A 蓝的无幂之夜

考点	难度
自定义函数	1

题目分析

如题目描述，由于禁止在代码中出现 `math.h`，`pow`，`abs` 这三个字符串，因此我们不调用 `math.h` 库中的 `pow` 函数和 `fabs` 函数，以及 `stdlib.h` 中的 `abs` 函数，而是自定义函数用于计算。

```
int myPow(int a, int b) {
    int ans = 1;
    for (int i = 0; i < b; i++) {
        ans = ans * a;
    }
    return ans;
}
int myAbs(int a) {
    if (a >= 0) {
        return a;
    } else {
        return -a;
    }
}
```

如果使用三目运算符和宏定义的话，`myAbs` 也可以写成如下形式

```
#define myAbs(a) ((a) > 0 ? (a) : (-(a)))
```

注意这些括号的使用，因为使用宏定义函数本质上是进行文本的替换，若不加上括号那么替换后的文本的运算顺序可能与你预期不相符，例如可以试试以下代码

```
#include <stdio.h>
#define myAbs(a) a > 0 ? a : -a
int main(void) {
    printf("%d", myAbs(1 - 3));
    return 0;
}
```

你会发现运行结果为 `-4`

在这个代码中，将`myAbs(1 - 3)`进行文本替换后得到的结果是

```
1 - 3 > 0 ? 1 - 3 : - 1 - 3
```

也即

```
-2 > 0 ? -2 : -4
```

这显然与我们所期望的效果不同，因此需要加上括号来矫正运算顺序。

如*Hint*所述，如果真的一定要使用库函数的话，也不是不可以，绕过字符串匹配的方法有很多，例如使用标记粘贴运算符

```
#define A(x) fab##s(x)
#define P(a,b) po##w(a,b)
```

以及使用宏延续运算符

```
#define H <math\
.h>
#define F po\
w
#define A fab\
s
```

示例代码

```
#include <stdio.h>
#define myAbs(a) ((a) > 0 ? (a) : (-(a)))
int myPow(int a, int b) {
    int ans = 1;
    for (int i = 0; i < b; i++) {
        ans = ans * a;
    }
    return ans;
}
int main(void) {
    int a, b;
    while (scanf("%d%d", &a, &b) != EOF) {
        printf("%d\n", myAbs(myPow(a, b) - myPow(b, a)));
    }
    return 0;
}
```

B 计算闪耀值

题目分析

根据题意，本题即计算 $\left\lfloor \frac{a!\ mod\ c}{b} \right\rfloor$ 的值与 $\left\lfloor \frac{b!\ mod\ c}{a} \right\rfloor$ 的值，并比较它们之差的绝对值是否大于9，自定义函数计算即可。注意阶乘结果的大小增长速度是相当快的，因此为了避免溢出，需要用到 $Hint$ 中提到的模乘的性质，**每乘一次就取一次模**。分式结果向下取整可以直接通过整除实现。

答疑过程中遇到有同学在计算阶乘模的时候，对每个 i 取模再乘到 sum 上，最后再对 sum 取模。这样的做法仍然**存在溢出风险**，如计算 $100!\ mod\ 101$ 的时候，因为 $1 \sim 100$ 之间的数都比 101 小，这样做实际上和直接计算 $100!$ 没有区别，仍然会造成溢出。因此不要只对乘法的乘数取模，而要对每一步乘法得到的结果取模。

此外，本题中对阶乘模的计算可以通过递归的方式实现。但是如果数据规模进一步增大，递归计算阶乘可能会因为递归层数过多而出现问题。使用循环的方式计算阶乘就不会出现这样的问题。因此，推荐在能使用循环的地方**尽量避免使用递归**。

示例代码

```c
#include <stdio.h>
#include <stdlib.h>

//函数定义语句也有声明的作用，自定义函数写在main函数前面就可以省去声明语句
int Modfact(int n, int mod) {
    int sum = 1;
    for (int i = 1; i <= n; ++i) {  //使用循环计算阶乘的模，在能用循环的时候要尽量避免使用递归
        sum = (sum * i) % mod;  //每乘一步都要取一次模
    }
    return sum % mod;   //保险起见，在返回值的地方再取一次模
}

int main() {
    int a, b, mod;
    scanf("%d%d%d", &a, &b, &mod);
    int ans1 = Modfact(a, mod) / b, ans2 = Modfact(b, mod) / a;
    printf("%d %d\n", ans1, ans2);
    if (abs(ans1 - ans2) > 9){
        printf("Mabushi~");
    }

    return 0;
}
```

