


# 算法设计与分析

主讲教师：王新宇



在算法设计方法中，有一种称为“贪心”的算法设计策略，它是求解最优化问题（特别是NP难的组合优化问题）的常见方法之一。

应用这种贪心策略设计的算法在大多数情况下易于实现且非常高效，能够得到很多问题的最优解。

在某些情况下，即使贪心法无法得到问题的全局最优解，其最终结果仍是全局最优解的很好近似。



# 第11章 贪心法

# 目 录

## 11.1 贪心法概述

## 11.2 贪心法的应用实例

## 11.3 小结



## 11.1 贪心法概述

# 11.1.1 贪心法的基本思想

## ➤ 一个应用实例——克鲁斯卡尔算法

设连通网 $N=(V, E)$ ，所求最小生成树 $TN=(V, TE)$ 。

**求解最小生成树的克鲁斯卡尔算法：**首先将连通网 $N$ 中的每个顶点作为一个独立的连通分量，形成一个包含 $N$ 的所有顶点、但没有边的初始生成树 $TN$ ；然后从连通网 $N$ 的边集 $E$ 中选取**权值最小**的边，若该边的两个顶点分别位于 $TN$ 的两个不同连通分量，则将该边加入 $TE$ ，否则舍弃该边；依次类推，直到 $TN$ 中的所有顶点位于一个连通分量为止。

克鲁斯卡尔算法**每次都是在边集 $E$ 中选择权值最小的能够连通两个连通分量的边**加入生成树 $TN$ ，使得生成树中所有边的当前权值之和最小，不考虑这样得到的生成树最终能否满足最小生成树的条件，这就是所谓的**贪心选择**。

# 11.1.1 贪心法的基本思想

## ➤ 什么是贪心法？

**贪心法**：在求解问题的过程中，总是通过做出当前看来最好的选择（贪心选择）求解问题的一种方法。

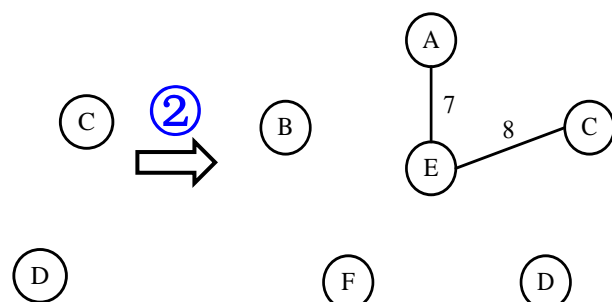
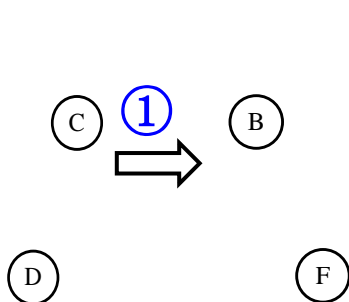
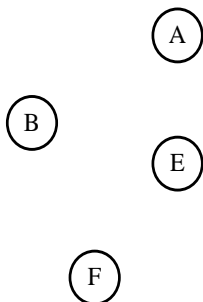
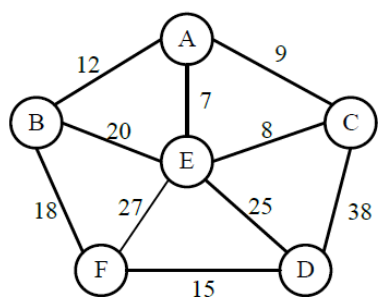
贪心法不从问题的全局最优进行考虑，它所做出的每一次选择都是某种意义上的**局部最优选择**。

因此，**贪心法不能对所有问题都得到全局最优解**。但对许多问题而言，它还是能够产生全局最优解的，如最小生成树问题、单源点最短路径问题和哈夫曼编码等。

在一些情况下，即使贪心法不能得到全局最优解，仍然可以得到很好的近似解。

# 11.1.1 贪心法的基本思想

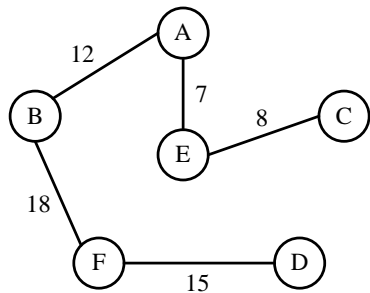
➤ 贪心法求角  $U=\{A, B, C, D, E, F\}$   $U=\{A, B, C, D, E, F\}$   $U=\{A, B, C, D, E, F\}$   
 $TE=\Phi$   $TE=\{(A, E)\}$   $TE=\{(A, E), (E, C)\}$



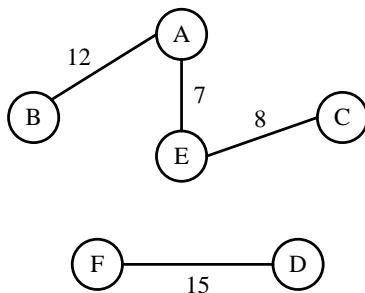
$U=\{A, B, C, D, E, F\}$   
 $TE=\{(A, E), (E, C), (A, B), (F, D), (B, F)\}$

$U=\{A, B, C, D, E, F\}$   
 $TE=\{(A, E), (E, C), (A, B), (F, D)\}$

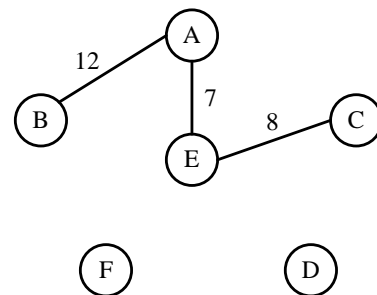
$U=\{A, B, C, D, E, F\}$   
 $TE=\{(A, E), (E, C), (A, B)\}$



⑤



④



最小生成树

用克鲁斯卡尔算法求解最小生成树的过程



# 11.1.1 贪心法的基本思想

**贪心法求解问题的一般过程：**贪心法将整个问题的求解过程划分为 $k$ 个阶段，从问题的某一个初始解出发，每一阶段依据一定的准则决策产生 $k$ 元组解 $(x_1, x_2, \dots, x_k)$ 的一个分量，逐步构造出整个问题的最优解。

贪心法在每一个阶段作为决策依据的准则称为**贪心准则**。

**每一次贪心选择都将问题转化为规模更小的子问题**，并期望通过每次所做的局部最优选择产生一个全局最优解。

## 11.1.2 贪心法的适用条件

### (1) 贪心选择性质

**贪心选择性质**是指问题的全局最优解可以通过一系列局部最优的选择（贪心选择）来达到。

贪心选择性质是贪心算法可行的第一个要素，也是**贪心算法与动态规划算法的主要区别**。

一个问题能否用贪心法求解，必须要确定它是否具有贪心选择性质，即确定由每一阶段所作的贪心选择是否能够最终产生问题的全局最优解。

## 11.1.2 贪心法的适用条件

### (2) 最优子结构性质

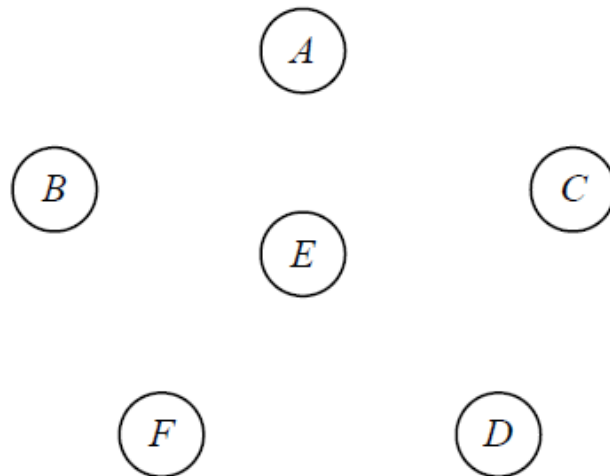
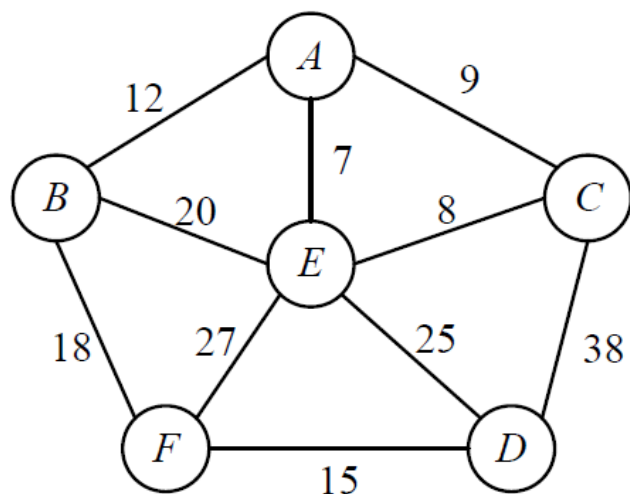
**最优子结构性质**是指一个问题的最优解包含其子问题的最优解。

