E7 - Solution

A - Sheep的指针

难度	考点
3	指针、指针传参

题目分析

C语言考点

本题目旨在让同学们理解C语言传递数组的方式

在C语言中，传递一个数组给函数时，C语言会默认地将传递参数转换成指向数组首元素的指针，即并非把整个数组复制一遍作为形参传入，而是传递指向数组首元素的指针（存放的值是数组首元素的地址，指向的数据是数组首元素）

另外要注意的是，原主函数中的数组名a代表的是一个数组，它的值是数组首元素的地址，有别于一般的指针变量，而对它做的一系列操作均与数组关联，比如$\&a$得到的值实际上是数组首元素的地址，也可以理解为数组的首地址，$sizeof(a)$返回的也是整个数组所占的空间

综上可知传递到函数当中后再求地址便与定义数组的函数求的地址不一样

所以将ca和pa直接输出便可得到数组首元素的地址

关于评测

1. 评测机的系统是开到了64位，所以对应的指针应该是8字节，而32位的机器则是4字节，对于本题目如果不理解可以直接将对指针求$sizeof$的部分替换成8，或者保守一点可以把代码写在程序里一并提交，在评测机上求$sizeof$
2. 对于更改程序的删减，要求使用最简形式，不然另开spj正确的标准也不好确定
3. 对于最终的字符串读入，C语言不支持与终端交互，故无法定义字符串常量，所以需要同学们手动求一下，但无论是选择按字符数量统计还是使用$strlen$函数，都需要注意字符串结束符

关于灵魂三问

- 对于一个指针型变量（`int *x`）取地址（`&x`），得到的是这个指针型变量本身的地址，毕竟指针也是一个变量，本身有个地址作为容身之处，而指针存的值也是一个地址，这两个地址要注意区分；对一个数组变量名（`int x[10]`）取地址（`&x`），得到的是数组名变量的地址，这个地址上存的值是数组的首地址，注意这两个地址虽然内涵不同，但在**此时**这两个地址的值其实相同。
- 对一个指针型变量（`int *x`）取大小（`sizeof(x)`），得到的是该指针的大小，但是我们都知道一个`int`变量占据4个字节，然而我们得的出来的`int*`的大小是8，这个问题在输出描述有提到：“所有输出结果均在64位环境下”。实际上`int*`的大小和我们的运行环境有关，在32位环境下，`int*`就是4字节，在64位环境下，`int*`就是8字节。可以理解为：地址的大小实际上不会根据变量所需的字节数变化，而是指的是CPU里面的通用寄存器的数据宽度为64位，占了8个字节，所以最终结果就是8。
- 再来看形参数组（`void f(int x[])`）。这其实是第二问的后半问和第三问的结合。我们在学函数调用数组之前，用的数组、指针一般都是实参变量，是有确定值的变量。而到了函数调用数组，这里的`x[]`就不再是拥有固定长度和位置的变量了，而是根据函数调用情况不断变化的指针变量，指向传递给它值

的那个实参数组。因此在对这种指针取地址时，取得的就是这个指针本身在内存中的地址；而不是这个指针存储的地址，也就是指针指向的数组的地址。这是形参和实参最根本的区别所在，此时对于`x[]`这个形参数组来说，数组名变量的地址和数组的首地址内涵不同，与第一问中实参数组所不同的是，实际值也不同。这样，就得到了题目中对不同地址的解释。而这个形参数组取大小的结果，也就和形参指针一样了。

最后的`ca[10]`是因为，这个地方的`ca`表面上定义了长度，实际上就和上面所说的，是一个指向实参数组的指针，本质上是一个形参，相当于`ca[]`，所以长度并没有影响。

