算法 设计与分析

第四章 贪心法（贪婪法）
Greedy Algorithms

Chapter 4

贪心算法引入

硬币找零问题

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 学过的贪心算法

- 哈夫曼树
- 单源最短路径——Dijkstra
- 最小生成树——Prim、Kruskal
- 最短磁道优先算法（操作系统）

Chapter 4

4.1 贪心算法基础

贪心法求解问题步骤：

1）确定合适的贪心选择标准；

2）证明在此标准下该问题具有贪心选择性质和最优子结构性质；

3）根据贪心选择标准，写出贪心选择的算法，求得最优解。

```
Greedy(C) { //C是问题的输入集合即候选集合
        S={ };  //初始解集合S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;
}
```

Chapter 4

4.1 贪心算法概述

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。

Chapter 4 4.1 贪心算法概述

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**。

Chapter 4 4.1 贪心算法概述

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。

Chapter 4 4.1 贪心算法概述

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。

4.1.3 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。

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
}
```

Chapter 4 4.2 贪心算法示例-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。**

Chapter 4 4.2 贪心算法示例-求解单源最短路径

Dijkstra算法是一种按各个顶点与源点之间路径长度的递增次序，生成源点到各个顶点的最短路径的方法——贪心算法，即先求出一条最短的路径，再参照它求出长度次短的一条路径，以此类推，直到求出所有为止。

算法步骤：将顶点集V划分成S和V-S两部分，其中S集合中的顶点到源点的最短路径已经确定。

(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。

4.2 贪心算法示例-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。

Chapter 4 4.2 贪心算法示例-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t，从而确定从源v到t的最短路径长度distance [t]。该最短路径：

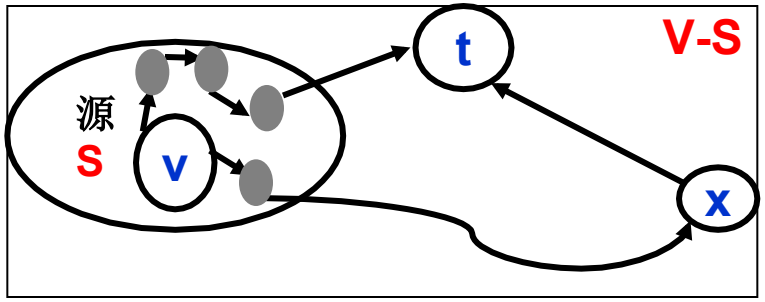
- 要么是<v,t>即v与t直接有边相连且该路径长度最短；
- 要么是经过S 集合中的某顶点u到达的。

即distance [t]=distance[u]+cost(u,t)。

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v到t没有更多的其他路径呢？

4.2 贪心算法示例-求解单源最短路径

事实上，如果存在一条从源v到t且长度比distance [t]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,t)和cost(v,t)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,t)= cost (v,t)< distance [t]。利用边权的非负性，可知cost(x,t)>=0，从而推distance [x]< distance [t]。此为矛盾。这就证明了distance [t]是从源到顶点u的最短路径长度。

4.2 贪心算法示例-求解单源最短路径

2）最优子结构性质

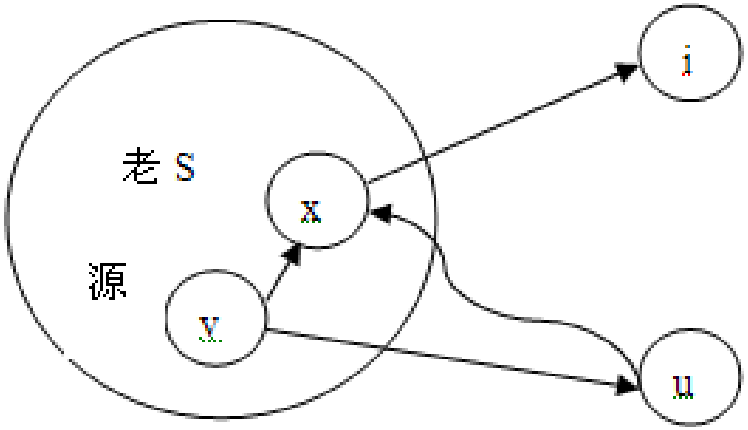
算法中确定的distance[u]确实是源点到顶点u的最短特殊路径长度。为此，只要考察算法在添加u到S中后,distance [u]的值所引起的变化。当添加u之后，可能出现一条到顶点i特殊的新路。

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。

4.2 贪心算法示例-求解单源最短路径

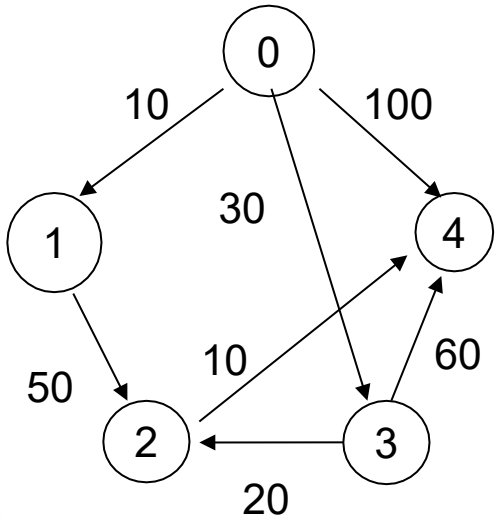
2）最优子结构性质

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。



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。



4.2 贪心算法示例-求解单源最短路径

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;
         }
      }
}
```

Chapter 4 4.2 贪心算法示例-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（$1 \leqslant i \leqslant n$）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。

Chapter 4 4.2 贪心算法示例-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，$0 \leqslant x_i \leqslant 1$。根据问题的要求，有如下约束条件和目标函数：

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

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

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$X=\{x_1, x_2, \cdots, x_n\}$。

Chapter 4 4.2 贪心算法示例-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。

Chapter 4 4.2 贪心算法示例-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.

4.2 贪心算法示例-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 作为部分背包问题的最优量度标准。

Chapter 4 4.2 贪心算法示例-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。

4.2 贪心算法示例-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。

4.2 贪心算法示例-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}。

4.2 贪心算法示例-求解部分背包问题

```
//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];
```

4.2 贪心算法示例-求解部分背包问题

```
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
   }
}
```

4.2 贪心算法示例-求解部分背包问题

```
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);
}
```

4.2 贪心算法示例-求解部分背包问题

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（$1 \leqslant i \leqslant n$）值递减排序得到1、2、…、$n$的序列，即$v_1/w_1 \geqslant v_2/w_2 \geqslant \cdots \geqslant v_n/w_n$。设$X=\{x_1, x_2, \cdots, x_n\}$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，$0 \leqslant x_i<1$ 。

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

按照求解的方法（贪心）此值为W

设$Y=\{y_1, y_2, \cdots, y_n\}$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

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

贪心法求解（部分背包问题）算法证明

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

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$x_i-y_i\geqslant 0$，且$v_i/w_i\geqslant v_{minj}/w_{minj}$。

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

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

4.2 贪心算法示例-求解部分背包问题算法证明

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

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

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

<=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。

Chapter 4.2 贪心算法示例-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W-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。

Chapter 4 .2 贪心算法示例-求解部分背包问题算法分析

【算法分析】对vi/wi排序的时间复杂性为0($n\log_2 n$)，while循环的时间为0(n)，所以本算法的时间复杂度为0($n\log_2 n$)。

Chapter 4

4.2 贪心算法示例-求解汽车加油问题

1. 问题分析

2. 算法分析（两个性质证明）

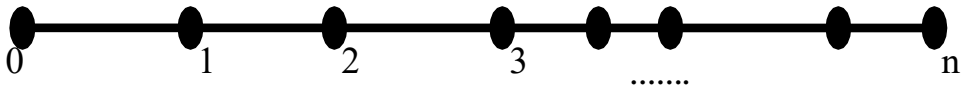
3. 设计与实现

Chapter 4

4.2 贪心算法示例-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

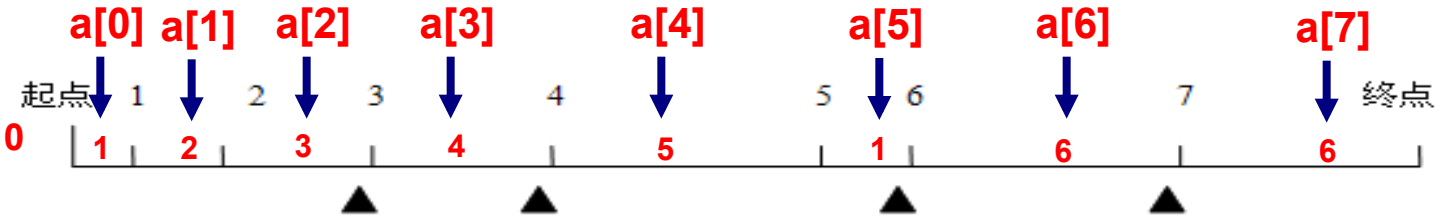
4.2 贪心算法示例-求解汽车加油问题

- 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。

4.2 贪心算法示例-求解汽车加油问题

- 例在汽车加油问题中，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-求解汽车加油问题

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。

4.2 贪心算法示例-求解汽车加油问题

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。

4.2 贪心算法示例-求解汽车加油问题

```
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-求解最优装载问题

【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（$1 \leqslant i \leqslant n$）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。

4.2 贪心算法示例-求解最优装载问题

【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。

4.2 贪心算法示例-求解最优装载问题

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

4.2 贪心算法示例-求解最优装载问题

```
//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
   }
}
```

4.2 贪心算法示例-求解最优装载问题

```
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-求解最优服务次序问题

【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-求解最优服务次序问题

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（即整个服务完成时间）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)；

平均等待时间=TA/n=t(1)+(n-1)/n*t(2)+…+(n+1-i)/n*t(i)+…+2/n*t(n-1)+1/n*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-求解最优服务次序问题

- 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

4.2 贪心算法示例-求解最优服务次序问题

【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-求解最优服务次序问题

【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-求解最优服务次序问题

```
/*
    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;
 }
```

Chapter 4

4.2 贪心法示例——多机调度问题

【问题描述】设有n个独立的作业{1，2，…，n}，由m台相同的机器{1，2，…，m}进行加工处理，作业i所需的处理时间为t_i（1≤i≤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。

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。

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

机器1

机器2

机器3

Chapter 4 4.2 贪心法示例——多机调度问题

【贪心策略】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。

作业i处理时间=等待机器的时间$wait_i$+机器对作业处理的时间t_i

Wait小　　t大的作业

整体上获得尽可能短的处理时间

- 当m≥n时，将机器i的[0，t_i)时间区间分配给作业i即可；
- 当m<n时，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。