最优子结构性质是一个问题可用贪心算法或动态规划算法求解的**关键特征**。

## 11.1.2 贪心法的适用条件

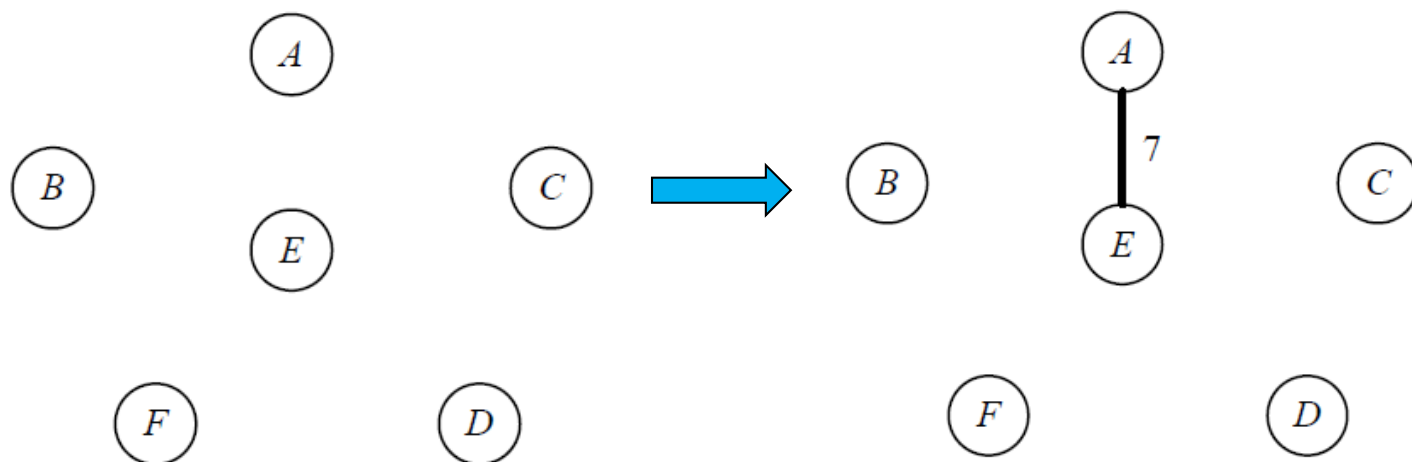
分析：最小生成树问题是否具有贪心选择性质和最优子结构性？

最小生成树问题通过每次选择权值最小的能够连通两个连通分量的边加入生成树这一贪心选择，能够最终获得连通网的最小生成树，因此具有贪心选择性质。



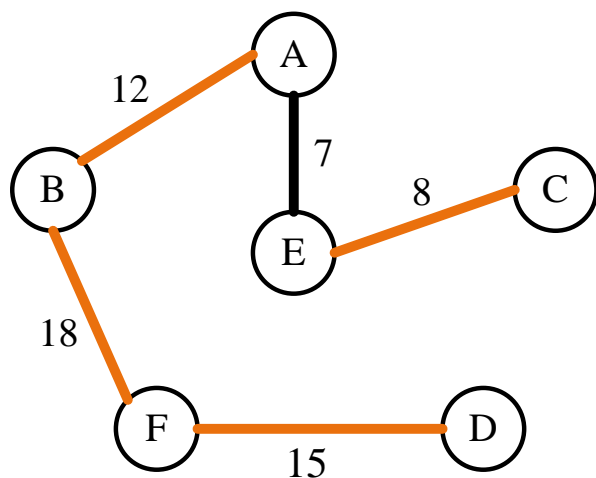
原问题：寻找5条边连接6个独立的连通分量形成一棵最小生成树

## 11.1.2 贪心法的适用条件



原问题：寻找5条边连接6个独立的连通分量形成一棵最小生成树

子问题：寻找4条边连接5个独立的连通分量形成一棵最小生成树



子问题的最优解包含在原问题的最优解之中。因此，最小生成树问题具有最优子结构性质。

# 11.1.3 贪心法和动态规划的区别

(1) 贪心法能够解决的问题必须具有贪心选择性质，动态规划没有这个要求。

(2) 在求解过程中，贪心法基于贪心准则确定每一阶段的决策，将当前问题转化为一个规模更小的子问题，不同阶段的子问题的解之间不存在依赖关系；动态规划基于优化函数最优值的递推方程计算当前问题的解，较大规模子问题的解决定于较小规模子问题的解。

(3) 贪心法的求解过程通常是自顶向下，随着求解过程，问题的规模不断减小，直至求解规模最小的子问题，在求解过程中每个子问题的解构成整个问题的解；动态规划的求解过程通常是自底向上，从最小子问题开始求解，直至计算得到整个问题的解。

## 11.1.4 贪心法的设计步骤

(1) **分解**: 通过分析, 将原始问题自顶向下分解为若干子问题, 把问题的求解过程划分为若干阶段, 每一个阶段对应一个子问题。

(2) **求解**: 设计**贪心准则**, 根据贪心准则逐阶段求解问题当前状态下的局部最优解。

(3) **合并**: 把各阶段的局部最优解合成原始问题的一个解。

➤ 两个关键

(1) **贪心准则**

(2) **算法的正确性证明**

## 11.1.4 贪心法的设计步骤

### ➤ 贪心法的算法框架

//SolutionType表示问题的解向量 $\mathbf{x}=(x_0, x_1, \dots, x_{n-1})$ 的类型, SType表示分量 $x_i$ 的类型, a表示问题的输入, n表示问题的规模。

```
SolutionType Greedy(SType a[], int n) {  
    SolutionType x = {};           //初始化解向量  
    for(int i = 0; i < n; i++) {   //n个阶段的求解过程  
        SType xi = Select(a);    //根据贪心准则做出当前阶段的决策  
        if(Feasible(xi))         //判断当前决策是否可行  
            x = Union(x, xi);    //可行, 将当前解xi合并入解向量  
    }  
    return x;                      //返回问题的最优解  
}
```



