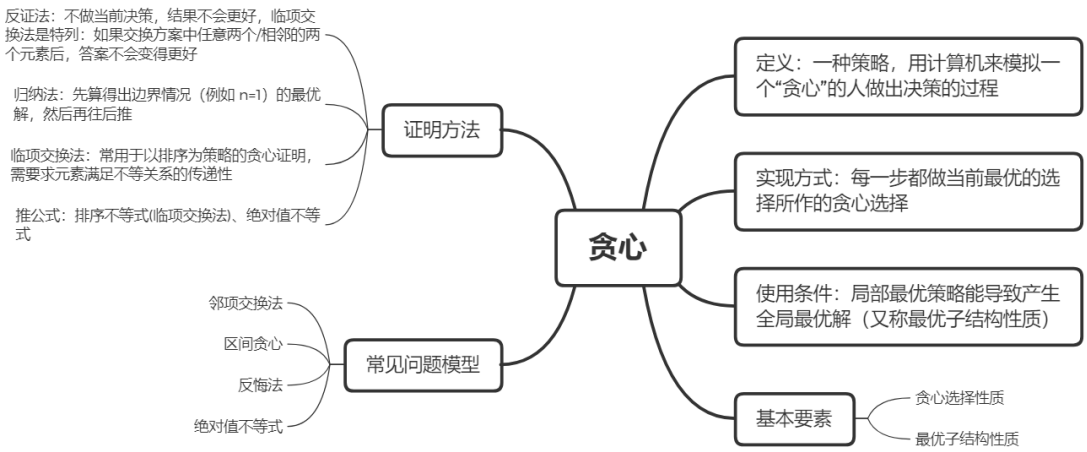
蓝桥杯30天算法冲刺集训

Day-2 贪心

贪心导图



简介

贪心算法（英语：greedy algorithm），是用计算机来模拟一个“贪心”的人做出决策的过程。这个人十分贪婪，每一步行动总是按某种指标选取最优的操作。而且他目光短浅，总是只看眼前，并不考虑以后可能造成的影响。

可想而知，并不是所有的时候贪心法都能获得最优解，所以一般使用贪心法的时候，都要确保自己能证明其正确性。

详细介绍

适用范围

贪心算法在有最优子结构的问题中尤为有效。最优子结构的意思是问题能够分解成子问题来解决，子问题的最优解能递推到最终问题的最优解，也即 **局部最优能推出全局最优。**

证明方法

贪心算法有两种证明方法：反证法和归纳法。一般情况下，一道题只会用到其中的一种方法来证明。

1. 反证法：如果交换方案中任意两个元素/相邻的两个元素后，答案不会变得更好，那么可以推定目前的解已经是最优解了。
2. 数学归纳法：先算得出边界情况（例如 $n = 1$ ）的最优解 F_1 ，然后再证明：对于每个 n ， F_{n+1} 都可以由 F_n 推导出结果，这个也就是按照“局部最优能推出全局最优”的过程一步一步从边界归纳出全局解的过程。

基本要素

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。这是贪心算法可行的第一个基本要素，也是贪心算法与动态规划算法的主要区别。

动态规划算法通常以自底向上的方式解各子问题，而贪心算法则通常以自顶向下的方式进行，以迭代的方式作出相继的贪心选择，每作一次贪心选择就将所求问题简化为规模更小的子问题。

对于一个具体问题，要确定它是否具有贪心选择性质，必须证明每一步所作的贪心选择最终导致问题的整体最优解。

2. 最优子结构性质

当一个问题的最优解包含其子问题的最优解时，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算法求解的关键特征。

基本思路

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（若干个决策）。

3.对每一子问题求解，得到子问题的局部最优解。

4.把子问题的解局部最优解合成原来解问题的一个解。

贪心算法适用的问题

贪心策略的前提是：**局部最优策略能导致产生全局最优解。**

实际上，贪心算法使用的情况比较少，一般对一个问题分析是否适用于贪心算法，可以先选择该问题下的几个实际数据进行分析可以做出判断。

贪心算法的实现框架

```
从问题的某一初始解出发：
while （能朝给定总目标前进一步）
{ 利用可行的决策，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。
```

