第 1 章 基础

1 C++语法基础

1.1 指针

1.1.1 变量的地址——指针

在程序中，数据都有其存储的地址。在程序每次的实际运行过程中变量在物理内存中的存储位置不尽相同。但我们可以通过一定的语句取得数据在内存中的地址。

地址也是数据。存放地址所用的变量类型有一个特殊的名字，叫做“指针变量”，有时也简称做“指针”。指针的声明方式如下所示：

```
1. type *var_name
```

type 是指针的基类型，它必须是一个有效的 C++ 数据类型，var-name 是指针变量的名称。星号 * 是用来指定一个变量是指针。以下是有效的指针声明：

```
1. int *x;
2. char *y;
3. double *z;
```

指针变量的大小在不同环境下有差异。在 32 位的机器上，地址用 32 位二进制整数表示，因此一个指针的大小为 4 字节。而 64 位机器上，地址用 64 位二进制整数表示，因此一个指针的大小就变成了 8 字节。

1.1.2 使用指针

我们可以使用 & 符号取得一个变量的地址，将地址赋值给指针变量，该指针就是一个指向该变量的指针。要想访问指针变量地址所指向的数据，需要对指针变量进行**解引用**，使用 * 符号。

```
1. int a = 100;
2. int *pointer_a = &a;
3. cout << *pointer_a << endl;   // 结果为：100
```

1.1.3 空指针

在指针变量声明的时候，如果没有确切的地址可以赋值，为指针变量赋一个空值是一个良好的编程习惯。赋为空值的指针被称为**空指针**。

在 C++11 之前，C++ 和 C 一样使用 NULL 宏表示空指针常量，NULL 的定义如下：

```
1. #define NULL 0
```

空指针和整数 0 的混用在 C++ 中会导致许多问题，为了解决这些问题，C++11 引入了 nullptr 关键字作为空指针常量。C++ 规定 nullptr 可以隐式转换为任何指针类型，这种转换结果是该类型的空指针值。

1.1.4 指针与数组

指针变量也可以和整数进行加减操作。对于 int 型指针，每加 1（递增 1），其指向的地址偏移 32 位（即 4 个字节）；若加 2，则指向的地址偏移 2 × 32＝64 位。同理，对于 char 型指针，每次递增，其指向的地址偏移 8 位（即 1 个字节）。偏移的位数取决于该数据类型所占的字节数。

我们常用 [] 运算符来访问数组中某一指定偏移量处的元素。比如数组a的第 2 个元素$a[1]$。这种写法和对指针进行运算后再引用是等价的，即 $a[1]$ 和 $*(a+1)$ 是等价的两种写法。

```
1. int a[] = {1, 2, 3, 4};
2. for (int i = 0; i < 4; i++) {
3.     printf("%d ", *(a + i));   // 结果为：1 2 3 4
4. }
```

1.1.5 例子

使用指针交换两个变量 a,b 的值。

```
1. int x = 1, y = 2;
2. int *px = &x, *py = &y;
3. int temp = *px;
4. *px = *py;
5. *py = temp;
6. cout << x << " " << y << endl;   // 输出：2 1
```

1.2 引用

引用变量是一个别名，也就是说，它是某个已存在变量的另一个名字。一旦把引用初始化为某个变量，就可以使用该引用名称来指向变量。引用的基本原则是在声明时必须指向对象，以及对引用的一切操作都相当于对原对象操作。引用的特点有：

- 引用不是对象，因此不存在引用的数组、无法获取引用的指针，也不存在引用的引用。
- 不存在空引用。引用必须连接到一块合法的内存，引用必须在创建时被初始化。
- 一旦引用被初始化为一个对象，就不能被指向到另一个对象。

引用主要分为两种，左值引用和右值引用。通常算法竞赛接触到的引用为左值引用，即绑定到左值的引用。

1.2.1 创建引用

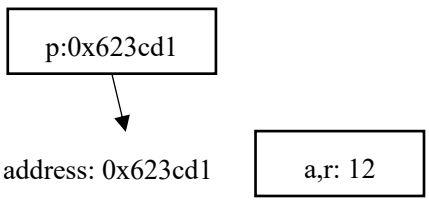
声明变量x的引用y以及对引用进行操作的示例如下：

```
1. int x = 1;
2. int &y = x;
3. y = 2;
4. printf("%d", x);   // 结果为：2
```

1.2.2 引用和指针的区别

如下代码所示的例子解释了引用和指针的区别。

```
1. int a = 12;
2. int *p = &a;   // 指针
3. int &r = a;    // 引用
```