C Josephus函数

题目分析

题目给出了 $J(n)$ 的递归表达式，按递归式写出递归函数计算即可。

示例代码

```c
#include <stdio.h>
int J(int);
int a[101], b[101], k;
int main()
{
    int t;
    scanf("%d%d", &k, &t);
    for (int i = 0; i < k; i++)
        scanf("%d", &a[i]);
    for (int i = 0; i < k; i++)
        scanf("%d", &b[i]);
    while (t--)
    {
        int n;
        scanf("%d", &n);
        printf("%d\n", J(n));
```

```
    }
    return 0;
}
int J(int N) //计算 J(N)
{            // N=nk+i，即i=N%k，n=N/k
    if (N < k)
        return a[N]; // N<k，即n=0，直接返回a[i]
    else
        return k * J(N / k) + b[N % k];
    //            递归计算J(n)   b[i]
}

```

D 地下墓穴的机关

题目分析

此题的函数表达式看似和本周学过的递归较像，但是：

在递归过程中，由于每一项与前三项相关，这样，在计算$f(n)$时，需要计算$f(n-3)$，$f(n-2)$，$f(n-1)$，而再计算这三项时，又要将前三项拆开计算，这样就会导致，每递归一层，计算的复杂度就会$\times 3$，从而到最后实际计算复杂度为$O(3^n)$

因此，此题用递归有非常大的超时风险，所以换一张方式思考，采用从1开始不断往后计算的方式，这样计算到$f(n)$时复杂度最多为$O(n)$，大大减少了时间开销。

示例代码

```
#include <stdio.h>
#define DIVISOR 1000000007

int Ms[1000];

int main()
{
    int n;
    Ms[0] = Ms[1] = Ms[2] = 1;

    scanf("%d", &n);
    for (int i = 3; i < n; i++) {
        Ms[i] = ((Ms[i - 3] + Ms[i - 2]) % DIVISOR + Ms[i - 1]) % DIVISOR;
    }

    printf("%d", Ms[n - 1]);

    return 0;
}
```

E 毕业，但是忘记了总学分

题目分析

首先统计已经修的总学分和要求的总学分之差，记为lim

并且计算已修的总学分乘以对应gpa之和

从lim开始，直到6.0,每次增加0.5，通过应有的gpa之和减去已修的总学分乘以gpa之和得到这门课的gpa

然后再求出gpa对应的分数输出

求gpa的过程和从gpa反解成绩的过程可以写成两个函数，简化程序

代码分析

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<ctype.h>
#include<string.h>
#include<time.h>
int n;
double m,s,cur;
double gpasum;
const double eps=1e-4;
//n,m,s按照题意
//cur为已修学分
//gpasum为没有除以总学分的gpa之和
double w[110];
double x[110];
double gpa(double x){
    if(x<60.0)return 0.0;
    return 4.0-3.0*(100.0-x)*(100.0-x)/1600.0;
}
double rev(double x){
    //gpa=(100.0-x)2/1600.0;
    x=4-x;
    x/=3.0;
    x*=1600.0;
    x=sqrt(x);
    return 100-x;
}
int main(){
    scanf("%d%lf%lf",&n,&m,&s);
    for(int i=1;i<=n-1;i++){
        scanf("%lf%lf",&w[i],&x[i]);
        cur+=w[i];
        gpasum+=w[i]*gpa(x[i]);
    }
    double lim=s-cur;
    for(double i=lim;i<=6.0;i+=0.5){
        double gparest=(cur+i)*m-gpasum;
        gparest/=i;
        double grade=rev(gparest);
        if(grade>0&&grade<100.0)
            printf("%.1lf %.3lf\n",i,grade);
    }
}
```

F 重炮的数学时间

题目分析

本题使用递归解决。

首先，我们需要使用两个数组，分别存储当前状态答案和当前状态有几个数字已经使用过。

假设我们当前已经搜索了 k 个数字（即答案数组中已经有 k 个数字），我们下一个搜索的数字应当是**当前答案数组中没有出现的最小数字**。我们找到这个最小数字，将这个数字标记为已经使用过，再进行下一步递归，继续寻找下一个数字。

当我们已经搜索完所有 n 个数字时，进行输出即可。

在搜索完一次答案之后，需要将当前这个状态记录到答案数组中的数字重新标记为未使用的状态。

标准程序