4.2 贪心法示例——多机调度问题

n=7, m=3, 各作业所需的处理时间如下：

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

（1）7个作业按处理时间递减排序。

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

4.2 贪心法示例——多机调度问题

(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}。

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

机器1

作业4:16

机器2

作业2:14

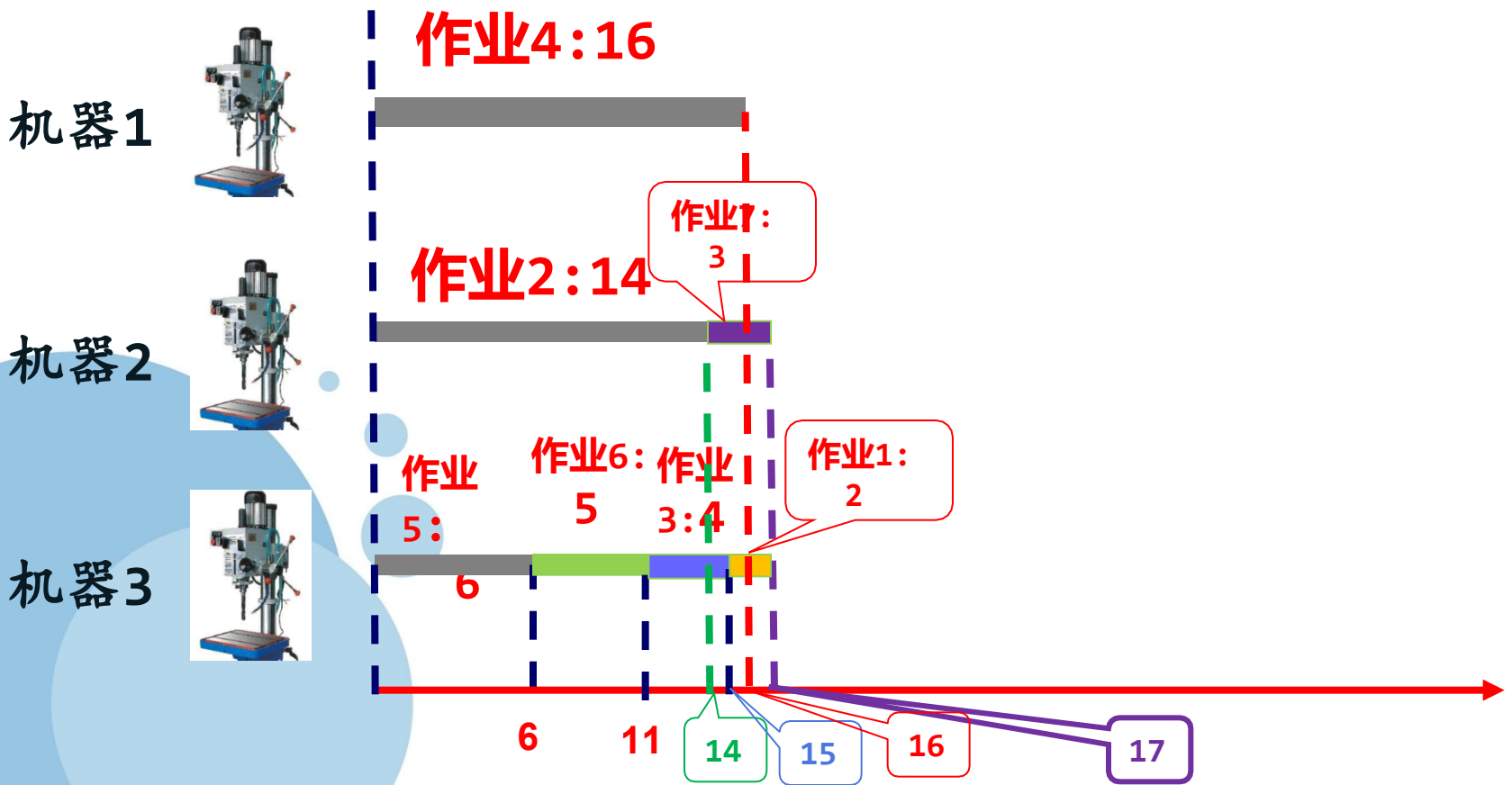
机器3

作业5:6

4.2 贪心法示例——多机调度问题

(3) 分配余下的作业：

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



Chapter 4

4.2 贪心法示例——多机调度问题

贪心策略：先分配最短作业

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

机器1  2--5--16 作业1,6,4，总时间23

机器2  3--6 作业7,5，总时间9

机器3  4--14 作业3,2，总时间18

作业调度方案

4.2 贪心法示例——多机调度问题

```
//问题表示
int n=7;
int m=3;
struct NodeType                                  //优先队列结点类型
{  int no;                                       //作业序号
   int t;                                        //执行时间
   int m_no;                                     //机器序号
   bool operator<(const NodeType &s) const
   {  return t>s.t;  }                           //按t越小越优先出队
};

struct NodeType
A[]={{1,2},{2,14},{3,4},{4,16},{5,6},{6,5},{7,3}};
```

4.2 贪心法示例——多机调度问题

```
void solve()                                    //求解多机调度问题
{  NodeType e;
   if (n<=m)
   {  printf("为每一个作业分配一台机器\n"); return; }

   sort(A,A+n);                                 //按t递减排序
   priority_queue<NodeType> qu;                 //小根堆
   for (int i=0;i<m;i++)                        //先分配m个作业，每台机器一个作业
   {  A[i].m_no=i+1;                                    //作业对应的机器编号
      printf("  给机器%d分配作业%d,执行时间为%2d,占用时间段:[%d,%d]\n",
                A[i].m_no,A[i].no,A[i].t,0,A[i].t);
      qu.push(A[i]);
   }
   for (int j=m;j<=n;j++)                               //分配余下作业
   {  e=qu.top();          qu.pop();                        //出队e
      printf("  给机器%d分配作业%d,执行时间为%2d,占用时间段:[%d,%d]\n",
                e.m_no,A[j].no,A[j].t,e.t,e.t+A[j].t);
      e.t+=A[j].t;
      qu.push(e);                                       //e进队
   }
}
```

Chapter 4

4.2 贪心法示例——多机调度问题

多机调度是NP难问题，贪心法只能求得一个较好的近似解。

反例：n=7，m=3，Ti={16,14,12,11,10,9,8}

机器1 

机器2 

机器3 

16--9: 25

14--10: 24

12--11--8: 31

贪心：31

16--11: 27

14--12: 26

10--9--8: 27

最优解：27

Chapter 4

4.2 贪心算法示例-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S，$S=\{1, \cdots, n\}$。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-求解活动安排问题

【问题求解】假设活动时间的参考原点为0。一个活动i（$1 \leqslant i \leqslant n$）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$b_i \geqslant e_j$或$b_j \geqslant e_i$）实际上就是指它们不相交。

用数组A存放所有的活动：

$A[i].b$（$1 \leqslant i \leqslant n$），存放活动起始时间，

$A[i].e$存放活动结束时间。

4.2 贪心算法示例-求解活动安排问题

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

4.2 贪心算法示例-求解活动安排问题

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
```

4.2 贪心算法示例-求解活动安排问题

```
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

【证明：贪心选择性质】一个全局最优解可以通过局部最优得到。

证明：设活动为A={a_1, a_2,…, a_n }，按最早结束时间排序后，ans[1]=1，表示第1个活动肯定被选择，不与其他任何活动冲突。对于剩余的活动A'={a_2,…a_k,...,a_x, a_n }，假设再次选择的活动是a_k，表示a_2到a_{k-1}活动，虽然结束时间早，但与活动a1冲突，不能选择。

如果不选择a_k，而是选择了一个更晚结束的活动a_x。

因为a_k和a_x都不与已选活动a_1冲突，但a_x结束时间更晚。那么选择a_k可以空出更多的空闲时间，可能可以安排更多的活动。

贪心法求解求解活动安排问题----算法证明

【证明：最优子结构性质】当一个大问题的最优解包含着它子问题的最优解时，称该问题具有最优子结构性质。

证明：假设(ans[1],ans[2],……ans[n])是整体最优解。

如果ans[1]=1,ans[2]=0/1，...,ans[k-1]=0/1，ans[k]=0/1，表示完成了对活动1到活动k的安排（最优安排），则面对与起始相同的情况，那么(ans[k+1],ans[k+2],……ans[n])为对活动k+1到活动n的局部最优解。

4.2 贪心算法示例-求解畜栏保留问题

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】求解蓄栏保留问题。农场有n头牛，每头牛会有一个特定的时间区间[b，e]在蓄栏里挤牛奶，并且一个蓄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蓄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。

蓄栏---------→资源

牛----------→ 需要享用资源的活动

对于一个资源（畜栏），希望安排更多的活动（牛）；之后如果还有没安排畜栏的牛，则继续新增一个畜栏。。。。

4.2 贪心算法示例-求解畜栏保留问题

【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，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，将它们安排在一个蓄栏中（蓄栏编号为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蓄栏中（蓄栏编号为2），以此类推。也就是说，最大兼容活动子集的个数就是最少蓄栏个数。

4.2 贪心算法示例-求解畜栏保留问题

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		2	7		5
1	5	8	12		2	11		4
4	7	9	13		5	15		10

最大兼容活动子集个数为3

4.2 贪心算法示例-求解畜栏保留问题

```
//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
```

4.2 贪心算法示例-求解畜栏保留问题

```
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
      }
   }
}
```

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）

```
void main()
{  solve();
   printf("求解结果\n");
   for (int i=1;i<=n;i++)
      printf("    牛%d安排的蓄栏: %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安排的蓄栏:1
牛3安排的蓄栏:2
牛5安排的蓄栏:3
牛4安排的蓄栏:1
牛1安排的蓄栏:4

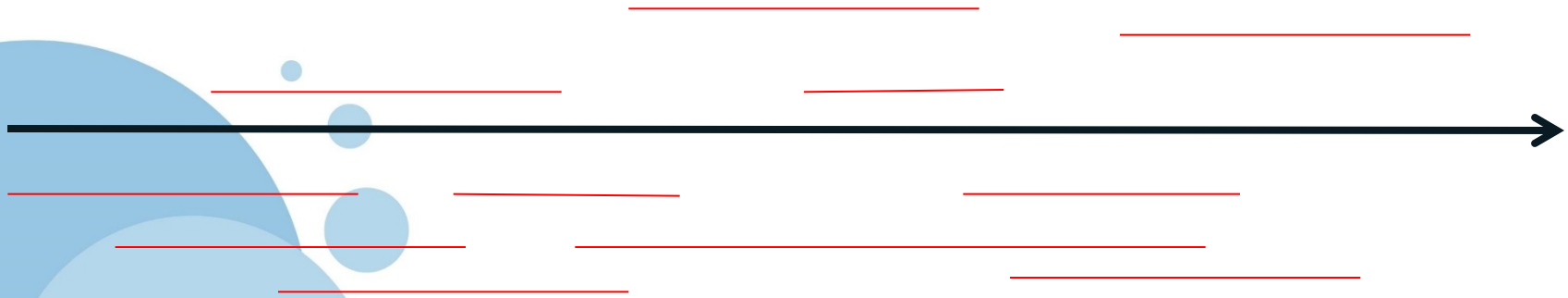
Chapter 4

4.2 贪心法示例——区间相交问题

【问题描述】给定x轴上n个闭区间[ ]。去掉尽可能少的闭区间，使剩下的闭区间都不相交。输出去掉的最少闭区间数。

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



Chapter 4 4.2 贪心算法示例求解区间相交问题

贪心法求解求解区间相交问题----问题分析

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。

4.2 贪心法示例——区间相交问题

```
typedef struct {
    int left;
   int right;
}interval ;
```

```
int GreedyArrange(interval num[ ], int n )
{
      sort(num);
      int count = 1;  //最大区间相容数
      int j=1;
      for (int i=2; i<=n; i++) {
            if (num[i].left>= num[j].right)
                  count++;  j=i;
       }

    return n-count;
}
```

Chapter 4

4.2 贪心法示例——股票买卖最佳时机

【问题描述】给定某只股票一段时间内每天的价格，如[8, 24, 2, 4, 10, 12, 13, 9]，问如何买卖（买卖最佳时机）使得收益最大。

（1）只能买卖一次

（2）可以买卖任意多次

贪心求解

（3）只能买卖指定次数，比如只能买卖3次

（4）有冻结期，买卖之后，需要冻结几天才能进行下次买卖

（5）有手续费，每次买卖都有手续费。

Chapter 4

4.2 贪心法示例——股票买卖最佳时机

（1）只能买卖一次，且无手续费

输入：8,9,2,5,4,7,1　　输出：buy=3 sell=6 profit=5

输入：3,9,2,5,4,7,1　　输出：buy=1 sell=2 profit=6

暴搜法：枚举买入时间i(1~n-1)，卖出时间(i+1~n)，寻找max{卖出价格-买入价格}。 $T(n)=O(n^2)$

贪心法：

- ✓ 卖出时间点有2~n
- ✓ 卖出日收益 = 卖出日当天价格 - 卖出日之前的最低价格
- ✓ 所有卖出日收益中，最大的

$T(n)=O(n)$

4.2 贪心法示例——股票买卖最佳时机

（1）只能买卖一次，且无手续费—算法描述

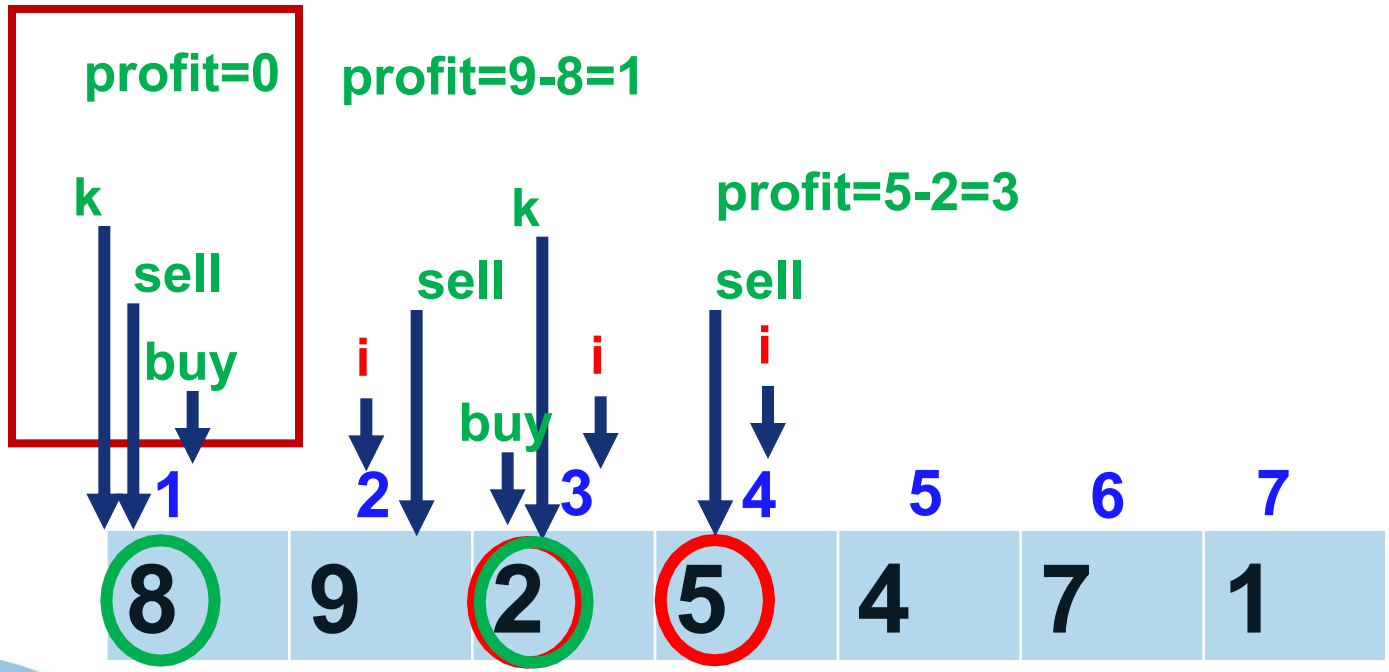
```
#define MAX 0x7fffffff
int main(){
      int prices[]={0,3,9,2,5,4,7,1};
      int minPrices = MAX,n=sizeof(prices)/sizeof(int)-1;
      int buy=1,sell=1, profit=0,k=1//k为当前股票价格最低日;