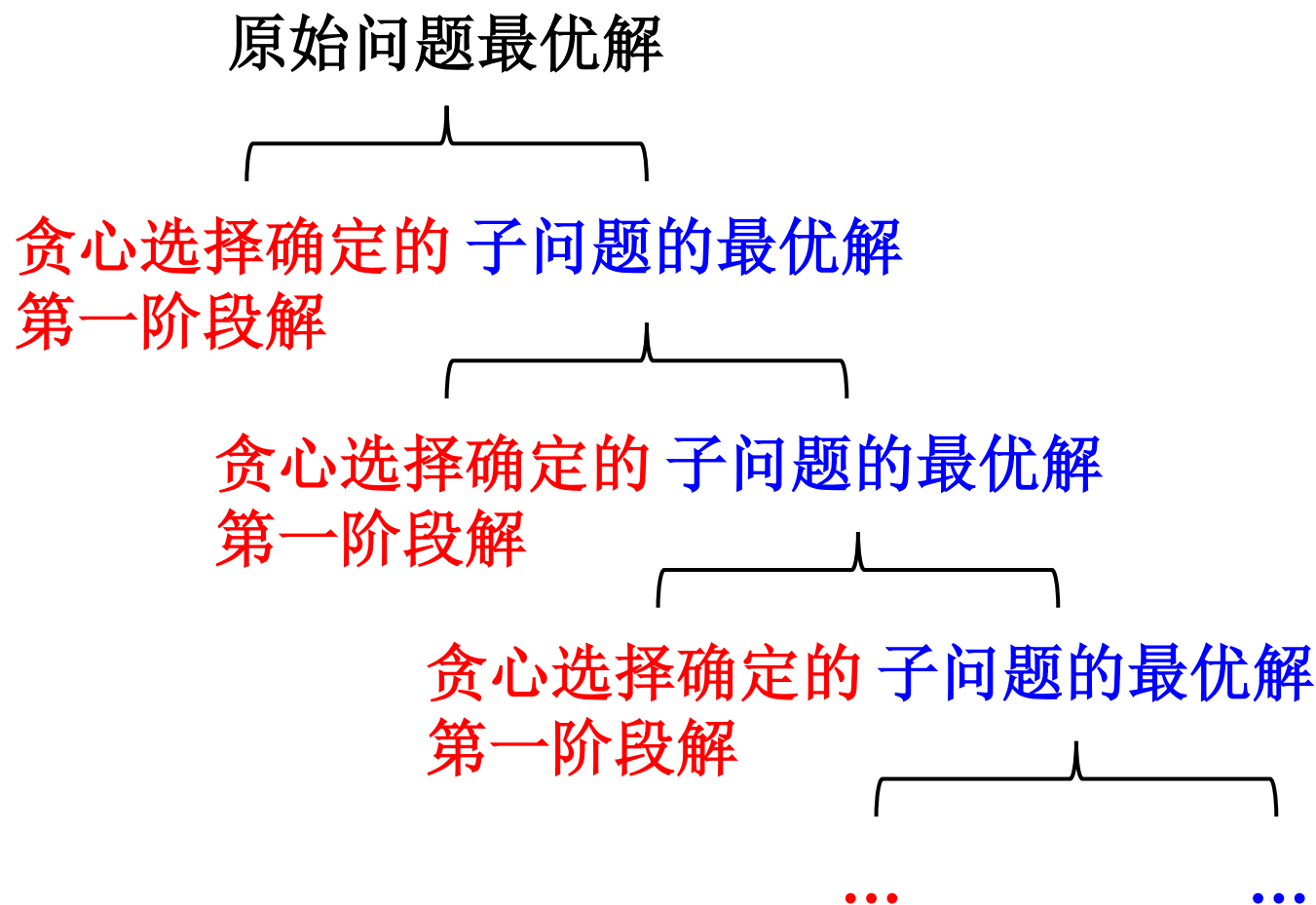
## 11.1.5 贪心算法的正确性证明

(1) **证明问题具有贪心选择性质**。证明方法：构造问题的一个最优解，对该解进行修正，使其第一阶段的决策成为一个贪心选择，从而证明总是能够找到一个以贪心选择开始的最优解。

(2) **证明问题具有最优子结构性质**。证明方法采用**反证法**：构造问题的一个最优解，假设其中包含的子问题的解并非最优，推出通过寻找到子问题的最优解能够得到比问题原本的最优解更优的一个解，与前提产生矛盾，表明假设不成立，从而证明问题的最优解包含其子问题的最优解。

(3) **对贪心选择次数用数学归纳法进行归纳**，即证明通过每一阶段的贪心选择，最终可以得到问题的全局最优解。

## 11.1.5 贪心算法的正确性证明





## 11.2 贪心法的应用实例

## 11.2.1 活动安排问题

已知 $n$ 个活动组成的集合 $A=\{1, \dots, n\}$ ，每个活动都需要使用同一个资源，而该资源在任何时刻只能被一个活动占用。

每个活动 $i$ 都有一个开始时间 $s_i$ 和结束时间 $f_i$ ，且 $s_i < f_i$ 。一旦开始执行某个活动，资源即被占用，一直持续到该活动执行完毕。

