算法 设计与分析

算法设计与分析

第四章 贪心算法

分硬币问题

- 条件：有以下几种面额的硬币：

1毛1分	5分	1分

- 问题：给顾客找一定数额的钱
- 目标：用的硬币数最少

贪心算法：什么都不想，每次都捡面值最大的拿，直到余额不足以使用最大的面额时，再考虑使用小一点的。

- 例如 找 1毛6分：先1毛1分，再5分；

  找 1毛5分：先1毛1分，再4个1分。（然而3个5分是最好的选择）

由此可见，贪心算法并不是总能得到最优解。

Chapter 4

4.1 贪心算法概述

基本思想

4.1.1 贪心算法的基本思想

- 贪心法的基本思路是在对问题求解时总是做出在当前看来是最好的选择，也就是说贪心法不从整体最优上加以考虑，所做出的仅是在某种意义上的局部最优解。
- 方法的“贪婪性”反映在对当前情况总是作最大限度的选择，即贪心算法总是做出在**当前看来**是最好的选择。

4.1 贪心算法概述

基本思想

4.1.1 贪心算法的基本思想

贪心法从问题的某一个初始解{}出发，采用逐步构造最优解的方法向给定的目标前进，每一步决策产生n-元组解（x_0，x_1，…，x_{n-1}）的一个分量。

贪心法每一步上用作决策依据的选择准则被称为最优量度标准（或**贪心准则**），也就是说，在选择解分量的过程中，添加新的解分量x_k后，形成的部分解（x_0，x_1，…，x_k)不违反可行解约束条件。

每一次贪心选择都将所求问题简化为**规模更小**的子问题，并**期望**通过每次所做的**局部最优选择**产生出一个**全局最优解**。

4.1 贪心算法概述

基本思想

4.1.1 贪心算法的基本思想

【存在问题】在很多情况下，所有局部最优解合起来不一定构成整体问题的最优解，所以贪心法不能保证对所有问题都得到整体最有解。

【解决方法】需要证明该算法的每一步做出的选择都必然最终导致问题的整体最优。

4.1.2 贪心法求解的问题应具有的性质

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。

证明方法：用数学归纳法证明。

先考虑问题的一个整体最优解，并证明：

可以修改这个最优解，使其从贪心选择开始，在做出贪心选择后，原问题可以转化为规模较小的类似问题，通过每一步的贪心选择，最后得到问题的整体最优解。

4.1.2 贪心法求解的问题应具有的性质

2. 最优子结构性质

如果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。

证明方法：用反证法证明。

首先：假设由这个问题的最优解X导出的子问题的解X'不是最优的。

然后：证明在这个假设下可以构造出比原问题的最优解X更优的解Y，从而引出矛盾。

得证：问题具有最优子结构性质。

4.1 贪心算法概述

基本思想 基本性质

表4-1 动态规划算法和贪心算法的区别

	基本思想	依赖子问题的解	解问题的方向	最优解	复杂程度
贪心选择	贪心选择	否	自顶向下	局部最优	简单有效
动态规划	递归定义填表	是	自底向上	整体最优	较复杂

4.1.3 贪心算法适合的问题

- 贪心算法通常用来解决具有最大值或最小值的优化问题。
- 它是从某一个初始状态出发，根据当前局部而非全局的最优决策，以满足约束方程为条件，以使得目标函数的值增加最快或最慢为准则，选择一个最快地达到要求的输入元素，以便尽快地构成问题的可行解。

4.1.4 贪心算法的基本步骤

- 基本步骤：

(1) 选定合适的贪心选择的标准；

(2) 证明在此标准下该问题具有贪心选择性质；

(3) 证明该问题具有最优子结构性质；

(4) 根据贪心选择的标准，写出贪心选择的算法，求得最优解。

4.1 贪心算法概述

基本思想 基本性质 适合的的问题 基本步骤

贪心法的一般求解过程

贪心法求解问题的算法框架如下：

```
SolutionType Greedy(SType a[],int n)
//假设解向量(x0,x1,…,xn-1)类型为SolutionType，其分量为SType类型
{  SolutionType x={};                  //初始时，解向量不包含任何分量
   for (int i=0;i<n;i++)               //执行n步操作
   {  SType xi=Select(a);              //从输入a中选择一个当前最好的分量
      if (Feasiable(xi))               //判断xi是否包含在当前解中
         solution=Union(x,xi);         //将xi分量合并形成x
   }
   return x;                           //返回生成的最优解
}
```

贪心法求解单源最短路径

【问题描述】

给定一个带权有向图G=（V，E），其中每条边的权是非负实数,另外，给定V中的一个顶点作为源点。现在要计算源点到其他各顶点的最短路径长度。这里路径长度是指路上各边权之和。

这个问题通常称为**单源最短路径问题。**

贪心法求解单源最短路径

Dijkstra算法是一种按路径长度递增序产生各顶点最短路径的贪心算法。

算法步骤：

(1)初始时,S中仅含有源。设u是V的某一个顶点，把从源到u且中间只经过S中顶点的路称为从源到u的特殊路径，并用数组distance记录当前每个顶点所对应的最短特殊路径长度。

