定，
  - 以及证明相应的贪心算法确实可求出最优解。



贪心法求解问题的算法框架如下：

```
Greedy(C)                                //C是问题的输入集合即候选集合
{
    S={ };                                //初始解集合为空集
    while (not solution(S)) //集合S没有构成问题的一个解
    {
        x=select(C);                    //在候选集合C中做贪心选择
        if feasible(S, x)                //判断集合S中加入x后的解是否可行
        {
            S=S+{x};
            C=C-{x};
        }
    }
    return S;
}
```

# 1 求解活动安排问题

**问题描述：**假设有一个需要使用某一资源的 $n$ 个活动所组成的集合 $E$ ， $E=\{1,\cdots,n\}$ 。该资源一次只能被一个活动所占用，每一个活动 $i$ 有一个开始时间 $s_i$ 和结束时间 $f_i$  ( $s_i < f_i$ )。

若活动 $i$ 和活动 $j$ 有 $s_i \geq f_j$ 或 $s_j \geq f_i$ ，则称这两个活动**兼容**。

**需要选择出由互相兼容的活动所组成的最大集合。**



**问题求解：**采用贪心策略如下：每一步总是选择这样一个活动来占用资源，它能够使得**余下**的未调度的时间最大化，使得兼容的活动尽可能多。

算法每次总是选择**具有最早完成时间**的相容活动加入集合A中。

直观上，按这种方法选择相容活动为未安排活动留下尽可能多的时间。

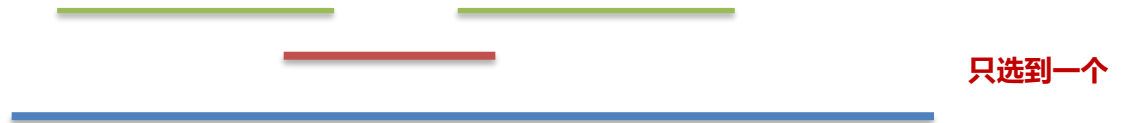
也就是说，该算法的贪心选择的意义是**使剩余的可安排时间段极大化**，从而安排尽可能多的相容活动。

# 其他贪心策略的反例

## □ 最早开始时间



## □ 最短任务



## □ 最少冲突



**例：** 设待安排的11个活动的开始时间和结束时间按**结束时间**的非减序排列如下：

i	1	2	3	4	5	6	7	8	9	10	11
开始时间 <b>S[i]</b>	1 ✓	3	0	5 ✓	3	5	6	8 ✓	8	2	12 ✓
结束时间 <b>f[i]</b>	4	5	6	7	8	9	10	11	12	13	14

