◆**贪心算法**:这种算法在每一步都作出在当前看来最好的选择。也就是说，它总是做出局部最优选择，希望从局部的最优选择得到整体最优解。

分子为1的分数

例如: 埃及分数

1. **问题：** 要求把一个真分数表示为最少的埃及分数之和的形式。5/7=1/2+1/7+1/14

2. **想法：** 每次选择真分数包含的最大埃及分数。

3. **说明：** 真分数A/B，其中B除以A的整数部分C，余数D，有：B=A*C+D

$$B/A = C + D/A < C+1$$

$$A/B > 1/(C+1)$$

即，1/(C+1)是真分数A/B的最大埃及分数

算法 1 埃及分数

输入： 真分数的分子A和分母B

输出： 最少的埃及分数之和

考察真分数m/n,
时间复杂度O(m)

- 1. E=B/A+1;
- 2. 输出1/E;
- 3. A=A*E-B; B=B*E;
- 4. 求A和B的最大公约数R，如果R不为1，则将A和B同时除以R;
- 5. 如果A等于 1，则输出1/B，算法结束；否则转向步骤1重复执行。

- 贪心算法不能保证对所有问题都得到整体最优解，但对许多问题它能产生整体最优解。如单源最短路经问题，最小生成树问题等。
- 在一些情况下，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。