(2)每次从集合V-S中选取到源点v0路径长度最短的顶点w加入集合S。S中每加入一个新顶点w，都要修改顶点v0到集合V-S中节点的最短路径长度值。

集合V-S中各节点新的最短路径长度值为原来最短路径长度值与顶点w的最短路径长度加上w到该顶点的路径长度值中的较小值。

(3)直到S包含了所有V中顶点，此时，distance就记录了从源到所有其他顶点之间的最短路径长度。

贪心法求解单源最短路径

【贪心策略】设置两个顶点集合S和V-S，集合S中存放已经找到最短路径的顶点，集合V-S中存放当前还未找到最短路径的顶点。

设置顶点集合S，并不断地作贪心选择来扩充这个集合。一个顶点属于集合S当且仅当从源到该顶点的最短路径长度已知。

贪心法求解单源最短路径

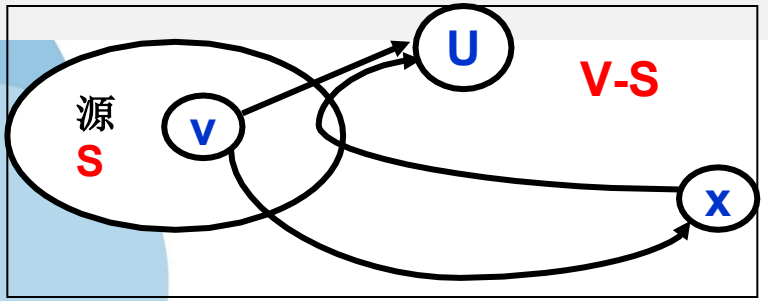
1）贪心选择性质

Dijkstra算法所作的贪心选择是从V-S中选择具有最短路径的顶点u，从而确定从源到u的最短路径长度distance [u]。这种贪心选择为什么会导致最优解呢？换句话说，为什么从源到u没有更多的其他路径呢？

事实上，如果存在一条从源到u且长度比distance [u]更短的路，设这条路初次走出S之外到达的顶点为x∈V-S，如下图所示。

在这条路径上，分别记cost(v,x),cost(x,u)和cost(v,u)为顶点v到顶点x，顶点x到顶点v到顶点u的路长，那么distance [x]<= cost (v,x)，

cost (v,x)+ cost (x,u)= cost (v,u)< distance [u]。利用边权的非负性，可知cost(x,u)>=0，从而推distance [x]< distance [u]。此为矛盾。这就证明了distance [u]是从源到顶点u的最短路径长度。



贪心法求解单源最短路径

2）最优子结构性质

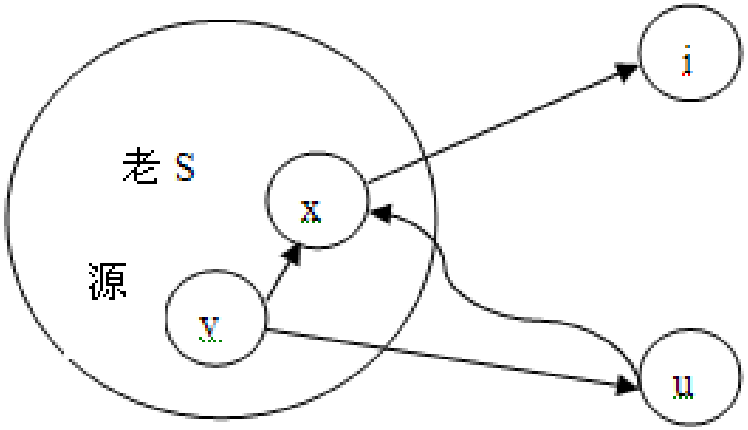
算法中确定的distance[u]确实是源点到顶点u的最短特殊路径长度。为此，只要考察算法在添加u到S中后,distance [u]的值所引起的变化。当添加u之后，可能出现一条到顶点i特殊的新路。

第一种情况：从u直接到达i。如果cost [u][i] + distance [u]< distance [i],则 cost [u][i] + distance [u]作为distance [i]新值。

4.10 单源最短路径问题

2）最优子结构性质

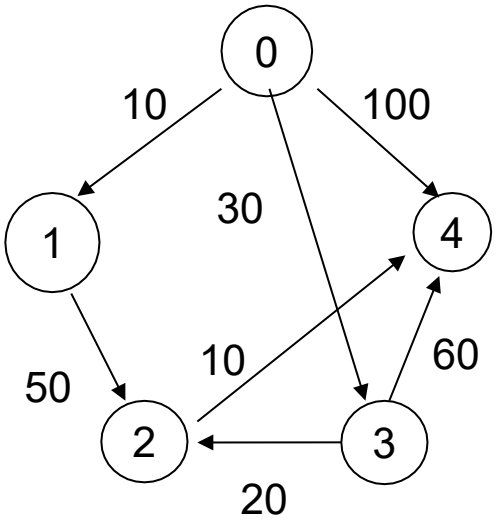
第二种情况：从u不直接到达i，如下图所示。回到老S中某个顶点x，最后到达i。当前distance [i]的值小于从源点经u和x，最后到达i的路径长度。因此，算法中不考虑此路。由此，不论distance [u]的值是否有变化，它总是关于当前顶点集S到顶点u的最短特殊路径长度。



