2021 AK 杯网络赛题解

----by lr580

> 以下所有题解仅提供一种或多种易于初学者理解的正确解法。并不必然代表下面提供的解法是最优解，且并不必然代表其他的解法不可行。因此，如果有别的思路，也欢迎在 SCNUOJ 讨论区分享你的解法~ ヾ(o·ω·)/

好朋友

根据题目提示里字典序的定义，对四人名字进行排序(心算即可)，然后直接输出答案即可。

我们发现提交的代码出现的一些常见错误如下：

1. 误以为 `bc,jl,sz,gd` 四人通过输入给定
2. 写错了 `bc,jl,sz,gd` 这四个人的人名
3. 用四行分别输出了答案

C参考程序

```c
#include <stdio.h>
int main()
{
    printf("bc gd jl sz");
    return 0;
}
```

Python参考程序

```python
print('bc gd jl sz')
```

超椭圆

根据题目给定的计算公式，和 $n=5.8$ 时的已知量，可以直接计算出答案。

这题的通过率还是很高的，但还是有一些常见错误：

1. 一种是 C 语言混用了 float, double, long double 的输入输出占位符导致无法正常输出，注意它们在 `printf` 下分别是 `%f`, `%lf`, `%Lf`。
2. 我们发现不少选手提交的C++ 代码使用了 `std::cout` 进行输出。注意 `std::cout` 默认格式并不是六位小数，而是六位有效数字。并且当输出的数字特别小时， `std::cout` 还会输出指数。这些都是不符合格式/精度要求的。会导致答案错误。

> 附：本题的强化版 `超椭圆 II` 已发布在 SCNUOJ 公共题库。感兴趣者可以去挑战。

C参考程序

```c
#include <stdio.h>
double a, b; //精度要求不高，所以float也可以过题，但我们更建议使用double
int main()
{
    scanf("%lf%lf", &a, &b);
    printf("%lf", 4 * a * b * 0.925970 * 0.925970 / 0.891690);
    return 0;
}
```

Python参考程序

```python
a, b = [float(i) for i in input().strip().split()]
print('%f' % (4*a*b*0.925970**2/0.891690))
# 注意要用%f格式化，直接print可能会输出指数，导致格式出错
```

饮茶

一种思路是分类讨论，对时间段 $[00:00, 15:00], (15:00, 19:00], (19:00, 24:00)$ 分别判断。

可以将输入的时间转化为总分钟数 $=$ 小时 $\times 60 +$ 分钟，将 $15:00, 19:00$ 同样处理。这样就可以直接比较分钟数的大小来确定在哪个时间段了，并且计算两个时间点相距多少分钟可以直接相减。最后再用取模和整数除法转换回小时和分钟即可。

我们发现提交的代码绝大部分用的是另一种思路，即都是直接对小时和分钟进行分类讨论的，这种做法较为复杂，需要很多个比较复杂的条件判断才能完成，大多数代码都没能完全写对这些条件，因此这道题提交代码数量非常的高，但正确率非常的低。

对于这种直接思路，一种较优的做法是可以按两类情况分析，即分钟 mm 为 0 和分钟不为 0 。分钟不为 0 时，由于三点和七点的分钟都是 0 ，所以时间作差，在三个时间段只需要让 $hh' = 15 - hh - 1$ 或 $19 - hh - 1$ 或 $24 - hh - 1 + 15$ ，分钟则都是 , $mm' = 60 - mm$ 。对分钟 mm 不为 0 的情况，如果按照上一种情况来计算，发现 $mm' = 60$ ，所以可以在上一种情况的基础上让 hh' 加 1 进位，再让 $mm' = 0$ 即可。这样也可以避开大段的条件判断或嵌套。

常见错误：

- 对 $15:00$ 和 $19:00$ 这两个时间点的判断出错，没有输出 `0 0`
- 对 15 或 19 点的 $mm \neq 0$ 的其他时间点的处理出错，因为判断错误导致将其与上一种情况一同处理了
- 对输入 $mm = 0$ 时的处理不妥当，导致输出了 $mm' = 60$
- 多个条件判断之间存在交集，导致输出了多次答案
- 在同一个条件判断表达式同时使用与或运算，但优先级出错

C参考程序(分钟数做法)

```c
#include <stdio.h>
int hh, mm, t1, t2;
int main()
{
    scanf("%d%d", &hh, &mm);
    t1 = hh * 60 + mm;
    if (t1 <= 15 * 60) //在15:00前或15:00
    {
        t2 = 15 * 60 - t1;
    }
```

```
    else if (t1 <= 19 * 60) //否则，在19:00前或19:00
    {
        t2 = 19 * 60 - t1;
    }
    else //否则，在19:00后，最早在第二天15:00饮茶
    {
        t2 = 15 * 60 + (24 * 60 - t1);
    }
    printf("%d %d", t2 / 60, t2 % 60);
    return 0;
}
```