贪心策略的选择

用贪心算法只能通过解局部最优解的策略来达到全局最优解，因此一定要注意判断问题是否适合采用贪心算法策略，找到解是否一定是问题的最优解。

要点

常见题型

在提高组难度以下的题目中，最常见的贪心有两种：

- 「我们将 XXX 按照某某顺序排序，然后按某种顺序（例如从小到大）选择。」。
- 「我们每次都取 XXX 中最大/小的东西，并更新 XXX。」（有时「XXX 中最大/小的东西」可以优化，比如用优先队列维护）

二者的区别在于一种是离线的，先处理后选择；一种是在线的，边处理边选择。

排序解法

用排序法常见的情况是输入一个包含几个（一般一到两个）权值的数组，通过排序然后遍历模拟计算的方法求出最优值。

后悔解法

思路是无论当前的选项是否最优都接受，然后进行比较，如果选择之后不是最优了，则反悔，舍弃掉这个选项；否则，正式接受。如此往复。

贪心算法与动态规划区别

1. 背景介绍：这两种算法都是选择性算法，就是从一个候选集合中选择适当的元素加入解集合。

贪心算法的选择策略即贪心选择策略，通过对候选解按照一定的规则进行排序，然后就可以按照这个排好的顺序进行选择了，选择过程中仅需确定当前元素是否要选取，与后面的元素是什么没有关系。

动态规划的选择策略是试探性的，每一步要试探所有的可行解并将结果保存起来，最后通过回溯的方法确定最优解，其试探策略称为决策过程。

2. 主要不同：两种算法的应用背景很相近，针对具体问题，有两个性质是与算法选择直接相关的，最优子结构性质和贪心选择性质。

最优子结构性质是选择类最优解都具有的性质，即全优一定包含局优。

当时我们也提到了贪心选择性质，满足贪心选择性质的问题可用贪心算法解决，不满足贪心选择性质的问题只能用动态规划解决。可见能用贪心算法解决的问题理论上都可以利用动态规划解决，而一旦证明贪心选择性质，用贪心算法解决问题比动态规划具有更低的时间复杂度和空间复杂度。

蓝桥杯真题

巧克力(蓝桥杯Java2021B组国赛)

本题的贪心策略不是太好想，如果单纯按单价从小到大排序很容易举出反例，比如有一种巧克力保质期短，单价不是最低但也比较低。

```
4 3
1 4 2
2 1 1
6 5 2
```

如果我们单纯按单价排序拿，我们会先拿单价为 1 的巧克力，拿完之后单价为 2 的巧克力就过期了，我们只能拿单价为 6 的巧克力，最后求出来最小花费为 14。

但实际上我们应该先拿单价为 2 的巧克力，之后拿单价为 1 的巧克力，最后拿单价为 6 的巧克力，这样最小花费为 10。

所以我们在按保质期从小到大排序后，应该按以下步骤进行贪心：

- 首先我们应该倒着贪心，不是从第一天开始，而是从最后一天开始，同时我们从保质期长的巧克力开始取，这样保质期短的巧克力不会过期。
- 然后我们需要用一个优先队列去维护队，如果这种巧克力在当前天数没有过期就放入到优先队列中

（1）如果队列为空，说明当前已经没有巧克力可以吃了，也就是说这 x 天没有办法每一天都吃到巧克力

（2）如果队列不为空，吃掉优先队列顶部的一块巧克力，累加这块巧克力的单价，如果这块巧克力数量为 0，弹出优先队列

```
////java21环境下存在问题，洛谷和蓝桥云课旧版本java可以过
import java.util.Arrays;
import java.util.Comparator;
import java.util.Scanner;
import java.util.TreeSet;

class Chocolate {
    public long price;
    public long remainDays;
    public long num;

}
class MyComparator implements Comparator<Chocolate> {

    @Override
    public int compare(Chocolate o1, Chocolate o2) {
        if (o1.price == o2.price) {
            return o1.remainDays < o2.remainDays ? -1 : 1;
        } else {
            return o1.price < o2.price ? -1 : 1;
        }

    }

}

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        long x = sc.nextLong();
        int n = sc.nextInt();
        Chocolate[] chocolates = new Chocolate[n];

        for (int i = 0; i < n; i++) {
            chocolates[i] = new Chocolate();
            chocolates[i].price = sc.nextLong();
            chocolates[i].remainDays = sc.nextLong();
            chocolates[i].num = sc.nextLong();
        }
        Arrays.sort(chocolates, new MyComparator());
        TreeSet<Integer> perDays = new TreeSet<Integer>();
        for (Integer i = 1; i <= x; i++) {
            perDays.add(i);
        }
        int i = 0;
        long cost = 0;
        while (i < n && perDays.size() > 0) {
            while (perDays.size() > 0 && chocolates[i].num > 0 &&
chocolates[i].remainDays >= perDays.first()) {
                cost += chocolates[i].price;
                chocolates[i].num--;
                perDays.pollFirst();
            }
```

```
            i++;
        }
        if (perDays.size() != 0) {
            System.out.println(-1);
        } else {
            System.out.println(cost);
        }

    }

}
```