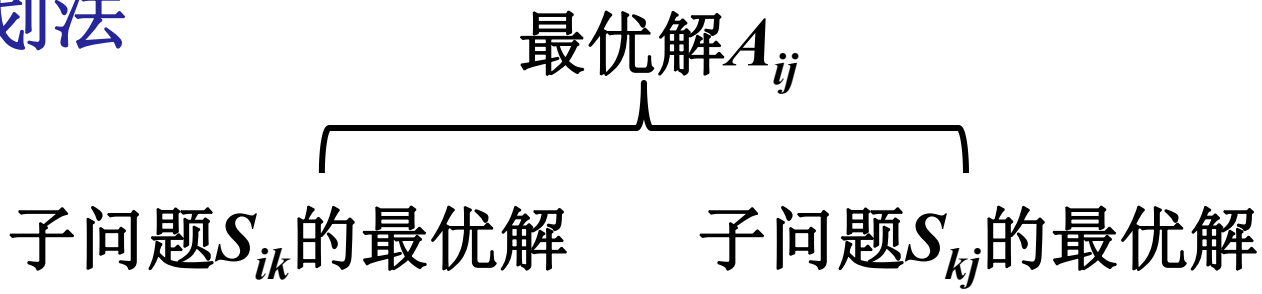
算法思想

采用逐步构造最优解的方法。在每个阶段，都作出一个看上去最优的决策（在一定的标准下）。决策一旦作出，就不可再更改。

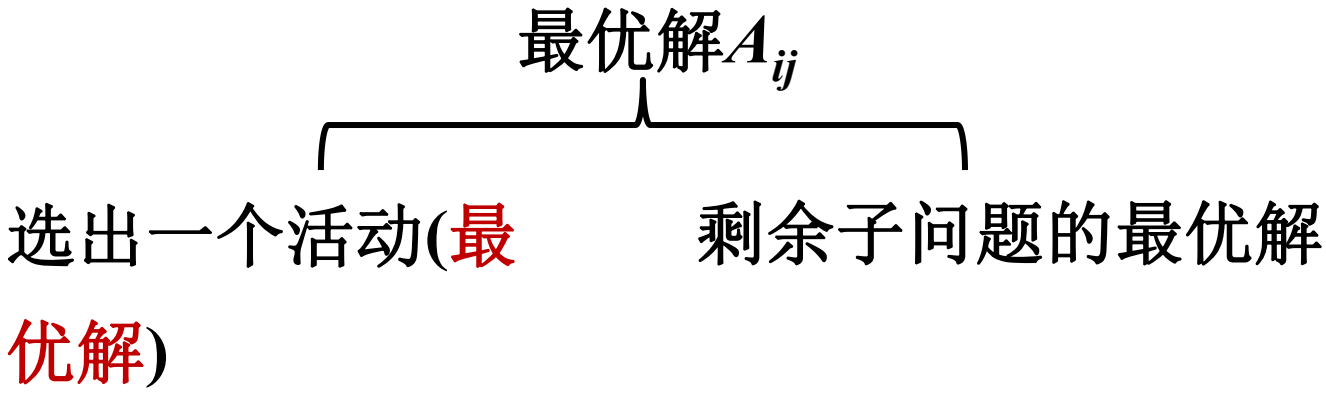
4.1 活动安排问题

- 有n个活动E={1,2,…,n},其中每个活动要使用同一资源,同一时间只允许一个活动使用该资源.
- 每个活动i都有一个开始时间s_i,和一个结束时间f_i .
- 如果选择活动i,则它在时间区间$[s_i, f_i)$内占用该资源;若区间 $[s_i, f_i)$ 与$[s_j, f_j)$不相交, 则称活动i与j是相容的(兼容的).
- 活动安排问题是要求在所给的活动集合中选出最大相容活动子集.

动态规划法


最优解A_{ij}
子问题S_{ik}的最优解
子问题S_{kj}的最优解

改变方法：


最优解A_{ij}
选出一个活动(最优解)
剩余子问题的最优解

[直观想法]

- 在安排时应该将结束时间早的活动尽量往前安排，好给后面的活动安排留出更多的空间，从而达到安排最多活动的目标。
- 贪心准则应当是：在未安排的活动中挑选结束时间最早的活动安排。

[算法思路]

1.将活动按结束时间排序,得到活动集

E={e_1,e_2…e_n};

2.先将e_1选入结果集合A中，即A={e_1};

3.依次扫描每一个活动 e_i: 如果e_i的s_i >最后一个选入A的活动e_j的 f_j , 则将e_i选入A中 , 否则放弃e_i。

[例]