示例代码

```
#include<stdio.h>
#include<string.h>
char s[105];
int main(){
    printf("#include <stdio.h>\n");
    printf("\n");
    printf("char ga[] = \"abcdefghijklm\";\n");
    printf("\n");
    printf("void my_array_func(char ca[10])\n");
    printf("{\n");
    printf("    printf(\" addr of array param = %%#x \\n\", ca);\n");//此行删除一个'&'
    printf("    printf(\" addr (ca[0]) = %%#x \\n\", &(ca[0]));\n");
    printf("    printf(\" addr (ca[1]) = %%#x \\n\", &(ca[1]));\n");
    printf("    printf(\" ++ca = %%#x \\n\\n\", ++ca);\n");
    printf("}\n");
    printf("\n");
    printf("void my_pointer_func(char *pa)\n");
    printf("{\n");
    printf("    printf(\" addr of ptr param = %%#x \\n\", pa);\n");//此行删除一个'&'
    printf("    printf(\" addr (pa[0]) = %%#x \\n\", &(pa[0]));\n");
    printf("    printf(\" addr (pa[1]) = %%#x \\n\", &(pa[1]));\n");
    printf("    printf(\" ++pa = %%#x \\n\", ++pa);\n");
    printf("}\n");
    printf("\n");
    printf("int main()\n");
    printf("{\n");
    printf("    printf(\" addr of global array = %%#x \\n\", &ga);\n");
    printf("    printf(\" addr (ga[0]) = %%#x \\n\", &(ga[0]));\n");
    printf("    printf(\" addr (ga[1]) = %%#x \\n\\n\", &(ga[1]));\n");
    printf("    my_array_func(ga);\n");
    printf("    my_pointer_func(ga);\n");
    printf("    return 0;\n");
    printf("}\n");
    printf("\n");
    scanf("%s",s);
    printf("%d\n8\n8\n",(int)strlen(s)+1);//这里直接输出两个8即可
    return 0;
}
```

B - Monica的比较

难度	考点
1	字符串

题目分析

主要考察了对字符串最基本的操作：遍历、求长度。
需要注意的是，一定不要写成 `for(i = 0; i <= strlen(str) - 1; i++)`。两个原因如下：

1. `strlen` 函数的时间复杂度是 $O(n)$ 的。也就是说，同样是遍历一遍字符串 s，`int len = strlen(s); for(int i = 0; i < len; i++);` 的复杂度是 $O(n)$ 的；而 `for(int i = 0;i < strlen(s); i++);` 的复杂度是 $O(n^2)$ 的。后者的写法可能会导致 TLE。
2. `strlen` 函数的返回值是 `unsigned` 无符号整型，对于 `i <= strlen(str) - 1` 这种表达式，只要运算符的任一边出现了无符号类型的变量，则整个式子中的变量都会被隐式地转换为无符号类型变量进行计算。例如字符串 s，长度为 $|s| = 1$，则在无符号类型运算下有 `strlen(s) - 2 = 4294967295`，也就是说循环的范围将非常大，很可能导致 TLE。
   所以，比较保险的方式是用 `'\0'` 判断是否访问到了字符串末尾，或者先用一个变量记录下 `strlen(s)` 的值。

示例代码

```
#include<stdio.h>
#include<string.h>
#include<math.h>
#define MN (1000+5)
char c[MN],s[MN];
int n;
int main(){
    scanf("%d",&n);
    for(int i=1;i<=n;++i){
        scanf("%s%s",c,s);
        int l1=strlen(c),l2=strlen(s);
        int l=l1<l2?l1:l2;
        int ans=abs(l1-l2);
        for(int j=0;j<l;++j)
            if(c[j]!=s[j])
                ans++;
        printf("%d\n",ans);
    }
    return 0;
}
```

C - 地址

难度	考点
2	地址、字符串处理

题目分析

对应地址的计算方法（注意对于`&a+y`型增加的是数组长度的地址）

输入	对应输出(十进制)
&a	0
&a[x]	$x \times 4$
&a[x]+y	$(x + y) \times 4$
a	0
a+y	$y \times 4$
&a+y	$100 \times y \times 4$ （100 为数组长度）

