一、定义

什么是贪心算法呢？所谓贪心算法是指，在对问题求解时，总是做出在当前看来最好的选择。也就是说，不从整体最优解出发来考虑，它所做出的仅是在某种意义上的局部最优解。

贪心算法不是对所有问题都能得到整体最优解，但对范围相当广泛的许多问题都能产生整体最优解或整体最优解的近似解。

贪心算法的基本思路如下：

1.建立数学模型来描述问题。

2.把求解的问题分成若干个子问题。

3.对每个子问题求解，得到每个子问题的局部最优解。

4.把每个子问题的局部最优解合成为原来问题的一个解。

实现该算法的过程：

从问题的某一初始状态出发；

while 能朝给定总目标前进一步 do

求出可行解的一个解元素；

由所有解元素组合成问题的一个可行解；

二、例题分析

[背包问题]有一个背包，背包容量是M=150。有7个物品，物品可以分割成任意大小。

要求尽可能让装入背包中的物品总价值最大，但不能超过总容量。

物品 A  B  C  D  E  F  G

重量 35 30 60 50 40 10 25

价值 10 40 30 50 35 40 30

记得当时学算法的时候，就是这个例子，可以说很经典。

分析：

目标函数： ∑pi最大

约束条件是装入的物品总重量不超过背包容量，即∑wi<=M( M=150)

（1）根据贪心的策略，每次挑选价值最大的物品装入背包，得到的结果是否最优？

（2）每次挑选所占重量最小的物品装入是否能得到最优解？

（3）每次选取单位重量价值最大的物品，成为解本题的策略?

贪心算法是很常见的算法之一，这是由于它简单易行，构造贪心策略简单。但是，它需要证明后才能真正运用到题目的算法中。一般来说，贪心算法的证明围绕着整个问题的最优解一定由在贪心策略中存在的子问题的最优解得来的。

对于本例题中的3种贪心策略，都无法成立，即无法被证明，解释如下：

（1）贪心策略：选取价值最大者。反例：

W=30

物品：A  B  C

重量：28 12 12

价值：30 20 20

根据策略，首先选取物品A，接下来就无法再选取了，可是，选取B、C则更好。

（2）贪心策略：选取重量最小。它的反例与第一种策略的反例差不多。

（3）贪心策略：选取单位重量价值最大的物品。反例：

W=30

物品：A  B  C

重量：28 20 10

价值：28 20 10

根据策略，三种物品单位重量价值一样，程序无法依据现有策略作出判断，如果选择A，则答案错误。

值得注意的是，贪心算法并不是完全不可以使用，贪心策略一旦经过证明成立后，它就是一种高效的算法。比如，求最小生成树的Prim算法和Kruskal算法都是漂亮的贪心算法。

［均分纸牌］有N堆纸牌，编号分别为1，2，…，n。每堆上有若干张,但纸牌总数必为n的倍数.可以在任一堆上取若干张纸牌,然后移动。移牌的规则为：在编号为1上取的纸牌，只能移到编号为2的堆上；在编号为n的堆上取的纸牌，只能移到编号为n-1的堆上；其他堆上取的纸牌，可以移到相邻左边或右边的堆上。现在要求找出一种移动方法，用最少的移动次数使每堆上纸牌数都一样多。例如：n=4，4堆纸牌分别为：① 9 ② 8 ③ 17 ④ 6 移动三次可以达到目的：从③取4张牌放到④ 再从③区3张放到②然后从②去1张放到①。

输入输出样例：4

9 8 17 6

屏幕显示：3

算法分析：设a[i]为第I堆纸牌的张数（0<=I<=n），v为均分后每堆纸牌的张数，s为最小移动次数。

我们用贪心算法，按照从左到右的顺序移动纸牌。如第I堆的纸牌数不等于平均值，则移动一次（即s加1），分两种情况移动：

1．若a[i]>v，则将a[i]-v张从第I堆移动到第I+1堆；

2．若a[i]<v，则将v-a[i]张从第I+1堆移动到第I堆。

为了设计的方便，我们把这两种情况统一看作是将a[i]-v从第I堆移动到第I+1堆，移动后有a[i]=v; a[I+1]=a[I+1]+a[i]-v.