1.2.3 引用作为函数参数

交换两个变量a, b值的函数如下所示。

```cpp
// 交换两个整数 a,b 的函数
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

int x = 1, y = 2;
swap(x, y);
printf("%d %d", x, y);   // 输出为：2 1
```

1.2.4 引用作为函数返回值

通过使用引用来替代指针，会使 C++ 程序更容易阅读和维护。

当函数返回一个引用时，则返回一个指向返回值的隐式指针。这样，函数就可以放在赋值语句的左边。

```cpp
int a[5] = {1, 2, 3, 4, 5};

int& setArray(int i) {
    int& val = a[i];
    return val;
}

setArray(2) = 6;
for (int i = 0; i < 5; i++) {
    printf("%d%c", a[i], i == 4 ? '\n' : ' ');
}
// 输出为：1 2 6 4 5
```

1.2.5 例子

扩展欧几里得函数使用引用传值简化代码：

```cpp
typedef long long ll;

ll exgcd(ll a, ll b, ll &x, ll &y) {
    if (!b) {
        x = 1, y = 0;
        return a;
    }
    ll gcd = exgcd(b, a % b, y, x);
    y -= (a / b) * x;
```

```
10.     return gcd;
11. }
```

1.3 函数

函数是一组一起执行一个任务的语句。每个 C++ 程序都至少有一个函数，即主函数 **main()**，如下所示，所有简单的程序都可以定义其他额外的函数。

```
1. // C++程序的主函数
2. int main() {
3.
4.         return 0
5. }
```

我们可以把代码划分到不同的函数中，在逻辑上，划分通常是根据每个函数执行一个特定的任务来进行的。

函数声明告诉编译器函数的名称、返回类型和参数。函数定义提供了函数的实际主体。

C++ 标准库提供了大量的程序可以调用的内置函数。例如，函数 **__builtin_popcount()** 用来计算 32 位无符号整数二进制形式下有多少个 1；函数 **memset()** 用来将某一块内存中的数据全部设置为指定的值。

函数还有很多叫法，比如方法、子过程等等。

1.3.1 函数的声明

声明一个函数，我们需要返回值类型、函数的名称以及参数列表，如下所示。

```
1.  return_type function_name(parameter list);
```

下面列出一个函数部分的详细介绍：

- **返回类型(return_type)：**一个函数可以返回一个值。return_type 是函数返回的值的数据类型。有些函数执行所需的操作而不返回值，在这种情况下，return_type 是关键字 void。
- **函数名称(function_name)：**函数的实际名称。
- **参数列表：**参数就像是占位符。当函数被调用时，按照函数的参数定义向参数传递若干个值，这被称为实际参数。参数列表包括函数参数的类型、顺序、数量。参数是可选的，也就是说，函数可能不包含参数。

函数声明会告诉编译器函数名称及如何调用函数。函数的实际主体可以单独定义。

1.3.2 函数的实现

只有函数的声明还不够，它只能让我们在调用时能够得知函数的**接口**类型（即接收什么数据、返回什么数据）,但其缺乏具体的内部实现。我们可以在**声明之后的其他任何地方**编写代码**实现**这个函数（也可以在另外的文件中实现，但是需要将分别编译后的文件在链接时一并给出）。

如果函数有返回值，则需要通过 return 语句，将值返回给调用方。

“求两个数的平均数”的函数声明及定义如下所示。

```
1. // 函数的声明
2. double average(int, int);
3. // 函数的实现
4. double average(int a, int b) {
5.         return 1.0 * (a + b) / 2;
6. }
```

如果是同一个文件中，我们也可以直接将声明和定义合并在一起，换句话说，也就是在声明时就完成定义。在程序设计比赛中，一般采取此种写法。一个函数 A 要想调用另外一个函数 B，函数 B 的声明必须在函数 A 之前。下面的写法等价于声明和实现分开写。

```
1. // 求两个数 a, b 的平均数
2. double average(int a, int b) {
3.         return 1.0 * (a + b) / 2;
4. }
```

函数内部定义的变量为**局部变量**，随着函数的结束变量被释放。函数一旦执行到 return 语句，则**直接结束**当前函数，不再执行后续的语句。同样，无返回值的函数执行到 return 语句也会结束执行。

“求两个数的最大值”的函数如下所示。

```
1. // 求两个数 x,y 的最大值
2. int max(int x, int y) {
3.         int z; //声明局部变量 z
4.         if (x > y) z = x;
5.         else z = y;
6.         return z;
7.         x = y * 2; //该语句不会被执行
8. }
```