```c
#include <stdio.h>

int used[100];
int ans[100];
void DFS(int n, int k);

int main(){
    int n;
    scanf("%d", &n);
    DFS(n, 0);
    return 0;
}

void DFS(int n, int k){ // k记录当前的答案数组里已经记录了几个数字
    if(k == n){  // n是我们总共需要几个数字
        for(int i = 0; i < n; i++)  // 当前答案数组记录了n个数字时，达到递归边界，进行输出。
            printf("%d ", ans[i]);
        printf("\n");
        return;
    }
    for(int i = 1; i <= n; i++){
        if(used[i] == 0){   //判断是否使用过这个数字
            used[i] = 1;    //记录已经使用过这个数字
            ans[k] = i;     //将这个数字存入答案数组
            DFS(n, k + 1);  //进行下一步递归
            used[i] = 0;    //回溯的时候将这个数字重新标记为未使用的状态
        }
    }
    return;
}
```

G 哪吒的水题（二）

题目分析

设供水站横坐标为 x，则供水站到 $Simon$ 家 (x_1, y_1) 的距离为 $\sqrt{(x-x_1)^2+y_1^2}$，到哪吒家 (x_2, y_2) 距离为 $\sqrt{(x-x_2)^2+y_2^2}$，因此总费用 $s(x)=s_1\sqrt{(x-x_1)^2+y_1^2}+s_2\sqrt{(x-x_2)^2+y_2^2}$.

显然，供水站最佳位置的横坐标在区间 $[x_1,x_2]$ 中，我们要在该区间内找到一个合适的 x 使得 $s(x)$ 最小，并输出此时的 x 和 $s(x)$.

法一：三分法

易知 $s(x)$ 一般情况下是关于 x 的**先减后增函数**，可以选择使用**三分法**求该函数极小值点。

三分法类似二分法，可以用于求**单峰函数**的**极值点**，下面以本题为例对其思路做简单说明。

设极小值点为 $(x_0,s(x_0))$. 初始化 $l=x_1,r=x_2$ ，每次循环，在区间 $[l,r]$ 中选择两个点 $mid1<mid2$ ，如区间 $[l,r]$ 的三等分点 $mid1=\frac{2l+r}{3},mid2=\frac{l+2r}{3}$ 。如果 $s(mid1)>s(mid2)$ 则说明 x_0 在区间 $[mid1,r]$ 中（若在 $[l,mid1)$ 中，则说明 $s(x)$ 在 $[x_0,r]$ 上单调递增，与 $s(mid1)>s(mid2)$ 矛盾），更新 $l=mid1$ ；否则说明 x_0 在区间 $[l,mid2]$ 中，更新 $r=mid2$ 。每次循环缩小区间长度为原来的 $\frac{2}{3}$ ，当 $r-l$ 缩小到足够小时，即可得到满足精度要求的答案。

示例代码

```c
#include <stdio.h>
#include <math.h>
#define eps (1e-8)
//math.h库中有个函数名字叫y1，所以不能给自己的变量起名字叫y1，什么奇怪的事情发生了
double a, b, c, d, s1, s2;//将坐标a, b, c, d设为全局变量，方便下面的s函数使用。
double s(double x)
{
    return s1 * sqrt((x - a) * (x - a) + b * b) + s2 * sqrt((x - c) * (x - c) + d * d);
}
int main()
{
    scanf("%lf%lf%lf%lf%lf%lf", &a, &b, &c, &d, &s1, &s2);
    double l = a, r = c, mid1 = (2 * l + r) / 3, mid2 = (l + 2 * r) / 3;
    while(r - l > eps) {
        if(s(mid1) > s(mid2)) l = mid1;
        else r = mid2;
        mid1 = (2 * l + r) / 3;
        mid2 = (l + 2 * r) / 3;
    }
    printf("%.3f %.3f", mid1, s(mid1));
}
```

法二：二分法

求函数极值点可以使用求导函数零点的方法。

$s(x)$ 是关于 x 的先减后增函数，$s'(x)=s_1\frac{x-x_1}{\sqrt{(x-x_1)^2+y_1^2}}+s_2\frac{x-x_2}{\sqrt{(x-x_2)^2+y_2^2}}$ ，一般情况下**先负后正**，在区间 $[x_1,x_2]$ 上只有一个零点，可以用**二分法**求该函数**零点**。

示例代码