在从第I+1堆取出纸牌补充第I堆的过程中可能回出现第I+1堆的纸牌小于零的情况。

如n=3，三堆指派数为1 2 27 ，这时v=10，为了使第一堆为10，要从第二堆移9张到第一堆，而第二堆只有2张可以移，这是不是意味着刚才使用贪心法是错误的呢？

我们继续按规则分析移牌过程，从第二堆移出9张到第一堆后，第一堆有10张，第二堆剩下-7张，在从第三堆移动17张到第二堆，刚好三堆纸牌都是10，最后结果是对的，我们在移动过程中，只是改变了移动的顺序，而移动次数不便，因此此题使用贪心法可行的。

Java源程序：

public class Greedy {

    public static void main(String[] args) {

  int n = 0, avg =0, s = 0;

  Scanner scanner = new Scanner(System.in);

  ArrayList<Integer> array = new ArrayList<Integer>();

  System.out.println("Please input the number of heaps:");

  n = scanner.nextInt();

  System.out.println("Please input heap number:");

  for (int i = 0; i < n; i++) {

   array.add(scanner.nextInt());

  }

  for(int i = 0; i < array.size(); i ++){

   avg += array.get(i);

  }

  avg = avg/array.size();

  System.out.println(array.size());

  System.out.println(avg);

  for(int i = 0; i < array.size()-1; i ++){

   s++;

   array.set(i+1, array.get(i+1)+array.get(i)-avg);

  }

  System.out.println("s:" + s);

 }

}

利用贪心算法解题，需要解决两个问题：

一是问题是否适合用贪心法求解。我们看一个找币的例子，如果一个货币系统有三种币值，面值分别为一角、五分和一分，求最小找币数时，可以用贪心法求解；如果将这三种币值改为一角一分、五分和一分，就不能使用贪心法求解。用贪心法解题很方便，但它的适用范围很小，判断一个问题是否适合用贪心法求解，目前还没有一个通用的方法，在信息学竞赛中，需要凭个人的经验来判断。

二是确定了可以用贪心算法之后，如何选择一个贪心标准，才能保证得到问题的最优解。在选择贪心标准时，我们要对所选的贪心标准进行验证才能使用，不要被表面上看似正确的贪心标准所迷惑，如下面的例子。

［最大整数］设有n个正整数，将它们连接成一排，组成一个最大的多位整数。

例如：n=3时，3个整数13，312，343，连成的最大整数为34331213。

又如：n=4时，4个整数7，13，4，246，连成的最大整数为7424613。

输入：n

N个数

输出：连成的多位数

算法分析：此题很容易想到使用贪心法，在考试时有很多同学把整数按从大到小的顺序连接起来，测试题目的例子也都符合，但最后测试的结果却不全对。按这种标准，我们很容易找到反例：12，121应该组成12121而非12112，那么是不是相互包含的时候就从小到大呢？也不一定，如12，123就是12312而非12123，这种情况就有很多种了。是不是此题不能用贪心法呢？

其实此题可以用贪心法来求解，只是刚才的标准不对，正确的标准是：先把整数转换成字符串，然后在比较a+b和b+a，如果a+b>=b+a，就把a排在b的前面，反之则把a排在b的后面。

java源程序：

public static void main(String[] args){

  String str = "";

  ArrayList<String> array = new ArrayList<String>();

  Scanner in = new Scanner(System.in);

  System.out.println("Please input the number of data:");

  int n = in.nextInt();

  System.out.println("Please input the data:");

  while (n-- > 0) {

   array.add(in.next());

  }

  for(int i = 0; i < array.size(); i ++)

   for(int j = i + 1; j < array.size(); j ++){

    if((array.get(i) + array.get(j)).compareTo(array.get(j) + array.get(i)) < 0){

     String temp = array.get(i);

     array.set(i, array.get(j));

     array.set(j, temp);

    }

   }

  for(int i = 0; i < array.size(); i ++){

   str += array.get(i);

  }

  System.out.println("str=:"+str);

 }

}

贪心算法所作的选择可以依赖于以往所作过的选择，但决不依赖于将来的选择，也不依赖于子问题的解，因此贪心算法与其他算法相比具有一定的速度优势。如果一个问题可以同时用几种方法解决，贪心算法应该是最好的选择之一。