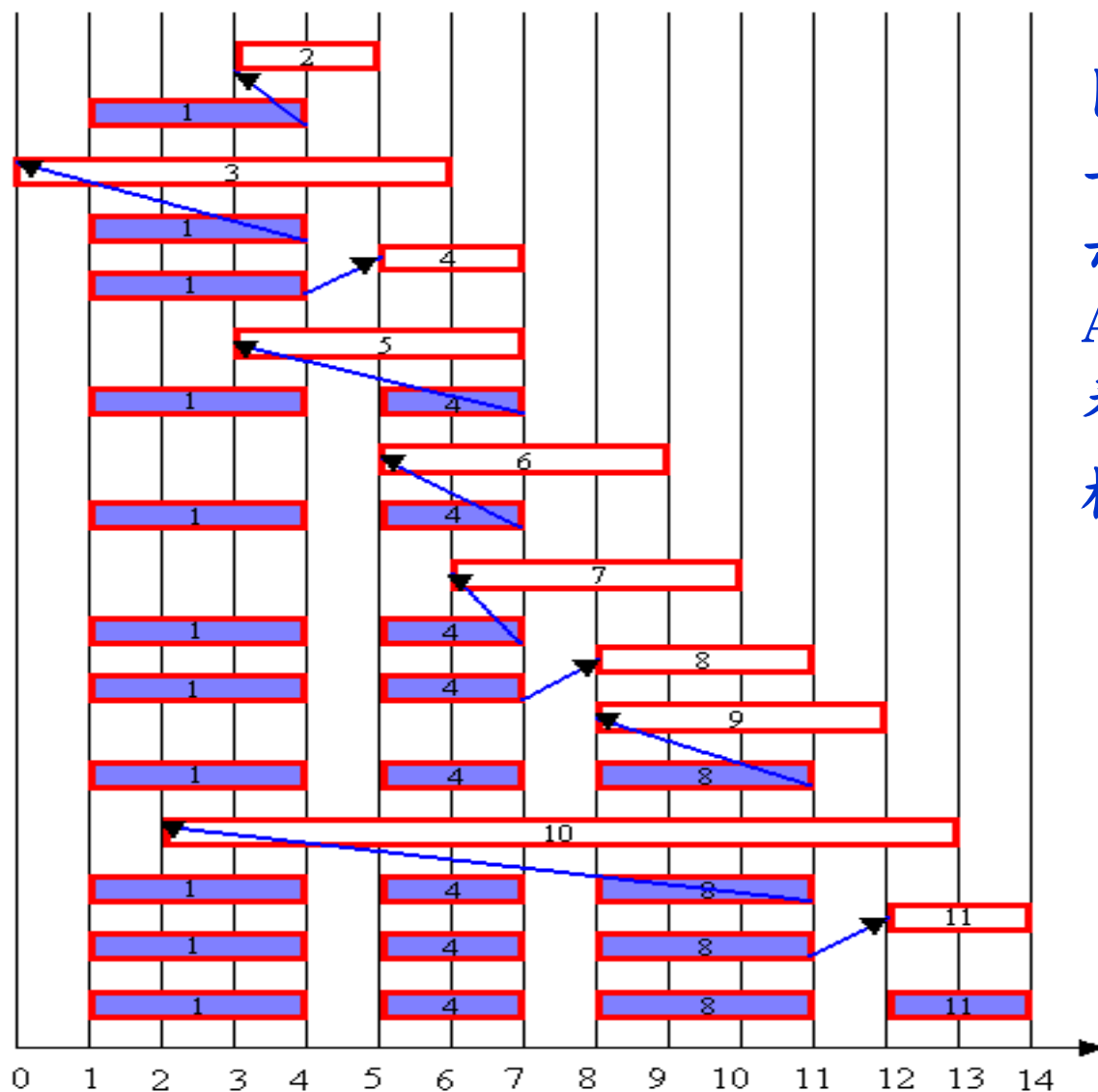
**贪心策略：** 具有**最早完成时间**的相容活动



<b>i</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>	<b>10</b>
<b>S[i]</b>	3	2	0	3	12	6	5	8	1	5
<b>f[i]</b>	5	13	6	8	15	10	7	9	6	9

<b>i</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>	<b>10</b>
<b>S[i]</b>	3	0	1	5	3	8	6	8	2	12
<b>f[i]</b>	5	6	6	7	8	9	10	11	13	15

# 活动安排问题的计算过程



图中每行相应于算法的一次迭代。阴影长条表示的活动是已选入集合A的活动，而空白长条表示的活动是当前正在检查相容性的活动。

由 $n$ 个活动（活动 $i$ 的起始时间和结束时间分别存放在 $s[i]$ 和 $f[i]$ 中， $0 \leq i \leq n-1$ ）产生的最大相容活动子集（所选活动的起始时间和结束时间分别存放在 $B[cnt]$ 和 $E[cnt]$ 中）的算法如下：

```
int greedySelector(int s[], int f[], int n, int B[], int E[])
{
    int i, cnt=0;           //cnt为B和E的下标，从0开始
    int select=0;          //select指向选到的活动，首先为第一个活动
    B[0]=s[0]; E[0]=f[0];  //选取第一个活动
    for (i=1; i<n; i++)    //i遍历所有活动
    {
        if (s[i] >= f[select]) //遇到了与select相容的活动
        {
            select = i;      //更新select，指向当前选取的活动的下标
            cnt++; B[cnt]=s[i]; E[cnt]=f[i]; //选取活动i
        }
    }
    return cnt+1;           //返回选取的活动个数
}
```

注意1：  $s[i]$ / $f[i]$ 是按照 $f[i]$ 的值排好的序列。

注意2： 以上代码也可以不用 $B[cnt]$ 和 $E[cnt]$ ，只用一个数组记录原活动数组中被选中的活动下标即可。具体看题目需求。

例如，对于下表所示的11个活动（已按结束时间递增排序） $X$ ，按最大不相交区间问题求解 $Y$ 的过程如下：

$i$	1	2	3	4	5	6	7	8	9	10	11
开始时间	1	3	0	5	3	5	6	8	8	2	12
结束时间	4	5	6	7	8	9	10	11	12	13	15