答疑(蓝桥杯C/C++2020B组国赛)

我们首先要思考一下每位同学需要等待的时间为 $s_i + a_i + e_i$，如果说我们让三项用时之和少的同学排在前面，让三项用时之和多的同学排在后面。

这也很好理解我们列一下两位同学的同学发消息时间之和的情况：
$(a_1 + s_1) + (a_1 + s_1 + e_1 + a_2 + s_2)$

我们假设第二名同学用时更多，他排在前面式子变成：$(a_2 + s_2) + (a_2 + s_2 + e_2 + a_1 + s_1)$

很明显 $(a_1 + s_1) + (a_1 + s_1 + e_1 + a_2 + s_2) < (a_2 + s_2) + (a_2 + s_2 + e_2 + a_1 + s_1)$

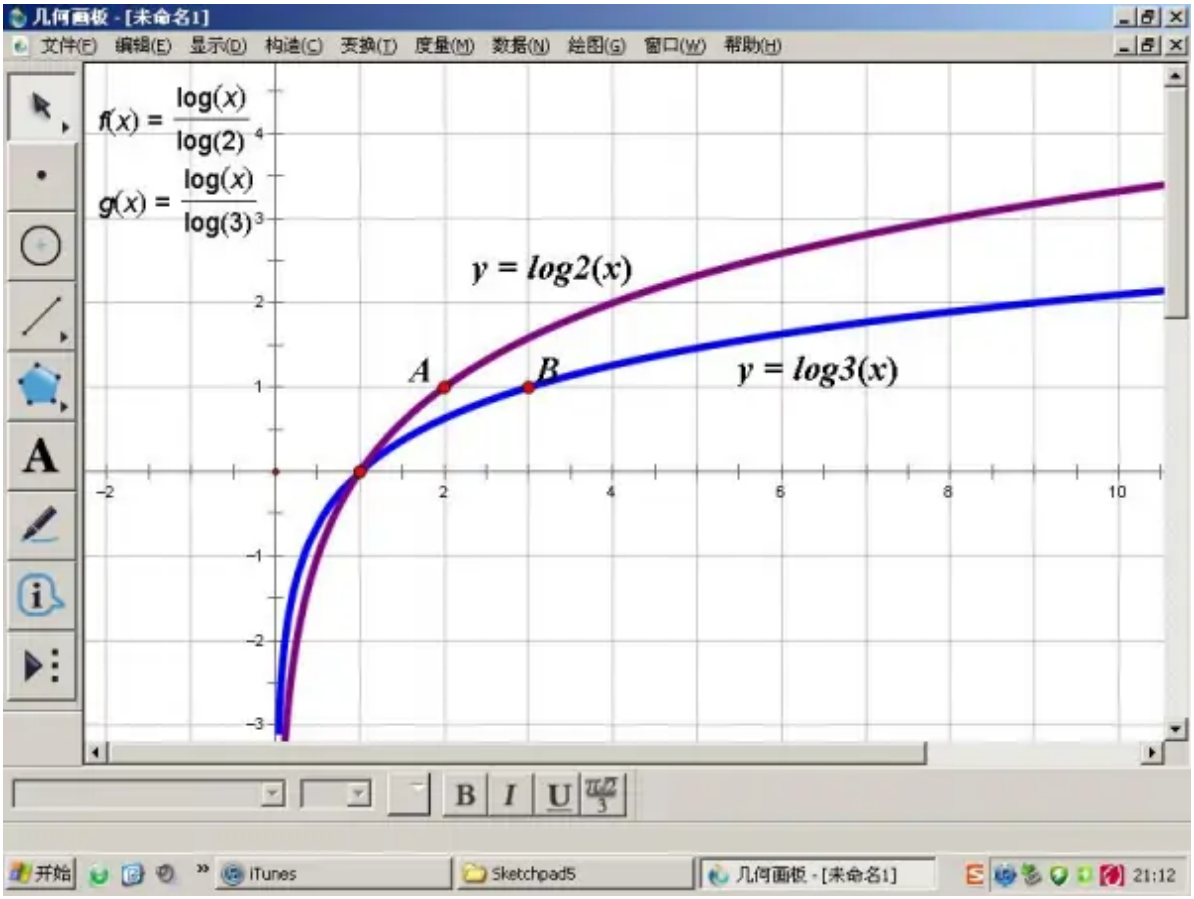
所以我们只需要按三项用时之和从小到大排序，然后计算发消息的时刻之和即可

```
import java.util.*;

class Main {
    static class Node implements Comparable<Node> {
        int a, s, e, sum;

        public Node(int a, int s, int e) {
            this.a = a;
            this.s = s;
            this.e = e;
            this.sum = a + s + e;
        }

        public int compareTo(Node other) {
            return Integer.compare(sum, other.sum);
        }
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        int N = 1010; // 根据 C++ 中的常量 N 的值设定数组大小
        Node[] t = new Node[N];

        int n = scanner.nextInt();

        for (int i = 1; i <= n; i++) {
            int a = scanner.nextInt();
            int s = scanner.nextInt();
            int e = scanner.nextInt();
            t[i] = new Node(a, s, e);
        }
```

```
        Arrays.sort(t, 1, n + 1);

        long res = 0, ans = 0;
        for (int i = 1; i <= n; i++) {
            res += t[i].a + t[i].s;
            ans += res;
            res += t[i].e;
        }
        System.out.println(ans);
    }
}
```