示例代码

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

char a[100];
int main()
{
    while (scanf("%s", a) != EOF)
    {
        int flag1 = 0, flag2 = 0;
        int x = 0, y = 0;
        int len = strlen(a);
        for (int i = 0; i < len; i++)
        {
            if (a[i] == '&')
                flag1 = 1;
            else if (a[i] == '[')
            {
                flag2 = 1;
                i++;
                while (isdigit(a[i]))
                    x = x * 10 + a[i] - '0', i++;
            }
            else if (a[i] == '+')
            {
                i++;
                while (isdigit(a[i]))
                    y = y * 10 + a[i] - '0', i++;
            }
        }
        int adr = 0;
        if (flag1 && !flag2)
            adr = 100 * y;
        else
            adr = x + y;
        printf("0x%08x\n", adr << 2);
    }
    return 0;
}
```

D - 厉害的指针数组

难度	考点
2	指针

题目分析

如hint中所说，用$lst[i]$指向字符串A_i的地址，当需要交换字符串次序时，可以直接修改指针数组的值，这比移动整个字符串要高效得多。

例如需要修改A_i, A_j的次序，可以交换$lst[i]$与$lst[j]$的值。

示例代码

```
#include<stdio.h>
#include<string.h>
int main()
{
    char a[105][105],*lst[105],*temp;
    int n,m,l1,l2;

    //初始化
    for(int i=0;i<=100;i++)
        lst[i]=a[i];//初始化，一开始，第i个字符串的地址就是a[i]

    //读入and处理
    scanf("%d%d ",&n,&m);
    for(int i=1;i<=n;i++)
        gets(a[i]);
    for(int i=1;i<=m;i++){
        scanf("%d%d",&l1,&l2);
        temp=lst[l1];//交换次序
        lst[l1]=lst[l2];
        lst[l2]=temp;
    }

    //输出
    for(int i=1;i<=n;i++){
        printf("%s\n",lst[i]);
    }
    return 0;
}
```

E - Count Sort!

难度	考点
2	排序

题目分析

即对于原始输入数据，建立一个计数数组，记录各个元素出现的次数，在计数过程记录所有元素的最小值和最大值。

完成计数后，下标从最小值到最大值遍历，其中计数排序数组中值大于0的表示该数被计数过，按计数次数输出该元素，依此类推最后可以输出排好序的数组。

示例代码

```
#include <stdio.h>
#define MAXN 1000005
#define _for(i, n) for ((i) = 0; (i) < (n); ++(i))
#define _sd(d) scanf("%d", &(d))
int a[MAXN] = {0};
int _min = 1919810, _max = -1;
void count_sort(int n)
{
    int i, el;
    _for(i, n)
    {
        _sd(el);
        if (el > _max)
            _max = el;
        if (el < _min)
            _min = el;
        a[el]++;
    }
    for (i = _min; i <= _max; ++i)
    {
        if (a[i]) printf("%d:%d\n", i, a[i]);
    }
}
int main()
{
    int n;
    _sd(n);
    count_sort(n);
    return 0;
}
```

扩展思考

与一般的排序（快排等）相比，可以对比其优劣势：

- 计数排序适用于元素数值范围小但是个数多（重复多）的情况
- 数值范围大时并不适用计数排序，一方面受限于空间，如果数组最大元素太大数组开不了那么多，而且中间很多数值不会被计数浪费空间

F - 指针判断

难度	考点
4	字符串，指针

题目分析

经观察易得，四种指针分别具有以下格式：

- 基本类型的指针：$< pre > * < name >$
- 数组指针：$< pre > (* < name >) < suf >$
- 指针的指针：$< pre > ** < name >$
- 函数指针：$< pre > (* < name >) < suf >$

其中 $< pre >$ 代表开头的类型，$< name >$ 代表指针名，$< suf >$ 代表数组各个维的大小或者函数参数表。将 $< name >$ 提到外面就可得到指针类型，再将中间的星号、括号删去，就可得到所指向的类型。因为数组指针和函数指针的格式相同，所以可以放在一起处理。

为了方便输出，代码中 suf 的含义和上面略有不同，具体细节看代码。

示例代码