$X=\{[1,4],[3,5],[0,6],[5,7],[3,8],[5,9],[6,10],[8,11],[8,12],[2,13],[12,14]\}$

$i=1$ : 选择第一个活动，其右端点为4， $Y=\{[1,4]\}$

$i=2$ : 活动 $[3,5]$ 的左端点小于4，不选取

$i=3$ : 活动 $[0,6]$ 的左端点小于4，不选取

$i=4$ : 活动 $[5,7]$ 的左端点大于4，选择它，其右端点为7， $j=4$ ， $Y=\{[1,4],[5,7]\}$

$i=5$ : 活动 $[3,8]$ 的左端点小于7，不选取

$i=6$ : 活动 $[5,9]$ 的左端点小于7，不选取

$i=7$ : 活动 $[6,10]$ 的左端点小于7，不选取

$i=8$ : 活动 $[8,11]$ 的左端点大于7，选择它，其右端点为11， $Y=\{[1,4],[5,7],[8,11]\}$

$i=9$ : 活动 $[8,12]$ 的左端点小于11，不选取

$i=10$ : 活动 $[2,13]$ 的左端点小于11，不选取

$i=11$ : 活动 $[12,14]$ 的左端点大于11，选择它， $Y=\{[1,4],[5,7],[8,11],[12,14]\}$

所以最后选择的最大相容活动子集为 $\{1,4,8,11\}$ 。

# 活动安排问题的正确性证明（用数学归纳法）★

## 1 贪心选择性质

即证明活动安排问题总存在一个最优解从贪心选择开始。

设 $E=\{1,2,\dots,n\}$ 为所给的活动集合。由于 $E$ 中的活动按结束时间的非递减排序，故活动1具有最早完成时间。

首先证明活动安排问题有一个最优解以贪心选择开始，  
即该最优解中包含活动1。

设 $A \subseteq E$ 是所给活动安排问题的一个最优解，且 $A$ 中的活动也按结束时间非递减排序， $A$ 中的第一个活动是 $k$ 。

若 $k=1$ ，则 $A$ 就是以贪心选择开始的最优解。

若 $k>1$ ，设 $B=A-\{k\} \cup \{1\}$ 。

因为 $f_1 \leq f_k$ ，且因为 $A$ 中的活动是相容的。故 $B$ 中的活动也是相容的。

又由于 $B$ 中的活动个数与 $A$ 中的活动个数相同，故 $A$ 是最优的， $B$ 也是最优的。即 $B$ 是以选择活动1开始的最优活动安排。

由此可见，总存在以贪心选择开始的最优活动安排方案。

$B$ 是将 $A$ 集合中去掉活动 $K$ ，替换入活动1的集合。  
证明 $B$ 也是最优解（它（ $B$ ）包含活动1）

## 2 最优子结构性质（最优解在子问题上，也是最优解）

在作出了贪心选择，即选择了活动1后，原问题简化为对 $E$ 中所有与活动1相容的活动进行活动安排的子问题。即若 $A$ 是原问题的最优解，则 $A'=A-\{1\}$ 是活动安排问题的 $E'=\{i \in E: S_i \geq f_1\}$ 的最优解。

反证法：若 $E'$ 中存在另一个解 $B'$ ，比 $A'$ 有更多的活动，则将1加入 $B'$ 中产生另一个解 $B$ ，比 $A$ 有更多的活动。与 $A$ 的最优性矛盾。

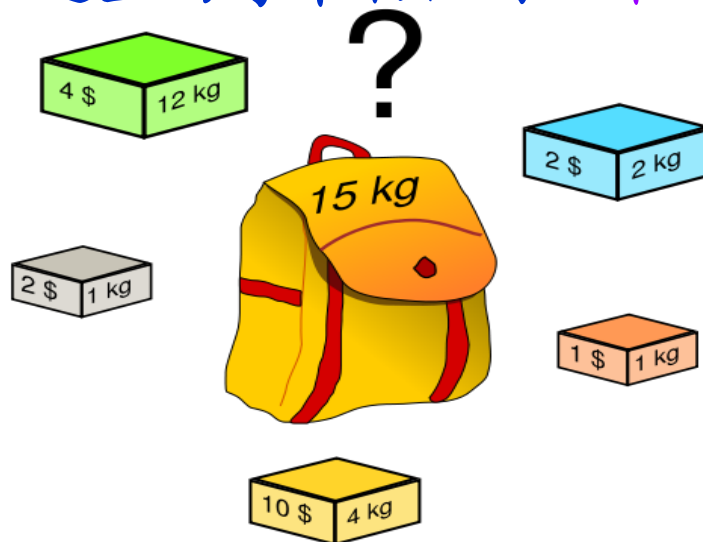
因此，每一步所作出的贪心选择都将问题简化为一个更小的与原问题具有相同形式的子问题。对贪心选择次数用归纳法可知，贪心算法greedySelector产生问题的最优解。

## 2 求解背包问题

**问题描述：**设有编号为1、2、...、 $n$ 的 $n$ 个物品，它们的重量分别为 $w_1$ 、 $w_2$ 、...、 $w_n$ ，价值分别为 $v_1$ 、 $v_2$ 、...、 $v_n$ ，其中 $w_i$ 、 $v_i$  ( $1 \leq i \leq n$ ) 均为正数。