防御力（蓝桥杯Java2018B组国赛）

题目中 $A = 2^d$ 和 $B = 2^d$ 可得 $d = log_2(A) = log_3(B)$



由上图函数图像可以知道 $log_2(x)$ 的增幅要优于 $log_3(x)$，所以我们要将增加 B 的值从大到小排，把增加 A 的值从小到大排。这样尽可能的保证，B 的增加值转换为 A 的时候 $d = log_2(x)$ 能使防御力增幅尽可能的大，因为 B 增幅较大的都排在前面。同样的 A 的增加值转化为 B 的时候 $d = log_3(x)$ 能使防御力损失尽可能的少，因为 A 增幅较大的都排在后面。

比如对于样例

```
1 2
4
2 8
101

//A从小到大，B从大到小

1 2
4
8 2
101
```

- $d = log_3(1+8) = 2$
- $A = 2^d = 2^2 = 4 \quad d = log_2(4+4) = 3$
- $B = 3^d = 3^3 = 27 \quad d = log_3(27+2)$
- 所以最后最大的防御力就是 $d = log_3(29)$

```java
import java.util.*;

class Main {
    static class Add {
        int inc;
        char type;
        int idx;

        public Add(int inc, char type, int idx) {
            this.inc = inc;
            this.type = type;
            this.idx = idx;
        }
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int N = 2000010;
        Add[] addsA = new Add[N];
        Add[] addsB = new Add[N];

        int n1 = scanner.nextInt();
        int n2 = scanner.nextInt();

        for (int i = 0; i < n1; i++) {
            int inc = scanner.nextInt();
            addsA[i] = new Add(inc, 'A', i + 1);
        }

        for (int i = 0; i < n2; i++) {
            int inc = scanner.nextInt();
            addsB[i] = new Add(inc, 'B', i + 1);
        }

        String str = scanner.next();

        Arrays.sort(addsA, 0, n1, new Comparator<Add>() {
            public int compare(Add a1, Add a2) {
                return Integer.compare(a1.inc, a2.inc);
```

```
            }
        });

        Arrays.sort(addsB, 0, n2, new Comparator<Add>() {
            public int compare(Add b1, Add b2) {
                return Integer.compare(b2.inc, b1.inc);
            }
        });

        int idxA = 0, idxB = 0;
        for (int j = 0; j < str.length(); j++) {
            char c = str.charAt(j);
            if (c == '1') {
                System.out.println("B" + addsB[idxB].idx);
                idxB++;
            } else if (c == '0') {
                System.out.println("A" + addsA[idxA].idx);
                idxA++;
            }
        }

        System.out.println("E");
    }
}
```

习题

[小蓝与磁盘]

[小蓝与字符串删除]

[小蓝与忍者游戏]