```c
#include <stdio.h>
#include <math.h>
#define eps (1e-8)
double a, b, c, d, s1, s2;
double s(double x)
{
```

```
    return s1 * sqrt((x - a) * (x - a) + b * b) + s2 * sqrt((x - c) * (x - c) + d * d);
}
double f(double x) //s关于x的导函数
{
    return s1 * (x - a) / sqrt((x - a) * (x - a) + b * b) + s2 * (x - c) / sqrt((x - c) * (x - c) + d * d);
}
int main()
{
    scanf("%lf%lf%lf%lf%lf%lf", &a, &b, &c, &d, &s1, &s2);
    double l = a, r = c, mid = (l + r) / 2;
    while(r - l > eps) {
        if(f(mid) > 0) r = mid;
        else l = mid;
        mid = (l + r) / 2;
    }
    printf("%.3f %.3f", mid, s(mid));
}
```

补充

1. 两种方法时间复杂度均为 $O(log(\frac{x_2-x_1}{eps}))$.
2. 特殊情况下，函数可能为单增函数或者单减函数，但是不影响做法的正确性。
3. 三分法选取 $mid1, mid2$ 时，选择区间的三等分点可以使每次循环缩小区间长度为原来的 $\frac{2}{3}$ ，如果选择靠近区间中点的两个点，可以使每次循环缩小区间长度接近原来的 $\frac{1}{2}$ 。

```
mid1 = (l + r) / 2 - eps;
//mid1 = (5001 * l + 5000 * r) / 10001;
mid2 = (l + r) / 2 + eps;
//mid2 = (5000 * l + 5001 * r) / 10001;
```

但注意避免死循环，这里的eps数量级要小于循环条件中的eps.

示例代码

```
#include <stdio.h>
#include <math.h>
#define eps1 (1e-6)
#define eps2 (1e-8)
double a, b, c, d, s1, s2;
double s(double x)
{
    return s1 * sqrt((x - a) * (x - a) + b * b) + s2 * sqrt((x - c) * (x - c) + d * d);
}
int main()
{
    scanf("%lf%lf%lf%lf%lf%lf", &a, &b, &c, &d, &s1, &s2);
    double l = a, r = c, mid1 = (l + r) / 2 - eps2, mid2 = (l + r) / 2 + eps2;
    while(r - l > eps1) {
        if(s(mid1) > s(mid2)) l = mid1;
        else r = mid2;
        mid1 = (l + r) / 2 - eps2;
        //mid1 = (5001 * l + 5000 * r) / 10001;
```