有一个背包可以携带的最大重量不超过 $W$ 。求解目标是，在不超过背包负重的前提下，使背包装入的总价值最大（即效益最大化）。

与0/1背包问题的区别是，这里的每个物品可以取一部分装入背包。





**问题求解：**这里采用贪心法求解。设 $x_i$ 表示物品 $i$ 装入背包的情况， $0 \leq x_i \leq 1$ 。根据问题的要求，有如下约束条件和目标函数：

$$\sum_{i=1}^n w_i x_i \leq W \quad 0 \leq x_i \leq 1 \quad (1 \leq i \leq n)$$

$$\text{MAX} \left\{ \sum_{i=1}^n w_i x_i \right\}$$

于是问题归结为寻找一个满足上述约束条件，并使目标函数达到最大的解向量 $X = \{x_1, x_2, \dots, x_n\}$ 。

例如,  $n=3$ ,  $(w_1, w_2, w_3)=(18, 15, 10)$ ,  $(v_1, v_2, v_3)=(25, 24, 15)$ ,  $W=20$ , 其中的4个可行解如下:

解编号	$(x_1, x_2, x_3)$	$\sum_{i=1}^n w_i x_i$	$\sum_{i=1}^n v_i x_i$
①	$(1/2, 1/3, 1/4)$	16.5	24.25
②	$(1, 2/15, 0)$	20	28.2
③	$(0, 2/3, 1)$	20	31
④	$(0, 1, 1/2)$	20	31.5

在这4个可行解中, 第④个解的效益最大, 可以求出它是这个背包问题的最优解。

**贪心策略：选择单位重量价值最大的物品。**

每次从物品集合中选择单位重量价值最大的物品，如果其重量小于背包容量，就可以把它装入，并将背包容量减去该物品的重量，然后就面临了一个最优子问题——它同样是背包问题，只不过背包容量减少了，物品集合减少了。

因此背包问题具有最优子结构性质。

对于下表所示一个背包问题， $n=5$ ，设背包容量 $W=100$ ，其求解过程如下：

$i$	1	2	3	4	5
$w_i$	10	20	30	40	50
$v_i$	20	30	66	40	60
$v_i/w_i$	2.0	1.5	2.2	1.0	1.2

- ① 将价值即 $v/w$ 递减排序，其结果为  
 $\{66/30, 20/10, 30/20, 60/50, 40/40\}$ ，物品重新按0~4编号。
- ② 设背包余下装入的重量为weight，其初值为 $W$ 。

$i$	1	2	3	4	5
$w_i$	10	20	30	40	50
$v_i$	20	30	66	40	60
$v_i/w_i$	2.0	1.5	2.2	1.0	1.2

③  $i=0$ 开始,  $w[0]<\text{weight}$ 成立, 表明物品0能够装入, 将其装入到背包中, 置 $x[0]=1$ ,  $\text{weight}=\text{weight}-w[0]=70$ ,  $i$ 增1即 $i=1$ ;  
 $w[1]<\text{weight}$ 成立, 表明物品1能够装入, 将其装入到背包中, 置 $x[1]=1$ ,  $\text{weight}=\text{weight}-w[1]=60$ ,  $i$ 增1即 $i=2$ ;  
 $w[2]<\text{weight}$ 成立, 表明物品2能够装入, 将其装入到背包中, 置 $x[2]=1$ ,  $\text{weight}=\text{weight}-w[2]=50$ ,  $i$ 增1即 $i=3$ ;  
 $w[3]<\text{weight}$ 不成立, 且 $\text{weight}>0$ , 表明只能将物品3部分装入, 装入比例 $=\text{weight}/w[3]=50/60=80\%$ , 置 $x[3]=0.8$ 。  
 算法结束, 得到 $X=\{1,1,1,0.8,0\}$ 。

```

double knap(double W, double w[], double v[],
    int n, double x[]) //求解背包问题并返回总价值
{
    int i;
    double V=0;           //V为总价值
    double weight=W;      //背包中能装入的余下重量
    for (i=0; i<n; i++)    //初始化x向量
        x[i]=0;
    i=0;
    while (w[i]<weight)    //物品i能够全部装入时循环
    {
        x[i]=1;           //装入物品i
        weight-=w[i];      //减少背包中能装入的余下重量
        V+=v[i];           //累计总价值
        i++;              //继续循环
    }
    if (weight>0)          //当余下重量大于0
    {
        x[i]=weight/w[i]; //将物品i的一部分装入
        V+=x[i]*v[i];      //累计总价值
    }
    return V;             //返回总价值
}

```

**算法证明：**假设对于 $n$ 个物品，按 $v_i/w_i$  ( $1 \leq i \leq n$ ) 值递减排序得到1、2、 $\dots$ 、 $n$ 的序列，即 $v_1/w_1 \geq v_2/w_2 \geq \dots \geq v_n/w_n$ 。设 $X=\{x_1, x_2, \dots, x_n\}$ 是本算法找到解。