C参考程序(直接做法)

```
#include <stdio.h>
int hh, mm, hh2, mm2;
int main()
{
    scanf("%d%d", &hh, &mm);
    if (hh < 15 || (hh == 15 && mm == 0)) //在15:00前或15:00
    {
        hh2 = 15 - 1 - hh;
    }
    else if (hh < 19 || (hh == 19 && mm == 0)) //否则，在19:00前或19:00
    {
        hh2 = 19 - 1 - hh;
    }
    else //否则，在19:00后，最早在第二天15:00饮茶
    {
        hh2 = 15 + 24 - 1 - hh;
    }
    mm2 = 60 - mm;
    if (mm2 == 60) //处理上述的第二种分钟情况
    {
        mm2 = 0;
        hh2++;
    }
    printf("%d %d", hh2, mm2);
    return 0;
}
```

Python参考程序(分钟数做法)

```
h, m = [int(i) for i in input().strip().split()]
t = h*60+m
if t <= 15*60:
    r = 15*60-t
elif t <= 19*60:
    r = 19*60-t
else:
    r = 24*60-t+15*60
print(r//60, r % 60)

```

Python参考程序(直接做法)

```python
hh, mm = [int(i) for i in input().strip().split()]
if hh < 15 or (hh == 15 and mm == 0):
    hh2 = 15-1-hh
elif hh < 19 or (hh == 19 and mm == 0):
    hh2 = 19-1-hh
else:
    hh2 = 15+24-1-hh
mm2 = 60-mm
if mm2 == 60:
    mm2 = 0
    hh2 += 1
print(hh2, mm2)

```

密码强度

判定大小写英文、数字可以直接用 ASCII 码比较，C 语言也可以调用`ctype`库的函数。 当判定既不是大小写英文也不是数字时，可以直接用`else`判定是特殊字符。

一种解法是可以设三个整型变量 a, b, c 。

- 若 $a = 0$ 表示没有出现过大小写字母，若 $a = 1$ 表示出现过大小写字母
- 若 $b = 0$ 表示没有出现过数字，若 $b = 1$ 表示出现过数字
- 若 $c = 0$ 表示没有出现过特殊字符，若 $c = 1$ 表示出现过特殊字符

那么，密码强度 $= a + b + c$ ，即出现过的字符类型的数目。

注意 C 语言如果用`getchar`或`scanf("%c", ...)`来读取字符串，需要预先读走第一行的换行符。这是一种赛时常见的错误代码。

赛时另一种常见的错误代码是对特殊字符的判断不全，不用`else`而手动枚举所有特殊符号的 ASCII 码范围时造成了遗漏。

还有少部分代码对边界判断不准确，如没把`a,A,z,Z`判定为字母。

C参考程序(ASCII码)

```c
#include <stdio.h>
int n, a, b, c;
char s;

int main()
{
    scanf("%d%c", &n, &s); //%c的意义是读走换行符
    for (int i = 0; i < n; ++i)
    {
        scanf("%c", &s);
        if ((s >= 'a' && s <= 'z') || (s >= 'A' && s <= 'Z'))
        {
            a = 1;
        }
        else if (s >= '0' && s <= '9')
        {
            b = 1;
```

```
        }
        else
        {
            c = 1;
        }
    }
    printf("%d", a + b + c);
    return 0;
}
```

C参考程序(ctype库函数)

```
#include <stdio.h>
#include <ctype.h>
int n, a, b, c;
char s[20];

int main()
{
    scanf("%d%s", &n, s); //不用字符串用字符也行
    for (int i = 0; i < n; ++i)
    {
        if (isalpha(s[i]))
        {
            a = 1;
        }
        else if (isdigit(s[i]))
        {
            b = 1;
        }
        else
        {
            c = 1;
        }
    }
    printf("%d", a + b + c);
    return 0;
}
```

Python参考程序

```
n = int(input())
a = b = c = 0
for i in input():
    v = ord(i)
    if (v >= ord('a') and v <= ord('z')) or (v >= ord('A') and v <=
ord('Z')):
        a = 1
    elif v >= ord('0') and v <= ord('9'):
        b = 1
    else:
        c = 1
print(a+b+c)