     for(int i = 2; i <= n;i++){
        if(prices[i]-prices[k]>profit){ //今天是否卖出
           profit = prices[i]-prices[k];
           buy=k;   sell=i;
         }
        if(prices[i] < prices[k])  k=i; //今天的价格是否更低
    }
    printf("buy=%d,sell=%d,profit=%d\n",buy,sell,profit);
    return 0;
}
```

Chapter 4

4.2 贪心法示例——股票买卖最佳时机

（1）只能买卖一次，且无手续费—算法示例



price[i]-price[k]=-6<profit(1)
则原有利润profit不更新；
同时：由于price[i]<price[k](即找到了股票价格更低的时机（3）则更新k=i(3);

price[i]-price[k]=3>profit(1)
则原有利润profit更新为3；
Buy更新为k即buy=3；sell=i;

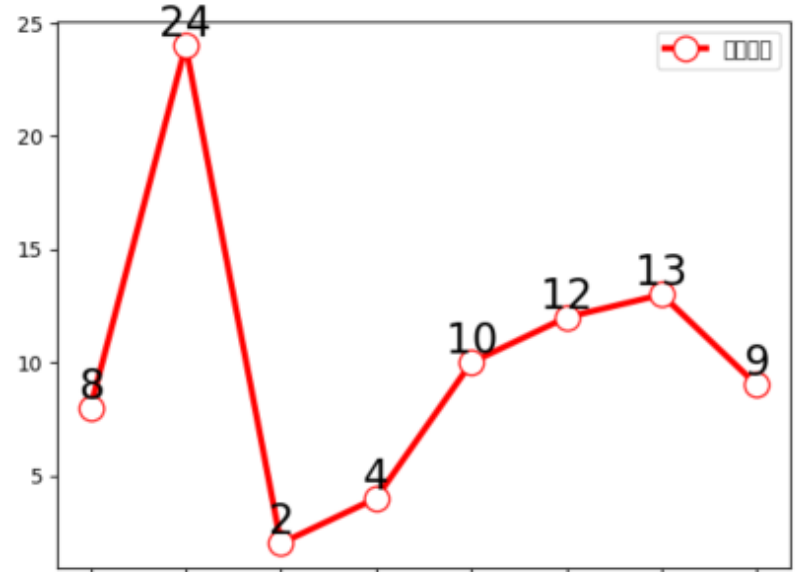
4.2 贪心法示例——股票买卖最佳时机

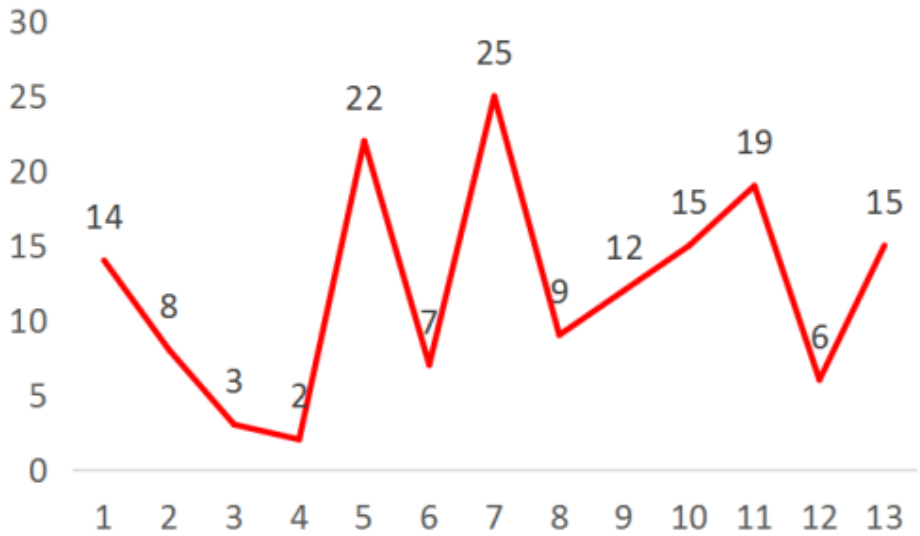
（2）可以多次买入和卖出，无手续费

输入：7 1 5 3 6 4　　输出：7

输入：8 24 2 4 10 12 13 9　　输出：27

输入：14 8 3 2 22 7 25 9 12 15 19 6 15　　输出：？？





买入时机：后一天价格 – 当前天价格 > 0

买入后，只要后一天的价格比前一天高，计算收益

卖出时机：后一天价格 – 当前天价格 < 0

Chapter 4

4.2 贪心法示例——股票买卖最佳时机

最基本的想法： 选一个价格低的时间买入，再选个价格高的时间卖出，再选一个低的时间买入.....循环反 复。 **问题是：** 何时为最低点，何时为最高点？

贪心法：能够将问题分解成子问题（子结构）

假如第0天买入，第3天卖出，那么利润为：

prices[3] - prices[0]（即第3天股票的价格-第0天的股票价格）。 能否转换为时间跨度为1天的问题？

prices[3] - prices[0]

=prices[3] - prices[2] + prices[2] - prices[1] + prices[1] - prices[0]。

= (prices[3] - prices[2])+ (prices[2] - prices[1])+ (prices[1] - prices[0])

此时就是把利润分解为每天为单位的维度，而不是从0天到第3天整体去考虑！

那么根据prices可以得到前i天每天的利润序列（假设第0天即为初始买入时间）

(prices[i] - prices[i - 1]) ,(prices[i-1] - prices[i - 2])+...+(prices[1] - prices[0])

4.2 贪心法示例——股票买卖最佳时机

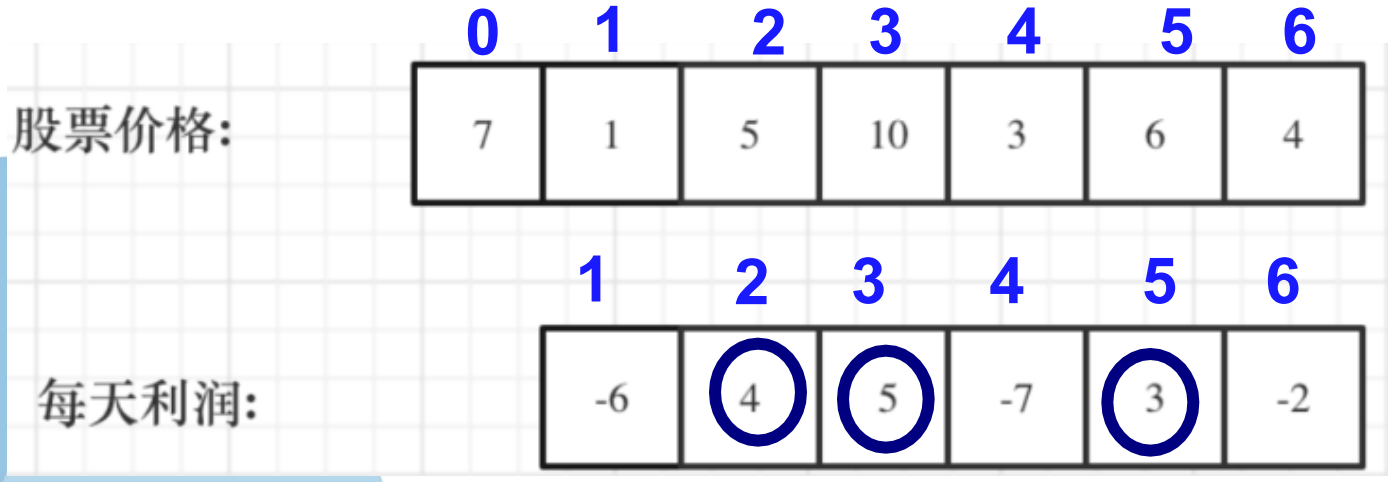
根据prices可以得到前i天每天的利润序列：
（假设第0天即为初始买入时间）：

(prices[i] - prices[i - 1]) ,(prices[i-1] - prices[i - 2]) …… .

..

...(prices[1] - prices[0])

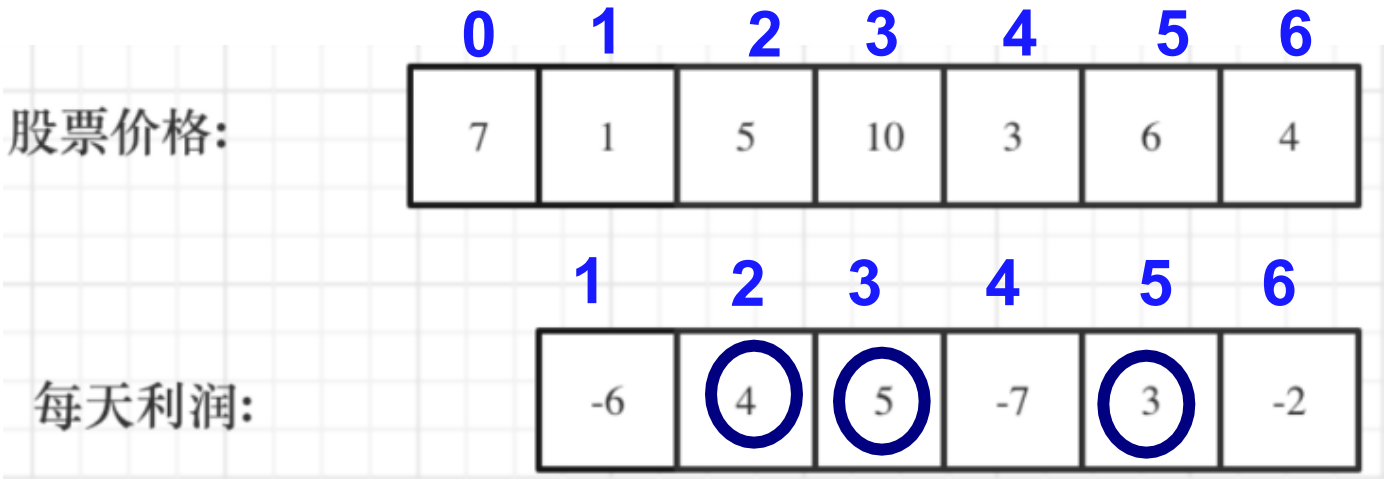
欲使利润最大，则上式中的每一项，只要非负项才能使股票获得收益。



贪心，只收集每天正利润：　　4 + 5 + 3 = 12

Chapter 4

4.2 贪心法示例——股票买卖最佳时机



从图中可以发现，其实我们需要收集每天的正利润就可以，收集正利润的区间，就是股票买 卖的区间，而我们只需要关注最终利润，不需要记录区间。

4.2 贪心法示例——股票买卖最佳时机

（2）可以多次买入和卖出，无手续费

