**贪心算法**

**贪心算法简介：**

　　贪心算法是指：在每一步求解的步骤中，它要求“贪婪”的选择最佳操作，并希望通过一系列的最优选择，能够产生一个问题的（全局的）最优解。

　　贪心算法每一步必须满足一下条件：

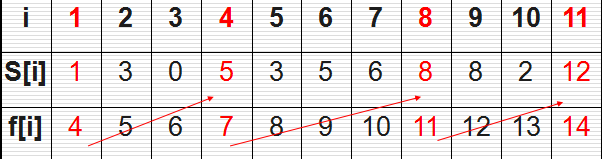
　　1、可行的：即它必须满足问题的约束。

　　2、局部最优：他是当前步骤中所有可行选择中最佳的局部选择。

　　3、不可取消：即选择一旦做出，在算法的后面步骤就不可改变了。

**贪心算法案例：**

1.活动选择问题  
 　　这是《算法导论》上的例子，也是一个非常经典的问题。有n个需要在同一天使用同一个教室的活动a1,a2,…,an，教室同一时刻只能由一个活动使用。每个活动ai都有一个开始时间si和结束时间fi 。一旦被选择后，活动ai就占据半开时间区间[si,fi)。如果[si,fi]和[sj,fj]互不重叠，ai和aj两个活动就可以被安排在这一天。该问题就是要安排这些活动使得尽量多的活动能不冲突的举行。例如下图所示的活动集合S，其中各项活动按照结束时间单调递增排序。



　　用贪心法的话思想很简单：活动越早结束，剩余的时间是不是越多？那我就早最早结束的那个活动，找到后在剩下的活动中再找最早结束的不就得了？

虽然贪心算法的思想简单，但是贪心法不保证能得到问题的最优解，如果得不到最优解，那就不是我们想要的东西了，所以我们现在要证明的是在这个问题中，用贪心法能得到最优解。

java代码实现：

[复制代码](javascript:void(0);)

1 public class ActiveTime {

2 public static void main(String[] args) {

3 //创建活动并添加到集合中

4 Active act1 = new Active(1, 4);

5 Active act2 = new Active(3, 5);

6 Active act3 = new Active(0, 6);

7 Active act4 = new Active(5, 7);

8 Active act5 = new Active(3, 8);

9 Active act6 = new Active(5, 9);

10 Active act7 = new Active(6, 10);

11 Active act8 = new Active(8, 11);

12 Active act9 = new Active(8, 12);

13 Active act10 = new Active(2, 13);

14 Active act11 = new Active(12, 14);

15 List<Active> actives = new ArrayList<Active>();

16 actives.add(act1);

17 actives.add(act2);

18 actives.add(act3);

19 actives.add(act4);

20 actives.add(act5);

21 actives.add(act6);

22 actives.add(act7);

23 actives.add(act8);

24 actives.add(act9);

25 actives.add(act10);

26 actives.add(act11);

27

28 List<Active> bestActives = getBestActives(actives, 0, 16);

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

30 System.out.println(bestActives.get(i));

31 }

32 }

33

34

35 /\*\*

36 \*

37 \* @param actives

38 \* 活动集合

39 \* @param startTime

40 \* 教室的开始使用时间

41 \* @param endTime

42 \* 教室的结束使用时间

43 \* @return

44 \*/

45 public static List<Active> getBestActives(List<Active> actives, int startTime, int endTime) {

46 //最佳活动选择集合

47 List<Active> bestActives = new ArrayList<Active>();

48 //将活动按照最早结束时间排序

49 actives.sort(null);

50 //nowTime 用来记录上次活动结束时间

51 int nowTime = startTime;

52 /\*\*

53 \* 因为我们已经按照最早结束时间排序，那么只要活动在时间范围内

54 \* actives.get(1)就应当是第一个活动的结束时间.