```

卡片

赛时有很多代码都是枚举卡片的，然而这题给的 a, b 较大 ($1 \leq a, b \leq 10^9$)。以枚举一边的卡片 1 数目和卡片 3 数目为例，直接枚举的时间复杂度可能是 $\mathrm{O}(ab) = \mathrm{O}(10^9 \cdot 10^9) = 10^{18}$ ，远大于评测姬在时限内的最大计算量 (数量级约为 10^8)，会运行超时，不能过题。考虑推导一个 $\mathrm{O}(1)$ 的结论。

$b = 0$ ，即没有卡片 3 时， a 为偶数即可均分。 $b = 1$ 时， 将卡片 1 先分 3 个在另一边，剩下的再两边均匀分， 当 $a \geq 3$ 且 a 为奇数时可均分。 $b \geq 2$ 时，将两张 3 一边各放一个，此时等效于分别有 a 张卡片 1， $b - 2$ 张卡片 3 的情况。如此递推，发现按 b 奇偶性分类讨论即可。

C参考程序

```c
#include <stdio.h>
int a, b;
int main()
{
    scanf("%d%d", &a, &b);
    if (b % 2)
    {
        if (a < 3 || a % 2 == 0)
        {
            printf("NO");
        }
        else
        {
            printf("YES");
        }
    }
    else
    {
        if (a % 2)
        {
            printf("NO");
        }
        else
        {
            printf("YES");
        }
    }
    return 0;
}
```

Python参考程序

```python
a, b = [int(i) for i in input().strip().split()]
if b % 2:
    if a < 3 or a % 2 == 0:
        print('NO')
    else:
        print('YES')
else:
    if a % 2:
        print('NO')
    else:
        print('YES')
```

体温数据

一种不使用数组的解法是：对于每组测试，维护体温数据的最大值、最小值和求和值。然后按照题意依次判断有误数据和危险数据。有误即最小值小于 36.2 或最大值大于 40.0 ，危险即在有误不成立的条件下，加上最大值大于 37.2 的条件。如非有误且非危险，最后输出平均值 $=$ 求和值 $\div 14$ 。

赛时我们发现很多提交 float 的代码都解答错误了。这其实是因为精度问题，对常量而言， `36.2` 这样的写法代表 double ，而 `36.2f` 才是 float 的常量。 如果一个 float 变量与 `36.2` 这样的 double 常量比较，会将这个 float 变量隐式类型转换为 double 并损失一部分精度。而如果 float 变量若都与 float 常量比较则不会出现问题。

C参考程序

```c
#include <stdio.h>
int t;
double x, minx, maxx, sumx; //同样地，我们更建议使用double而不是float
int main()
{
    scanf("%d", &t);
    while (t--)
    {
        scanf("%lf", &x);              //先读第一个
        sumx = minx = maxx = x;        //初始化
        for (int i = 2; i <= 14; ++i) //然后读剩下的
        {
            scanf("%lf", &x);
            sumx += x;
            if (x > maxx)
            {
                maxx = x;
            }
            else if (x < minx) //事实上不可能同时更新maxx,minx
            {
                minx = x;
            }
        }
        if (minx < 36.2 || maxx > 40.0)
        {
            printf("error\n");
        }
        else if (maxx > 37.2)
        {
            printf("danger\n");
        }
        else
        {
            printf("%lf\n", sumx / 14);
        }
        //用float的话记得写成36.2f,40.0f,37.2f,14.0f
        //不然由于36.2等是double，会让float的x强转double的x损失精度进而错误
    }
    return 0;
}
```

Python参考程序

```python
for t in range(int(input())):
    x = [float(i) for i in input().strip().split()]
    if min(x) < 36.2 or max(x) > 40.0:
        print('error')
    elif max(x) > 37.2:
        print('danger')
    else:
        print('%f' % (sum(x)/14))
```

选菜

从这一题开始变得逐渐有意(难)思(度)了（•`ω•´)✧ 。一种解法是首先遍历一次，找出最美味的菜对应的最大美味值。然后再遍历一次，这一次跳过把所有美味值等于最大美味值的菜，在未标记售罄的菜里找最美味的菜，并且如果发现相同美味的菜时判断美味值差的绝对值作取舍，最后输出即可。该解法的时间复杂度是 $\mathrm{O}(n)$。

特别注意：C 语言 int 的范围是 $[-2^{31}, 2^{31}-1]$，$2^{31} \approx 2.1 \times 10^9$，而 $0 \le a_i, b_i \le 10^9$，因此 $0 \le a_i b_i \le 10^{18}$ 超过了 int 能存储的范围。注意到 long long 的范围是 $[-2^{63}, 2^{63}-1]$，$2^{63} \approx 9.2 \times 10^{18}$，所以本题应当使用 long long 。