若活动 $i$ 和活动 $j$ 满足 $s_i \geq f_j$ 或 $s_j \geq f_i$ ，则称活动 $i$ 和活动 $j$ 是相容的。

现在要求寻找一个最优活动安排方案，使得被安排的活动数量达到最多，即求活动集合 $A$ 的最大相容活动集合。

## 11.2.1 活动安排问题

### 【问题分析】

#### (1) 分解问题

原问题

寻找一个最优安排

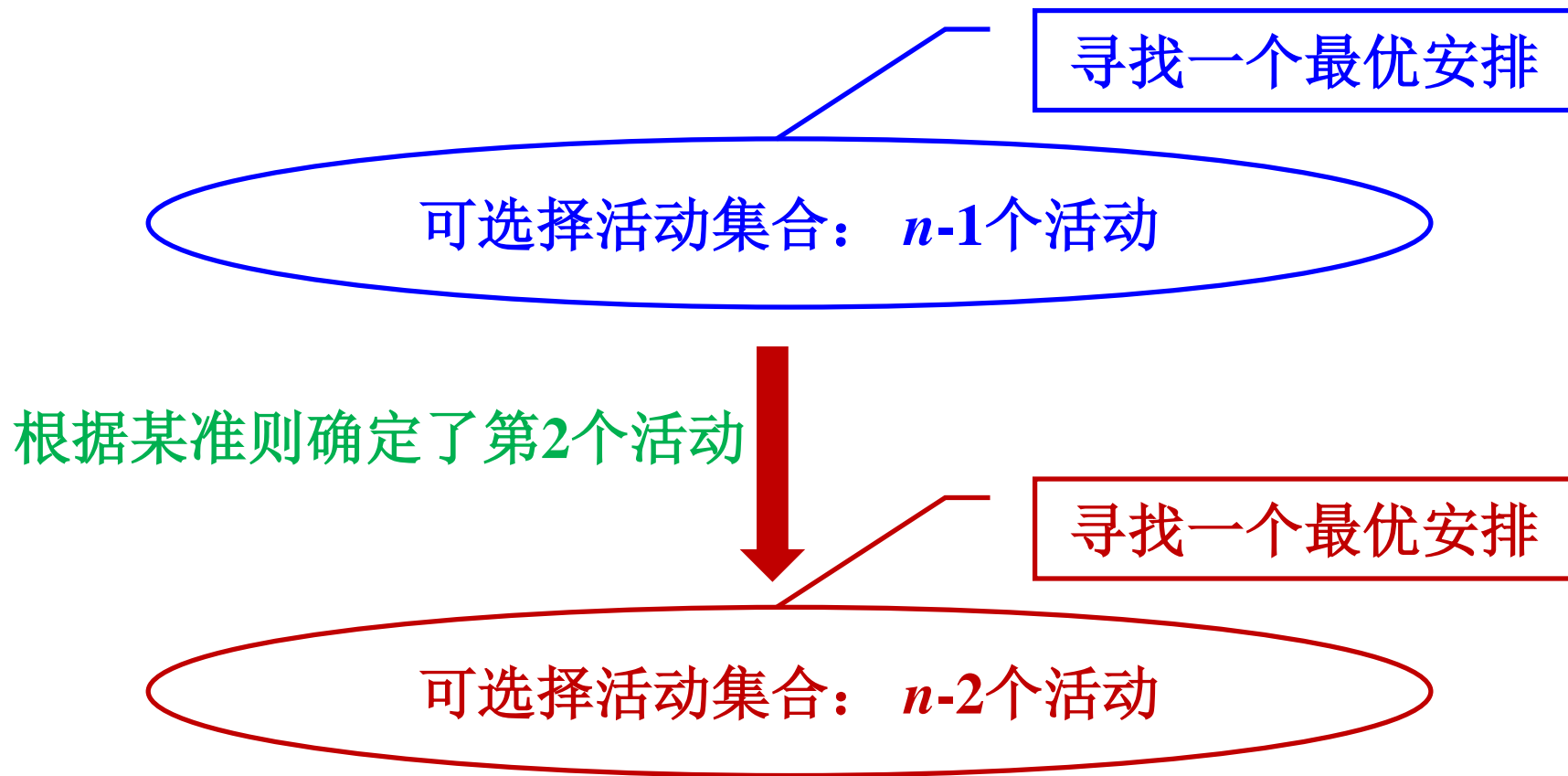
可选择活动集合： $n$ 个活动

根据某准则确定了第1个活动

寻找一个最优安排

可选择活动集合： $n-1$ 个活动

## 11.2.1 活动安排问题



重复上述过程，直至找不到可以安排的活动为止。

## 11.2.1 活动安排问题

### (2) 设计贪心准则

**策略1:** 把活动按照**开始时间**由小到大排序, 使得 $s_1 \leq s_2 \leq \dots \leq s_n$ , 从前往后挑选活动, 只要当前活动与前一个挑选的活动不冲突, 就选择该活动。

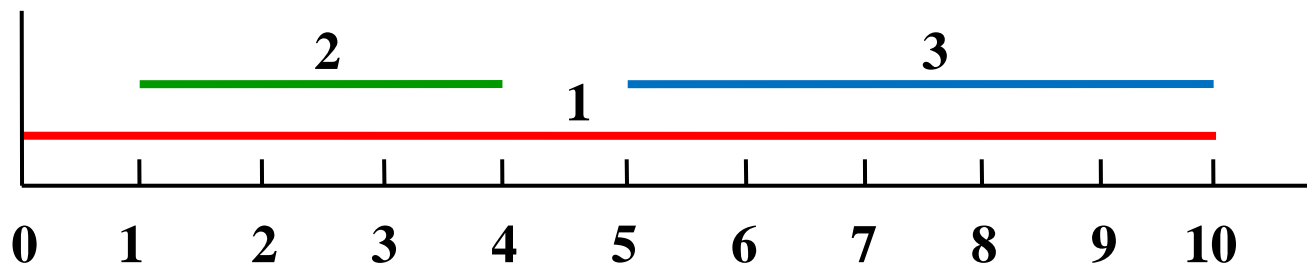
**策略2:** 把活动按照**占用资源的时间长短**由小到大排序, 使得 $f_1 - s_1 \leq f_2 - s_2 \leq \dots \leq f_n - s_n$ , 从前往后挑选活动, 只要当前活动与前一个挑选的活动不冲突, 就选择该活动。

**策略3:** 把活动按照**结束时间**由小到大排序, 使得 $f_1 \leq f_2 \leq \dots \leq f_n$ , 从前往后挑选活动, 只要当前活动与前一个挑选的活动不冲突, 就选择该活动。

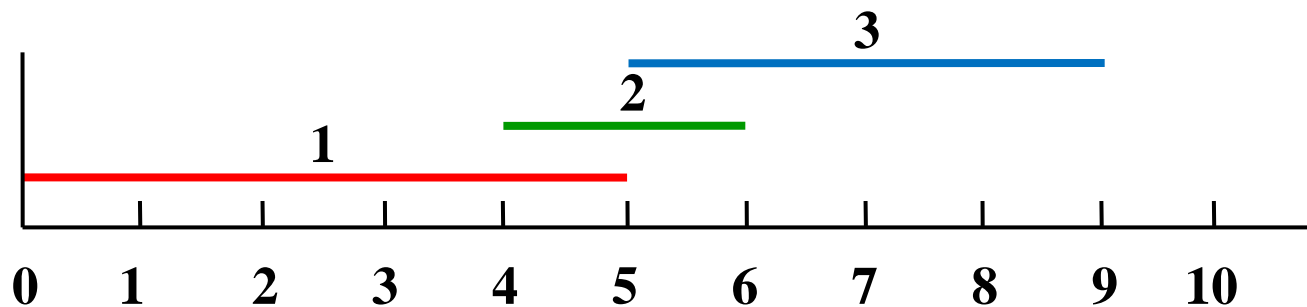
## 11.2.1 活动安排问题

### □ 两个反例

策略1: 若 $A=\{1, 2, 3\}$ ,  $s_1=0$ ,  $f_1=10$ ,  $s_2=1$ ,  $f_2=4$ ,  $s_3=5$ ,  $f_3=10$



策略2: 若 $A=\{1, 2, 3\}$ ,  $s_1=0$ ,  $f_1=5$ ,  $s_2=4$ ,  $f_2=6$ ,  $s_3=5$ ,  $f_3=9$



表明策略1和策略2无法找到问题的最优解。



## 11.2.1 活动安排问题

### 【算法实现】

一维结构体数组A存放n个活动，其中，A[i]表示第i个活动（ $1 \leq i \leq n$ ），A[i].no记录第i个活动的编号，A[i].s记录第i个活动的开始时间，A[i].f记录第i个活动的结束时间。

一维数组E作为标记数组记录选择的活动的情况，其中，E[i]记录活动A[i]的选择情况。

pre记录前一个被选择的活动的结束时间。

## 11.2.1 活动安排问题

```
struct Activity {
```

```
    int no;           //活动编号
```

```
    int s, f;       //活动开始时间， 活动结束时间
```

```
    //重载小于运算符， 用于按活动的结束时间递增排序
```

```
    bool operator<(const Activity &b) const
```

```
    { return f < b.f; }
```

```
};
```

## 11.2.1 活动安排问题

```
void ActivitySelect(Activity *A, int n, bool *E) {
```

```
    sort(A + 1, A + n + 1);
```

```
    for(int i = 1; i <= n; i++) E[i] = false;
```

```
    int pre = 0;
```

```
    for(int i = 1; i <= n; i++) {
```

```
        if(A[i].s >= pre) {
```

```
            E[i] = true;
```

```
            pre = A[i].f;
```

```
        }
```

```
    }
```

```
}
```

S1: 按照活动的结束时间对A中的活动由小到大排序。

## 11.2.1 活动安排问题

```
void ActivitySelect(Activity *A, int n, bool *E) {
```

```
    sort(A + 1, A + n + 1);
```

```
    for(int i = 1; i <= n; i++) E[i] = false;
```

```
    int pre = 0;
```

```
    for(int i = 1; i <= n; i++) {
```

```
        if(A[i].s >= pre) {
```

```
            E[i] = true;
```

```
            pre = A[i].f;
```

```
        }
```

```
    }
```

```
}
```

S2: 初始化标记数组  
E的元素为false，表  
示活动均未被选择；  
并初始化pre=0。

## 11.2.1 活动安排问题

```
void ActivitySelect(Activity *A, int n, bool *E) {
```

```
    sort(A + 1, A + n + 1);
```

```
    for(int i = 1; i <= n; i++) E[i] = false;
```

```
    int pre = 0;
```

```
    for(int i = 1; i <= n; i++) {
```

```
        if(A[i].s >= pre) {
```

```
            E[i] = true;
```

```
            pre = A[i].f;
```

```
        }
```

```
    }
```

```
}
```

S3: 逐个检查第1, 2, ..., n个活动, 进行如下处理。

## 11.2.1 活动安排问题

```
void ActivitySelect(Activity *A, int n, bool *E) {
```

```
    sort(A + 1, A + n + 1);
```

```
    for(int i = 1; i <= n; i++) E[i] = false;
```

```
    int pre = 0;
```

```
    for(int i = 1; i <= n; i++) {
```

```
        if(A[i].s >= pre) {
```

```
            E[i] = true;
```

```
            pre = A[i].f;
```

```
        }
```

```
    }
```

```
}
```

判断活动A[i]与前一个被选择的  
活动是否相容，若相容，  
则选择A[i]作为可以安排的  
活动，修改E[i]和pre。

## 11.2.1 活动安排问题

### 【算法分析】

算法的时间主要花费在排序上。

排序使用了 C++ 的函数 `sort`，该函数的时间复杂度为  $O(n\log n)$ ，所以整个算法的时间复杂度为  $O(n\log n)$ 。

## 11.2.1 活动安排问题

### 【一个简单实例的求解过程】

例如，对于下表的 $n=11$ 个活动（已按活动结束时间递增排序）

| 序号 $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8  | 9  | 10 | 11 |
|--------|---|---|---|---|---|---|---|----|----|----|----|
| 编号     | 9 | 1 | 4 | 7 | 3 | 6 | 8 | 2  | 11 | 10 | 5  |
| 开始时间   | 1 | 3 | 0 | 2 | 2 | 5 | 6 | 8  | 10 | 8  | 4  |
| 结束时间   | 3 | 5 | 6 | 7 | 8 | 9 | 9 | 12 | 13 | 15 | 16 |