```
#include <stdio.h>

char s[105], pre[10], name[105];
char* star[3] = { "*", "*", "(*)" }; //用于输出星号
char suf[3][105] = { "", "*" };
int preLen, nameLen, type; //type 0~3 分别表示基本类型的指针，指针的指针，数组指针和函数指针

int main()
{
    while (~scanf("%s", s))
    {
        int i = preLen = nameLen = type = 0;
        while (s[i] >= 'a' && s[i] <= 'z')  //读入pre
            pre[preLen ++] = s[i ++];
        pre[preLen] = '\0';
        if (s[i] == '(') //遇到左括号说明是数组/函数指针
            i += 2, type = 2;
        else if (s[i+1] == '*') //两个星号说明是指针的指针
            i += 2, type = 1;
        else
            i += 1, type = 0;
        while (s[i] >= 'a' && s[i] <= 'z') //读入name
            name[nameLen ++] = s[i ++];
        name[nameLen] = '\0';
        if (type == 2) //如果是数组/函数指针，则将后缀读入suf[2]
            sscanf(s+i+1, "%s", suf[2]);
        printf("%s%s%s %s -> %s%s\n", pre, star[type], suf[type],
            name, pre, suf[type]);
        //基本类型的指针：pre + "*" + "" + name + "->" + pre + ""
        //指针的指针：  pre + "*" + "*" + name + "->" + pre + "*"
        //数组、函数指针：pre + "(*)" + suf[2] + "->" + pre + suf[2]
    }
    return 0;
}
```

G - good->perfect

难度	考点
5	字符串

题目分析

我们要做的有两项工作：

1. 找到”好“的片段
2. 将其”替换“为perfect

第一步我们需要按照定义，找到”好”的片段。一个自然的想法是，找字母‘g’作为开头，然后从g的位置往后找字母‘d’，最后统计这两个字母之间‘o’的个数，以及这之间有没有其他字符。

第二步，我们其实并不需要真正在程序中将最终答案的字符串完整地存储下来，而只需要输出它的样子。也就是说，我们可以从前往后找，当检测到一个”好“片段，就不要输出这一片段，改为输出"perfect"。其他部分原样输出即可。当然，将答案字符串完整存下来也是可行的，只是需要额外的代码。而且要小心数组越界问题，因为转化之后的字符串可能比原来的要长。

当然还有其他做法，只要正确都是可行的，大家选取自己最顺手的方法来编程即可。

示例代码1

用最直观的想法编程。代码稍显复杂。

```
#include <stdio.h>
#include <string.h>
char s[1005];
int i, j, k, len, flag;
int main()
{
    while (gets(s) != NULL)
    {
        len = strlen(s);
        //i表示'g'的位置，j表示'd'的位置
        for (i = 0; i < len; i++)
        {
            if (s[i] == 'g')//找g
            {
                j = i + 1;
                while (j < len && s[j] != 'd')//从g的位置找d，注意还有条件j<len,防止数组越界
                    j++;
                if (s[j] == 'd')//如果真的找到了d（而不是字符串末尾）
                {
                    flag = 1;//flag表示”有无除o以外的其他字符“
                    for (k = i + 1; k <= j - 1; k++)
                    {
                        if (s[k] != 'o')
                            flag = 0;
                    }
                    if (flag == 1 && j - i - 1 >= 2)
                        //如果没有其他字符，且o的个数（i和j之间相隔的距离）比2大
                    {
                        printf("perfect");
                        i = j;
```

```
                        continue;
                        //如果找到好片段，输出perfect，再令i=j，进入下一趟循环。
                        //（别忘了continue之后，for循环也会执行i++）
                    }
                }
            }
            printf("%c", s[i]);//如果没找到好片段，原样输出
        }
        printf("\n");
    }
    return 0;
}
```

示例代码2

这段代码与上面的小不同在于，*j*表示'g'之后**第一个不是'o'的位置**，据此判断"好片段"。代码实现相较于上面的比较简单。