```
        mid2 = (l + r) / 2 + eps2;
        //mid2 = (5000 * l + 5001 * r) / 10001;
    }
    printf("%.3f %.3f", mid1, s(mid1));
}
```

4. 这道题精度要求比较仁慈，精确到 3 位小数，由于数据范围在 10^4 以内，结果居然有人用暴力遍历的做法找最小值，也可以通过。如果精度要求高一点点或者数据范围大一点点，暴力遍历就一定会超时。因此还是希望大家掌握三分法和二分法。

PS: 另外这道题和光的折射十分类似， s_1, s_2 可看作光在两个介质中的折射率，供水站位置应满足折射定律，即入射角和出射角的正弦之比应该等于 $\frac{s_2}{s_1}$ ，也可以利用该定律用二分法求使函数值等于 $\frac{s_2}{s_1}$ 的 x 方法来做。其实折射定律可以通过求导法得出，本质上和思路二是一样的。

Author: 哪吒

H 前往苍盐海

考点	难度
递归	2

题目分析

本题由“整数分解”改编而来，关键在于引进了“非负整数”，因此0 成了需要考虑的问题。

对于 0 而言，要注意不能将其作为最后一天的赶路公里数，否则相当于实际赶路天数减少了。

- 思路：
  - 首先，用 `for` 循环列举所有合法的到达天数，即m~l，将目标到达天数，作为自定义函数的参数之一；
  - 然后，分析题目得到，当前已经到达天数和剩余公里数，两个因素决定了不同状态；
  - 接着，由上述两点确定，自定义函数声明为：

    ```
    void func(int distance, int day,int needDay)//distance：剩余公里数，day:目前天数,needDay:需要的天数 ;
    ```

  - 下一步，确定递归方程，递归终止条件，进行函数编写；
  - 最后，在主函数中调用自定义函数，并输出总方案数

示例代码

```
#include<stdio.h>

int n,m,l,sum;
int route[25];
void func(int distance, int day,int needDay);//distance：剩余公里数，day:目前天数,needDay:需要的天数

int main()
{
    int i;
    scanf("%d%d%d", &n, &m,&l);
```

```
    for(i=m;i<=l;i++)
       func(n, 1,i);
    printf("%d\n", sum);
    return 0;
}
void func(int distance, int day,int needDay)
{
    int i;
    if (day>needDay )
    {
        if (distance) return;
        sum ++;
        for (i = 1; i <= needDay; i++)
            printf("%d ", route[i]);
        printf("\n");
        return ;
    }
    for (i = 0; i <=distance; i ++)
      if(i<distance||(i==distance&&day==needDay))
        {
            route[day] = i;
            func(distance - i, day + 1,needDay);
        }

}
```

I 反约瑟夫

题目分析

这道题其实是约瑟夫问题的逆问题，即已知出列结果，求初始序列。

模拟即可：首先设一个长度为n的空序列，因为第一个出列的是1，所以空序列的第m个位置填入1，从1开始，经过$m-1$个没有数的位置，然后填入2。重复此操作直至所有空位置被填满，即为初始序列。

示例代码

```
#include <stdio.h>

int a[1010], n, s, m;
int main() {
    scanf ("%d%d", &n, &m);
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) { // 循环经过m个空位置
            s++;// s表示当前的数组位置，当超过n时要回到数组开头（因为是环）
            if (s > n)
                s = 1;
            if (a[s] != 0) // 不是空位置，取消j++的操作
                j--;
        }
        a[s] = i; // 空位置填上对应数
    }
    for (int i = 1; i <= n; i++)
        printf ("%d ", a[i]);
    return 0;
}
```

J 二百由旬之一闪

难度	考点
2	二分，贪心

问题分析

根据题意，我们可以得到一个平凡的大前提：**一个不使锋利度降低的修剪操作，对后续是否修剪是没有影响的。**

而对于题目中的描述，我们可以得到一个粗略的猜想：尽量把使剑变钝的操作留在后面。

这个猜想比较容易验证。剑的锋利度是一定的，如果剑变钝了，会导致在后面原本能修剪的树，现在又要花费 1 锋利度才能修剪。如果我们每次砍完树后最后的锋利度为 0，也就是扣除锋利度的操作是一定的，那么让锋利度降低的操作放在后面显然是较优的；甚至我们为了让后面更多的树能不消耗锋利度而被修剪，使最后的锋利度不为 0，这样的最优方案也是存在的。那么尽量使锋利度保持在一个较高的值是我们所期望的，也即把使剑变钝的操作留在后面。

关于上述锋利度不为 0 的最优方案的叙述，可以考虑如下的情况：

```
5 2
5 1 2 2 2
```

显然，我们应当输出：

```
01111
```

在这个最优方案中，剑的锋利度始终为 2。

按照上面的描述，我们可以得到一个较平凡的结论 1：**对于一个满足题意的砍树方案，如果存在一棵树 A 没有被修剪，且位于 A 之前的一棵树 B 出现了锋利度减少的情况，那么放弃修剪 B，改为修剪 A 一定也是满足题意的一种方案**，例如样例中：