贪心法求解单源最短路径

例

- 对以下有向图，应用Dijkstra算法计算从源顶点0到其他顶点间最短路径，按其算法步骤执行过程如表所示。



贪心法求解单源最短路径示例

Dijkstra算法的迭代过程

迭代	S	u	distance [1]	distance [2]	distance [3]	distance [4]
初始	{0}	-	10		30	100
1	{0,1}	1	10	60	30	100
2	{0,1,3}	2	10	50	30	90
3	{0,1,3,2}	2	10	50	30	60
4	{0,1,3,2,4}	4	10	50	30	60

```
void Dijkstra(int cost[][],int n,int
    v0,int distance[],int prev[])
{
   int *s=new int[n];
   int mindis,dis;
   int i,j,u;
   //初始化
   for(i=0;i<n;i++)
   {
      distance[i]=cost[v0][i];
      s[i]=0;
      if(distance[i]==MAX)
      prev[i] = -1;
      else
      prev[i] = v0;
   }
   distance[v0] = 0;
   s[v0] =1; //标记v0
   //在当前还未找到最短路径的顶点中，
   //寻找具有最短距离的顶点
```

```
for(i=1;i<n;i++)  //每次循环求得一条最短路径
   {
      mindis=MAX;
      u = v0;
      for (j=0;j<n;j++) //求离出发点最近的顶点
      if(s[j]==0&&distance[j]<mindis)
      {
         mindis=distance [j];
         u=j;
      }
      s[u] = 1;//将该点加入S集合
      for(j=0;j<n;j++) //修改递增路径序列（集合）
      if(s[j]==0 && cost[u][j]<MAX)
      {
        dis=distance[u] +cost[u][j];
         // 如果新的路径更短，就替换掉原路径
         if(distance[j]>dis)
         {
            Distance[j] = dis;
            prev[j] = u;
         }
      }
}
```

2、贪心法求解（部分背包问题）

【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（$1 \le i \le n$）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：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。·

2、贪心法求解（部分背包问题）

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

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

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

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$X=\{x_1, x_2, \ldots, x_n\}$。

2、贪心法求解（部分背包问题）

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。

2、贪心法求解（部分背包问题）

- (1)”效益”优先,使每装入一件物品就使背包获得最大可能的效益值增量.

  按物品收益从大到小排序0,1,2

  解为： (x0，x1，x2)=(1，2/15，0)

  收益： 25+24*2/15=28.2

  此方法解非最优解。原因:只考虑当前收益最大,而背包可用容量消耗过快.

- (2)选重量作为量度,使背包容量尽可能慢地被消耗.

  按物品重量从小到大排序:2,1,0;

  解为： (x0，x1，x2)=(0，2/3，1)

  收益： 15+24*2/3=31

  此方法解非最优解。原因:虽然容量消耗慢，但效益没有很快的增加.

2、贪心法求解（部分背包问题）

- (3)选利润/重量为量度，使每一次装入的物品应使它占用的每一单位容量获得当前最大的单位效益。

  按物品的vi/wi重量从大到小排序:1,2,0;

  解为： (x0，x1，x2)=(0，1，1/2)

  收益： 24+15/2=31.5

  此方法解为最优解。可见，可以把vi/wi 作为背包问题的最优量度标准。

2、贪心法求解（部分背包问题）

【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，就可以把它完全装入，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。

2、贪心法求解（部分背包问题）

对于下表一个背包问题，$n=5$，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）将单位价值即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1～5编号。

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

（2）设背包余下装入的重量为weight=W。

2、贪心法求解（部分背包问题）

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

（3）从i=1开始，w[1]<weight成立，表明物品1能够完全装入，将其装入到背包中，置x[1]=1，weight=weight-w[1]=70，i增1即i=2。

w[2]<weight成立，表明物品2能够完全装入，将其装入到背包中，置x[2]=1，weight=weight-w[2]=60，i增1即i=3。

w[3]<weight成立，表明物品3能够完全装入，将其装入到背包中，置x[3]=1，weight=weight-w[3]=50，i增1即i=4。

w[4]<weight不成立，且weight>0，表明只能将物品4部分装入，装入比例=weight/w[4]=50/60=80%，置x[4]=0.8。

算法结束，得到X={1，1，1，0.8，0}。

贪心法求解（部分背包问题）算法描述

```
//问题表示
int n=5;
double W=100;                          //限重
struct NodeType
{  double w;
   double v;
   double p;                           //p=v/w
   bool operator<(const NodeType &s) const
   {
       return p>s.p;                   //按p递减排序
   }
};
NodeType A[]={{0},{10,20},{20,30},{30,66},{40,40},{50,60}};
                                                  //下标0不用

//求解结果表示
double V;                              //最大价值
double x[MAXN];
```

贪心法求解（部分背包问题）算法描述

```
void Knap()                          //求解背包问题并返回总价值
{  V=0;                                    //V初始化为0
   double weight=W;                  //背包中能装入的余下重量
   memset(x,0,sizeof(x));            //初始化x向量
   int i=1;

   while (A[i].w<weight)             //物品i能够全部装入时循环
   {  x[i]=1;                        //装入物品i
      weight-=A[i].w;                      //减少背包中能装入的余下重量
      V+=A[i].v;                     //累计总价值
      i++;                           //继续循环
   }

   if (weight>0)                     //当余下重量大于0
   {  x[i]=weight/A[i].w;            //将物品i的一部分装入
      V+=x[i]*A[i].v;                      //累计总价值
   }
}
```