```
#include <stdio.h>
#include <string.h>
char s[1005];
int len, i, j, cnt;
int main()
{
    while (gets(s) != NULL)
    {
        len = strlen(s);
        for (i = 0; i < len; i++)
        {
            if (s[i] == 'g')
            {
                cnt = 0;
                for (j = i + 1; j < len; j++)//j表示'g'之后第一个不是'o'的位置
                {
                    if (s[j] == 'o')
                        cnt++;
                    else
                        break;
                }//如果s[j] == 'd'，则表示i与j之间所有字母都是'o'
                if (s[j] == 'd' && cnt >= 2)
                {
                    printf("perfect");
                    i = j;
                    continue;
                }
            }
            printf("%c", s[i]);
        }
        printf("\n");
    }
    return 0;
}
```

H - 变0为1

难度	考点
4	循环、曼哈顿距离

题目分析

假设我们当前只有一个询问 (a,b)，且时刻 0 时只有一个值为 1 的网格 (x,y)，那么问题就变成了，每次可以移动一格，最快要移动几次才能从 (x,y) 出发到达 (a,b)。答案很显然是 $|x-a|+|y-b|$，这实际上是 (x,y) 与 (a,b) 的**曼哈顿距离**。

有 T 个询问， k 个值为 1 的网格的情况也是同理。

对于每个询问分别处理，求所有值为 1 的网格到当前询问格的曼哈顿距离，取 $\min$ 即可。

示例代码

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

int n,m,k,T;
int x[1005],y[1005];

int min(int a,int b){
    return a>b?b:a;
}

int main(){
    scanf("%d%d%d%d",&n,&m,&k,&T);
    for(int i=1;i<=k;i++)
        scanf("%d%d",&x[i],&y[i]);
    while(T--){
        int a,b,ans=2e9;
        scanf("%d%d",&a,&b);
        for(int i=1;i<=k;i++)
            ans=min(ans,abs(a-x[i])+abs(b-y[i]));
        printf("%d\n",ans);
    }
}
```

I - Special Judge

难度	考点
4	读题、模拟、字符串

题目分析

本题只要结合样例，把题面读清楚了，其实是很简单的。

首先判断`prog.out`是不是比`ans.out`长，然后对于每一行挨个字符进行比较就行了。最后再判断一下`prog.out`是不是比`ans.out`短，全部过关就输出`[Accepted]`。

考虑到“文件”具有的特殊性质，我们可以利用`scanf`函数自动去掉行末空格和回车。

示例代码

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <ctype.h>
#include <stdlib.h>
#define ll long long
#define maxn 1000+10
const double eps = 1e-11;

char prog[maxn][maxn];
char ans[maxn][maxn];
int progLen = 0, ansLen = 0;

int main() {
    while (scanf("%s", prog[progLen]) != EOF) {
        if (strcmp(prog[progLen], "prog.out") == 0) {
            continue;
        }
        if (strcmp(prog[progLen], "ans.out") == 0) {
            break;
        }
        ++progLen;
    }
    while (scanf("%s", ans[ansLen]) != EOF) {
        if (strcmp(ans[ansLen], "--END--") == 0) {
            break;
        }
        ++ansLen;
    }

    if (progLen > ansLen) {
        printf("0.0000\n[Wrong Answer]: Your output is too long.\n");
        return 0;
    }
    int i = 0;
    for (i = 0; i < ansLen; ++i) {
        if (i >= progLen) break;
        int l1 = strlen(prog[i]);
        int l2 = strlen(ans[i]);
        int Min = l1 < l2 ? l1 : l2;
        for (int j = 0; j < Min; ++j) {
            if (prog[i][j] != ans[i][j]) {
                double score = (i * 1.0) / (ansLen * 1.0);
                printf("%.4f\n[Wrong Answer]: On line %d column %d, read %c, expect
%c.\n", score, i + 1, j + 1, prog[i][j], ans[i][j]);
                return 0;
            }
        }
```

```
        if (l1 != l2) {
            double score = (i * 1.0) / (ansLen * 1.0);
            if (l1 > l2) {
                printf("%.4f\n[Wrong Answer]: On line %d column %d, read %c, expect
%c.\n", score, i + 1, l2 + 1, prog[i][l2], '?');
            } else {
                printf("%.4f\n[Wrong Answer]: On line %d column %d, read %c, expect
%c.\n", score, i + 1, l1 + 1, '?', ans[i][l1]);
            }
            return 0;
        }

    }
    if (progLen < ansLen) {
        double score = (progLen * 1.0) / (ansLen * 1.0);
        printf("%.4f\n[Wrong Answer]: Your output is too short.\n", score);
        return 0;
    }
    printf("1.0000\n[Accepted]\n");
    return 0;
}
```

J - 工科元素分析

难度	考点
5	模拟

题目分析

完全按照题目中的描述模拟即可。

不过有些同学会遇到一个奇怪的精度问题，比如这样的数据：