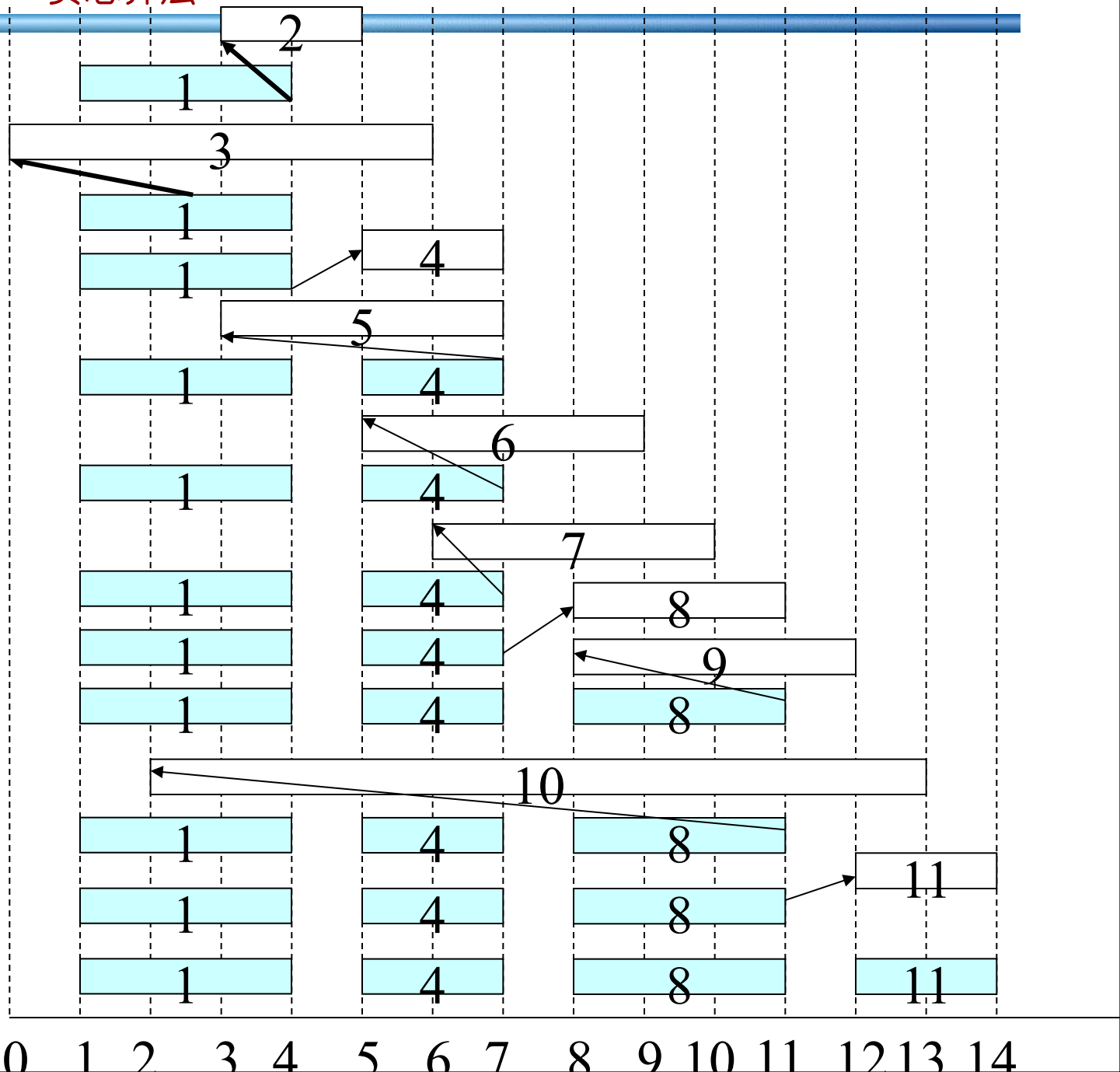
◆设待排的11个活动起止时间按结束时间的非减序排列）（预处理过程）

i	1	2	3	4	5	6	7	8	9	10	11
s[i]	1	3	0	5	3	5	6	8	8	2	12
f[i]	4	5	6	7	8	9	10	11	12	13	14

最大相容活动子集(1, 4, 8, 11),

也可表示为等长n元数组:(1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1)

i	si	fi
1√	1	4
2	3	5
3	0	6
4√	5	7
5	3	8
6	5	9
7	6	10
8√	8	11
9	8	12
10	2	13
11√	12	14



[活动安排问题贪心算法]

```
Template <class T >
void GreedySelector( int n, T s[ ], T f[ ], bool A[ ] )
{ A[1] =true;
   int j =1;
  //从第二个活动开始检查是否与前一个相容
  for (int i=2; i<=n;i+ +) {
    if ( s[i]>= f [j] ) {
        A[i] = true;
        j=i;}
     else A[i] = false;
 } }
```

各活动的起始时间和结束时间存储于数组s和f中且按结束时间的非减序排列

存放所选择的活动

- True:选择
- False:未选择

[算法分析]

- ✓ 当输入的活动已按结束时间的非减序排列时，算法只需O(n)的时间来安排n个活动，使最多的活动能相容地使用公共资源。
- ✓ 当输入的活动未按结束时间排序时，首先选择一个算法已按结束时间排序，最好的排序算法需要O(nlogn)时间，然后再安排n个活动。

总结：T(n)=O(n) (排序时)

T(n)=O(nlogn) (未排序时)

- 贪心算法并不总能求得问题的整体最优解，但对于活动安排问题，贪心算法GreedySelector却总能求得整体最优解，即它最终所确定的相容活动集合A的规模最大。
- 证明:数学归纳法证明。

贪心算法设计步骤：

- ✓1、对其做出一次选择（找到一个最优解），剩下一个子问题需要求解；
- ✓2、证明做出贪心选择后，原问题总是存在最优解，即贪心选择总是安全的；
- ✓3、证明做出贪心选择后，剩余的子问题满足性质：其最优解与贪心选择组合即可得到最优子结构。

4.2 贪心算法的基本要素

- 贪心算法通过一系列的选择来得到一个问题的解。它所做的每一个选择都是当前状态下某种意义的最好选择，即贪心选择。
- 希望通过每次所做的选择导致最终结果是问题的一个最优解。
- 希望从局部的最优选择得到整体最优解。

一个具体的问题，怎么知道是否可用贪心算法解此问题？以及能否得到问题的最优解呢?

