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。

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

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。

例子：n 场演唱会的问题

```
import java.util.ArrayList;
import java.util.List;

public class ActiveTime {
    public static void main(String[] args) {
        // 创建活动并添加到集合中
        Active act1 = new Active(1, 4);
        Active act2 = new Active(3, 5);
        Active act3 = new Active(0, 6);
        Active act4 = new Active(5, 7);
        Active act5 = new Active(3, 8);
        Active act6 = new Active(5, 9);
        Active act7 = new Active(6, 10);
        Active act8 = new Active(8, 11);
```

```
        Active act9 = new Active(8, 12);
        Active act10 = new Active(2, 13);
        Active act11 = new Active(12, 14);
        List<Active> actives = new ArrayList<Active>();
        actives.add(act1);
        actives.add(act2);
        actives.add(act3);
        actives.add(act4);
        actives.add(act5);
        actives.add(act6);
        actives.add(act7);
        actives.add(act8);
        actives.add(act9);
        actives.add(act10);
        actives.add(act11);
        List<Active> bestActives = getBestActives(actives, 0, 16);
        for (int i = 0; i < bestActives.size(); i++) {
            System.out.println(bestActives.get(i));
        }
    }

    /**
     *
     * @param actives      活动集合
     * @param startTime  教室的开始使用时间
     * @param endTime      教室的结束使用时间
     * @return
     */
    public static List<Active> getBestActives(List<Active> actives, int startTime, int endTime) {
        // 最佳活动选择集合
        List<Active> bestActives = new ArrayList<Active>();
        // 将活动按照最早结束时间排序
        actives.sort(null);
        // nowTime 用来记录上次活动结束时间
        int nowTime = startTime;
        /**
         * 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 actives.get(1)就应当是第一个活动的结束时间.
         * 则我们记录第一次活动结束的时间，在结合剩下的活动中， 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， 知道选出所有活动
         */
        for (int i = 0; i < actives.size(); i++) {
            Active act = actives.get(i);
            if (act.getStartTime() >= nowTime && act.getEndTime() <= endTime) {
```

```
                    bestActives.add(act);
                    nowTime = act.getEndTime();
                }
            }
            return bestActives;
        }
}

/**
 * 活动类
 *
 * @CreatTime 下午 9:45:37
 *
 */
class Active implements Comparable<Active> {
    private int startTime;// 活动开始时间
    private int endTime;// 活动结束时间

    public Active(int startTime, int endTime) {
        super();
        this.startTime = startTime;
        this.endTime = endTime;
    }

    public int getStartTime() {
        return startTime;
    }

    public void setStartTime(int startTime) {
        this.startTime = startTime;
    }

    public int getEndTime() {
        return endTime;
    }

    public void setEndTime(int endTime) {
        this.endTime = endTime;
    }

    @Override
    public String toString() {
        return "Active [startTime=" + startTime + ", endTime=" + endTime + "]";
    }
```

```
    // 活动排序时按照结束时间升序
    @Override
    public int compareTo(Active o) {
        if (this.endTime > o.getEndTime()) {
            return 1;
        } else if (this.endTime == o.endTime) {
            return 0;
        } else {
            return -1;
        }
    }
}
```

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。

```
public class CoinChange {
    public static void main(String[] args) {
        // 人民币面值集合
        int[] values = { 1, 2, 5, 10, 20, 50, 100 };
        // 各种面值对应数量集合
        int[] counts = { 3, 1, 2, 1, 1, 3, 5 };
        // 求 442 元人民币需各种面值多少张
        int[] num = change(442, values, counts);
        print(num, values);
    }

    public static int[] change(int money, int[] values, int[] counts) {
        // 用来记录需要的各种面值张数
        int[] result = new int[values.length];

        for (int i = values.length - 1; i >= 0; i--) {
            int num = 0;
            // 需要最大面值人民币张数
            int c = min(money / values[i], counts[i]);
            // 剩下钱数
            money = money - c * values[i];
            // 将需要最大面值人民币张数存入数组
            num += c;
            result[i] = num;
```

```
        }
        return result;
    }

    /**
     * 返回最小值
     */
    private static int min(int i, int j) {
        return i > j ? j : i;
    }

    private static void print(int[] num, int[] values) {
        for (int i = 0; i < values.length; i++) {
            if (num[i] != 0) {
                System.out.println("需要面额为" + values[i] + "的人民币" + num[i] + "张");
            }
        }
    }
}
```

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。