如果所有的 $x_i$ 都等于1，这个解明显是最优解。否则，设 $\min j$ 是满足 $x_{\min j} < 1$ 的最小下标。考虑算法的工作方式，很明显，当 $i < \min j$ 时， $x_i = 1$ ，当 $i > \min j$ 时， $x_i = 0$ ，并且。设 $X$ 的值为 $V(X) = \sum_{i=1}^n v_i x_i$ 。

设 $Y=\{y_1, y_2, \dots, y_n\}$ 是该背包问题的一个最优可行解，因此有 $\sum_{i=1}^n w_i y_i \leq W$ ，从而有 $\sum_{i=1}^n w_i (x_i - y_i) = \sum_{i=1}^n w_i x_i - \sum_{i=1}^n w_i y_i \geq 0$ ，这个解的值为 $V(Y) = \sum_{i=1}^n v_i y_i$ 。则

$$V(X) - V(Y) = \sum_{i=1}^n v_i (x_i - y_i) = \sum_{i=1}^n w_i \frac{v_i}{w_i} (x_i - y_i)$$

当 $i < \min j$ 时,  $x_i = 1$ , 所以 $x_i - y_i \geq 0$ , 且 $v_i/w_i \geq v_{\min j}/w_{\min j}$ 。  
 当 $i > \min j$ 时,  $x_i = 0$ , 所以 $x_i - y_i \leq 0$ , 且 $v_i/w_i \leq v_{\min j}/w_{\min j}$ 。  
 当 $i = \min j$ 时,  $v_i/w_i = v_{\min j}/w_{\min j}$ 。

$$\begin{aligned}
 \text{则 } V(X) - V(Y) &= \sum_{i=1}^n w_i \frac{v_i}{w_i} (x_i - y_i) = \sum_{i=1}^{\min j - 1} w_i \frac{v_i}{w_i} (x_i - y_i) + \sum_{i=\min j}^{\min j} w_i \frac{v_i}{w_i} (x_i - y_i) + \sum_{i=\min j + 1}^n w_i \frac{v_i}{w_i} (x_i - y_i) \\
 &\geq \sum_{i=1}^{\min j - 1} w_i \frac{v_{\min j}}{w_{\min j}} (x_i - y_i) + \sum_{i=\min j}^{\min j} w_i \frac{v_{\min j}}{w_{\min j}} (x_i - y_i) + \sum_{i=\min j + 1}^n w_i \frac{v_{\min j}}{w_{\min j}} (x_i - y_i) \\
 &= \frac{v_{\min j}}{w_{\min j}} \sum_{i=1}^n w_i (x_i - y_i) \geq 0
 \end{aligned}$$

这样与 $Y$ 是最优解的假设矛盾, 也就是说没有哪个可行解的价值会大于 $V(X)$ , 因此解 $X$ 是最优解。

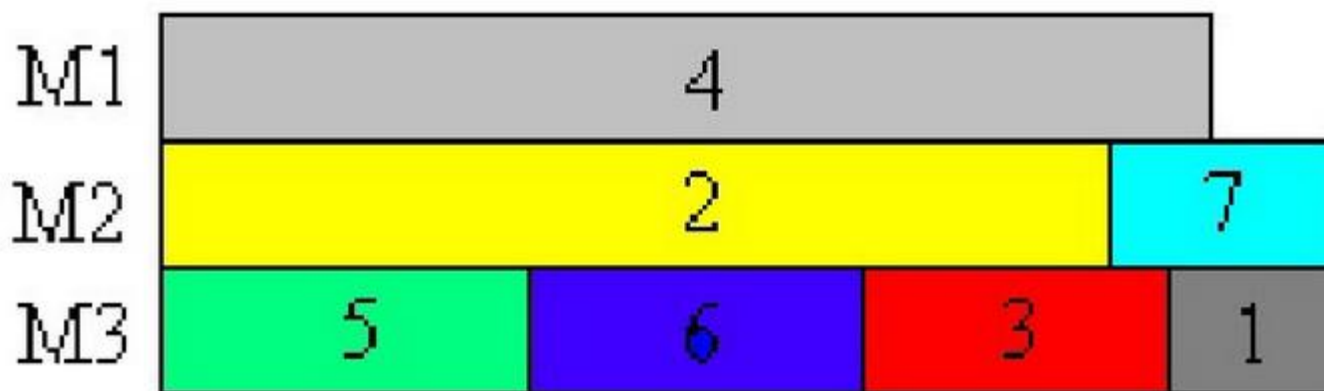


**算法分析：**快速排序的时间复杂性为 $O(n\log_2 n)$ , while  
循环的时间为 $O(n)$ , 所以本算法的时间复杂度为 $O(n\log_2 n)$ 。

### 3 求解多机调度问题

**问题描述：**设有 $n$ 个独立的作业 $\{1, 2, \dots, n\}$ ，由 $m$ 台相同的机器 $\{1, 2, \dots, m\}$ 进行加工处理，作业 $i$ 所需的处理时间为 $t_i$  ( $1 \leq i \leq n$ )，每个作业均可在任何一台机器上加工处理，但未完工前不允许中断，任何作业也不能拆分成更小的子作业。