产生最大相容活动  
集合的过程

| 序号 $i$ | A[ $i$ ].s | E[ $i$ ] | pre | 序号 $i$ | A[ $i$ ].s | E[ $i$ ] | pre |
|--------|------------|----------|-----|--------|------------|----------|-----|
| 1      | 1          | true     | 3   | 8      | 8          | false    | 9   |
| 2      | 3          | true     | 5   | 9      | 10         | true     | 13  |
| 3      | 0          | false    | 5   | 10     | 8          | false    | 13  |
| 4      | 2          | false    | 5   | 11     | 4          | false    | 13  |
| 5      | 2          | false    | 5   |        |            |          |     |
| 6      | 5          | true     | 9   |        |            |          |     |
| 7      | 6          | false    | 9   |        |            |          |     |

最优活动安排：编号9的活动->编号1的活动->编号6的活动->编号11的活动



## 11.2.1 活动安排问题

### 【算法的正确性证明】

设 $n$ 个活动已经按照结束时间递增排序，活动 $i(i=1, 2, \dots, n)$ 表示有序活动序列中序号为 $i$ 的活动。

证明：若 $X$ 是活动安排问题 $A$ 的最优解，则 $X=X' \cup \{1\}$ ，且 $X'$ 是活动安排问题 $A'=\{i \in A \mid f_i \geq f_1, i=2, 3, \dots, n\}$ 的最优解，即证明问题具有贪心选择性质和最优子结构性质。

## 11.2.1 活动安排问题

### (1) 证明贪心选择性质

即证明总存在一个以活动1开始的最优解。

设在活动安排问题 $A$ 的最优解 $X$ 中，第一个选中的活动为 $k(k \neq 1)$ 。

用活动1代替 $X$ 中的活动 $k$ 构造出另一个解 $Y$ ，由于 $f_1 \leq f_k$ ，因此 $Y$ 中的活动也相容，且 $Y$ 与 $X$ 的活动数相同，表明 $Y$ 也是最优的，即总存在一个以活动1开始的最优解。

## 11.2.1 活动安排问题

### (2) 证明最优子结构性质

在做出对活动1的贪心选择后，原问题转化为在活动2、3..... $n$ 中寻找与活动1相容的活动的子问题。

证明最优子结构性质就是要证明：若 $X$ 为原问题 $A$ 的一个最优解，则 $X'=X-\{1\}$ 是活动选择问题 $A'=\{i \in A \mid f_i \geq f_1, i=2, 3, \dots, n\}$ 的一个最优解。

反证法：假设能找到 $A'$ 的一个比 $X'$ 包含更多活动的解 $Y'$ ，则将活动1加入 $Y'$ 后就得到 $A$ 的一个比 $X$ 包含更多活动的解 $Y$ ，与 $X$ 是最优解的前提矛盾。

因此，假设不成立，表明每一次做出的贪心选择都将问题转化为一个更小的与原问题具有相同形式的子问题，满足最优子结构性质。

## 11.2.2 最优装载问题

有编号为1, 2, ...,  $n$ 的 $n$ 个集装箱要装上一艘轮船, 集装箱 $i$ 的质量为 $w_i$ ; 轮船装载质量的限制为 $c$ , 但无体积限制。请确定一个装载方案将尽可能多的集装箱装上轮船。

最优装载问题的目标函数可以表示为

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

约束条件为

$$\sum_{i=1}^n w_i x_i \leq c, \quad x_i \in \{0, 1\}, \quad i=1, 2, \dots, n$$

其中,  $x_i=0$ 表示第 $i$ 个集装箱不装上轮船;  $x_i=1$ 表示第 $i$ 个集装箱装上轮船。

## 11.2.2 最优装载问题

### 【问题分析】

#### (1) 划分子问题

原问题

待装载集装箱集合： $n$ 个集装箱

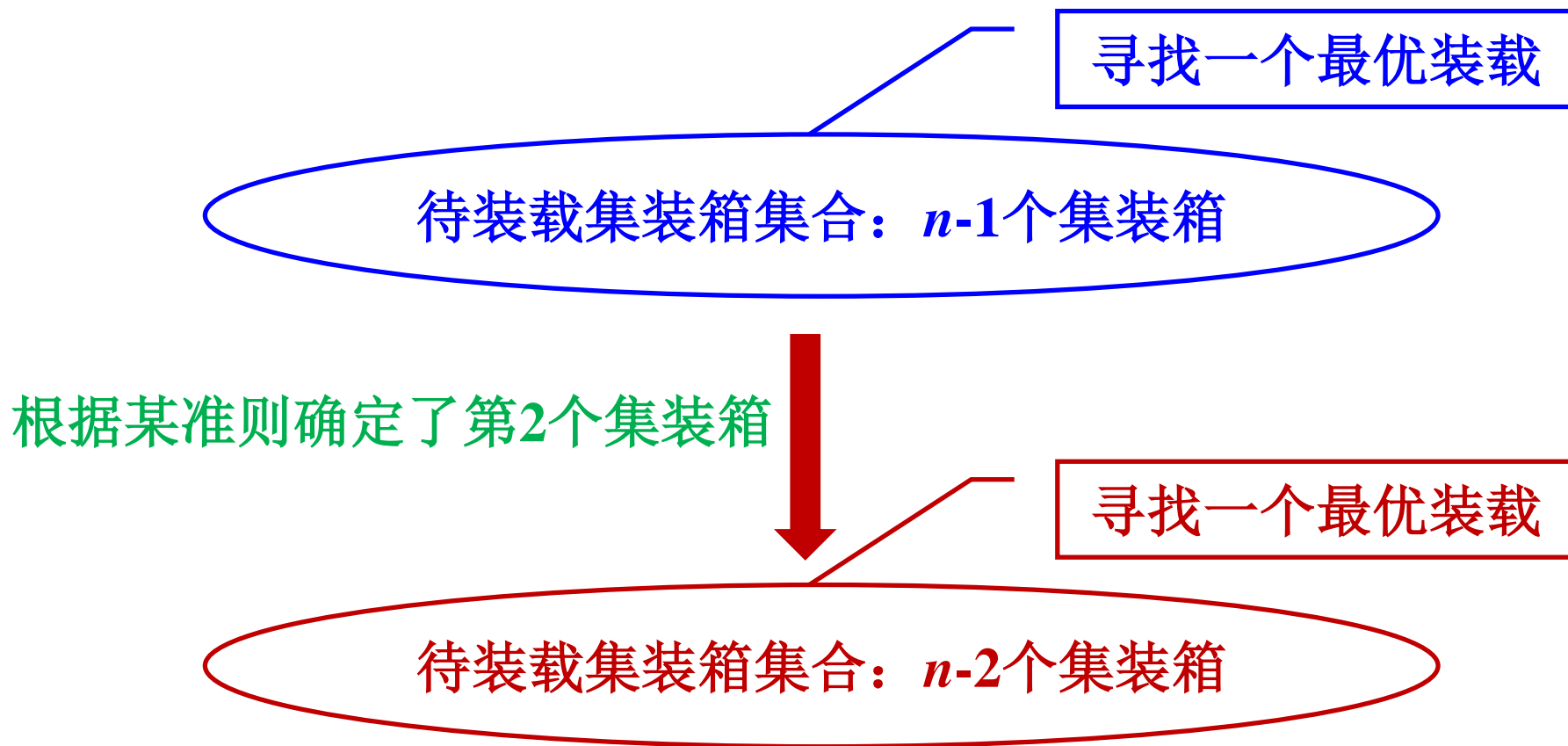
寻找一个最优装载

根据某准则确定了第1个集装箱

待装载集装箱集合： $n-1$ 个集装箱

寻找一个最优装载

## 11.2.2 最优装载问题



重复上述过程，直至找不到可以装载的集装箱为止。

## 11.2.2 最优装载问题

### (2) 设计贪心准则

在最优装载问题中，轮船没有体积限制，只有装载的质量限制。

在这种情况下，要使装载的集装箱数量尽可能多，显然先装载的集装箱的质量要尽可能轻，这样才能留出更大的装载容量来装上更多的集装箱。

贪心准则：优先选择质量轻的集装箱装上轮船。

## 11.2.2 最优装载问题

### 【算法实现】

一维结构体数组A记录集装箱信息，其中，A[i]表示第i个集装箱，A[i].no表示集装箱的编号，A[i].w表示集装箱的质量（ $1 \leq i \leq n$ ）。