```
64 313 4103 392
0 0 0 0 0 0 0
4 6
```

这组数据用分数手动算出伤害为767，但是实际上由于C的浮点数特性，计算出来结果是766，这是由于在计算增伤系数λ_2时，有时候是一个无限循环小数，但是在$float, double$中会成为有限小数，所以最后计算出来的结果是$766.499999999\cdots$，用$round()$进行四舍五入就成了766。

实际上在样例解释中也有例子，对于除不尽的小数，应先计算出系数的7位小数，然后再参与运算。所以在代码中，先算出增伤系数，再用增伤系数参与运算。

示例代码

```
#include <stdio.h>
#include <math.h>
const int Elemental_Reaction[10][10] = { //元素反应
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 1, 4, 3, 5, 2, 9, 0, 0},
```

```
    {0, 1, 0, 4, 6, 0, 8, 9, 0, 0},
    {0, 4, 4, 0, 4, 0, 4, 0, 0, 0},
    {0, 3, 6, 4, 0, 0, 7, 9, 0, 0},
    {0, 5, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 2, 8, 4, 7, 0, 0, 9, 0, 0},
    {0, 9, 9, 0, 9, 0, 9, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}};
struct Character
{ //角色基础信息
    int Level, Attack, Defend, Elemental_Mastery;
    double Delta_Element[10];
} character;
void Common_Element_Attack(int Element)
{ //普通攻击，无元素反应
    double Hurt = 1.00 * character.Attack * (1.00 + character.Delta_Element[Element]);
    printf("%lld", (long long)round(Hurt));
}
void Evaporation_Reaction(int first_element, int second_element)
{ //蒸发反应
    int alpha = character.Elemental_Mastery / 80, beta = character.Elemental_Mastery %
80;
    double Hurt, lambda = (2.50 - (1.50 - 0.15 * beta / 80.00) * pow(0.9, alpha));
    Hurt = 1.00 * character.Attack * (1.00 + character.Delta_Element[second_element]) *
lambda;
    if (first_element == 2)
        Hurt *= 1.50;
    else
        Hurt *= 2.00;
    printf("%lld", (long long)round(Hurt));
}
void Melting_Reaction(int first_element, int second_element)
{ //融化反应
    int alpha = character.Elemental_Mastery / 80, beta = character.Elemental_Mastery %
80;
    double Hurt, lambda = (2.50 - (1.50 - 0.15 * beta / 80.00) * pow(0.9, alpha));
    Hurt = 1.00 * character.Attack * (1.00 + character.Delta_Element[second_element]) *
lambda;
    if (first_element == 6)
        Hurt *= 2.00;
    else
        Hurt *= 1.50;
    printf("%lld", (long long)round(Hurt));
}
double Basic_Hurt(int Level)
{ //基础伤害
    if (Level <= 20)
        return 36.3;
    else if (20 < Level && Level <= 30)
        return 61.30;
    else if (30 < Level && Level <= 40)
        return 92.90;
    else if (40 < Level && Level <= 50)
        return 145.00;
    else if (50 < Level && Level <= 60)
        return 221.00;
```