多机调度问题要求给出一种作业调度方案，使所给的 $n$ 个作业在尽可能短的时间内由 $m$ 台机器加工处理完成。



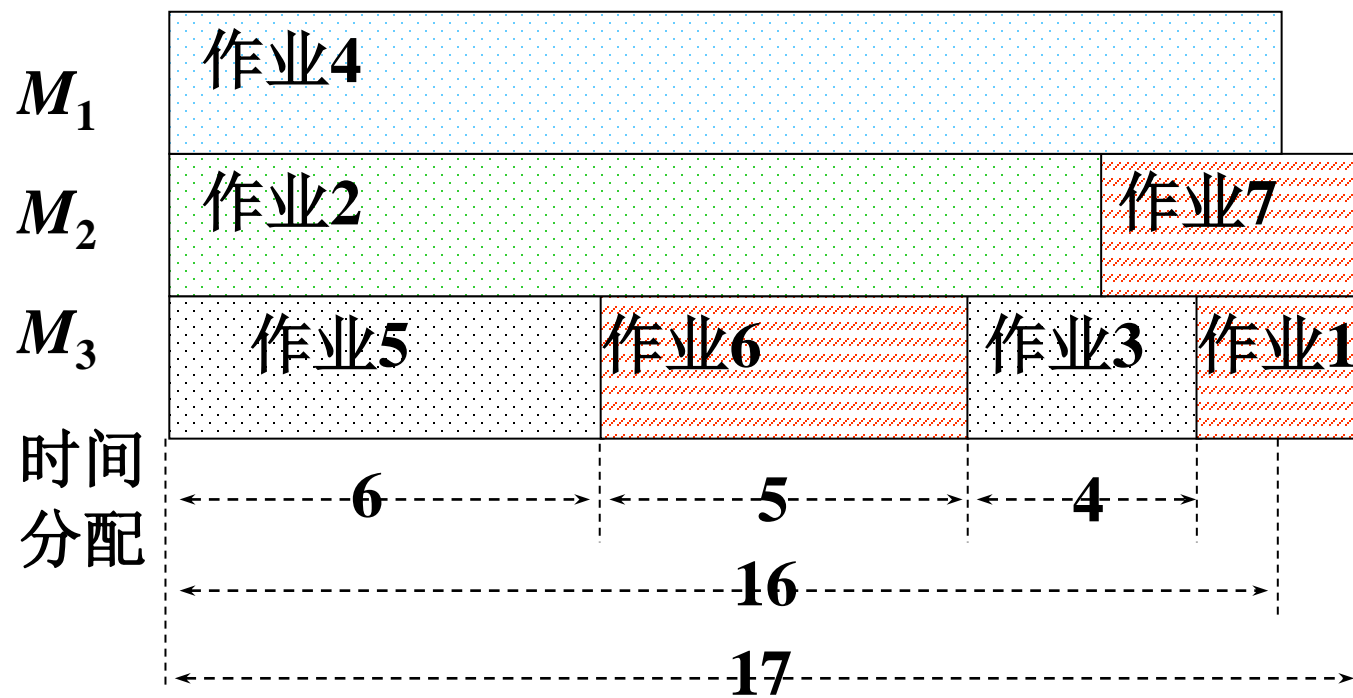
**问题求解：**贪心法求解多机调度问题的贪心策略是**最长处理时间作业优先**，即**把处理时间最长的作业分配给最先空闲的机器**，这样可以保证处理时间长的作业优先处理，从而在整体上获得尽可能短的处理时间。