一维数组x作为标记数组记录对集装箱的决策，其中，x[i]记录对第i个集装箱的决策。

total记录装载的总质量。



## 11.2.2 最优装载问题

```
struct Container {  
  
    int no;    //集装箱编号  
  
    int w;    //集装箱质量  
  
    //重载小于运算符，用于按质量递增排序  
  
    bool operator<(const Container &s) const  
  
    { return w < s.w; }  
  
};
```

## 11.2.2 最优装载问题

```
int Load(Container *A, int n, int *x, int c) {
```

```
    int total = 0;
```

```
    sort(A + 1, A + n + 1);
```

S1: 将集装箱按  
质量递增排序。

```
    for(int i = 1; i <= n; i++) x[i] = 0;
```

```
    for(int i = 1; i <= n && total + A[i].w <= c; i++) {
```

```
        x[i] = 1;
```

```
        total += A[i].w;
```

```
    }
```

```
    return total;
```

```
}
```

## 11.2.2 最优装载问题

```
int Load(Container *A, int n, int *x, int c) {
```

```
    int total = 0;
```

```
    sort(A + 1, A + n + 1);
```

```
    for(int i = 1; i <= n; i++) x[i] = 0;
```

S2: 初始化标记  
数组x。

```
    for(int i = 1; i <= n && total + A[i].w <= c; i++) {
```

```
        x[i] = 1;
```

```
        total += A[i].w;
```

```
    }
```

```
    return total;
```

```
}
```

## 11.2.2 最优装载问题

```
int Load(Container *A, int n, int *x, int c) {  
    int total = 0;  
    sort(A + 1, A + n + 1);  
    for(int i = 1; i <= n; i++) x[i] = 0;  
    for(int i = 1; i <= n && total + A[i].w <= c; i++) {  
        x[i] = 1;  
        total += A[i].w;  
    }  
    return total;  
}
```

S3: 对第 $i(=1, 2, \dots, n)$ 个集装箱, 若 $\text{total} + A[i].w$ 不超过轮船的装载质量限制, 则将第 $i$ 个集装箱装上船, 更新 $x[i]$ 和 $\text{total}$ 。

## 11.2.2 最优装载问题

```
int Load(Container *A, int n, int *x, int c) {  
    int total = 0;  
    sort(A + 1, A + n + 1);  
    for(int i = 1; i <= n; i++) x[i] = 0;  
    for(int i = 1; i <= n && total + A[i].w <= c; i++) {  
        x[i] = 1;  
        total += A[i].w;  
    }  
    return total;  
}
```

S4: 返回装载的  
集装箱总质量。

## 11.2.2 最优装载问题

### 【算法分析】

算法的时间主要花费在排序上。

`sort`函数的时间复杂度为 $O(n\log n)$ ，所以，整个算法的时间复杂度为 $O(n\log n)$ 。

## 11.2.2 最优装载问题

### 【算法的正确性证明】

设 $n$ 个集装箱已按质量递增排序，集装箱 $i(i=1, 2, \dots, n)$ 表示有序集装箱序列中序号为 $i$ 的集装箱； $X=(x_1, x_2, \dots, x_n)$ 是最优装载问题的一个最优解。

#### (1) 证明贪心选择性质

设 $k=\min\{i|x_i=1\}$ 且 $k \neq 1$ 为最优解 $X$ 中第1个被装上轮船的集装箱的序号，  
证明：将 $k$ 替换为1，即用质量最轻的集装箱1替换集装箱 $k$ ，得到的解仍是原问题的一个最优解。

构造一个解 $Y=(y_1, y_2, \dots, y_n)$ ，其中， $y_1=1$ ， $y_k=0$ ， $y_i=x_i(1 < i \leq n \text{ 且 } i \neq k)$ ，则

$$\sum_{i=1}^n w_i y_i = \sum_{i=1}^n w_i x_i - w_k + w_1 \leq \sum_{i=1}^n w_i x_i \leq c$$

因此， $Y$ 是满足变量约束条件的一个可行解。并且， $\sum_{i=1}^n y_i = \sum_{i=1}^n x_i$ ，即 $Y$ 的集装箱数目与 $X$ 相同，所以， $Y$ 也是一个最优解，表明对于最优装载问题，总是能够找到一个以贪心选择开始的最优解。

## 11.2.2 最优装载问题

### (2) 证明最优子结构性质

在做出对集装箱1的贪心选择后，原问题就变成了在集装箱2, ...,  $n$  中寻找装载质量不超过 $c-w_1$ 的最优装载子问题。证明最优子结构性质就是要证明：若 $X$ 为原问题的一个最优解，则 $X'=X-\{1\}$ 是装载质量不超过 $c-w_1$ 的装载问题的一个最优解。

反证法：假设子问题能够找到一个方案 $Y'$ ，装载的集装箱数量比 $X'$ 更多，且不超过装载质量的限制 $c-w_1$ ，则将集装箱1加入 $Y'$ 后就得到原始问题的一个比 $X$ 装载更多集装箱的方案 $Y$ ，与 $X$ 是最优解的前提相矛盾。

因此，假设不成立，表明每一次做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，满足最优子结构性质。



## 11.2.3 背包问题

有编号为 $1, 2, \dots, n$ 的 $n$ 种物品和一个背包，物品 $i$ 的质量是 $w_i$ ，价值为 $v_i$ ，背包的容量为 $c$ 。在选择装入背包的物品时，对物品 $i$ 可以取一部分装入，但最多只能装入1个。应当如何选择物品，使得装入物品后的背包价值最大呢？

设 $x_i (i=1, 2, \dots, n)$ 表示装入背包的第 $i$ 种物品的数量，则背包问题的目标函数可以表示为

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

约束条件为

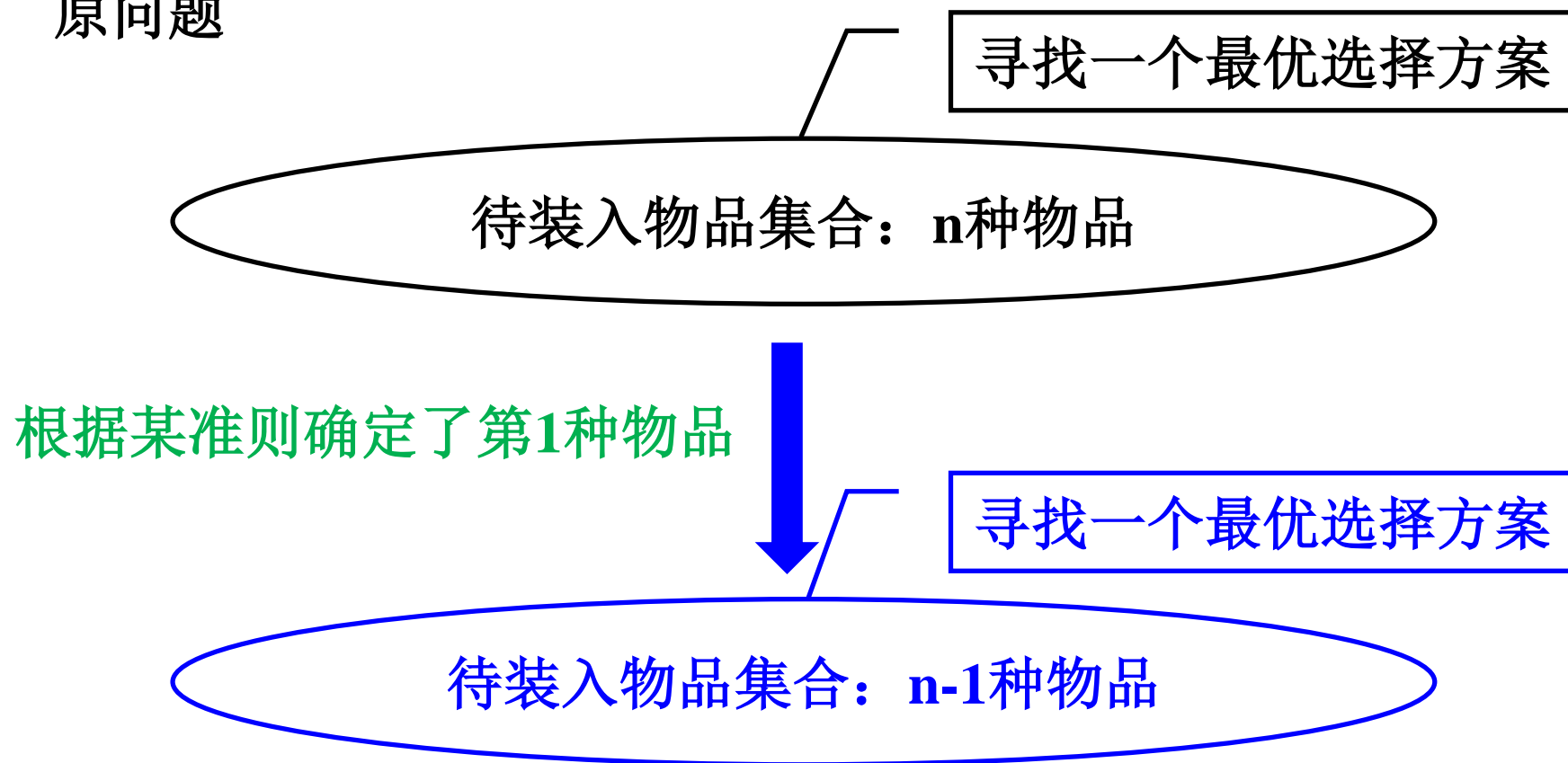
$$\sum_{i=1}^n w_i x_i \leq c, \quad 0 \leq x_i \leq 1 (i=1, 2, \dots, n)。$$