> 由于数据量小，另一种解法是可以直接结构体排序，然后在排序后的基础上剔除掉所有售罄菜，然后输出处理后的第一个即可。该解法的时间复杂度取决于排序的时间复杂度。这里不给出该解法的代码，有兴趣的可以自行尝试。

C参考程序

```c
#include <stdio.h>
#define maxn 1010
int n, a[maxn], b[maxn], k = 1;
long long best; //最大美味值

int abs(int x) //绝对值函数，也可以不自己写而调用stdlib.h的
{
    return x >= 0 ? x : -x;
}

int main()
{
    scanf("%d", &n);
    for (int i = 1; i <= n; ++i) //下标=编号
    {
        scanf("%d", &a[i]);
    }
    for (int i = 1; i <= n; ++i)
    {
        scanf("%d", &b[i]);
    }
    for (int i = 2; i <= n; ++i)
    {
        if (1LL * a[i] * b[i] > 1LL * a[k] * b[k]) //注意1LL
        {
            k = i;
        }
```

```
    }
    best = 1LL * a[k] * b[k]; //注意1LL
    k = 0;                    //假设没找到是0
    for (int i = 1; i <= n; ++i)
    {
        if (1LL * a[i] * b[i] == best)
        {
            continue;
        }
        if (k == -1 || 1LL * a[i] * b[i] > 1LL * a[k] * b[k] || (1LL * a[i] * b[i] == 1LL * a[k] * b[k] && abs(a[i] - b[i]) < abs(a[k] - b[k])))
        {
            k = i;
        }
    }
    if (k)
    {
        printf("%d %lld", k, 1LL * a[k] * b[k]);
    }
    else
    {
        printf("sold out");
    }
    return 0;
}
```

Python参考程序

```
n = int(input())
a = [int(i) for i in input().strip().split()]  # 下标+1=编号
b = [int(i) for i in input().strip().split()]
mxi = 0
for i in range(1, n):
    if a[i]*b[i] > a[mxi]*b[mxi]:
        mxi = i
best = a[mxi]*b[mxi]
mxi = -1
for i in range(0, n):
    if a[i]*b[i] == best:
        continue
    if mxi == -1 or a[i]*b[i] > a[mxi]*b[mxi] or (a[i]*b[i] == a[mxi]*b[mxi] and abs(a[i]-b[i]) < abs(a[mxi]-b[mxi])):
        mxi = i
if mxi != -1:
    print(mxi+1, a[mxi]*b[mxi])
else:
    print('sold out')

```

升级

赛时很多代码都交了直接枚举的做法。然而本题不可以直接一级一级地模拟，会超时。理由如下：

设从 1 级升到 x 级共需要 y 经验，由等差数列知识可知：

$$y = 1 + 2 + \cdots + (x-1) = \frac{x(x-1)}{2}$$

对 $0 \le n \le 10^{16}$，对最大的 $n = 10^{16}$，计算得一共会升到第 $141421356 \approx 1.4 \times 10^8$ 级。也就是说每次询问都有可能需要循环 1.4×10^8 次，而一共最多有 10^5 次询问，所以显然会超时。事实上，可以证明时间复杂度是 $\mathrm{O}(t\sqrt{n})$，证明思路可由下文推知，这里不作具体证明。

一种解法是解方程。那么当获得 n 点经验时，设可以升 m 级。则有：

$$n = \frac{m(m-1)}{2}$$

$$\Rightarrow m^2 - m - 2n = 0$$

由一元二次方程知识可知：

$$\Delta = 1 + 8n > 0, m_1 = \frac{1 + \sqrt{1+8n}}{2}, m_2 = \frac{1 - \sqrt{1+8n}}{2}$$

由于 $1^2 < (\sqrt{1+8n})^2 = 1 + 8n$，所以 $m_2 < 0$，舍去。

记下取整符号是 $\lfloor p \rfloor$，如 $\lfloor 1.9 \rfloor = 1$，$\lfloor 2.0 \rfloor = 2$。由于等级是整数，所以最终答案应当是：

$$m = \lfloor m_1 \rfloor = \left\lfloor \frac{1 + \sqrt{1+8n}}{2} \right\rfloor$$

同样地，注意到 n 特别大，超出了 int 的范围，本题应该使用 long long。

该解法的时间复杂度是 $\mathrm{O}(t)$。

第二种解法是二分答案法。假设可以升到 m 级，求 $n' = \frac{m(m-1)}{2}$，显然 $n' = \frac{m(m-1)}{2}$ 对自变量 m，因变量 n' 是单调函数，满足二分的条件，所以可以二分。如果发现 $n' < 0$，那么说明获得 n' 经验时不足以升到 m 级，此时应该往下二分；否则，可以往上二分。