```
4 2
2 5 4 1
```

此处输出 `1110` 是满足题意的，则 `1011` 显然也满足题意。

由这个结论，我们可以进行一个初步的构想：确定一棵树的位置，从这个起点开始使剑变钝，起点之前不去强行修剪不能修剪的树，从起点开始到最后能把后面所有的树都修剪一遍。

我们当然可以枚举每棵树的位置，然后看从此处开始到后面有没有提前为 0 导致不能修剪的情况。注意到 n 的范围是 $1 \leq n \leq 10^5$，而上述算法的平均时间复杂度是 $O(n^2)$，这是我们不能接受的。不过由于砍树是有先后顺序的，如果从某个位置开始锋利度开始下降可以顺利修剪完，那么从这个位置后的某个位置开始一定也可以修剪完，所以我们可以采用二分法来找这个位置。

参考代码 #1

```
#include <stdio.h>
#include <string.h>
```

```
3
4  int a[100005];//记录树所需锋利度
5  int s[100005];//记录砍树方案
6  int n, S, i;
7
8  int check(int mid)//检查从mid处开始钝剑能否砍完后面的树
9  {
10     int f = S;
11     for (i = mid; i < n; i++)
12     {
13         if (f < a[i])
14             f--;
15     }
16     return f >= 0;//能砍完则返回1，不能则返回0
17 }
18
19 int main()
20 {
21     int T, l, m, r, pos;
22     scanf("%d", &T);
23     while (T--)
24     {
25         memset(s, 0, sizeof(s));//初始化砍树方案为全部不砍
26         scanf("%d%d", &n, &S);
27         for (i = 0; i < n; i++)
28             scanf("%d", &a[i]);
29         l = 0, r = n - 1;
30         while (l <= r)
31         {
32             m = (l + r) >> 1;
33             if (check(m))
34             {
35                 r = m - 1;
36                 pos = m;//合理的砍树方案则更新pos的值
37             }
38             else
39                 l = m + 1;
40         }
41         for (i = 0; i < pos; i++)//最后再模拟一遍得出最终砍树方案
42         {
43             if (S >= a[i])
44                 s[i] = 1;
45         }
46         for (; i < n; i++)
47             s[i] = 1;
48         for (i = 0; i < n; i++)
49             printf("%d", s[i]);
50         printf("\n");
51     }
52     return 0;
53 }
```

更进一步，我们可以想到，如果我们想让锋利度减少的操作在后面发生，那么其实可以倒着思考，从后面的位置砍树，然后让锋利度增加。我们从最后使锋利度不断增加，来模拟实际砍树过程中将锋利度降低的操作放在后面，因此有如下思路：

1. 我们从最后的树往前遍历，假设锋利度 f 最后为 0，如果当前所需锋利度大于 f，则令 f 加 1，并修剪这棵树；如果所需锋利度小于等于 f，则 f 不变，也修剪这棵树；
2. 重复上述操作直到 $f = S$，此时锋利度不可能继续增加，则剩下的树能修剪则修剪，不能修剪就放弃。

注意，此处的 f 并不是正向修剪过程中一个实际的量，它只是我们为了贪心而抽象出的一个锋利度，因为从前文叙述来看，最优方案的最终锋利度不一定为 0，我们只是用这种方法来保证所有锋利度减少的操作都在最后发生。

如果有同学难以理解这个过程，笔者在文末从数值上给出了略为繁琐的说明。如果大家有更好的说明方式，还请不吝赐教。

参考代码 #2

```c
#include <stdio.h>
#include <string.h>

int a[100005];
int s[100005];