按照最长处理时间作业优先的贪心策略，当 $m \geq n$ 时，只  
要将机器 $i$ 的 $[0, t_i)$ 时间区间分配给作业 $i$ 即可；

当 $m < n$ 时，首先将 $n$ 个作业依其所需的处理时间从大到小  
排序，然后依此顺序将作业分配给空闲的处理机。

例如，有7个独立的作业{1,2,3,4,5,6,7}，由3台机器{1,2,3}加工处理，各作业所需的处理时间如下表所示。

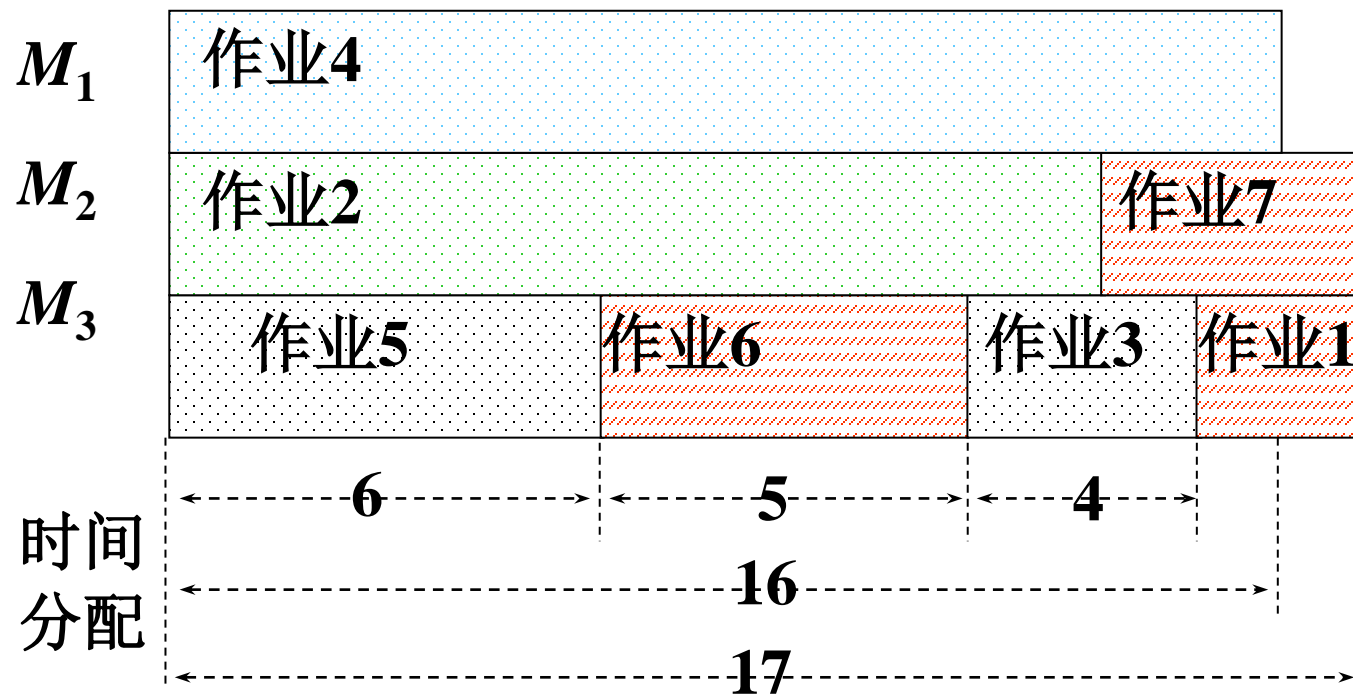
作业编号	1	2	3	4	5	6	7
作业的处理时间	2	14	4	16	6	5	3



三台机器的调度问题示例

例如，有7个独立的作业{1,2,3,4,5,6,7}，由3台机器{1,2,3}加工处理，各作业所需的处理时间如下表所示。

作业编号	1	2	3	4	5	6	7
作业的处理时间	2	14	4	16	6	5	3



三台机器的调度问题示例

这里 $n=7$ ,  $m=3$ , 采用贪心法求解的过程如下:

① 7个作业按处理时间递减排序, 其结果如表5.4所示。

作业编号	4	2	5	6	3	7	1
作业的处理时间	16	14	6	5	4	3	2

② 先将排序后的前3个作业分配给3台机器。此时机器的分配情况为 $\{\{4\},\{2\},\{5\}\}$ , 对应的总处理时间为 $\{16,14,6\}$ 。

③ 分配余下的作业。

分配作业6: 3台机器中机器3在时间6后最先空闲, 将作业6分配给它, 此时机器的分配情况为 $\{\{4\},\{2\},\{5,6\}\}$ , 对应的总处理时间为 $\{16,14,6+5=11\}$ 。

分配作业3: 3台机器中机器3在时间11后最先空闲, 将作业3分配给它, 此时机器的分配情况为 $\{\{4\},\{2\},\{5,6,3\}\}$ , 对应的总处理时间为 $\{16,14,11+4=15\}$ 。

分配作业4: 3台机器中机器2在时间14后最先空闲, 将作业7分配给它, 此时机器的分配情况为 $\{\{4\},\{2,7\},\{5,6,3\}\}$ , 对应的总处理时间为 $\{16,14+3=17,15\}$ 。