特别注意，二分的右边界不应取 n，首先是 $n = 1$ 时答案为 2，更重要的是 n 最大值为 10^{16}，所以 $\frac{n(n-1)}{2} \approx 10^{32}$，远远比 long long 还大。由上文可知，最高等级约为 1.41×10^8，所以可以取二分右边界为 $\min(n+1, 1.42 \times 10^8)$。该解法的时间复杂度是 $\mathrm{O}(t \log n)$。

C参考程序(解方程法)

```c
#include <stdio.h>
#include <math.h>
long long n, t;
int main()
{
    scanf("%lld", &t);
    while (t--)
    {
        scanf("%lld", &n);
        printf("%lld\n", (long long)((1 + sqrt(1 + 8 * n)) / 2));
    } //如果不强转long long，计算结果为double，lld格式无法正确输出
    return 0;
}
```

C参考程序(二分法)

```c
#include <stdio.h>
long long t, n, lf, rf, cf, ans, v;
signed main()
{
    scanf("%lld", &t);
    while (t--)
    {
        scanf("%lld", &n);
        lf = 1;                                    //二分左边界
        rf = (n + 1 > 1.42e8) ? 1.42e8 : n + 1; //二分右边界
        while (lf <= rf)
        {
            cf = (lf + rf) / 2; //二分答案值
            v = cf * (cf - 1) / 2;
            if (v > n) //升到cf级需要v经验，获得的经验n不够v，升不到cf级
            {
                rf = cf - 1;
            }
            else
            { //能够升到cf级
                ans = cf;
                lf = cf + 1;
            }
        }
        printf("%lld\n", ans);
    }
    return 0;
}
```

Python参考程序(解方程法)

```python
for t in range(int(input())):
    print(int((1+(1+8*int(input()))**0.5)/2))
```

Python参考程序(二分法)

```python
for t in range(int(input())):
    n = int(input())
    lf, rf = 1, n+1
    while lf <= rf:
        cf = (lf+rf)//2
        v = cf*(cf-1)/2
        if v > n:
            rf = cf-1
        else:
            lf = cf+1
    print(rf)
```

修复建筑

恭喜你，如果你赛时能做到这里的话，证明你具有相当的实力。从本题开始的题目都是较难的题了，请做好心理准备(｀・ω・´)

本题使用贪心算法求解。一种思路是首先是可以使用分解的思维，将 1 栋损坏值为 a_i 需要 b_i 天修复的建筑拆分为 b_i 栋损坏值为 $\frac{a_i}{b_i}$ ，需要 1 天修复的建筑。由于拆分前后修复的总用时以及每天带来的经济损失完全相同，所以这一种拆分前后不改变题目性质，是合理正确的拆分。

拆分后，将建筑按损坏值排序，直观上看，不难思考得出，每天应该贪心地修复损坏值最高的建筑，这样的总经济损失值最小。严谨的证明如下：

设排序后建筑的损坏值为 $d_1 \leq d_2 \leq \cdots \leq d_n$ ，当天修复第 k 栋建筑，则当天总损失值为 $\sum_{i=1}^{n} d_i$，次日总损失值为 $\sum_{i=1}^{n} d_i - d_k$ 。次日修复哪栋建筑又可以转化为减少一栋建筑后的子问题。

修复损坏值最高的建筑，次日的损失值为 $\sum_{i=1}^{n} d_i - d_n$ ，且次日的子问题为 $d_1 \leq d_2 \leq \cdots \leq d_{n-1}$ ，继续修复损失值最大的，则第三日损失为 $\sum_{i=1}^{n-1} d_i - d_{n-1}$ ，由此类推，可得，总损失值为：

$$s = nd_1 + (n-1)d_2 + \cdots 2d_{n-1} + d_n = \sum_{i=1}^{n}(n-i+1)d_i$$

设第 i 天修复建筑 h_i ，则总损失值为：

$$s(h_i) = nd_{h_n} + (n-1)d_{h_{n-1}} + \cdots + 2d_{h_2} + d_{h_1} = \sum_{i=1}^{n} id_{h_i}$$

在上述贪心策略里，即 $h_i = n - i + 1$ ，是 n 的一个排列。要证贪心正确，即证对所有 n 的排列 h_i'，均有 $s(h_i) \leq s(h_i')$。