- 从许多可以用贪心算法求解的问题中看到这类问题一般具有2个重要的性质：贪心选择性质和最优子结构性质。

1. 贪心选择性质

- 通过做出局部最优（贪心）选择来构造全局最优解。直接做出当前问题中看来最优的选择，而不必考虑子问题的解。
- 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。

- 贪心算法可行的第一个基本要素，也是贪心算法与动态规划算法的主要区别。
- 动态规划方法，每一步骤都要进行一次选择，但选择通常依赖于子问题的解。
- 贪心算法，进行选择时可能依赖之前做出的选择，但不依赖任何子问题的解。

必须证明每个步骤做出的贪心选择能生成全局最优解.

这种证明通常考察某个问题的整体最优解，然后用贪心选择替换某个其它选择来修改此解，从而得到相似但更小的问题。然后，用数学归纳法证明，通过每一步做贪心选择，最终可以得到问题的整体最优解。

定理：考虑任意非空子问题S_k，令a_m是S_k中结束时间最早的活动，则a_m在S_k的某个最大兼容活动子集中。

证明：令A_k是S_k的一个最大兼容活动子集，且a_j是A_k中结束时间最早的活动。

✓若$a_j=a_m$，则证明a_m在S_k的某个最大兼容活动子集中。

✓若$a_j\neq a_m$，令集合$A_k' = A_k - \{a_j\} \cup \{a_m\}$，即将$A_k$中的$a_j$替换$a_m$

因为A_k中的活动都是不相交的，a_j是A_k中结束时间最早的活动，而$f_m \leq f_j$，所以A_k'中的活动都是不相交的。

由于$|A_k'| = |A_k|$，因此得出结论A_k'也是S_k的一个最大活动子集，且它包含a_m。

2. 最优子结构性质

- 当一个问题的最优解包含其子问题的最优解时，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算法求解的关键特征。

3. 贪心算法与动态规划算法的差异

- 相同点：都具有最优子结构性质
- 区别：

	动态规划法	贪心算法
选择特征	往往依赖于相关子问题的解。只有解出相关子问题才能作出选择。	当前状态下作出最好选择，即局部最优选择，但不依赖于子问题的解
设计特征	自底向上	自顶向下

- 对于具有最优子结构的问题应该选用贪心算法还是动态规划算法求解?
- 是否能用动态规划算法求解的问题也能用贪心算法求解?

背包问题

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。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?