```
int maxProfit(int prices[],int n)
{
int result = 0;
for (int i = 1; i < n;i++)
  result += max(prices[i] - prices[i - 1], 0);
return result;
}
```

Chapter 4

4.2 贪心法示例——移除k个数字

【问题描述】给定一个以字符串表示的非负整数num，移除这个数中的 k 位数字，使得剩下的数字最小。其中num的长度小于10000 且 ≥ k，num不会包含任何前导零。

例如，输入：num = ″1432219″, k = 3

输出：″1219″

解释：移除掉三个数字 4, 3和2形成一个新的最小的数字1219。

num="2430024890892150749282036219",k=8

怎么办?

Chapter 4

4.2 贪心法示例——移除k个数字

【问题分析】对于num="124682385", k=1，删除谁？

124682385

如果num'="12462385", k=1，删除谁？

12462385

如果num'="1242385", k=1，删除谁？

1242385

Chapter 4

4.2 贪心法示例——移除k个数字

贪心策略：

当左边数(高位）>右边（低位）数时，删除左边数。

num="243002489089215074928203621 9",k=8

思考：

- ✓当所有数字都扫描完成后，K仍然>0，应该做怎么样的处理？
- ✓当数字中有0出现时，应该有怎么样的特殊处理？
- ✓如何将最后结果存储为字符串并返回？

Chapter 4

4.2 贪心法示例——移除k个数字

算法：从左到右依次扫描字符串，比较ch[i]和ch[i+1]：

（1）如果ch[i] >ch[i+1]，舍弃ch[i]，k--，i++；

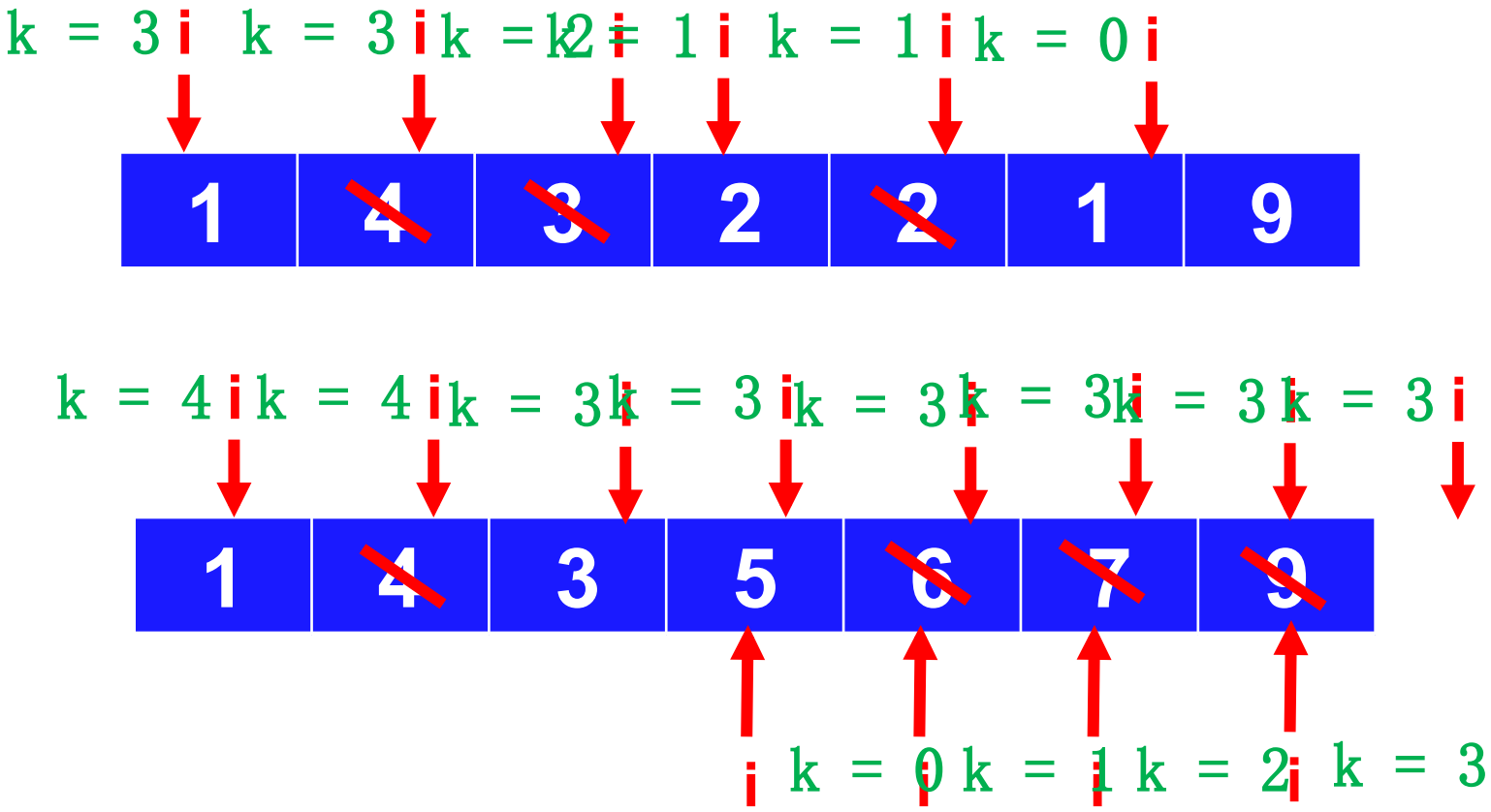
（2）如果ch[i] <ch[i+1]，k不变，i++。

（3）如果ch[i] =ch[i+1]，k不变，i++。

（4）如果结束后，发现K值大于0，则表示全部为升序，从后面删除剩余的K个字符即可。

Chapter 4

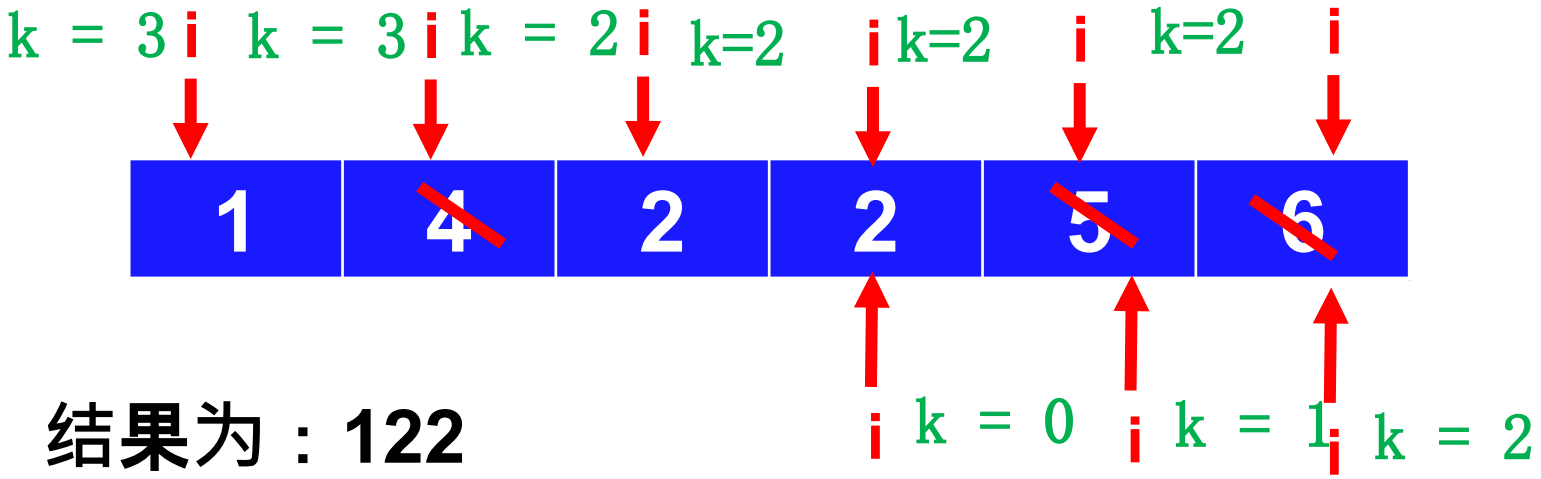
4.2 贪心法示例——移除k个数字



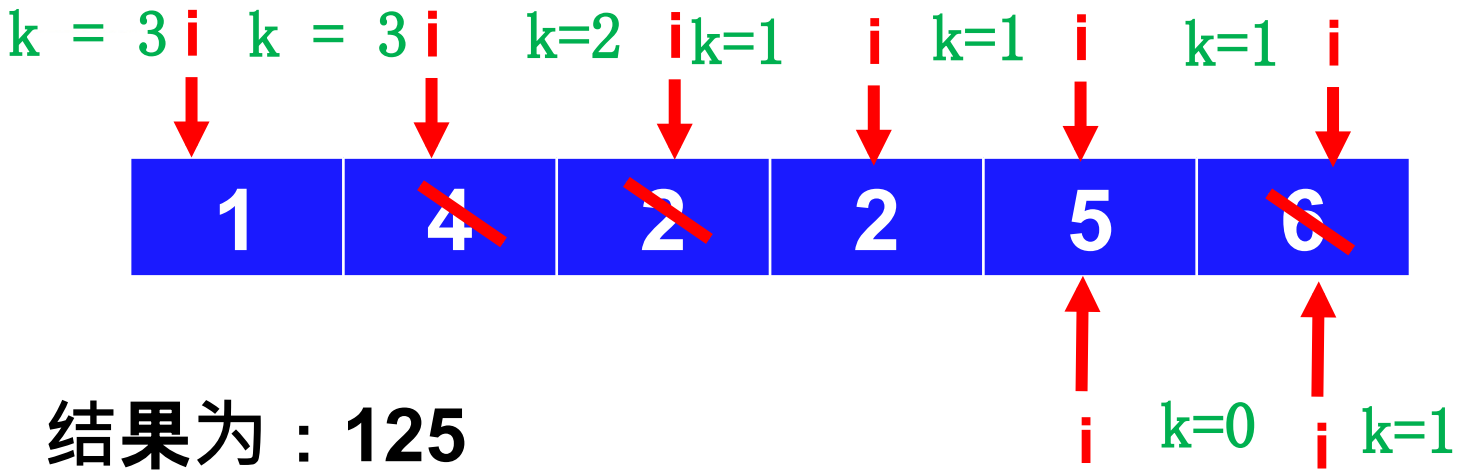
Chapter 4

4.2 贪心法示例——移除k个数字

为什么相邻元素ch[i]和ch[i+1]相等的时候，不删ch[i]？



结果为：122



结果为：125

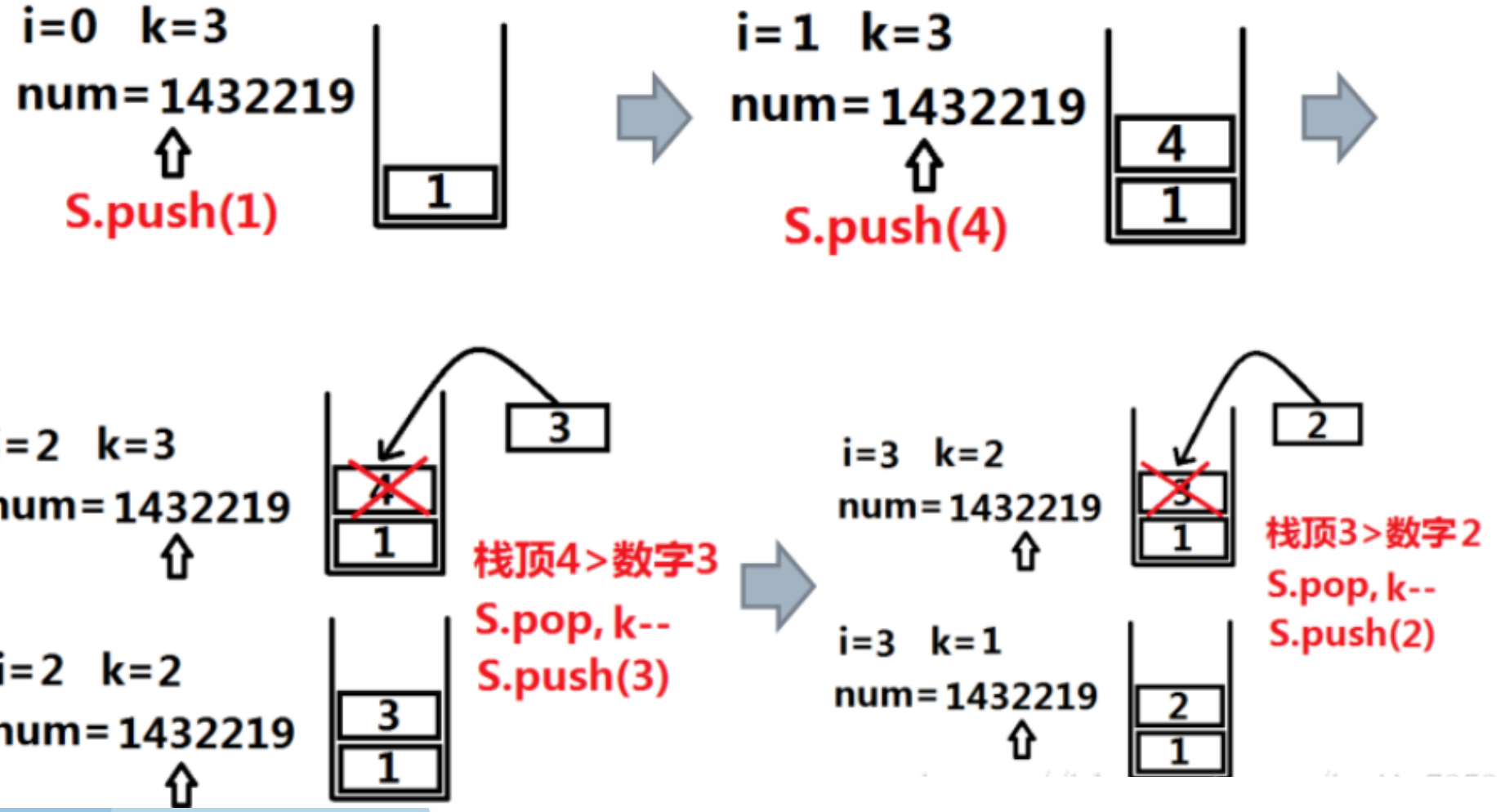
Chapter 4

4.2 贪心法示例——移除k个数字