1.3.3 函数的调用

和变量一样，函数需要先被声明，才能使用。使用函数的行为，叫做“调用”。我们可以在任何函数内部调用其他函数，包括这个函数自身。函数调用自身的行为，称为**递归**。

在大多数语言中，调用函数的写法，是函数名称加上一对括号。如果函数需要参数，则我们将其需要的参数按顺序填写在括号中，以逗号间隔。函数的调用可以看做是一个表达式，函数的返回值就是表达式的值。

```
// 判断实数 x 是否在一定精度范围内为 0
bool isZero(double x) {
        if(x < 0) x = -x;
        return x < 1e-6 ? true : false;
}

double t = 1e-7;
isZero(t); // 结果为 true
```

1.3.4 函数的参数

如果函数要使用参数，则必须声明接受参数值的变量。这些变量称为函数的形式参数。形式参数就像函数内的其他局部变量，在进入函数时被创建，退出函数时被销毁。

当调用函数时，有如下三种向函数传递参数的方式：

1. 传值调用：该方法把参数的实际值赋值给函数的形式参数。在这种情况下，修改函数内的形式参数对实际参数没有影响。
2. 指针调用：该方法把参数的地址赋值给形式参数。在函数内，该地址用于访问调用中要用到的实际参数。这意味着，修改形式参数会影响实际参数。
3. 引用调用：该方法把参数的引用赋值给形式参数。在函数内，该引用用于访问调用中要用到的实际参数。这意味着，修改形式参数会影响实际参数。

下面的例子解释了三种传参方式的区别。

```
void MyFunction1(int x, int y) {
        x += 2;
        y *= 3;
}

void MyFunction2(int *x, int *y) {
        *x += 2;
        *y *= 3;
}

void MyFunction3(int &x, int &y) {
        x += 2;
        y *= 3;
}

int a = 0, b = 1;
```

```
MyFunction1(a, b); // a,b 结果为：0,1

a = 0, b = 1;
MyFunction2(&a, &b); // a,b 结果为：2,3

a = 0, b = 1;
MyFunction3(a, b); // a,b 结果为：2,3
```

1.3.5 例子

1. 求出斐波那契数列中的第n个数：

```
int fibonacci(int n) {
    return n > 2 ? fibonacci(n - 1) + fibonacci(n - 2) : 1;
}
```

2. 对二维数组a[2][2]求和：

```
// 传递参数是二维数组
int sum1(int a[2][2]) {
    int sum = 0;
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            sum += a[i][j];
        }
    }
    return sum;
}
// 传递参数是省略行数的二维数组
int sum2(int a[][2]) {
    int sum = 0;
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            sum += a[i][j];
        }
    }
    return sum;
}
// 传递参数为数组指针：定义了一个指针，指向列大小为 2 的数组
int sum3(int (*a)[2]) {
    int sum = 0;
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            sum += a[i][j];
        }
```

```
    }
    return sum;
}
// 传递参数为数组首地址，把二维数组当做一维数组处理
int sum4(int *a, int r, int c) {
    int sum = 0;
    for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++) {
            sum += a[i * r + j];
        }
    }
    return sum;
}

int a[2][2] = {1, 2, 3, 4};
cout << sum1(a) << endl;            // 结果为：10
cout << sum2(a) << endl;            // 结果为：10
cout << sum3(a) << endl;            // 结果为：10
cout << sum4(a[0], 2, 2) << endl;   // 结果为：10
```

1.4 结构体

结构体（struct），可以看做是一系列称为成员元素的组合体，是 C++ 中一种用户自定义的可用的数据类型，它允许我们存储不同类型的数据项。在 C++ 中 struct 被扩展为类似 class 的类说明符。类和结构体联系程度很高，对于算法竞赛来说，掌握结构体一般情况下是足够的。

1.4.1 定义结构体

为了定义结构体，必须使用 struct 语句。struct 语句定义了一个包含多个成员的新的数据类型，struct 语句的格式和示例如下：

```
// 结构体声明方式
struct strcut_name {
    type1 member1;   // 成员 1
    type2 member2;   // 成员 2
    type3 member3;   // 成员 3
    ...
} object_name1;   // 结构体声明变量方式 1

struct_name object_name2;   // 结构体声明变量方式 2

```

```
// Books 结构体示例
struct Books {
    int id;
    string name;
    string author;
    double price;
};
```