## 11.2.3 背包问题

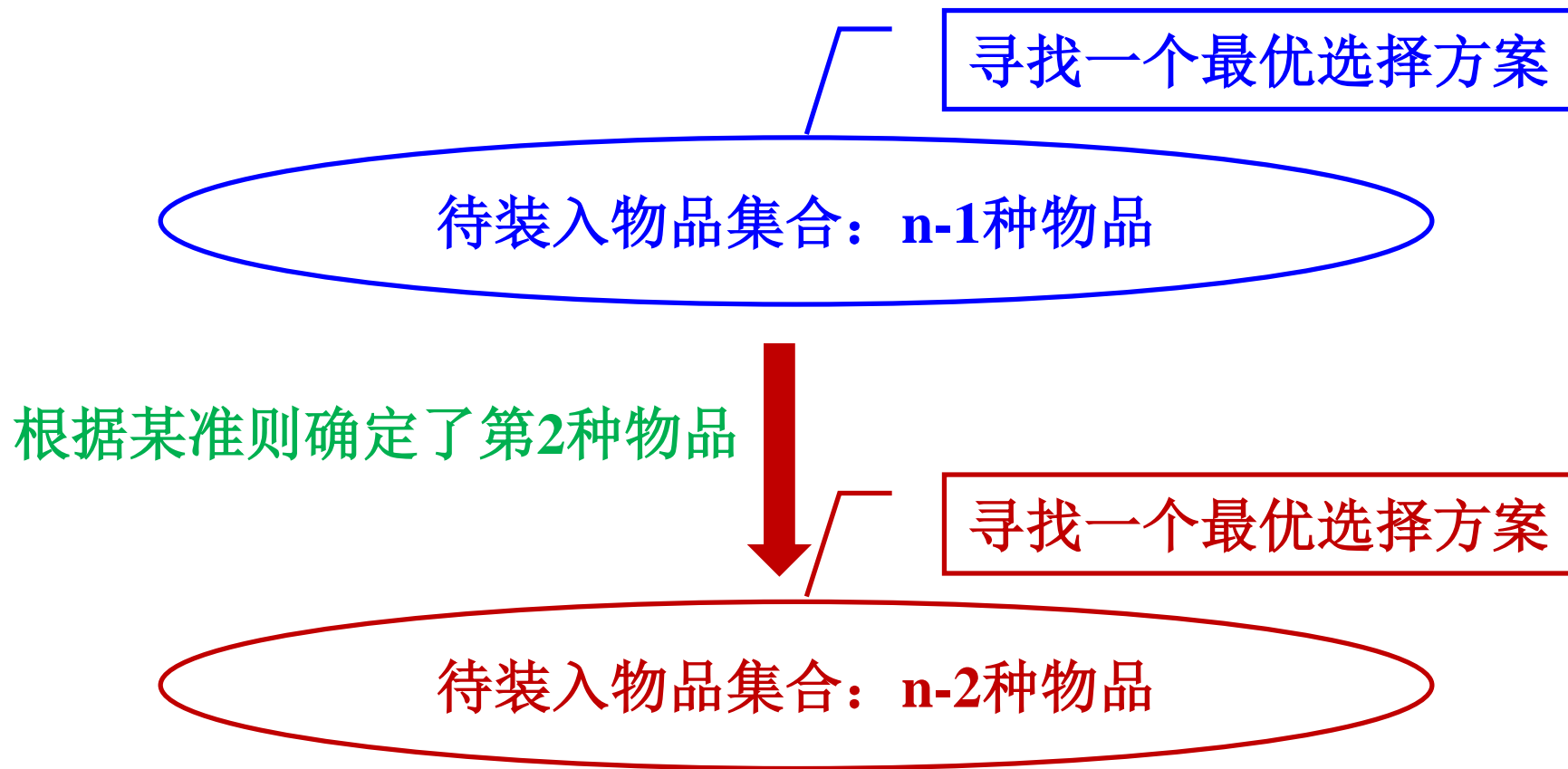
### 【问题分析】

#### (1) 划分子问题

原问题



## 11.2.3 背包问题



重复上述过程，直至找不到可以装入的物品为止。

## 11.2.3 背包问题

### (2) 设计贪心准则

在背包问题中，一种物品可以取一部分装入背包，因此可以将整个背包装满，即能够装入背包的物品总质量必然等于背包的容量。

显然，为了使装入物品后的背包价值尽可能大，应使装入物品的单位质量价值尽可能大。

贪心准则：优先选择单位质量价值大的物品，若该物品的质量不超过背包剩余容量，则全部装入该物品，否则装入该物品的一部分使背包装满。

## 11.2.3 背包问题

### 【算法实现】

一维结构体数组A记录物品信息，其中，A[i]表示第i种物品，A[i].no表示物品的编号，A[i].w表示物品的质量，A[i].v表示物品的价值，A[i].p表示物品单位质量的价值（ $1 \leq i \leq n$ ）。

一维数组x记录物品的装入数量，其中，x[i]记录第i种物品的装入数量。

value记录装入物品后的背包价值。

## 11.2.3 背包问题

```
struct Item {  
  
    int no;           //物品编号  
  
    double w;         //物品质量  
  
    double v;         //物品价值  
  
    double p;         //物品单位质量的价值  
  
    //重载小于运算符，用于按照物品的单位质量价值递减排序  
  
    bool operator<(const Item &s) const  
  
    { return p > s.p; }  
  
};
```

## 11.2.3 背包问题

```
double Knapsack(Item *A, int n, double c, double *x) {  
    double value = 0;  
    int i;  
    sort(A + 1, A + n + 1);  
    for(i = 1; i <= n; i++) x[i] = 0;  
    for(i = 1; i <= n && A[i].w < c; i++) {  
        x[i] = 1;  
        value += A[i].v;  
        c -= A[i].w;  
    }  
    if(c > 0)  
    {    x[i] = c / A[i].w;    value += x[i] * A[i].v;    }  
    return value;  
}
```

S1: 将物品按照单位质量的价值递减排序。

## 11.2.3 背包问题

```
double Knapsack(Item *A, int n, double c, double *x) {  
    double value = 0;  
    int i;  
    sort(A + 1, A + n + 1);  
    for(i = 1; i <= n; i++) x[i] = 0;  
    for(i = 1; i <= n && A[i].w < c; i++) {  
        x[i] = 1;  
        value += A[i].v;  
        c -= A[i].w;  
    }  
    if(c > 0)  
    {    x[i] = c / A[i].w;    value += x[i] * A[i].v;    }  
    return value;  
}
```