在选择物品i装入背包时，可以选择物品i的部分，而不一定要全部装入背包，$1 \le i \le n$。

不允许重复装入。

设x_i表示物品i装入背包的情况，根据问题的要求，有如下约束条件和目标函数：

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

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

于是，背包问题归结为寻找一个满足约束条件，并使目标函数达到最大的解向量$X=(x_1, x_2, \cdots, x_n)$。

贪心选择：

✓每次捡最轻的物品装；

只考虑到多装些物品，但由于单位价值未必高，总价值不能达到最大；

✓每次捡价值最大的装；

每次选择的价值最大，但同时也可能占用了较大的空间，装的物品少，未必能够达到总价值最大

✓每次装包时既考虑物品的重量又考虑物品的价值，也就是说每次捡单位价值最大的装。

确实能够达到总价值最大。

基本步骤：

✓首先计算每种物品单位重量的价值v_i/w_i，然后，依贪心选择策略，将尽可能多的单位重量价值最高的物品装入背包。

✓若将这种物品全部装入背包后，背包内的物品总重量未超过C，则选择单位重量价值次高的物品并尽可能多地装入背包。

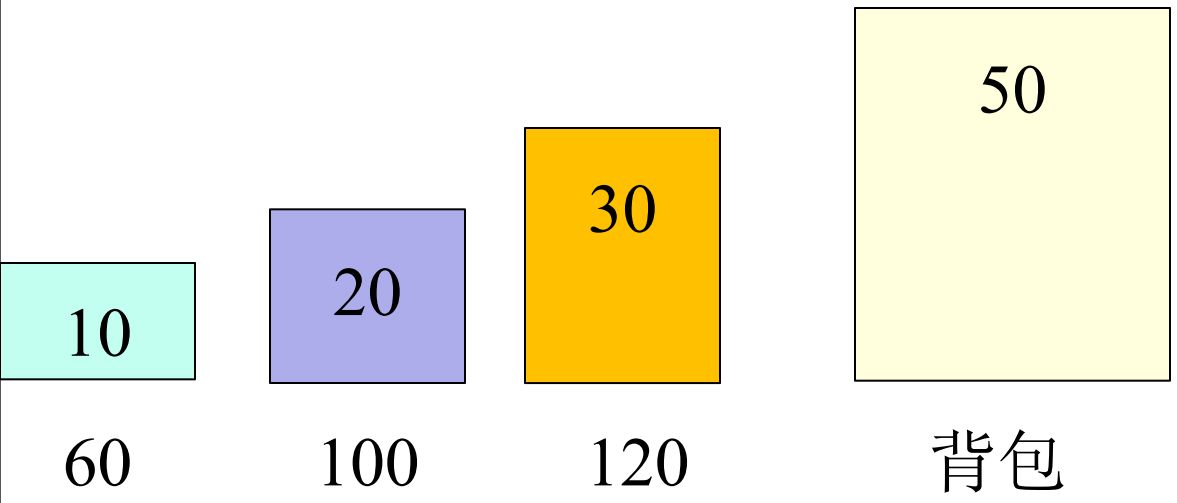
✓依此策略处理，直到背包装满为止。

```
void  Knapsack(int n, float M, float v[], float  w[], float  x[] )
 //价值数组v[1:n]、重量数组w[1:n];
 //它们元素的排列顺序满足v[i]/w[i]≥v[i+1]/w[i+1];
 // M是背包容量，x是解向量.
 {   float c=M ;     // 使背包剩余容量初始化为M
     int   i;
     Sort(n,v,w);
     for( i=1; i<=n; i++ ) x[i]=0;   // 将x
     for (i=1; i<=n; i++)  {
       if w[i] >c   break;
       x[i]=1;  c= c-w[i];
      }
     if ( i≤n  )    x[i]= c / w[i];
 }
```

将各种物品依其单位重量的价值从大到小排序。算法的计算时间上界为O(nlogn)。

[例]

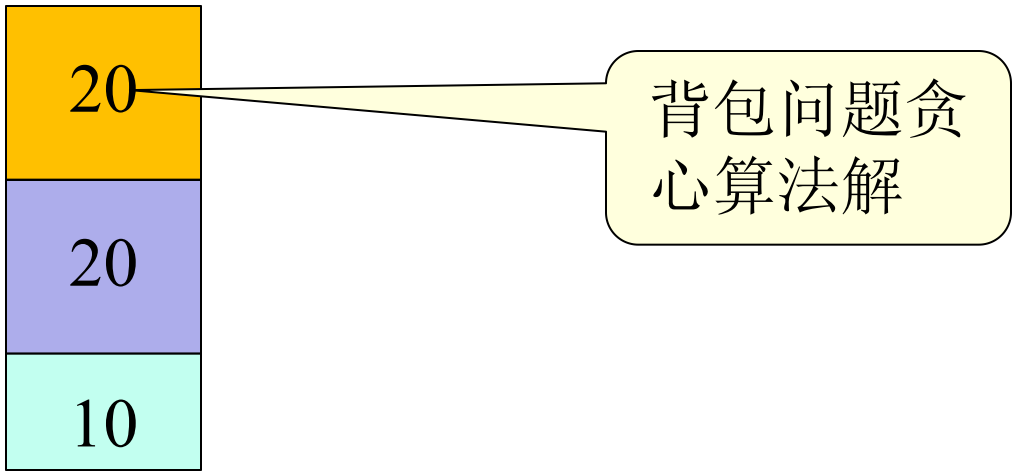
- 一个正在抢劫商店的小偷发现3种物品，物品1重10千克；价值60万元；物品2重20千克，价值100万元；物品3重30千克；价值120万元，他的背包容量为50千克
- 他应该拿哪些商品呢？