2、贪心法求解（部分背包问题）算法描述

```
void main()
{  printf("求解过程\n");
   for (int i=1;i<=n;i++)                    //求v/w
     A[i].p=A[i].v/A[i].w;
   printf("(1)排序前\n");dispA();
   sort(A+1,A+n+1);                          //A[1..n]排序
   printf("(2)排序后\n"); dispA();
   Knap();
   printf("(3)求解结果\n");                  //输出结果
   printf("    x: [");
   for (int j=1;j<=n;j++)
      printf("%g, ",x[j]);
   printf("%g]\n",x[n]);
   printf("    总价值=%g\n",V);
}
```

贪心法求解（部分背包问题）算法证明

证明贪心法求解完全背包问题具有

1）贪心选择性质

即证明：设背包按其单位价值量vi/wi由高到低排序，(x1,x2,...,xn)是背包问题的一个最优解。

贪心法求解（部分背包问题）算法证明

【算法证明】假设对于n个物品，按v_i/w_i（$1\le i\le n$）值递减排序得到1、2、…、n的序列，即$v_1/w_1 \ge v_2/w_2 \ge \dots \ge v_n/w_n$。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minj是满足$x_{minj}<1$的最小下标。考虑算法的工作方式，

很明显，当$i<minj$时，$x_i=1$，

当$i>minj$时，$x_i=0$；

当$i=minj$时，$0\le x_i<1$ 。

设X的价值为$V(X)=\sum_{i=1}^{n} v_i x_i$ 并且此时的X为最优解。

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 $\sum_{i=1}^{n} w_i y_i \le W$，

从而有 $\sum_{i=1}^{n} w_i (x_i - y_i) = \sum_{i=1}^{n} w_i x_i - \sum_{i=1}^{n} w_i y_i \ge 0$，这个解的价值为$V(Y)=\sum_{i=1}^{n} v_i y_i$。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$i<minj$时，$x_i=1$，

当$i>minj$时，$x_i=0$；

当$i=minj$时，$0<x_i<1$。

X

x_1	x_2	x_3	…	x_{minj-1}	x_{minj}	x_{minj+1}	…	x_{n-1}	x_n
1	1	1	1	1	介于0·1之间	0	0	0	0

y_1	y_2	y_3	…	y_{minj-1}	y_{minj}	y_{minj+1}	…	y_{n-1}	y_n

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

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

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

贪心法求解（部分背包问题）算法证明

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

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

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

<=0

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

<=0

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

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

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。

贪心法求解（部分背包问题）算法证明

证明贪心法求解部分背包问题具有2）最优子结构性质

贪心选择物体1之后，问题转化为背包重量为m-w1*x1，物体集为{物体2，物体3，...，物体n}的背包问题。且该问题的最优解包含在初始问题的最优解中。

对于部分背包和0-1背包这两类问题都具有最优子结构性质。对于部分背包问题，类似地，若它的一个最优解包含物品j，则从该最优解中拿出所含的物品j的那部分重量w，剩余的将是n-1个原重物品1，2，...，j-1，j+1，...，n及**重为wj-w**的物品j中可装入容量为**c-w**的背包且具有最大价值的物品。

【算法分析】对vi/wi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($n\log_2 n$)。

4.2 贪心算法示例

贪心法求解 汽车加油问题

本节内容

1. 问题分析

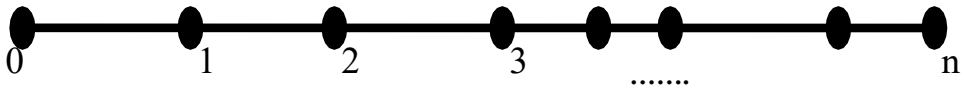
2. 算法分析（两个性质证明）

3. 设计与实现

贪心法求解汽车加油问题

【问题描述】

一辆汽车加满油后可以行驶N千米。旅途中有若干个加油站，如图所示。指出若要使沿途的加油次数最少，设计一个有效的算法，指出应在那些加油站停靠加油（前提：行驶前车里加满油）。



汽车加油图例

贪心法求解汽车加油问题---问题分析

- 由于汽车是由始向终点方向开的，我们最大的麻烦就是不知道在哪个加油站加油可以使我们既可以到达终点又可以使我们加油次数最少。
- **我们可以假设不到万不得已我们不加油，即除非我们油箱里的油不足以开到下一个加油站，我们才加一次油。**
- 在局部找到一个最优的解。每加一次油我们可以看作是一个新的起点（具有最优子结构），用相同的递归方法进行下去。最终将各个阶段的最优解合并为原问题的解得到我们原问题的求解。

【贪心策略】汽车行驶过程中，**应走到自己能走到并且离自己最远的那个加油站**，在那个加油站加油后再按照同样的方法贪心选择下一个加油站。

贪心法求解汽车加油问题---问题分析

- 例4-3
- 在汽车加油问题中，设各个加油站之间的距离为（假设没有环路）：1, 2, 3, 4, 5, 1, 6, 6. 汽车加满油以后行驶的最大距离为7，则根据贪心算法求得最少加油次数为4，需要在3, 4, 6, 7加油站加油，如下图所示：