分配作业1: 3台机器中机器3在时间15后最先空闲, 将作业1分配给它, 此时机器的分配情况为 $\{\{4\},\{2,7\},\{5,6,3,1\}\}$ , 对应的总处理时间为 $\{16,17,15+2=17\}$ 。

```

void Mscheduling(int P[], int T[], PlanType S[], int n, int m)
//求调度方案S
{
    int i, j, k;
    for (i=0; i<m; i++) //将m个作业分配给m台机器
    {
        S[i].num=S[i].sumt=0;
        S[i].seq[S[i].num]=P[i]; //将作业P[i]分配给机器i
        S[i].sumt=T[i]; //累加处理时间
        S[i].num++; //累计处理作业数
    }
    for (i=m; i<n; i++) //分配余下的作业
    {
        j=0;
        for (k=1; k<m; k++) //求所有机器中处理时间总数最小的下标j
            if (S[k].sumt<S[j].sumt) j=k;
        S[j].seq[S[j].num]=P[i]; //将作业P[i]分配给机器j
        S[j].sumt+=T[i]; //累加处理时间
        S[j].num++; //累计处理作业数
    }
}

```

由于多机调度问题是**NP难问题**，到目前为止还没有有效的解法，上述算法是采用贪心法求得的一个较好的**近似解**。

**算法分析：**快速排序的时间复杂性为 $O(n\log_2 n)$ ，两次for循环的时间为 $O(n)$ ，所以本算法的时间复杂度为 $O(n\log_2 n)$ 。



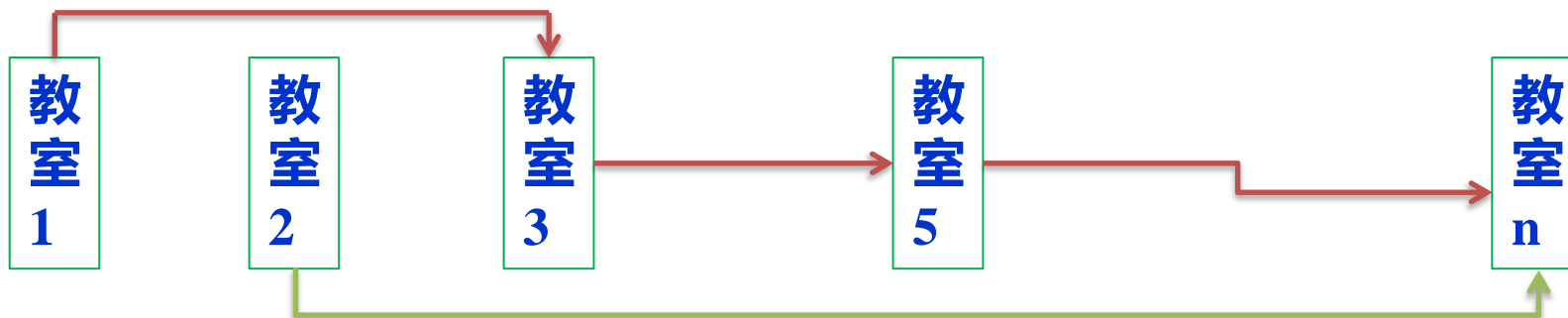
## 4 补充搬桌子问题

**问题描述：**某教学大楼一层有 $n$ 个教室，从左到右依次编号为1、2、...、 $n$ 。现在要把一些课桌从某些教室搬到另外一些教室，每张桌子都是从编号较小的教室搬到编号较大的教室，每一趟，都是从左到右走，且一次只能搬一张桌子，搬完一张课桌后，可以继续从当前位置或往右走搬另一张桌子，中间不能折返，走到最右端教室后可以折返搬下一趟。

**输入数据：**先输入 $n$ (教室)、 $m$ (桌子)，

然后紧接着 $m$ 行输入这 $m$ 张要搬课桌的起始教室和目标教室。

**输出数据：**最少需要跑几趟。



桌子1: 教室3 => 教室5

桌子2: 教室1 => 教室3

桌子3: 教室5 => 教室n

。 。 。

桌子<sub>m</sub>: 教室2 => 教室n

**贪心策略：把课桌按起点从小到大排序，每次都是搬离当前位置最近的课桌。**


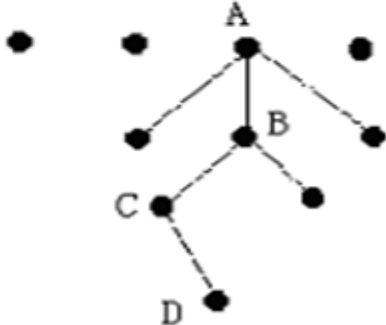
每次搬**目的编号最小**的（一趟尽可能搬最多数目）桌子，这个方法可以吗？



上例，按照**目的编号最小最早**策略，第一趟移动了3张桌子；剩余还需要移动两趟，总计3趟。

按照就近搬动策略，第一趟移动了2张桌子；剩余只需移动一趟，总计2趟。

**该策略思想：因为都要搬完，所以不多跑路，不浪费时间。**

	贪心算法	动态规划
不同点	所占空间较小	所占空间较大
	某次具体选择必是最优选择	某次具体选择不一定是最终最优选择
	解题大体框架： 线形 	解题大体框架： 图或树 

**END**