```
void removeK(char ch[], int n,int k){
    int i=0;
    while(k>0 && i<n-1){
      if(ch[i] > ch[i+1]) {ch[i]=' '; k--;}
       i++; }
    if(k>0)
     for(i=n-1;i>=0&&k>0;i--)
      {ch[i]=' ';  k--;   }
}
```

Chapter 4

4.2 贪心法示例——移除k个数字



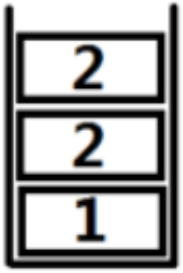
Chapter 4

4.2 贪心法示例——移除k个数字

i=4 k=1

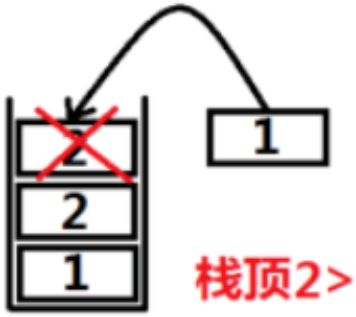
num=1432219

S.push(2)



i=5 k=1

num=1432219



栈顶2>数字1

S.pop, k--

S.push(1)

i=5 k=0

num=1432219

i=6 k=0

num=1432219

S.push(9)

栈底遍历至栈顶，即为最终结果”1219”

Chapter 4

4.2 贪心法示例——摇摆序列

【问题描述】一个整数序列，如果两个相邻元素的差恰好正负（或负正）交替出现，则该序列称为摇摆序列，一个小于2个元素的序列直接为摇摆序列。

给一个随机序列，求这个序列满足摇摆序列定义的最长子序列的长度。

例如：序列[1,7,4,9,2,5]，相邻元素的差(6,-3,5,-7,3)，该序列为摇摆序列

但，序列[1,4,7,2,5]相差(3,3,-5,3)不是摇摆序列

Chapter 4

4.2 贪心法示例——摇摆序列

【思考与分析】[1, 17, 5, 10, 13, 15, 10, 5, 16, 8]，整体不是摇摆序列。

观察该序列的前6位：[1, 17, 5, 10, 13, 15…]；其中5, 10, 13, 15部分为上升段，有三个子序列是摇摆序列：

[1, 17, 5, 10….]　　[1, 17, 5, 13, …]　　[1, 17, 5, 15…..]

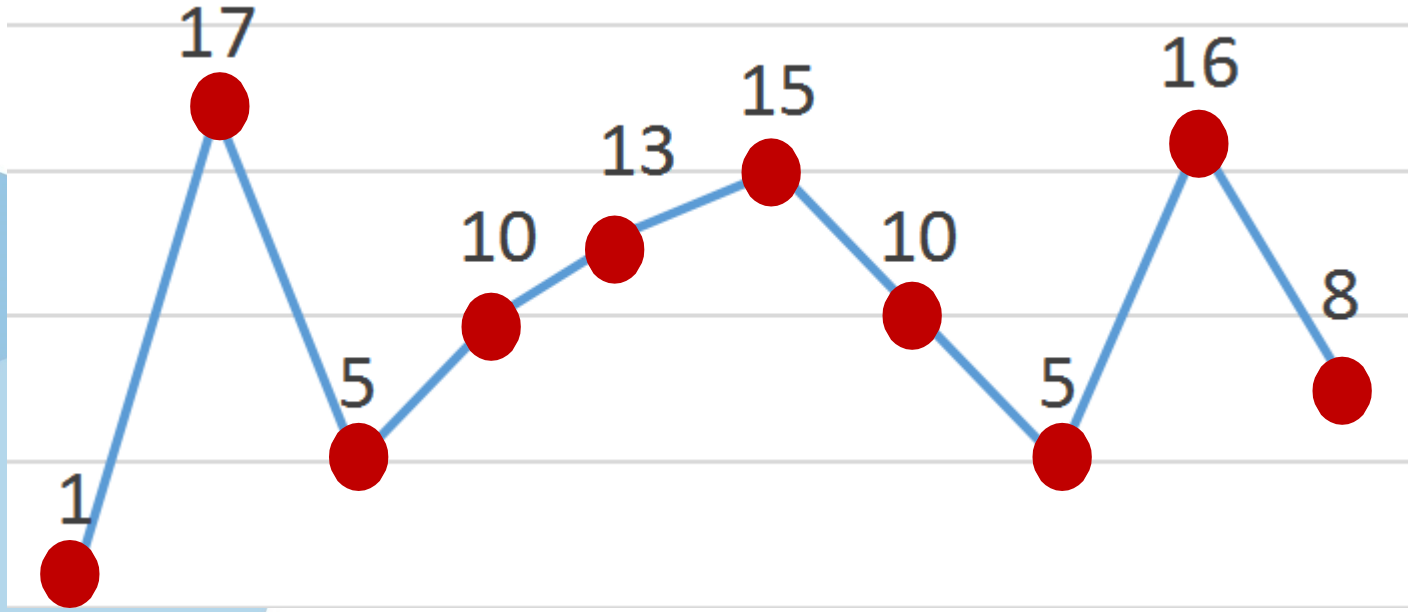
在不清楚原始序列的第7位的情况下，只看前6位，摇摆子系列的第四位从10，13，15中选择一个数，我们应该选择哪一个？

选最大的数：15

Chapter 4

4.2 贪心法示例——摇摆序列

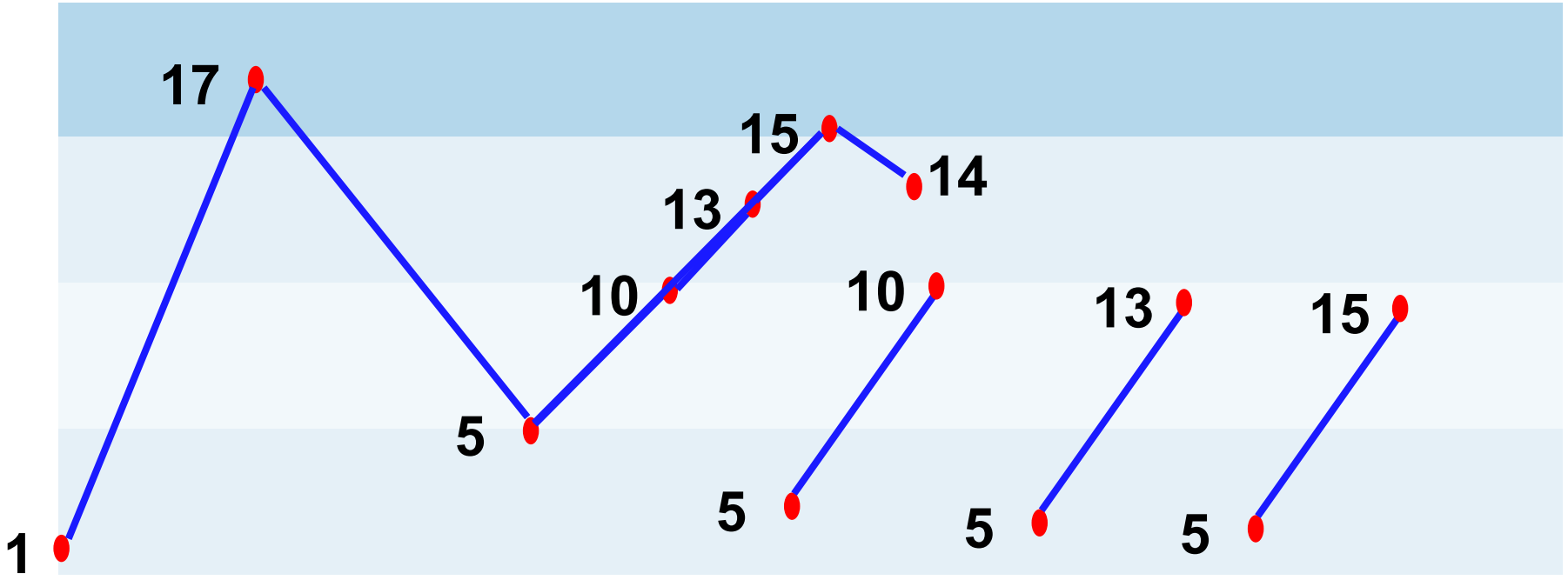
贪心策略： 当序列有一段连续的递增(或递减)时，未形成摇摆子序列，**我们只需要保留**这段连续的递增(或递减)的**首尾**元素，这样更有可能使得尾部的后一个元素成为摇摆子序列的下一个元素。 **1 17 5 10 13 15 10 5 16 8**



Chapter 4

4.2 贪心法示例——摇摆序列

为什么要保留连续的递增(或递减)的首尾元素？



对于单调递增或递减的坡度，只需要保持头尾值，
这样不会影响求最长的摆动序列的长度。

Chapter 4

4.2 贪心法示例——摇摆序列

【要解决的问题】

Q1：首尾元素如何处理？即若序列中只有两个元素时的状况

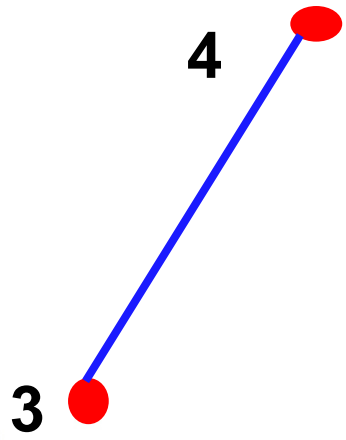
Q2：序列中可能出现上升（递增）坡、下降（递减）坡、平坡。

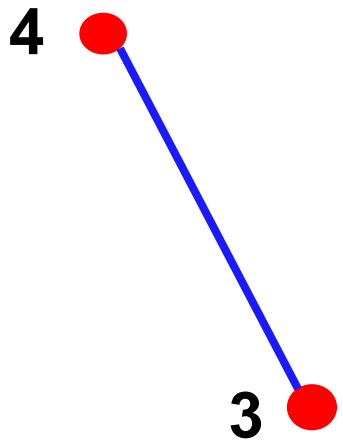
Q3：单调增（上升）坡或单调减（下降）坡上有相邻元素值相等的状况（平坡），如何处理？

Chapter 4

4.2 贪心法示例——摇摆序列

【约定】若序列中只有两个元素：默认存在两个摆动。

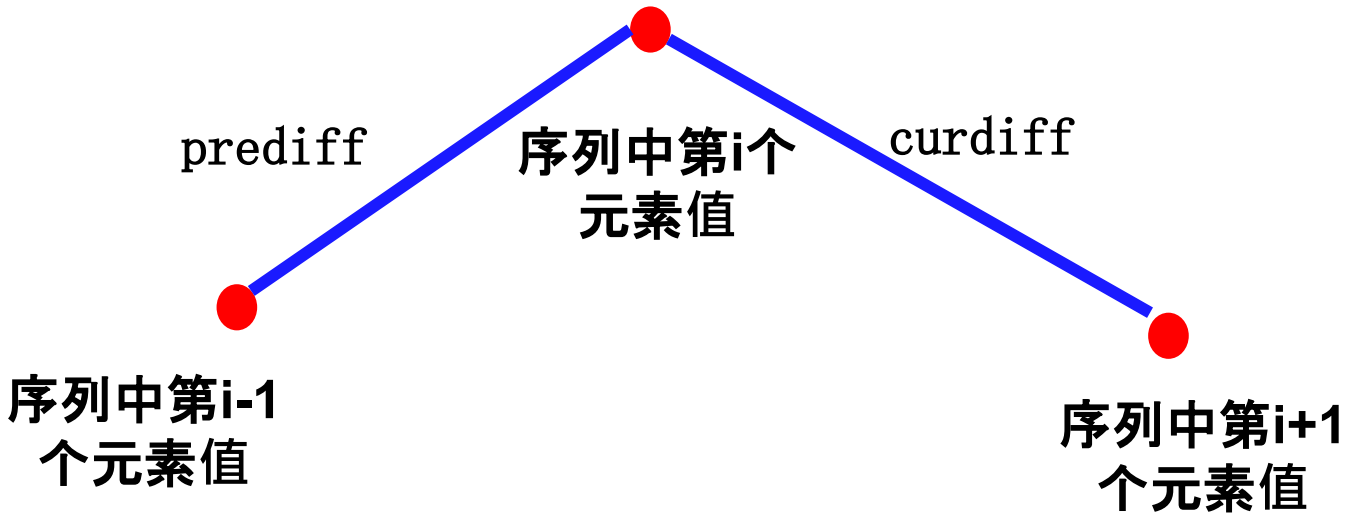




4.2 贪心法示例——摇摆序列

【相关变量】序列存放在nums[0..n-1]中

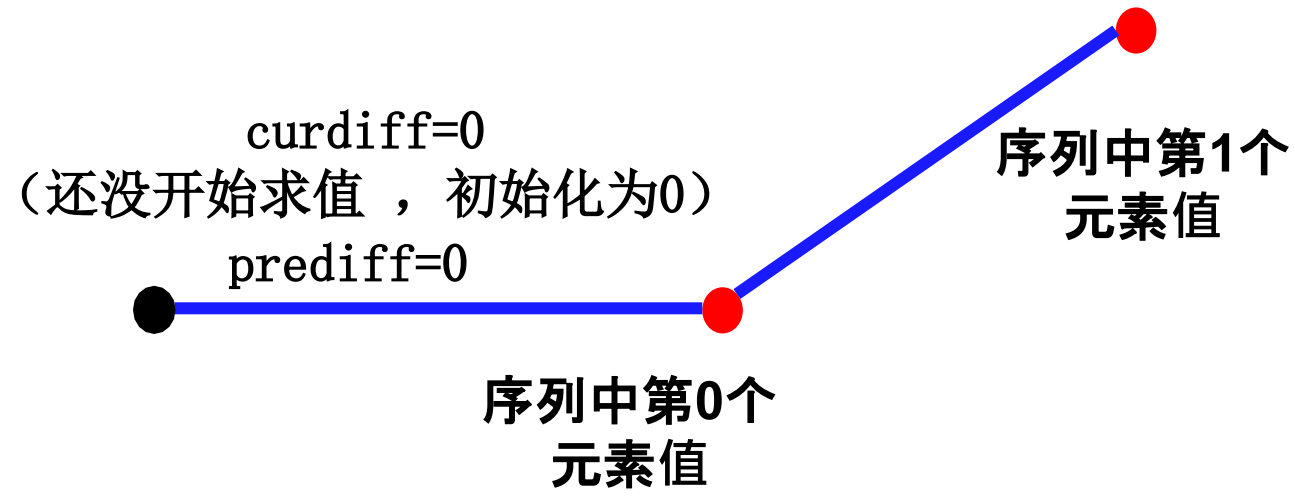
1：prediff和curdiff；2：result:记录摆动序列的长度



4.2 贪心法示例——摇摆序列

【相关变量】序列存放在nums[0..n-1]中

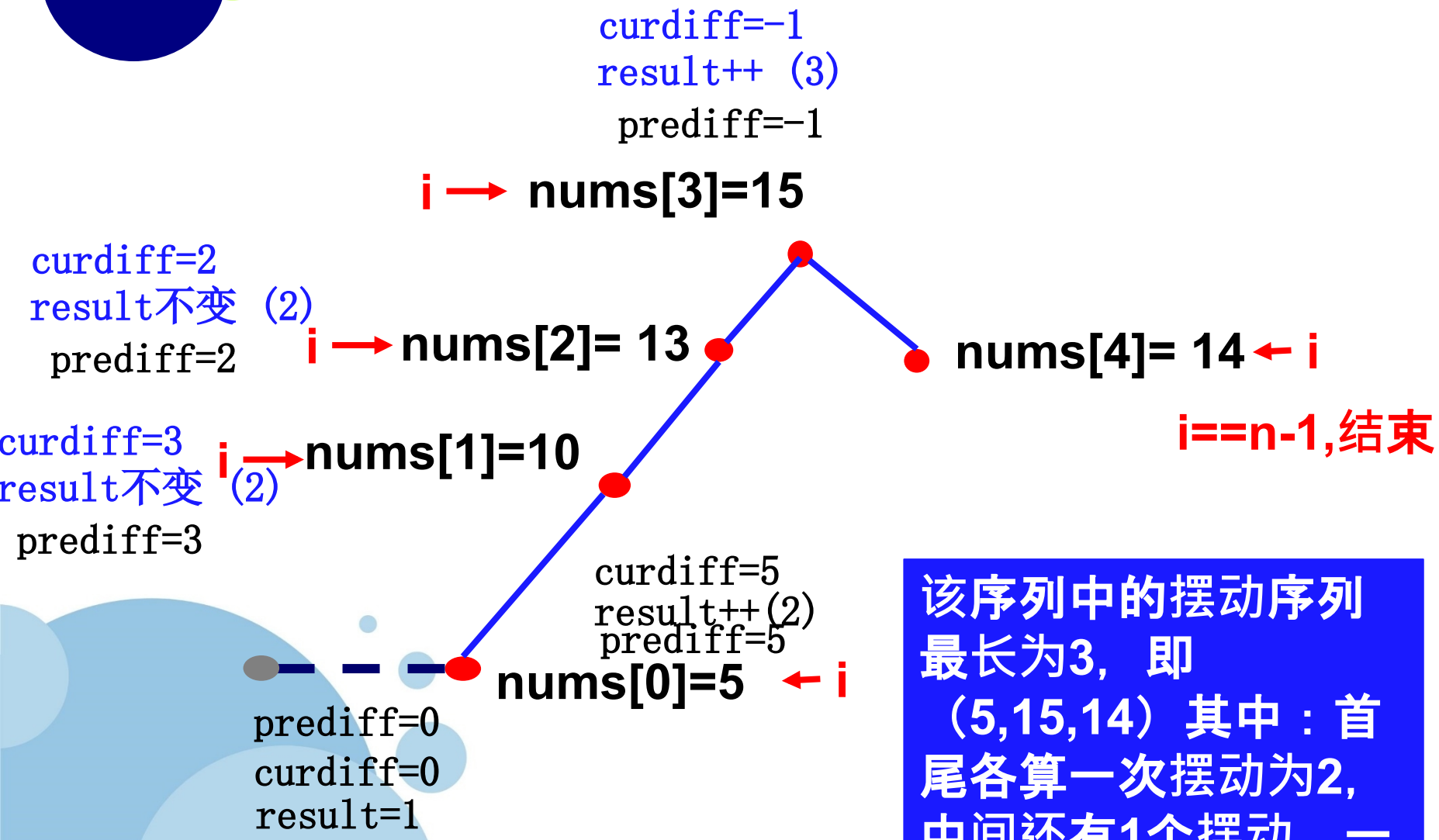
prediff和curdiff的初值：



4.2 贪心法示例——摇摆序列