55 \* 则我们记录第一次活动结束的时间，在结合剩下的活动中，

56 \* 选取开始时间大于nowTime且结束时间又在范围内的活动，则为第二次活动时间，

57 \* 知道选出所有活动

58 \*/

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

60 Active act = actives.get(i);

61 if(act.getStartTime()>=nowTime&&act.getEndTime()<=endTime){

62 bestActives.add(act);

63 nowTime = act.getEndTime();

64 }

65 }

66 return bestActives;

67 }

68 }

69

70 /\*\*

71 \* 活动类

72 \* @CreatTime 下午9:45:37

73 \*

74 \*/

75 class Active implements Comparable<Active>{

76 private int startTime;//活动开始时间

77 private int endTime;//活动结束时间

78

79 public Active(int startTime, int endTime) {

80 super();

81 this.startTime = startTime;

82 this.endTime = endTime;

83 }

84

85 public int getStartTime() {

86 return startTime;

87 }

88

89 public void setStartTime(int startTime) {

90 this.startTime = startTime;

91 }

92

93 public int getEndTime() {

94 return endTime;

95 }

96

97 public void setEndTime(int endTime) {

98 this.endTime = endTime;

99 }

100

101 @Override

102 public String toString() {

103 return "Active [startTime=" + startTime + ", endTime=" + endTime + "]";

104 }

105

106 //活动排序时按照结束时间升序

107 @Override

108 public int compareTo(Active o) {

109 if(this.endTime>o.getEndTime()){

110 return 1;

111 }else if(this.endTime == o.endTime){

112 return 0;

113 }else{

114 return -1;

115 }

116 }

117

118

119 }

[复制代码](javascript:void(0);)

运行结果：

Active [startTime=1, endTime=4]

Active [startTime=5, endTime=7]

Active [startTime=8, endTime=11]

Active [startTime=12, endTime=14]

可以看出，求得的结果正好是最优解。

2.钱币找零问题  
这个问题在我们的日常生活中就更加普遍了。假设1元、2元、5元、10元、20元、50元、100元的纸币分别有c0, c1, c2, c3, c4, c5, c6张。现在要用这些钱来支付K元，至少要用多少张纸币？用贪心算法的思想，很显然，每一步尽可能用面值大的纸币即可。在日常生活中我们自然而然也是这么做的。在程序中已经事先将Value按照从小到大的顺序排好。

java代码实现：

[复制代码](javascript:void(0);)

1 package GreedyAlgorithm;

2

3 public class CoinChange {

4 public static void main(String[] args) {

5 //人民币面值集合

6 int[] values = { 1, 2, 5, 10, 20, 50, 100 };

7 //各种面值对应数量集合

8 int[] counts = { 3, 1, 2, 1, 1, 3, 5 };

9 //求442元人民币需各种面值多少张

10 int[] num = change(442, values, counts);

11 print(num, values);

12 }

13

14 public static int[] change(int money, int[] values, int[] counts) {

15 //用来记录需要的各种面值张数

16 int[] result = new int[values.length];

17

18 for (int i = values.length - 1; i >= 0; i--) {

19 int num = 0;

20 //需要最大面值人民币张数

21 int c = min(money / values[i], counts[i]);

22 //剩下钱数

23 money = money - c \* values[i];

24 //将需要最大面值人民币张数存入数组

25 num += c;

26 result[i] = num;

27 }

28 return result;

29 }

30

31 /\*\*

32 \* 返回最小值

33 \*/

34 private static int min(int i, int j) {

35 return i > j ? j : i;

36 }

37

38 private static void print(int[] num, int[] values) {

39 for (int i = 0; i < values.length; i++) {

40 if (num[i] != 0) {

41 System.out.println("需要面额为" + values[i] + "的人民币" + num[i] + "张");

42 }

43 }

44 }

45 }

[复制代码](javascript:void(0);)

运行结果：

需要面额为2的人民币1张

需要面额为5的人民币2张

需要面额为10的人民币1张

需要面额为20的人民币1张

需要面额为100的人民币4张

可以看出，求出的结果也刚好等于442元。正好为最优解。但是，当面额及数量为下种特殊情况时，贪心算法就无法给出最优解。

//人民币面值集合

6 int[] values = { 3, 5, 10, 20, 50, 100 };

7 //各种面值对应数量集合

8 int[] counts = { 3, 2, 1, 1, 3, 5 };  
需要求得money = 416元

运行结果如下：

需要面额为5的人民币1张

需要面额为10的人民币1张

需要面额为100的人民币4张

 于是我们可以看出，有些情况，贪心算法确实可以给出最优解，然而，还有一些问题并不是这种情况。对于这种情况，我们关心的是近似解，或者只能满足于近似解，贪心算法也是有价值的。