```
    else if (60 < Level && Level <= 70)
        return 324.00;
    else if (70 < Level)
        return 521.00;
}
double Basic_Strength(int Level)
{ //基础护盾强度
    if (Level <= 30)
        return 303.80;
    else if (30 < Level && Level <= 40)
        return 585.00;
    else if (40 < Level && Level <= 50)
        return 786.80;
    else if (50 < Level && Level <= 60)
        return 1030.10;
    else if (60 < Level && Level <= 70)
        return 1314.80;
    else if (70 < Level && Level <= 80)
        return 1596.80;
    else if (80 < Level)
        return 1851.10;
}
double Reaction_Coefficient(int reaction_kind)
{ //反应系数
    switch (reaction_kind)
    {
        case 3:
            return 4.00;
        case 4:
            return 1.20;
        case 5:
            return 0.40;
        case 6:
            return 4.80;
        case 7:
            return 1.00;
        default:
            break;
    }
    return 0;
}
void Overload_Reaction()
{ // 超载反应
    double Reaction_Hurt, lambda = 2.40 * 25.00 * (1.00 * character.Elemental_Mastery /
(9.00 * (character.Elemental_Mastery + 1400.00)));
    Reaction_Hurt = Basic_Hurt(character.Level) * Reaction_Coefficient(3) * (1.00 +
lambda);
    printf(" %lld", (long long)round(Reaction_Hurt));
}
void Diffusion_Reaction(int Element)
{ //扩散反应
    double Reaction_Hurt, lambda = 2.40 * 25.00 * (1.00 * character.Elemental_Mastery /
(9.00 * (character.Elemental_Mastery + 1400.00)));
    Reaction_Hurt = Basic_Hurt(character.Level) * Reaction_Coefficient(4) * (1.00 +
lambda) * (1.00 + character.Delta_Element[Element]);
    printf(" %lld", (long long)round(Reaction_Hurt));
```

```
}
void Combustion_Reaction()
{ //燃烧反应
    double Reaction_Hurt = Basic_Hurt(character.Level) * Reaction_Coefficient(5);
    printf(" %lld", (long long)round(Reaction_Hurt));
}
void Inductive_Electirc_Reaction()
{ //感电反应
    double Reaction_Hurt, lambda = 2.40 * 25.00 * (1.00 * character.Elemental_Mastery /
(9.00 * (character.Elemental_Mastery + 1400.00)));
    Reaction_Hurt = Basic_Hurt(character.Level) * Reaction_Coefficient(6) * (1.00 +
lambda);
    printf(" %lld", (long long)round(Reaction_Hurt));
}
void Superconducting_Reaction()
{ //超导反应
    double Reaction_Hurt, lambda = 2.40 * 25.00 * (1.00 * character.Elemental_Mastery /
(9.00 * (character.Elemental_Mastery + 1400.00)));
    Reaction_Hurt = Basic_Hurt(character.Level) * Reaction_Coefficient(7) * (1.00 +
lambda);
    printf(" %lld", (long long)round(Reaction_Hurt));
}
void Freezing_Reaction()
{ //冻结反应
    printf("Frozen");
}
void Crystallization_Reaction()
{ //结晶反应
    double Shield_Strength, lambda;
    if (character.Elemental_Mastery == 0)
        lambda = 0.00;
    else
        lambda = 4.44 / (1.00 + 1400.00 / character.Elemental_Mastery);
    Shield_Strength = 1.00 * character.Defend + Basic_Strength(character.Level) * (1.00 +
lambda);
    printf("%lld", (long long)round(Shield_Strength));
}
void solve()
{
    int u, v;
    scanf("%d%d", &u, &v);
    switch (Elemental_Reaction[u][v])
    {
        case 1:
            Evaporation_Reaction(u, v);
            break;
        case 2:
            Melting_Reaction(u, v);
            break;
        case 3:
            Common_Element_Attack(v);
            Overload_Reaction();
            break;
        case 4:
            Common_Element_Attack(v);
            Diffusion_Reaction(u == 3 ? v : u);
```

```
            break;
        case 5:
            Common_Element_Attack(v);
            Combustion_Reaction();
            break;
        case 6:
            Common_Element_Attack(v);
            Inductive_Electirc_Reaction();
            break;
        case 7:
            Common_Element_Attack(v);
            Superconducting_Reaction();
            break;
        case 8:
            Freezing_Reaction();
            break;
        case 9:
            Crystallization_Reaction();
            break;
        default:
            Common_Element_Attack(v);
            break;
    }
    printf("\n");
}
void inital()
{
    scanf("%d%d%d%d", &character.Level, &character.Attack, &character.Defend,
&character.Elemental_Mastery);
    for (int i = 1; i <= 7; i++)
    {
        scanf("%lf", &character.Delta_Element[i]);
        character.Delta_Element[i] /= 100.00; //百分制
    }
}
int main()
{
    int q;
    scanf("%d", &q);
    while (q--)
    {
        inital();
        solve();
    }
    return 0;
}
```