```
int wiggleMaxLength(int nums[],int n)
{if (n==1) return 1);
int curDiff = 0; // 当前一对差值
int preDiff = 0; // 前一对差值
int result = 1;
// 记录峰值个数，序列默认序列最右边有一个峰值
for (int i = 0; i < n-1; i++)//最后一个已经计入“摆动”
{curDiff = nums[i + 1] - nums[i];
 if((preDiff <=0&&curDiff>0)||(preDiff >= 0 && curDiff<0))
  result++;
 preDiff = curDiff;
}
return result;
}
```

4.2 贪心法示例——摇摆序列

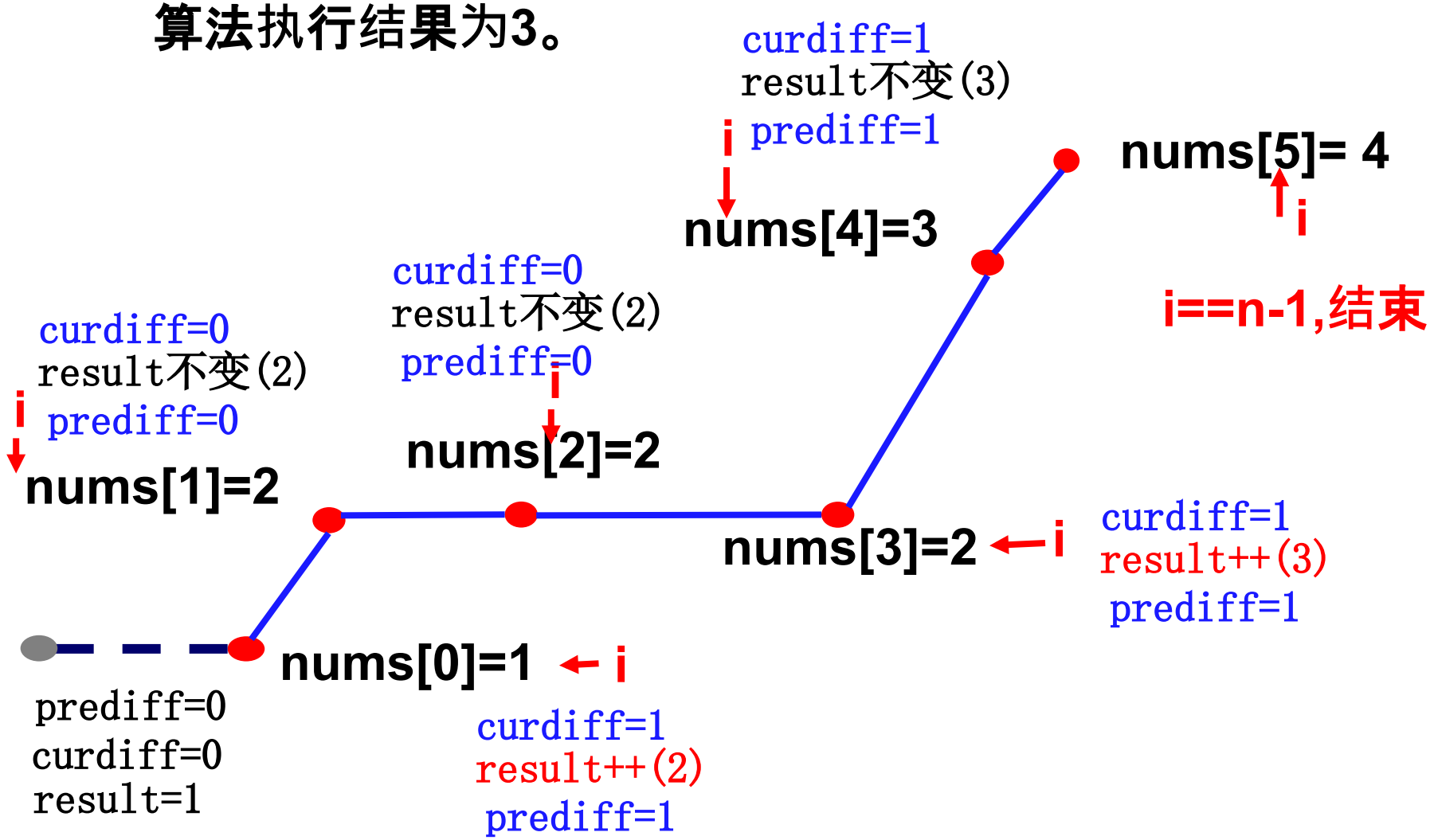


该序列中的摆动序列最长为3，即（5,15,14）其中：首尾各算一次摆动为2，中间还有1个摆动，一共有3个摆动。

Chapter 4

4.2 贪心法示例——摇摆序列

对于序列（1,2,2,2,3,4）其最长摆动序列为2。而算法执行结果为3。



4.2 贪心法示例——摇摆序列