商品1单位重量价值 6

商品2单位重量价值 5

商品3单位重量价值 4



(d) 60+100+80

0-1背包问题：

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。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?

在选择装入背包的物品时，对每种物品 i只有2种选择，即装入背包或不装入背包。不能将物品i装入背包多次，也不能只装入部分的物品i。

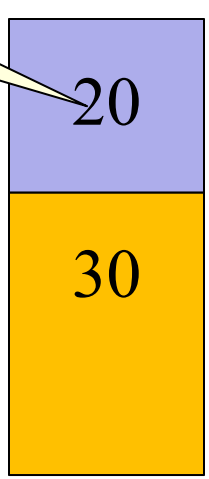
商品1单位重量价值 6

商品2单位重量价值 5

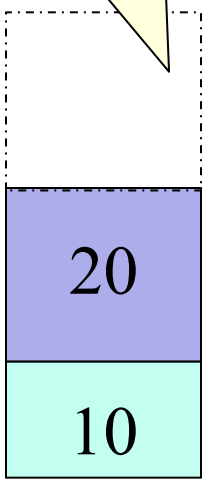
商品3单位重量价值 4

0-1背包问题的最优解

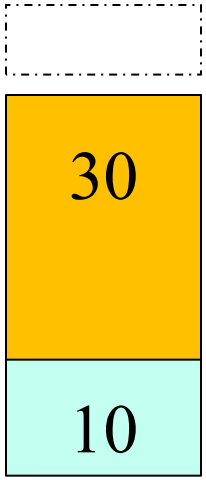
0-1背包问题贪心算法解



(a) 120+100



(b) 60+100



(c) 60+120

分析：

- 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。

4.3 最优装载

[问题描述]

有一批集装箱要装上一艘载重量为c的轮船。其中集装箱i的重量为w_i。最优装载问题要求确定在装载体积不受限制的情况下，将尽可能多的集装箱装上轮船。

满足 $\sum_{i=1}^{n} w_i x_i \le c$ $\max \sum_{i=1}^{n} x_i$

1.算法描述

最优装载问题可用贪心算法求解。采用重量最轻者先装的贪心选择策略，可产生最优装载问题的最优解。

```
void Loading(int x[], Type  w[], float c, int n)
    {
        int *t= new int [n+1];
        Sort.(w,t,n);
        for (int i = 0; i < n && w[t[i]] <= c; i++) {
          x[t[i]] = 1;
          c -= w[t[i]];
         }
    }
```

x[i]记载第i个集装箱是否装入，x[i]=1装入，x[i]=0未装入

t[i]中存储第i轻集装箱在原来序列中的下标

[例]

物品重量分别为15，10，27，18，船载重为50

定义：w[ ]={15, 10, 27, 18}　物品重量

T [ ]={ 1 , 0, 3, 2}　　物品从轻到重排序后序号

C=50　　船载重

贪心算法计算步骤：

w[T[0]]=10<c, c= c-10=40

w[T[1]]=15<c, c=c-15=25

w[T[2]]=18<c, c=c-18=7

w[T[3]]=27>c

4.7 多机调度问题

[问题描述]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，作业i所需时间为t_i。约定，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，作业不能拆分成更小的子作业。

该问题是NP完全问题，到目前为止还没有有效的解法。用贪心选择策略有时可设计出较好的近似算法。

[贪心近似算法]