由高中数学知识(人教数学选修 4-5)的排序不等式可知：

设 $c_1, c_2, \cdots, c_n$ 是升序数组 $b_1 \leq b_2 \leq \cdots \leq b_n$ 的任何一个排列，$a_1 \leq a_2 \leq \cdots \leq a_n$ 是任意一个长为 n 的升序数组，设乱序和：

$$S = a_1c_1 + a_2c_2 + \cdots + a_nc_n = \sum_{i=1}^{n} a_ic_i$$

且设反序和：

$$S_0 = a_nb_1 + a_2b_{n-1} + \cdots + a_nb_1 = \sum_{i=1}^{n} a_ib_{n-i+1}$$

根据排序不等式结论，有反序和 $\leq$ 乱序和，即：

$$S_0 \leq S$$

将上述结论代入上文推导过程，可知 h_i 是反序排列，$s(h_i)$ 是反序和，对任意 n 的排列 h_i'，有乱序和 $s(h_i')$，故均有 $s(h_i) \leq s(h_i')$ ，也就是说贪心策略下的经济损失小于等于任何其他策略下的经济损失。由此，严格证明了每次修复损坏值最大的建筑的贪心策略是正确的。

当然，在比赛中不大可能有时间严格证明贪心策略的正确性，但是由于该贪心较符合直观，赛时想到了也可以直接使用本策略。

本题 $\sum_{i=1}^{n} b_i \leq 10^7$， $\sum_{i=1}^{n} b_i$ 最大时，可以分解出 10^7 个建筑。由于损坏值是 $\frac{a_i}{b_i}$ ，并不是小范围整数值，难以进行桶排序等线性复杂度的排序，其他复杂度更高的排序将更无从下手，所以对这个数量级的建筑进行排序是不现实的。下面考虑对贪心策略进行优化。

重新将分解的建筑按本来拆分的方法组合起来分组，得到 n 组建筑。即有，对每个组 i，有 b_i 个建筑，组内建筑损坏值均为 $\frac{a_i}{b_i}$ ，它们的损坏值相同，所以在贪心策略修复建筑时，它们一定是在连续的 b_i 天内修复的。设这 b_i 天内，其他所有未修复建筑的损坏值总和为 d ，那么这 b_i 天的经济损失为：

$$
\begin{aligned}
s_i &= b_i d + (b_i + (b_i - 1) + (b_1 - 2) + \cdots + 2 + 1) \cdot \frac{a_i}{b_i} \\
&= b_i d + \frac{b_i(b_i + 1)}{2} \cdot \frac{a_i}{b_i} \\
&= b_i d + \frac{a_1(b_i + 1)}{2}
\end{aligned}
$$

根据贪心策略可知，只需要将原本输入的建筑按 $\frac{a_i}{b_i}$ 值大小排序，然后从大到小计算 s_i 即可得到答案。有 n 组，由于 $1 \le n \le 10^3$，所以可以使用任意排序算法。未修复建筑的损坏值总和 d 的计算，首先用 $\mathrm{O}(n)$ 预处理首次修复前的 $d = \sum_{i=1}^{n} \frac{a_i}{b_i} b_i = \sum_{i=1}^{n} a_i$，之后每次计算时，修复好了一组建筑，所以先在原本的基础上减去修复的这组建筑，即用 $d - \frac{a_i}{b_i} b_i = d - a_i$ 代替原本的 d_i，然后计算 s_i，时间复杂度是 $\mathrm{O}(1)$，需要计算 n 次，因此损失值计算的总时间复杂度是 $\mathrm{O}(n) + n\mathrm{O}(1) = \mathrm{O}(n) + \mathrm{O}(n) = \mathrm{O}(n)$。算法总时间复杂度取决于排序算法的复杂度。

注意到 $1 \le a_i, b_i, \sum_{i=1}^{n} a_i \le 10^7$，所以在计算过程中，最大可能会有 $b_i d \approx 10^{14}$，$\frac{a_1(b_i + 1)}{2} \approx 10^{14}$，所以 C/C++ 选手应使用 long long。

C参考代码

```c
#include <stdio.h>
#define mn 100010
long long n, d, a[mn], b[mn];
double ans;
signed main()
{
    scanf("%lld", &n);
    for (int i = 0; i < n; ++i)
    {
        scanf("%lld", &a[i]);
        d += a[i];
    }
    for (int i = 0; i < n; ++i)
    {
        scanf("%lld", &b[i]);
    }
    for (int i = 0; i < n; ++i)
    { //冒泡排序
        for (int j = n; j > i; --j)
        {
            if (1.0 * a[j] / b[j] > 1.0 * a[j - 1] / b[j - 1])
            {
                long long ta = a[j], tb = b[j];
                a[j] = a[j - 1], b[j] = b[j - 1];
                a[j - 1] = ta, b[j - 1] = tb;
            }
        }
    }
    for (int i = 0; i < n; ++i)
    {
        d -= a[i];
        ans += d * b[i] + a[i] * (b[i] + 1) / 2.0;
    }
    printf("%lf", ans); //虽然float能过题，但不建议使用
    return 0;
}
```