- 计算过程如下表所示。

	1号	2号	3号	4号	5号	6号	7号	终点
加满油后行驶公里数	1	3	6	4	5	6	6	6
剩余行驶数	6	4	1	3	2	1	1	1
是否加油	0	0	1	1	0	1	1	

4.2 贪心算法示例

贪心法求解汽车加油问题---算法分析

- 1）贪心选择性质

设在加满油后可行驶的N千米这段路程上任取两个加油站A、B，且A距离始点比B距离始点近，则若在B加油不能到达终点那么在A加油一定不能到达终点，如下图：

- 由图可知：因为m+N<n+N，即在B点加油可行驶的路程比在A点加油可行驶的路程要长n-m千米，所以只要终点不在A、B之间且在B的右边的话，根据贪心选择，为使加油次数最少就会选择距离加满油的点远一些的加油站去加油，因此，加油次数最少满足贪心选择性质。

贪心法求解汽车加油问题---算法分析

2）最优子结构性质

- 当一个大问题的最优解包含着它的子问题的最优解时，称该问题具有最优子结构性质。

  (b[1],b[2],.....b[n]) ➔ 整体最优解

  b[1]=1,(b[2],b[3],.....b[n]) ➔ 局部最优解

  每一次加油后与起点具有相同的条件，每个过程都是相同且独立。

贪心法求解汽车加油问题---算法设计

```
int Greedy(int a[],int n,int k) {
    int *b=new int[k+1]; //加油站加油最优解b1~bk
    int num = 0;
    int s=0;    //加满油后行驶的公里数
    for(int i = 0;i < =k;i++) {
        if(a[i] > n) {
            cout<<“no solution\n”;  //a[i]记录的是第i~第i+1个加油点之间的距离。
                                    //a[0]记录的是起点到第1个加油点之间的距离。
            return;
        } }
    for(int i = 0,s = 0;i < =k;i++) {
        s += a[i];
        if(s > n) {
            num++;
            b[i]=1;
            s = a[i];
        }    }
  return num;    }
```

4.2 贪心算法示例

贪心法求解最优装载问题----问题描述

【问题描述】有n个集装箱要装上一艘载重量为W的轮船，其中集装箱i（$1 \le i \le n$）的重量为w_i。

不考虑集装箱的体积限制，现要选出尽可能多的集装箱装上轮船，使它们的重量之和不超过W。

贪心法求解最优装载问题----问题求解

【问题求解】这里的最优解是选出尽可能多的集装箱个数，并采用贪心法求解。

当重量限制为W时，wi越小可装载的集装箱个数越多，所以采用优先选取重量轻的集装箱装船的贪心思路。

贪心法求解最优装载问题----问题求解

对w_i从小到大排序得到$\{w_1, w_2, \ldots, w_n\}$，设最优解向量为$x=\{x_1, x_2, \ldots, x_n\}$，显然，$x_1=1$，则$x'=\{x_2, \ldots, x_n\}$是装载问题$w'=\{w_2, \ldots, w_n\}$，$W'=W-w_1$的最优解，满足**贪心最优子结构性质**。

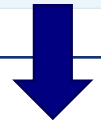
贪心法求解最优装载问题----算法描述

```
//问题表示
int w[]={0,5,2,6,4,3};                  //各集装箱重量,不用下标0的元素
int n=5,W=10;
//求解结果表示
int maxw;                               //存放最优解的总重量
int x[MAXN];                            //存放最优解向量

void solve()                            //求解最优装载问题
{  memset(x,0,sizeof(x));               //初始化解向量
   sort(w+1,w+n+1);                     //w[1..n]递增排序
   maxw=0;
   int restw=W;                         //剩余重量
   for (int i=1;i<=n &&  w[i]<=restw;i++)
   {  x[i]=1;                           //选择集装箱i
      restw-=w[i];                      //减少剩余重量
      maxw+=w[i];                       //累计装载总重量
   }
}
```

贪心法求解最优装载问题----算法示例

```
int w[]={0,5,2,6,4,3};  //各集装箱重量,不用下标0的元素
int n=5,W=10;
```



最优方案

选取重量为2的集装箱

选取重量为3的集装箱

选取重量为4的集装箱

总重量=9

【算法分析】算法的主要时间花费在排序上，时间复杂度为$O(n\log_2 n)$。

4.2 贪心算法示例

贪心法求解最优服务次序问题----问题描述

【问题描述】

最优服务次序问题：设有n个顾客同时等待同一项服务，顾客i 需要的服务时间为ti,1 ≤ i ≤ n，应如何安排这n个顾客的服务次序才能使平均等待时间达到最小。平均等待时间是n 个顾客等待服务时间的总和除以n。

4.2 贪心算法示例

贪心法求解最优服务次序问题----问题分析

假设原问题为T，而我们已经知道了某个最优服务系列，即最优解为

A={t(1),t(2),…，t(n)}(其中t(i)为第i个用户需要的服务时间)，则每个用户等待时间为:

T(1)=t(1); T(2)=t(1)+t(2);

……

T(n)=t(1)+t(2)+t(3)+…+t(n);

那么总等待时间，即最优值为:

TA=n*t(1)+(n-1)*t(2)+…+(n+1-i)*t(i)+…+2*t(n-1)+t(n)；

由于平均等待时间是n个顾客等待时间的总和除以n，故本题实际上就是求使顾客等待时间的总和最小的服务次序。

【贪心策略】：对服务时间最短的顾客先服务的贪心选择策略。首先对需要服务时间最短的顾客进行服务，即做完第一次选择后，原问题T 变成了对n-1 个顾客服务的新问题T’ 。新问题和原问题相同，只是问题规模由n减小为n-1。基于此种选择策略，对新问题T’ ，选择n-1顾客中选择服务时间最短的先进行服务，如此进行下去，直至所有服务都完成为止。

4.2 贪心算法示例

贪心法求解最优服务次序问题----问题分析

- 有6个顾客{x1,x2,…,x6}同时等待用一服务，它们需要的服务时间分别为30,50,100,20,120,70.求使顾客等待时间的总和最小的服务次序。

解：顾客服务时间按从小到大排列结果为

{x4,x1,x2,x6,x3,x5}。

按贪心算法知服务时间最短的顾客先服务。则总时间TA为：

TA=6*20+5*30+4*50+3*70+100*2+120=1000。

平均等待时间： Average=1000/6166.7

贪心法求解最优服务次序问题----算法分析

【贪心选择性质】

先来证明该问题具有贪心选择性质，即最优服务A 中t(1)满足条件：t(1)<=t(i) (2 ≤ i ≤ n)。

证明：用反证法:

假设t(1)不是最小的，不妨设t(1)>t(i) (i>1)。设另一服务序列B={t(i),t(2),…, t(1),...t(n)}

那么TA-TB=

n*[t(1)-t(i)]+(n+1-i)*[t(i)- t(1)]=(1-i)*[t(i)-t(1)]>0

<0 <0

即TA>TB，这与A 是最优服务相矛盾，即问题得证。

故最优服务次序问题满足贪心选择性质。

4.2 贪心算法示例

贪心法求解最优服务次序问题----算法分析

【最优子结构性质】

在进行了贪心选择后，原问题T就变成了如何安排剩余的n-1 个顾客的服务次序的问题T’，是原问题的子问题。若A 是原问题T 的最优解，则A’={t(2),… t(i)...t(n)}是服务次序问题子问题T’的最优解。

证明：假设A’不是子问题T’的最优解，其子问题的最优解为B’，则有TB’<TA’，而根据TA 的定义知，TA’+t(1)=TA。因此，TB’+t(1)<TA’+t(1)=TA，即存在一个比最优值TA更短的总等待时间，而这与TA为问题T的最优值相矛盾。

因此，A’是子问题T’的最优值。从以上贪心选择及最优子结构性质的证明，可知对最优服务次序问题用贪心算法可求得最优解。

根据以上证明，最优服务次序问题可以用**最短服务时间优先**的贪心选择可以达到最优解。故只需对所有服务先按服务时间从小到大进行排序，然后按照排序结果依次进行服务即可。**平均等待时间即为TA/n.**

4.2 贪心算法示例

贪心法求解最优服务次序问题----算法描述

```
/*
    功能：计算平均等待时间
    输入：各顾客等待时间a[n],n是顾客人数
    输出：平均等待时间average
*/
double GreedyWait(int a[],int n)
{
     double average=0.0;
    Sort(a);//按服务时间从小到大排序
    for(int i=0;i<n;i++)
    {
          average += (n-i)*a[i];
    }
     average /=n;
    return average;
 }
```

4.2 贪心算法示例

贪心法求解求解活动安排问题----问题描述

【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S，$S=\{1, \ldots, n\}$。该资源任何时刻只能被一个活动所占用，活动i有一个开始时间b_i和结束时间e_i（$b_i<e_i$），其执行时间为e_i-b_i，假设最早活动执行时间为0。

- 一旦某个活动开始执行，中间不能被打断，直到其执行完毕。
- 若活动i和活动j有$b_i \geq e_j$或$b_j \geq e_i$，则称这两个活动兼容。

设计算法求一种最优活动安排方案，使得所有安排的活动个数最多。

4.2 贪心算法示例

贪心法求解求解活动安排问题----问题求解

【问题求解】假设活动时间的参考原点为0。一个活动 i（$1 \le i \le n$）用一个区间$[b_i, e_i)$表示，当活动按结束时间（右端点）递增排序后，两个活动$[b_i, e_i)$和$[b_j, e_j)$兼容（满足$b_i \ge e_j$或$b_j \ge e_i$）实际上就是指它们不相交。

用数组A存放所有的活动：

A[i].b（$1 \le i \le n$），存放活动起始时间，

A[i].e存放活动结束时间。

贪心法求解求解活动安排问题----求解示例

例如，对于下表的$n=11$个活动（已按结束时间递增排序）A：

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

产生最大兼容活动集合的过程：

活动1	√
活动2	×
活动3	×
活动4	√
活动5	×
活动6	×
活动7	×
活动8	√
活动9	×
活动10	×
活动11	√

最大兼容活动集合：

活动1　活动4　活动8　活动11

求解结果

贪心法求解求解活动安排问题----算法描述

```
//问题表示
struct Action                          //活动的类型声明
{   int b;                             //活动起始时间
    int e;                             //活动结束时间
    bool operator<(const Action &s) const    //重载<关系函数
    {
        return e<=s.e;                 //用于按活动结束时间递增排序
    }
};
int n=11;
Action A[]={{0},{1,4},{3,5},{0,6},{5,7},{3,8},{5,9},{6,10},{8,11},
        {8,12},{2,13},{12,15}};        //下标0不用

//求解结果表示
bool flag[MAX];                        //标记选择的活动
int Count=0;                           //选取的兼容活动个数
```