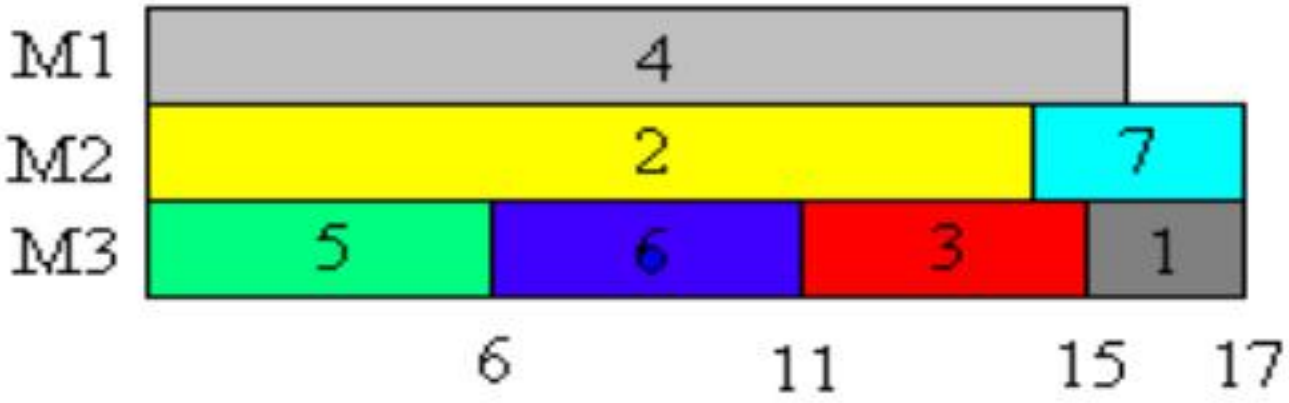
采用最长处理时间作业优先的贪心策略.

- 当$n \le m$时, 只要将机器i的$[0, t_i]$时间区间分配给作业i即可;
- 当$n > m$时, 将n个作业依其所需的处理时间从大到小排序，然后依次将作业分配给空闲的处理机。

例题

设7个独立作业{1, 2, 3, 4, 5, 6, 7}由3台机器M1，M2和M3加工处理。各作业所需的处理时间分别为:{ 2, 14, 4, 16, 6, 5, 3 }。

按算法greedy产生的作业调度如下图所示:



所需的加工时间为17

多机调度问题的贪心近似算法

```
class JobNode {
    friend void Greedy(JobNode * , int, int);
    friend void main(void);
    public:
        operator int () const {return time; }
    private:
        int ID,
            time; };
class MachineNode {
    friend void Greedy(JobNode *, int, int);
    public:
        operator int( ) const { return avail; }
    private:
        int ID,
            avail; }
```

作业结点类

机器结点类

```
template<class Type>
void Greedy(Type a[]，int n，int m)
{if (n<=m){
   cont<<”为每个作业分配一台机器.“<<endl
   return；}
Sort(a，n)；    //将作业按时间从大到小排序
MinHeap<MachineNode>H(m)；
MachineNode x；
for(int i=1；i<=m; i++){
  x．avail=0；
  x．ID=i;
  H．Insert(x)；}
for(int i=n；i>=1；i--){   //作业分配过程
   H．DeleteMin(x)；    //从堆中取出堆顶x机器
   cout<<”将机器”<<x．ID<<“从” <<x．avail<<"到”
        <<(x．avail十a[i]．time)
        <<”的时间段分配给作业”<<a[i]. ID<<endl；
   x．avail+=a[i]．time；  //机器的时间分配
  H．insert(x)；}}    //重新插入,调整堆
```

如果机器数大于作业数直接分配

按照机器结点编号顺序建立堆

- 时间复杂度分析
  - n<=m 时间O(1)
  - n>m 排序耗时O ($n\log n$)

    初始化堆O(m)

    DeleteMin 和Insert 耗时O ($n\log m$)

  共需时间为： O ($n\log n$+ $n\log m$)=O($n\log n$)