int main()
{
    int T, n, S, i, f;
    scanf("%d", &T);
    while (T--)
    {
        f = 0;
        memset(s, 0, sizeof(s));
        scanf("%d%d", &n, &S);
        for (i = 0; i < n; i++)
            scanf("%d", &a[i]);
        for (i = n - 1; i >= 0 && f < S; i--)
        {
            if (f < a[i])//为什么？
                f++;
            s[i] = 1;
        }
        for (; i >= 0; i--)
        {
            if (f >= a[i])
                s[i] = 1;
        }
        for (i = 0; i < n; i++)
            printf("%d", s[i]);
        puts("");
    }
    return 0;
}
```

说明

参考代码 2 中有两个疑点，也是影响该策略正确性与否的两个要点：

1. 为什么假定最后的锋利度为 0？前面也给出了锋利度不为 0 但最优的情况。
2. 代码中出现了如下的语句：`if (f < a[i]) f++;`。当 $f \geq a_i$ 时，正逆思考都是相同的，因为锋利度足够修剪，不会使 f 的值发生变化。但是上述语句正向思考时，会发现有可能出现 $f = a_i$ 的

情况，这时候不会使锋利度降低，逆向的时候也就不应该使锋利度增大。

总结一下，即我们的策略不完全是可逆的，因此最优性有待商榷。笔者下面将从数值而非逻辑的角度来尝试分析其最优性。

对于疑点 1：

首先，我们的策略得出的方案虽然不一定是最优的方案，但一定是一个可以实行的方案，因为即便出现了 $f = a_i$ 的情况，最后的锋利度也是有富余的。而我们的方案是使 f 在增长到 S 前将树全部修剪，我们记 f 的初始值为 k，如果我们设定 $0 < k \leq S$，f 会比我们的方案更快地到达阈值，从而不能修剪高于阈值的树。具体地，有如下反例：

```
5 5
6 6 6 6 6
```

这样得出的方案一定是劣于我们设定 $k = 0$ 的方案的，因为 f 的值会提前增长到 5，导致前面所需锋利度为 6 的树无法被修剪。

如果我们设定 $k < 0$，有如下反例：

```
5 2
5 5 5 5 5
```

得出的方案是错误的。

显然 k 的值不可能大于 S，这里略去讨论。

因此，我们在此策略中设定 $k = 0$，既保证了所得方案的正确性，又保证了一定程度上的更优。不过，这并不能说明我们的策略就是最优的。

对于疑点 2：

根据疑点 2 的描述，我们仅需关注**正向**修剪时 $f = a_i$，而**逆向**修剪时 $f < a_i$ 的情况，除此之外其他情况都是显然可逆且合理的。而对于这样的操作，我们可以由结论 1 来引申解释：当正向修剪时，设遇到一棵树 j，此时 $f_j = a_i$，当我们修改 j 以前的砍树方案，如果为了多修剪一棵树而使锋利度降低，导致我们不能修剪 j 这棵树，显然违背了结论 1，因为后续如果有与 j 所需锋利度相同的树，我们也不能修剪了，可谓得不偿失；由大前提，若我们修改 j 以后的砍树方案，是与 j 无关的，而除了 $f = a_i$ 的情况，其他操作都是合理的，无需修改。因此，不存在一种修改方案，比我们的策略得到的方案更优，也即我们的策略是正确且最优的。

K 岩王爷没有摩拉

题目分析

首先我们否定暴力的方法，暴力的时间复杂度是$O(n^2)$，毫无疑问会TLE。我们注意到a_i的数据范围是`1-1500`，这个范围很小，我们可以用数组下标去统计$[1-1500]$中每个整数出现的次数，具体可以了解桶排序算法。现在我们只需考虑每个数在公式里做出的贡献，我们可以发现数列中每个数都会被用上n次。所以只需用双重循环遍历`1-1500`即可

参考代码

```
#include <stdio.h>
long long count[2010];
int main() {
    long long n,ans=0;
```

```
    scanf("%lld",&n);
    long long tmp;
    int i,j;
    for(i = 0; i < n; i++) {
        scanf("%lld",&tmp);
        count[tmp]++;//用下标统计每个整数出现的次数
    }
    for(i = 0; i <= 1500; i++) {
        for(j = 0; j <= 1500; j++) {
            tmp=i+j-1000;
            if(tmp<0)tmp*=-1;//相当于取绝对值
            if(i==j)ans+=tmp*count[i]*(count[j]+1);//需要特判，如果i,j相等，相当于要多统计一次，因为最后答案要除以2
            else ans+=tmp*count[i]*count[j];
        }
    }
    printf("%lld",ans/2);//实际每个数的贡献算了2n次,答案需要除以2。
    return 0;
}
```