C++参考代码

```cpp
#include <bits/stdc++.h>
using namespace std;
#define mn 100010
struct building
{
    long long a, b;
    bool operator<(const building &p) const
    { //把a/b>p.b/p.a化成乘法形式，避免精度误差
        return a * p.b > p.a * b;
    }
} x[mn];
long long n, s;
double ans;
signed main()
{
    scanf("%lld", &n);
    for (int i = 0; i < n; ++i)
    {
        scanf("%lld", &x[i].a);
        s += x[i].a;
    }
    for (int i = 0; i < n; ++i)
    {
        scanf("%lld", &x[i].b);
    }
    sort(x, x + n);
    for (int i = 0; i < n; ++i)
    {
        s -= x[i].a;
        ans += s * x[i].b + x[i].a * (x[i].b + 1) / 2.0;
    }
    printf("%lf", ans); //虽然float能过题，但不建议使用
    return 0;
}
```

Python参考代码

```python
class building(object):
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __gt__(self, x):  # 定义小于号，即定义大小关系比较依据
        return self.a/self.b < x.a/x.b


n = int(input())
a = [int(i) for i in input().strip().split()]
b = [int(i) for i in input().strip().split()]
d = sum(a)
t = []
for i in range(n):
    t.append(building(a[i], b[i]))
t.sort()
res = 0
```

```
for i in t:
    d -= i.a
    res += d*i.b+i.a*(i.b+1)/2
print(res)

```

天空即为极限

这是一道大模拟题，也就是说只要将复杂的题意完全用代码实现即可过题。完成这题需要对代码有非常熟练的掌握能力，能够维护复杂的长代码，十分考验功底。下面按照题目描述顺序列出这道题可能比较难实现的功能：

1. jl 八方向移动

   如果使用 if 8 次，那么代码会非常冗长。可以预设两个常量数组 dy, dz 代表坐标偏移量，下标 d 是方向数值，那么 dy[d], dz[d] 的意义分别是方向为 d 时 y, z 坐标的偏移量(单位：格)，那么移动只需要两个坐标量分别加上 dy[d]，dz[d] 即可。

2. 幻翼的跟踪方向判定

   对幻翼，可以不使用八方向数值进行方向移动判定，否则可能需要用 if 判断 9 次才能确定方向数值，然后再复用上述的 jl 八方向移动的思路移动幻翼。

   一种更优解是：当距离在 64 格内时，设符号函数 $sgn(x)$，坐标差 $\Delta y, \Delta z$ 为 jl 的坐标值减去幻翼坐标值，那么在 y, z 轴的偏移量分别是 $sgn(\Delta y), sgn(\Delta z)$ 格。这样可以避开冗长的讨论。

3. 幻翼被攻击后一回合不能移动

   对每个幻翼设一个布尔值代表上一秒是否被攻击，如果没有被攻击的话在这一秒进行上述的跟踪方向判定，否则不判定，并将该布尔值设为假。

4. jl 的更改飞行方向

   题目已经有序给出了更改的各个时间点。可以设一个变量代表下一次变更方向的下标，当判定到达这一秒时进行更改操作并让下标自增，否则不进行处理。

整体逻辑可以用循环处理每一秒，在循环体按照题意先后处理 jl 可能的更改飞行方向， jl 的移动，然后对每一只未死亡幻翼先后判定幻翼的移动、幻翼的攻击，然后单独判断 jl 可能的扣血及是否死亡。

需要注意可能出错的地方：

- jl 对幻翼的攻击并不总是造成 7 点伤害，对某一只幻翼第三次攻击的会造成 6 点伤害。
- jl 一秒内攻击多只幻翼时，只受到一次攻击。
- 边界问题：小于等于 64 有可能误判为小于 64 秒，可能漏判第 t 秒的情况等。
- 初始化：上一个测试完毕后，上一个测试的数据没有清除，比如对幻翼的总伤害数值变量。