贪心法求解求解活动安排问题----算法描述

```
void solve()                          //求解最大兼容活动子集
{  memset(flag,0,sizeof(flag));       //初始化为false
   sort(A+1,A+n+1);                   //A[1..n]按活动结束时间递增排序
   int preend=0;                      //前一个兼容活动的结束时间
   for (int i=1;i<=n;i++)             //扫描所有活动
   {  if (A[i].b>=preend)             //找到一个兼容活动
      {  flag[i]=true;                //选择A[i]活动
         preend=A[i].e;               //更新preend值
      }
   }
}
```

贪心法求解求解活动安排问题----算法分析

【算法分析】算法的主要时间花费在排序上，排序时间为$O(n\log_2 n)$，所以整个算法的时间复杂度为$O(n\log_2 n)$。

贪心法求解求解活动安排问题----算法证明

【算法证明】通常证明一个贪心选择得出的解是最优解的一般的方法是，构造一个初始最优解，然后对该解进行修正，使其第一步为一个贪心选择，证明总是存在一个以贪心选择开始的求解方案。

对于本问题，所有活动按结束时间递增排序，就是要证明：若X是活动安排问题A的最优解，$X=X'\cup\{1\}$，则X'是$A'=\{i\in A : e_i\geq b_1\}$的活动安排问题的最优解。

贪心法求解求解活动安排问题----算法证明

【贪心选择性质】

首先证明总存在一个以活动1开始的最优解。

如果第一个选中的活动为k（$k\neq1$），可以构造另一个最优解Y，Y中的活动是兼容的，Y与X的活动数相同。

那么用活动1取代活动k得到Y'，因为$e_1\leq e_k$，所以Y'中的活动是兼容的，即Y'也是最优的，这就说明总存在一个以活动1(即活动结束时间最小的活动)开始的最优解。

贪心法求解求解活动安排问题----算法证明

【*最优子结构性质*】当做出了对活动1的贪心选择后，原问题就变成了在活动2、…、n中找与活动1兼容的那些活动的子问题。亦即，如果X为原问题的一个最优解，则$X'=X-\{1\}$也是活动选择问题$A'=\{i\in A \mid b_i\geq e_1\}$的一个最优解。

反证法：如果能找到一个A'的含有比X'更多活动的解Y'，则将活动1加入Y'后就得到A的一个包含比X更多活动的解Y，这就与X是最优解的假设相矛盾。

因此，在每一次贪心选择后，留下的是一个与原问题具有相同形式的最优化问题，即最优子结构性质。

贪心法求解求解活动安排问题--拓展（畜栏保留问题）

【问题描述】求解畜栏保留问题。农场有n头牛，每头牛会有一个特定的时间区间[b，e]在畜栏里挤牛奶，并且一个畜栏里任何时刻只能有一头牛挤奶。

现在农场主希望知道最少畜栏能够满足上述要求，并给出每头牛被安排的方案。对于多种可行方案，输出一种即可。

4.2 贪心算法示例

贪心法求解求解活动安排问题--拓展（畜栏保留问题）

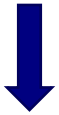
【问题求解】牛的编号为1～n，每头牛的挤奶时间相当于一个活动，与前面活动安排问题不同，这里的活动时间是闭区间，例如[2，4]与[4，7]是交叉的，它们不是兼容活动。

采用与求解活动安排问题类似的贪心思路，将所有活动这样排序：结束时间相同按开始时间递增排序，否则按结束时间递增排序。

求出一个最大兼容活动子集，将它们安排在一个畜栏中（畜栏编号为1）；如果没有安排完，再在剩余的活动再求下一个最大兼容活动子集，将它们安排在另一个畜栏中（畜栏编号为2），以此类推。也就是说，最大兼容活动子集的个数就是最少畜栏个数。

贪心法求解求解活动安排问题--拓展（畜栏保留问题）

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



1	3	4	6
1	5	8	12
4	7	9	13

2	7
2	11
5	15

5
4
10

最大兼容活动子集个数为3

贪心法求解求解活动安排问题--拓展（畜栏保留问题）

```
//问题表示
struct Cow                          //奶牛的类型声明
{  int no;                          //牛编号
   int b;                           //起始时间
   int e;                           //结束时间
   bool operator<(const Cow &s) const    //重载<关系函数
   {  if (e==s.e)                   //结束时间相同按开始时间递增排序
         return b<=s.b;
      else                          //否则按结束时间递增排序
         return e<=s.e;
   }
};
int n=5;
Cow A[]={{0},{1,1,10},{2,2,4},{3,3,6},{4,5,8},{5,4,7}};
                                    //下标0不用
//求解结果表示
int ans[MAX];                       //ans[i]表示第A[i].no头牛的畜栏编号
```