```
int wiggleMaxLength(int nums[],int n)
{if (n==1) return 1);
int curDiff = 0; // 当前一对差值
int preDiff = 0; // 前一对差值
int result = 1;
// 记录峰值个数，序列默认序列最右边有一个峰值
for (int i = 0; i < n-1; i++)//最后一个已经计入“摆动”
{curDiff = nums[i + 1] - nums[i];
 if((preDiff <=0&&curDiff>0)||(preDiff >= 0 && curDiff<0))
  result++;
 preDiff = curDiff;
}
return result;
}
```

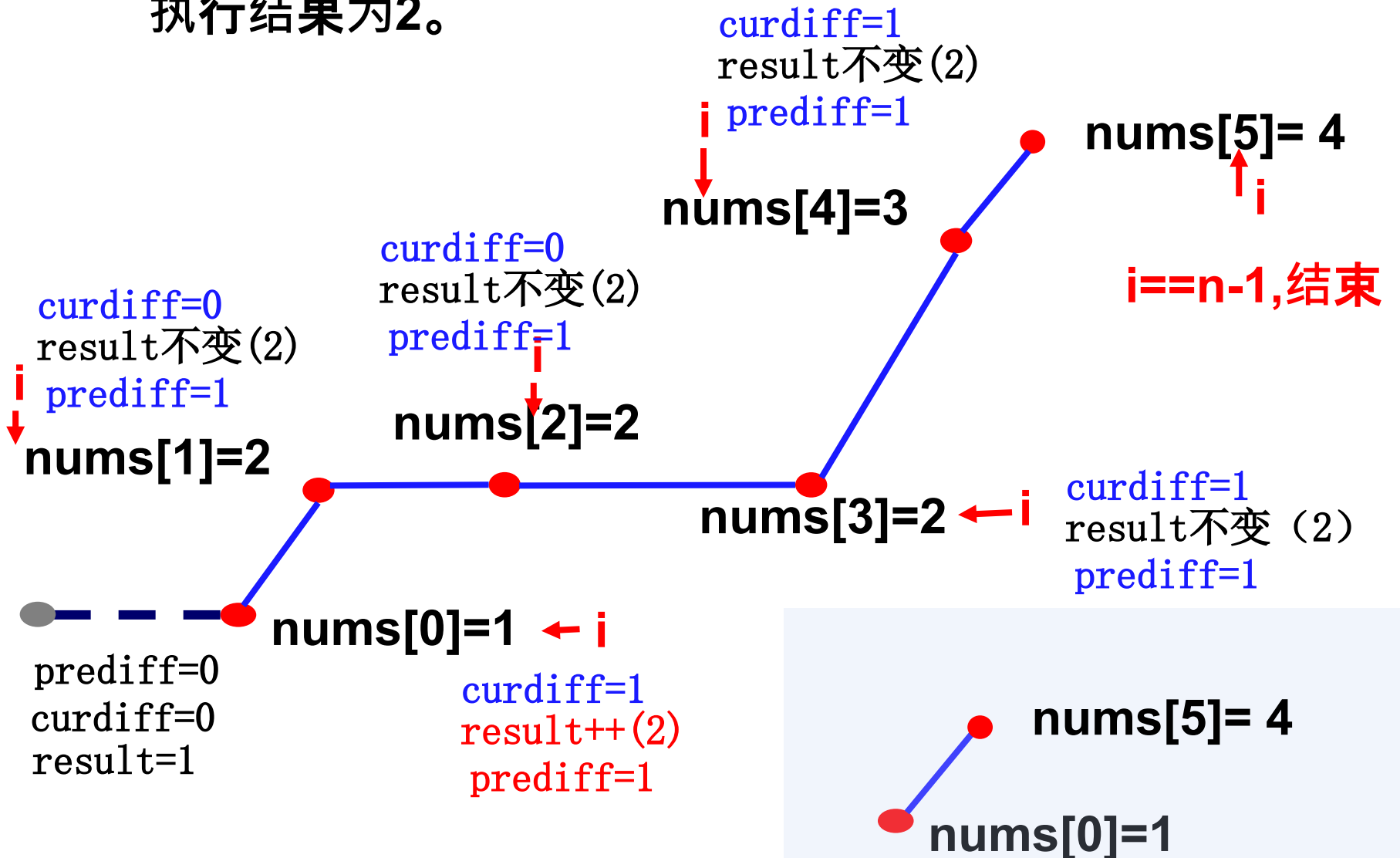
prediff不必每次紧跟curdiff,当出现摆动的时候，采用curdiff更新prediff。（prediff记录的是上次摆动时的差值）

4.2 贪心法示例——摇摆序列

```
int wiggleMaxLength(int nums[],int n)
{if (n==1) return 1);
int curDiff = 0; // 当前一对差值
int preDiff = 0; // 前一对差值
int result = 1;
// 记录峰值个数，序列默认序列最右边有一个峰值
for (int i = 0; i < n-1; i++)//最后一个已经计入“摆动”
{curDiff = nums[i + 1] - nums[i];
 if((preDiff <=0&&curDiff>0)||(preDiff >= 0 && curDiff<0))
   {result++; preDiff = curDiff; }
}
return result;
}
```

4.2 贪心法示例——摇摆序列

对于序列（1,2,2,2,3,4）其最长摆动序列为2。改进后算法执行结果为2。



Chapter 4

4.2 贪心法示例——跳跃游戏

【问题描述】给定一个非负整数数组，你最初位于数组的第一个位置。数组中的每个元素代表你在该位置可以跳跃的最大长度。判断是否能够到达最后一个位置。

输入1：[2,3,1,1,4]　　输出1：true

解释1：从位置1跳1步到3，然后跳3步到达最后一个位置。

输入2：[3,2,1,0,4]　　输出2：false

解释2：无论怎样，总会到达值为0的位置，所以永远不可能到达最后一个位置。

Chapter 4

4.2 贪心法示例——跳跃游戏

当前位置元素如果是3：

下一个决策：可以跳1步，可以跳2步，最多跳3步。

Q：那究竟是跳1步，还是2步，还是3步呢？究竟跳几步才是最优？

A：其实跳几步无所谓，关键在于可跳的**覆盖范围**！每次取最大的跳跃步数，这个就是可以跳跃的覆盖范围。在覆盖范围内，不关心是怎么跳的，一定是可以跳过来。

【问题转化】

问题就转化为**跳跃覆盖范围**究竟可不可以覆盖到**终点**！

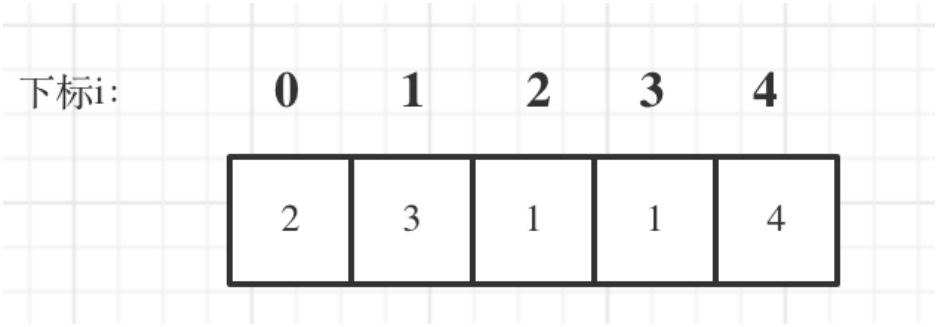
每次移动取最大跳跃步数（得到最大的覆盖范围），每移动一个单位，就更新最大覆盖范围。

4.2 贪心法示例——跳跃游戏

贪心算法：

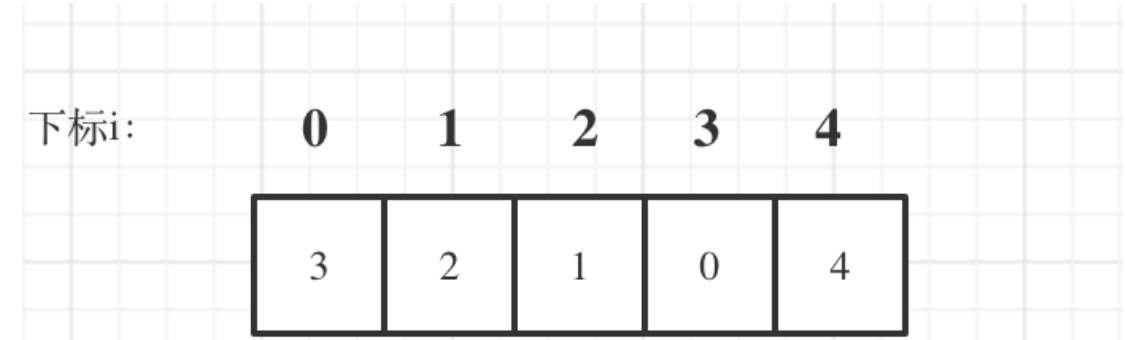
局部最优解： 每次取最大跳跃步数（取最大覆盖范围）。

整体最优解： 最后得到整体最大覆盖范围，看是否能覆盖到终点。



Chapter 4

4.2 贪心法示例——跳跃游戏



cover

i

i+nums[i]>cover,cover=i+nums[i]=3；i++

cover

i

i+nums[i](3)==cover,cover为3；i++；

cover

i

i+nums[i](3)==cover,cover为3；i++；

cover

i

i+nums[i](3)==cover,cover为3；

i++；

cover

i

i=4>cover,cover=3

未到达终点4，返回0

4.2 贪心法示例——跳跃游戏