其他实现细节请参考代码。

C参考代码

```
#include <stdio.h>
#define mn 1010
//0不动、1左上、2左、3左下、4下、5右下、6右、7右上、8上
int dy[] = {0, -1, -1, -1, 0, 1, 1, 1, 0};
int dz[] = {0, 1, 0, -1, -1, -1, 0, 1, 1};
const int att_jl = 7, att_phantom = 3, detect = 64, hp_init = 20;
int T, t, p, d0, y_jl, z_jl, hp0, cnt, alive; //jl的方向,血,累积攻击,是否存活
int n, y[mn], z[mn], hp[mn], hitted[mn];      //幻翼坐标,上回合是否被打
int m[mn], d[mn], nxmove;                     //nxmove是下次移动数组对应下标
int min(int x, int y)
```

```
{
    return x > y ? y : x;
}
int abs(int x)
{
    return x < 0 ? -x : x;
}
int sgn(int x)
{
    return x > 0 ? 1 : (x < 0 ? -1 : 0);
}
signed main()
{
    scanf("%d", &T);
    while (T--)
    {
        scanf("%d%d%d%d%d", &t, &p, &d0, &y_jl, &z_jl);
        for (int i = 0; i < p; ++i)
        {
            scanf("%d%d", &m[i], &d[i]);
        }
        nxmove = 0; //记得初始化，防止上次测试干扰
        scanf("%d", &n);
        for (int i = 0; i < n; ++i)
        {
            scanf("%d%d", &y[i], &z[i]);
            hp[i] = hp_init;
            hitted[i] = 0; //记得初始化，防止上次测试干扰
        }
        alive = 1, hp0 = hp_init, cnt = 0; //记得初始化，防止上次测试干扰
        int i;                             //放for外则变量生存期可供lose输出
        for (i = 1; i <= t; ++i)
        {
            if (m[nxmove] == i)
            {
                d0 = d[nxmove];
                nxmove++;
            }
            y_jl += dy[d0];
            z_jl += dz[d0];
            int attack = 0; //假设本回合jl未受到攻击
            for (int j = 0; j < n; ++j)
            {
                if (hp[j] <= 0)
                {
                    continue;
                }
                if (hitted[j])
                {
                    hitted[j] = 0;
                    continue;
                }
                if (abs(y[j] - y_jl) + abs(z[j] - z_jl) <= 64)
                {
                    y[j] += sgn(y_jl - y[j]);
                    z[j] += sgn(z_jl - z[j]);
                }

```

```
                if (y[j] == y_jl && z[j] == z_jl)
                {
                    attack = 1;
                    cnt += min(att_jl, hp[j]);
                    hp[j] -= att_jl;
                    hitted[j] = 1;
                }
            }
            if (attack)
            {
                hp0 -= att_phantom;
                if (hp0 <= 0)
                {
                    alive = 0;
                    break;
                }
            }
        }
        if (alive)
        {
            printf("survive %d\n", cnt);
        }
        else
        {
            printf("lose %d\n", i);
        }
    }
    return 0;
}
```

Python参考代码

```
def sgn(x):
    return 1 if x > 0 else -1 if x < 0 else 0


dy = [0, -1, -1, -1, 0, 1, 1, 1, 0]
dz = [0, 1, 0, -1, -1, -1, 0, 1, 1]
for T in range(int(input())):
    t, p, d0, y0, z0 = [int(i) for i in input().strip().split()]
    m, d = [], []
    for i in range(p):
        mi, di = [int(i) for i in input().strip().split()]
        m.append(mi)
        d.append(di)
    alive = True
    hp0 = 20
    cnt = times = nx = 0
    n = int(input())
    hp, hitted, died, y, z = [], [], [], [], []
    for i in range(n):
        yi, zi = [int(i) for i in input().strip().split()]
        y.append(yi)
        z.append(zi)
        hitted.append(False)
        hp.append(20)
    for i in range(1, t+1):
```

```
        if nx < p and m[nx] == i:
            d0 = d[nx]
            nx += 1
        y0 += dy[d0]
        z0 += dz[d0]
        attacked = False
        for j in range(n):
            if hp[j] <= 0:
                continue
            if hitted[j]:
                hitted[j] = False
                continue
            if abs(y[j]-y0)+abs(z[j]-z0) <= 64:
                y[j] += sgn(y0-y[j])
                z[j] += sgn(z0-z[j])
            if y[j] == y0 and z[j] == z0:
                attacked = True
                cnt += min(7, hp[j])
                hp[j] -= min(7, hp[j])
                hitted[j] = True
        if attacked:
            hp0 -= 3
            if hp0 <= 0:
                alive = False
                times = i
                break
    if alive:
        print('survive', cnt)
    else:
        print('lose', times)

```