贪心法求解求解活动安排问题--拓展（畜栏保留问题）

```
void solve()                          //求解最大兼容活动子集个数
{  sort(A+1,A+n+1);                   //A[1..n]按指定方式排序
   memset(ans,0,sizeof(ans));                 //初始化为0
   int num=1;                         //畜栏编号

   for (int i=1;i<=n;i++)             //i、j均为排序后的下标
   {  if (ans[i]==0)                          //第i头牛还没有安排畜栏
      {  ans[i]=num;                          //第i头牛安排畜栏num
         int preend=A[i].e;                   //前一个兼容活动的结束时间
         for (int j=i+1;j<=n;j++)             //查找一个最大兼容活动子集
         {  if (A[j].b>preend && ans[j]==0)
            {  ans[j]=num;//将兼容活动子集中活动安排在num畜栏中
               preend=A[j].e;                 //更新结束时间
            }
         }
         num++;                  //查找下一个最大兼容活动子集,num增1
      }
    }
}
```

贪心法求解求解活动安排问题--拓展（畜栏保留问题）

```
void main()
{   solve();
    printf("求解结果\n");
    for (int i=1;i<=n;i++)
        printf("    牛%d安排的畜栏: %d\n",A[i].no,ans[i]);
}
```

i	1	2	3	4	5
开始时间	1	2	3	5	4
结束时间	10	4	6	8	7

求解结果

牛2安排的畜栏:1
牛3安排的畜栏:2
牛5安排的畜栏:3
牛4安排的畜栏:1
牛1安排的畜栏:4

贪心法求解求解区间相交问题----问题描述

【问题描述】

给定x 轴上n 个闭区间。去掉尽可能少的闭区间，剩下的闭区间都不相交。输出计算出的去掉的最少闭区间数。

贪心法求解求解区间相交问题----问题分析

最小删去区间数目=区间总数目-最大相容区间数目。

区间选择问题即若干个区间要求互斥使用某一公共区间段，目标是选择最大的相容区间集合。 假定集合$S=\{x1, x2, \cdots, xn\}$中含有n个希望使用某一区间段，每个区间xi有开始时间$lefti$和完成时间$righti$，其中，$0 \leqslant lefti < righti < \infty$。如果某个区间$xi$被选中使用区间段，则该区间在半开区间$(lefti, righti]$这段时间占据区间段。如果区间$xi$和$xj$在区间$(lefti, righti]$和$(leftj, rightj]$上不重叠，则称它们是相容的，即如果$lefti \geqslant rightj$或者$leftj \geqslant righti$，区间$xi$和$xj$是相容的。 区间选择问题是选择最大的相容区间集合。

最大相容区间问题可以用贪心算法解决，可以将集合S的n个区间段以右端点的非减序排列，每次总是选择具有最小右端点相容区间加入集合A中。直观上，按这种方法选择相容区间为未安排区间留下尽可能多的区间段。也就是说，该算法贪心选择的意义是使剩余的可安排区间段极大化，以便安排尽可能多的相容区间。

贪心法求解求解区间相交问题--算法分析（两个性质）

【贪心选择性质】

（1）设A是区间选择问题一个最优解，A中区间按右端点非减序排列；

（2）设K是A中第一区间，若k=1，则A就是一个以贪心选择开始的最优解；若k>1，再设B=A-{k}U{1}.

由于$right1 \leq rightk$且A中区间是相容的，故B中区间也是相容的。又由于B中区间个数与A中区间个数相同，且A是最优的，故B也是最优的。也就是B是以贪心选择区间1开始的最优区间选择。

结论：总存在一个以贪心选择开始的最优区间选择方案。

贪心法求解求解区间相交问题--算法分析（两个性质）

【最优子结构性质】（有问题）

在选择了区间1后，原问题简化为对E中所有与区间1相容的区间进行区间选择的子问题，若A是原问题的一个最优解，则：

$A' = A - \{1\}$ 是区间选择问题E'={i　E: $left_i \geq right_i$} 的最优解。

反证法：假设，它包含比A更多间，$B' \cup \{1\} = B$ ，比A有更多区间B是最优解，与A为最优解矛盾。

- 因此，每一步所作贪心选择都将问题简化为一个更小且与原问题具有相同形式的子问题，用数学归纳法知，区间选择算法使用贪心算法最终能产生原问题的最优解。

贪心法求解求解区间相交问题--算法描述

```
class interval   //区间类
{
public:
        int left;   // 左端点
        int right;  //右端点
};

bool cmp(interval a,
     interval b)
 {
      if (a. right <b. right)
     return true;
      else return false;
}
```

```
int GreedyArrange(int n, interval inte[ ])
{
    sort(inte, inte+n, cmp);
    int count = 1;  //最大区间相容数
    int j=0;
     for (int i=1; i<n; i++) {
      if (inte[i].left inte[j].right) {
     count++; j=i;
    }
   }
   return n-count;
}
```

Chapter 4

4.6 本章小结

- 贪心算法适用于最优化问题。它是通过做一系列的选择给出某一问题的最优解。对算法中的每一个决策点做出当时看起来最佳的选择。

  贪心算法的基本步骤:

  1. 选择合适的贪心选择的标准;
  2. 证明在此标准下该问题具有贪心选择性质;
  3. 证明该问题具有最优子结构性质;
  4. 根据贪心选择的标准，写出贪心选择的算法，求得最优解。

  贪心算法通常包括排序过程，这是因为贪心选择的对象通常是一个数值递增或递减的有序关系，自顶向下计算。