```
int  canJump(int nums[],int n) {
 int cover = 0;
 if (n== 1) return true; // 只有1个元素，就是能达到
 for (int i = 0; i <= cover; i++) {
 if(i + nums[i]>cover) cover = i + nums[i];
 if (cover >= n- 1) return 1; // 说明可以覆盖到终点了
}
 return 0;
 }
```

Chapter 4

4.2 贪心法示例——跳跃游戏2

【思考】给定一个非负整数数组(长度<=100)，你最初位于数组的第一个位置。数组中的每个元素(值在1-1000之间)代表你在该位置可以跳跃的最大长度。

假设总可以到达数组的最后一个位置，设计算法，如何使用最少的跳跃次数到达数组的最后一个位置。

输入：[2, 3, 1, 1, 4]　　输出：2

解释：从位置1跳1步到3，然后跳3步到达最后一个位置。

输入：[2, 3, 0, 1, 4]　　输出：2

解释：从位置1跳1步到3，然后跳3步到达最后一个位置

Chapter 4

4.2 贪心法示例——跳跃游戏2

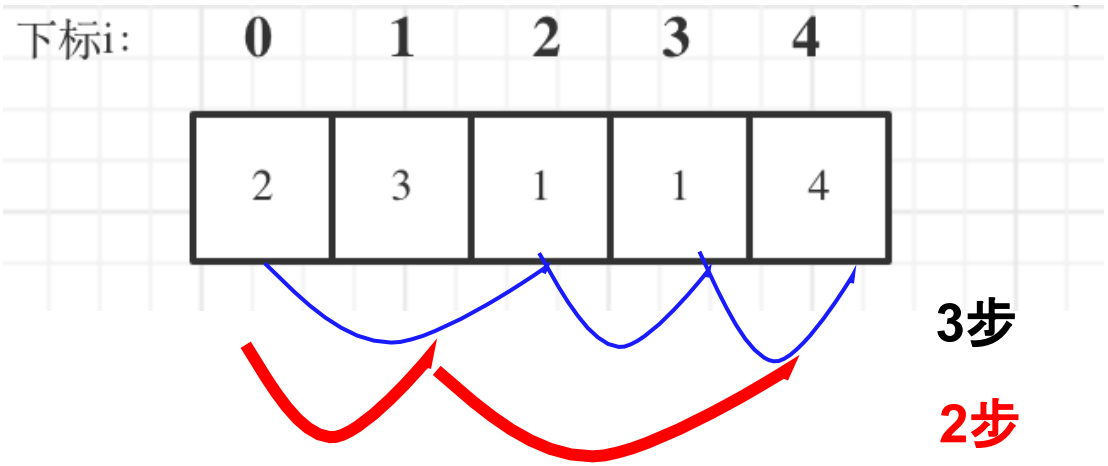
【目标】要计算跳跃到终点的最小步数。

【贪心的思路】

局部最优：从当前位置尽可能跳最远，这样距离终点更近，剩余需要跳跃的步数会更少。

整体最优：通过每一一步都尽可能多跳，从而达到最小步数。

是否能够得到全局最优？



4.2 贪心法示例——跳跃游戏2

【解决办法】仍从覆盖范围出发，不管怎么跳，覆盖范围内一定是可以跳到的，以最小的步数增加覆盖范围，覆盖范围一旦覆盖了终点，得到的就是最少步数！

【具体做法】需要统计两个覆盖范围：

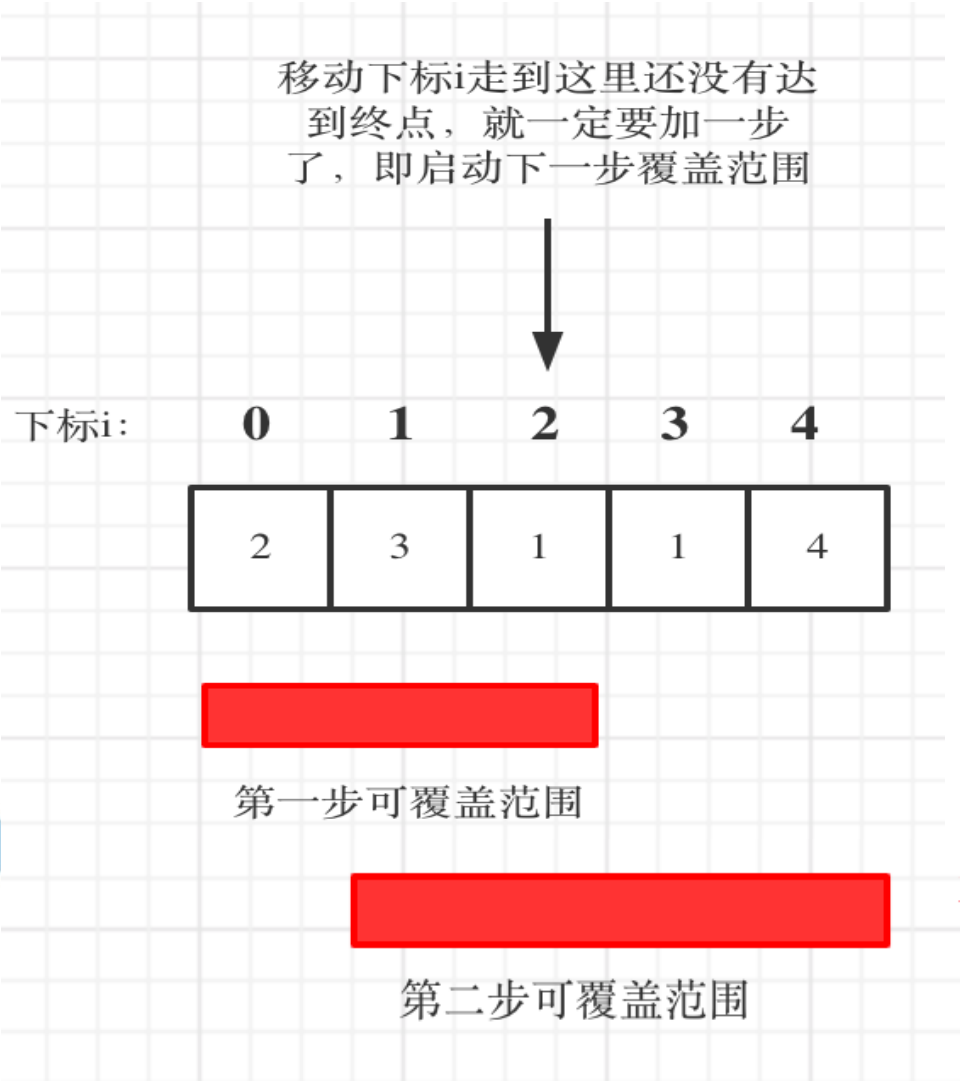
当前这一步的最大覆盖：curdistance;

下一步最大覆盖:nextdistance;

如果移动下标i达到了当前这一步的最大覆盖最远距离curdistance，还没有到终点，即curdistance ！=n-1，那么就必须再走一步（用nextdistance）来增加覆盖范围，直到覆盖范围覆盖了终点（n-1）。

Chapter 4

4.2 贪心法示例——跳跃游戏2

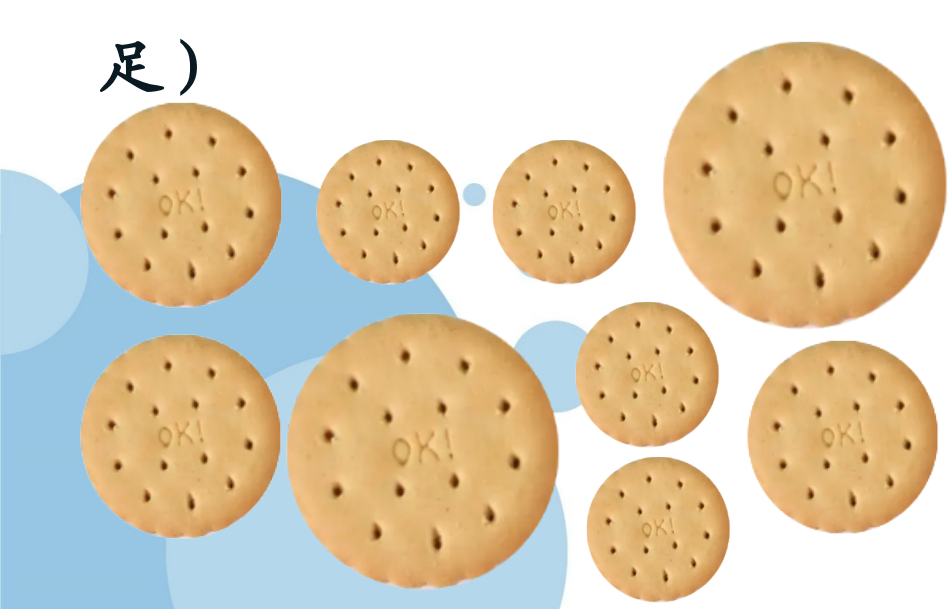


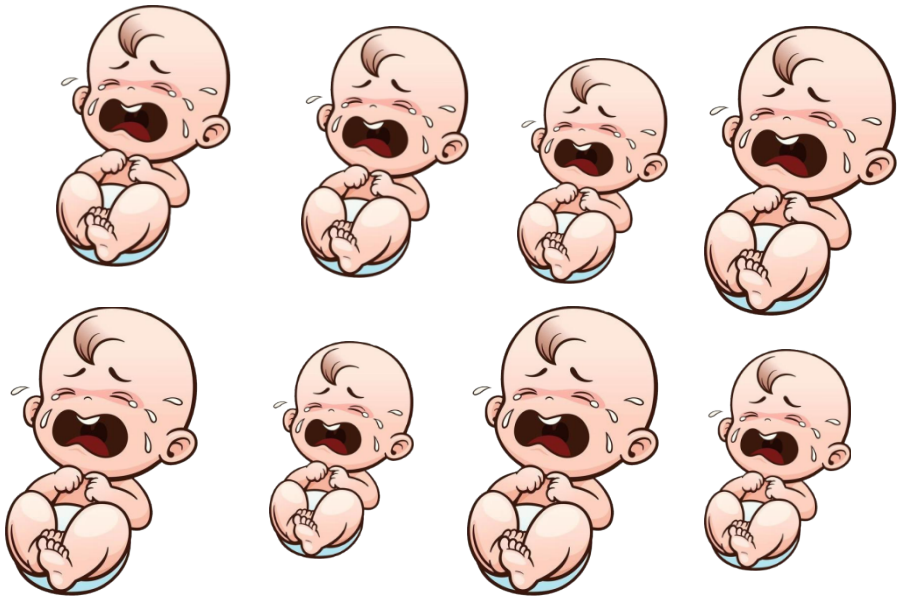
```
int jump(int nums[],int n) {
 if (n == 1) return 0;
 int curDistance = 0; // 当前覆盖最远距离下标
 int nextDistance = 0; // 下一步跳跃覆盖最远距离即下标
int ans = 0; // 记录到达终点的的跳跃步数
  for ( i = 0; i < n; i++) {
 if(nums[i] + i>nextDistance)
    nextDistance = nums[i] + i; // 更新下一步覆盖最远距离下标
if (i == curDistance) { // 遇到当前覆盖最远距离下标
        if (curDistance != n - 1) { // 如果当前覆盖最远距离下标不是终点
         ans++; // 需要跳跃下一步
        curDistance = nextDistance; // 更新当前覆盖最远距离下标
 if (nextDistance >= n - 1) break; // 下一步的覆盖范围已经可以达到终点，
结束循环
}
else break; // 当前覆盖最远距离下标是集合终点，不做ans++操作了，直接结束}
 }
return ans;
}
```

Chapter 4

4.2贪心法示例—课堂思考—分饼干

【问题描述】已知一些孩子和一些饼干，每个孩子有需求因子g(≧0)，每个饼干有大小s，当某个饼干的大小s>=某个孩子的需求因子g时，代表该饼干可以满足该孩子，求使用这些饼干，最多能满足多少孩子？（每个孩子最多只能用1个饼干满足）





Chapter 4

4.2 贪心法示例—课堂思考—分饼干

目标：尽可能多的满足更多的孩子

贪心策略1——从小孩胃口考虑：对胃口小的孩子，用尽可能小的饼干去满足该孩子，后面的孩子用同样的策略。即小胃口用小饼干。

小孩编号	1	2	3	4	5
胃口g	3	2	11	6	7

饼干编号	1	2	3	4	5	6	7	8
饼干大小s	3	3	2	1	1	5	5	6

Chapter 4

4.2 贪心法示例—课堂思考—分饼干

算法步骤1：

（1）对需求因子数组g和饼干大小数组s从小到大排序。

（2）从最小胃口孩子开始，用最小的饼干尝试满足他。若不能满足，则换下一个饼干。只有满足该孩子后，才能尝试给下一个孩子分饼干。直到饼干尝试用完或孩子都已得到满足。

小孩编号	1	2	3	4	5
胃口g	3	2	11	6	7

饼干编号	1	2	3	4	5	6	7	8
饼干大小s	3	3	2	1	1	5	5	6

Chapter 4

4.2 贪心法示例—课堂思考—分饼干

目标：尽可能多的满足更多的孩子

贪心策略2——从饼干角度考虑：用饼干去满足最接近其大小且可以满足的孩子，后面的孩子用同样的策略。即用大饼干满足大胃口孩子。

Chapter 4

4.2 贪心法示例—课堂思考—分饼干

算法步骤2：

（1）对需求因子数组g和饼干大小数组s从小到大排序。

（2）从最大饼干开始，去尝试满足胃口最大的孩子。若不能满足，则换下一个孩子。只有该饼干被分配后，才能尝试下一个饼干。直到孩子都得到满足或饼干已经尝试用完。

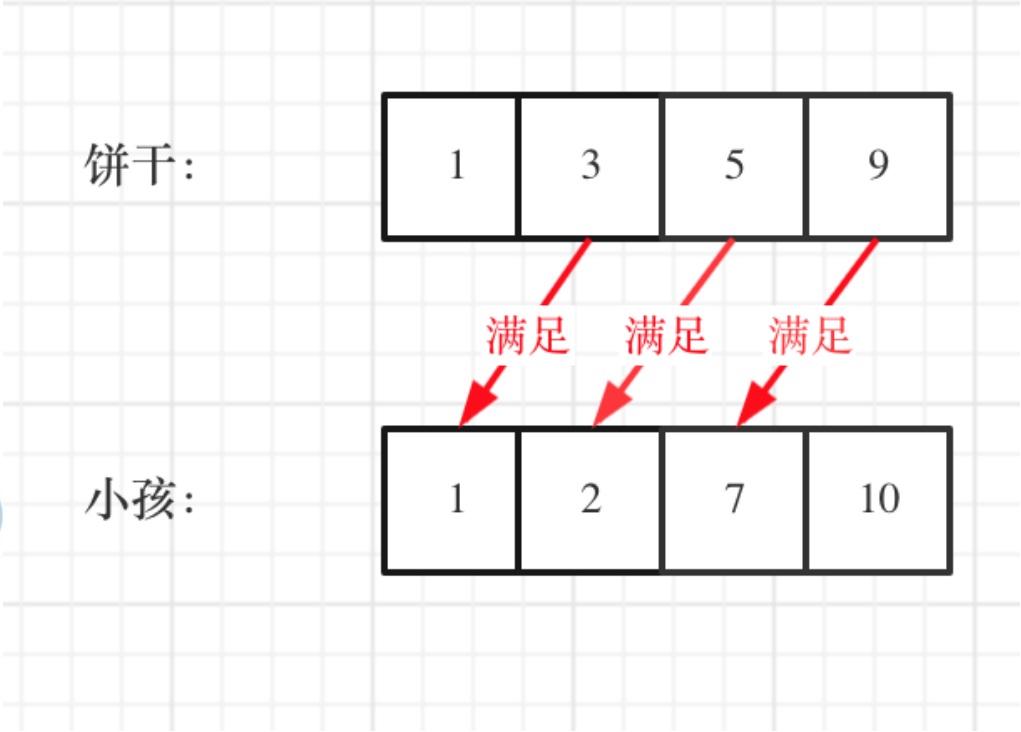
小孩编号	1	2	3	4	5
胃口g	3	2	11	6	7

饼干编号	1	2	3	4	5	6	7	8
饼干大小s	3	3	2	1	1	5	5	6

Chapter 4

4.2 贪心法示例—课堂思考—分饼干

贪心策略2——从饼干角度考虑：用饼干去满足最接近其大小且可以满足的孩子，后面的孩子用同样的策略。即用大饼干满足大胃口孩子。



Chapter 4

4.2 贪心法示例——分饼干

```
int findContentChildren(int g[],int gsize, int s[],int ssize)
{ sort(g,gsize); sort(s,ssize));
 int index = ssize- 1; // 饼干数组的最大下标
 int result = 0;
 for (int i=gsize-1; i>= 0; i--) {
    if (index >= 0 && s[index] >= g[i]){
       result++; index--;
     }
  }
return result;
}
```

Chapter 4

4.3 本章小结

✓贪心算法通过做一系列的贪心选择，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：

- 选择合适的贪心策略；
- 证明在此策略下，该问题具有贪心选择性质和最优子结构性质；
- 根据贪心策略，写出贪心选择的算法，求得最优解。

✓贪心算法经常需要排序。

✓贪心法不能保证问题总能得到最优解。

✓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。