S2: 初始化标记数组。



## 11.2.3 背包问题

```
double Knapsack(Item *A, int n, double c, double *x) {  
    double value = 0;  
    int i;  
    sort(A + 1, A + n + 1);  
    for(i = 1; i <= n; i++) x[i] = 0;  
    for(i = 1; i <= n && A[i].w < c; i++) {  
        x[i] = 1;  
        value += A[i].v;  
        c -= A[i].w;  
    }  
    if(c > 0)  
    {    x[i] = c / A[i].w;    value += x[i] * A[i].v;    }  
    return value;  
}
```

S3: 对第 $i(=1, 2, \dots, n)$ 种物品, 若该物品的质量不超过背包剩余容量, 则全部装入第 $i$ 种物品, 更新 $x[i]$ 、 $value$ 和背包剩余容量。

## 11.2.3 背包问题

```
double Knapsack(Item *A, int n, double c, double *x) {  
    double value = 0;  
    int i;  
    sort(A + 1, A + n + 1);  
    for(i = 1; i <= n; i++) x[i] = 0;  
    for(i = 1; i <= n && A[i].w < c; i++) {  
        x[i] = 1;  
        value += A[i].v;  
        c -= A[i].w;  
    }  
    if(c > 0)  
    {    x[i] = c / A[i].w;    value += x[i] * A[i].v;    }  
    return value;  
}
```

S4: 若第*i*种物品的质量超过背包剩余容量，则取部分装入背包使背包装满，更新*x[i]*和*value*。

## 11.2.3 背包问题

```
double Knapsack(Item *A, int n, double c, double *x) {  
    double value = 0;  
    int i;  
    sort(A + 1, A + n + 1);  
    for(i = 1; i <= n; i++) x[i] = 0;  
    for(i = 1; i <= n && A[i].w < c; i++) {  
        x[i] = 1;  
        value += A[i].v;  
        c -= A[i].w;  
    }  
    if(c > 0)  
    {    x[i] = c / A[i].w;    value += x[i] * A[i].v;    }  
    return value;  
}
```

S5: 返回背包价值。

## 11.2.3 背包问题

### 【算法分析】

算法的时间主要花费在排序上。

`sort`函数的时间复杂度为 $O(n\log n)$ ，因此整个算法的时间复杂度为 $O(n\log n)$ 。

## 11.2.3 背包问题

### 【一个简单实例的求解过程】

例如，已知一个容量 $c=115$ 的背包和下表所示的6种物品信息。

| 编号 (no)    | 1  | 2  | 3  | 4  | 5  | 6   |
|------------|----|----|----|----|----|-----|
| 质量 ( $w$ ) | 10 | 20 | 30 | 40 | 50 | 60  |
| 价值 ( $v$ ) | 15 | 40 | 90 | 30 | 60 | 150 |

将物品按照单位质量价值递减排序后的结果如下：

| 序号 ( $i$ )     | 1  | 2   | 3  | 4   | 5   | 6    |
|----------------|----|-----|----|-----|-----|------|
| 编号 (no)        | 3  | 6   | 2  | 1   | 5   | 4    |
| 质量 ( $w$ )     | 30 | 60  | 20 | 10  | 50  | 40   |
| 价值 ( $v$ )     | 90 | 150 | 40 | 15  | 60  | 30   |
| 单位质量价值 ( $p$ ) | 3  | 2.5 | 2  | 1.5 | 1.2 | 0.75 |

## 11.2.3 背包问题

|                |    |     |    |     |     |      |
|----------------|----|-----|----|-----|-----|------|
| 序号 ( $i$ )     | 1  | 2   | 3  | 4   | 5   | 6    |
| 编号 (no)        | 3  | 6   | 2  | 1   | 5   | 4    |
| 质量 ( $w$ )     | 30 | 60  | 20 | 10  | 50  | 40   |
| 价值 ( $v$ )     | 90 | 150 | 40 | 15  | 60  | 30   |
| 单位质量价值 ( $p$ ) | 3  | 2.5 | 2  | 1.5 | 1.2 | 0.75 |

设 $X=(x_1, x_2, x_3, x_4, x_5, x_6)$ 记录背包问题的最优解，value记录背包的价值，求解过程如下：

- (1) 考察序号为1的物品， $w_1 < c$ ，故 $x_1=1$ ，value=90， $c=c-w_1=85$ 。
- (2) 考察序号为2的物品， $w_2 < c$ ，故 $x_2=1$ ，value=240， $c=c-w_2=25$ 。
- (3) 考察序号为3的物品， $w_3 < c$ ，故 $x_3=1$ ，value=280， $c=c-w_3=5$ 。
- (4) 考察序号为4的物品， $w_4 > c$ ，故 $x_4=c/w_4=0.5$ ，value=287.5， $c=c-w_4x_4=0$ 。

背包问题的最优解 $X=(1, 1, 1, 0.5, 0, 0)$ ，即全部装入编号为3、6和2的物品，装入1/2的编号为1的物品，能够获得最大的背包价值287.5。

## 11.2.3 背包问题

### 【算法的正确性证明】

假设所有物品已经按照单位质量的价值递减排序，形成一个有序的物品序列，在序列中，物品的序号为1, 2, 3..... $n$ 。

设 $X=(x_1, x_2, \dots, x_n)$ 是应用贪心算法找到的一个解，证明 $X$ 是背包问题的最优解。

若所有 $x_i=1(i=1, 2, \dots, n)$ ，则 $X$ 显然是背包问题的最优解。

否则，设 $\min$ 是满足 $x_i < 1$ 的最小下标，根据贪心算法的解的特殊性可知：当 $i < \min$ 时， $x_i=1$ ，当 $i > \min$ 时， $x_i=0$ 。此时， $\sum_{i=1}^n w_i x_i = c$ ，装入物品后的背包价值为  $v(X)=\sum_{i=1}^n v_i x_i$ 。欲证明 $X$ 在当前情况下是背包问题的最优解，只需证明不存在另一个能够使背包价值比 $v(X)$ 更大的可行解。

## 11.2.3 背包问题

假设存在背包问题的另一个可行解 $Y=(y_1, y_2, \dots, y_n)$ , 其必然满足约束条件 $\sum_{i=1}^n w_i y_i \leq c$ , 从而

$$\sum_{i=1}^n w_i (x_i - y_i) = \sum_{i=1}^n w_i x_i - \sum_{i=1}^n w_i y_i \geq 0$$

设 $Y$ 对应的背包价值为 $v(Y) = \sum_{i=1}^n v_i y_i$ , 可得

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

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

当 $i > \min$ 时, 由于 $x_i = 0$ , 因此 $x_i - y_i \leq 0$ , 且 $v_i/w_i \leq v_{\min}/w_{\min}$ 。

当 $i = \min$ 时,  $v_i/w_i = v_{\min}/w_{\min}$ 。



## 11.2.3 背包问题

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

$$= \sum_{i=1}^{\min-1} w_i \frac{v_i}{w_i} (x_i - y_i) + \sum_{i=\min}^{\min} w_i \frac{v_i}{w_i} (x_i - y_i) + \sum_{i=\min+1}^n w_i \frac{v_i}{w_i} (x_i - y_i)$$

$$\geq \sum_{i=1}^{\min-1} w_i \frac{v_{\min}}{w_{\min}} (x_i - y_i) + \sum_{i=\min}^{\min} w_i \frac{v_{\min}}{w_{\min}} (x_i - y_i) + \sum_{i=\min+1}^n w_i \frac{v_{\min}}{w_{\min}} (x_i - y_i)$$

$$= \frac{v_{\min}}{w_{\min}} \sum_{i=1}^n w_i (x_i - y_i) \geq 0$$

即 $v(X) \geq v(Y)$ ，表明不存在另外的可行解能够形成比 $v(X)$ 更大的背包价值。所以，应用贪心算法寻找到的解 $X$ 是背包问题的最优解。



## 11.3 小结

## 11.3 小结

- 贪心法在求解问题时，不从整体最优进行考虑，总是做出当前看来最好的选择，即做出某种意义上的局部最优选择。
- 贪心法未必能够得到问题的最优解，可以应用贪心法求解的问题需要具有贪心选择性质和最优子结构性质。
- 设计贪心算法的关键在于选择正确的贪心准则，并且需要证明贪心算法的正确性，即证明设计的贪心算法可以得到问题的最优解。



定一个

小目标

Do not give up

只要心里还存着不甘心  
就还不到放弃的时候



小目标

宽约75厘米X高约42厘米