1.4.2 初始化结构体

当定义结构体变量时，可以通过两种方式初始化它：使用初始化列表或构造函数。

使用初始化列表定义和初始化变量的方式是：先指定变量名，后接赋值运算符和初始化列表。也可以仅初始化结构体变量的部分成员。如果某个结构成员未被初始化，则所有跟在它后面的成员都需要保留为未初始化。使用初始化列表时，C++ 不提供跳过成员的方法。

```
Books book1;   // 未初始化结构体变量
Books book2 = {1, "How to win ICPC", "tourist", 56.8};
Books book3 = {1, "How to win ICPC"};   // 仅仅初始化 id 和 name 两个成员
Books book3 = {1, 25.9};                // 报错
```

使用构造函数来初始化结构体成员变量，这和初始化类成员变量是相同的。与类构造函数一样，结构体的构造函数必须是与结构体名称相同的公共成员函数，并且没有返回类型。因为默认情况下，所有结构体成员都是公开的，所以不需要使用关键字 public。

采用重载的方式来自定义构造函数，一个结构体可以有多个重载的构造函数，但是对于形参列表完全相同的函数能且只能存在一个。如示例代码的两种构造函数会**发生冲突**。

在如下示例代码的第一个构造函数中，关键字 this 代表该结构体的内置类型指针，也可以省略不写该指针。

需要注意的是，如果定义了构造函数却没有定义默认构造函数，那么将无法声明未初始化的结构体变量，因为重载的构造函数会取代编译器自动生成的默认构造函数。为了解决这个问题再手动加上默认的构造函数即可。

```
struct Books {
    int id;
    string name;
    string author;
    double price;

    // 手动声明默认构造函数
    Books() {}

    // 构造函数 1
    Books(int id, string name, string author, double price) {
```

```
12.         this->id = id;
13.         this->name = name;
14.         this->author = author;
15.         this->price = price;
16.     }
17.
18.     // 构造函数 2
19.     Books(int id, string name, string author, double price)
20.         : id(id), name(name), author(author), price(price) {}
21. };
22.
23. Books book;   // 若没有默认构造函数此语句报错
```

1.4.3 访问结构体成员

使用**成员访问运算符（.）**访问结构的成员。也可以使用 “指针名->成员元素名”或者使用“(*指针名).成员元素名”进行访问。仍然以上述 Books 结构体为例，示例代码如下：

```
1. Books book = {1, "How to win ICPC", "tourist", 56.8};
2. Books *p_book = &book;
3. cout << book.author << " " << p_book->author << " "
4.      << (*p_book).author;   // 结果为：tourist tourist tourist
```

1.4.4 运算符重载

C++ 允许在同一作用域中的某个函数和运算符指定多个定义，分别称为函数重载和运算符重载。重载声明是指一个与之前已经在该作用域内声明过的函数或方法具有相同名称的声明，但是它们的参数列表和定义（实现）不相同。

当程序调用一个重载函数或重载运算符时，编译器通过把您所使用的参数类型与定义中的参数类型进行比较，决定选用最合适的定义。选择最合适的重载函数或重载运算符的过程，称为重载决策。

C++ 自带的运算符，最初只定义了一些基本类型的运算规则。当我们要在用户自定义的数据类型上使用这些运算符时，就需要定义运算符在这些特定类型上的运算方式。

1.4.4.1 注意事项

重载运算符有以下限制：

1. 只能对现有的运算符进行重载，不能自行定义新的运算符。
2. 以下运算符不能被重载：::（作用域解析），.（成员访问），.*（通过成员指针的

成员访问），**?:**（三目运算符）。

3. 重载后的运算符，其运算优先级，运算操作数，结合方向不得改变。
4. 对 **&&**（逻辑与）和 **||**（逻辑或）的重载失去短路求值。

自增自减运算符分为两类，前置和后置。为了能将两类运算符区别开来，对于后置自增自减运算符，重载的时候需要添加一个类型为 int 的空置形参。因此，对于类型 T，典型的重载自增运算符的定义如下：

重载定义（以++为例）	成员函数	非成员函数
前置++	T& T::operator++();	T& operator++(T& a);
后置++	T T::operator++(int);	T operator++(T& a, int);

1.4.4.2 两种重载方式

重载运算符分为两种情况，重载为成员函数或非成员函数。

当重载为成员函数时，因为隐含一个指向当前成员的 this 指针作为参数，此时函数的参数个数与运算操作数相比少一个。

而当重载为非成员函数时，函数的参数个数与运算操作数相同。

下面给出对于结构体 node 的+运算符重载的两种示例。

```
struct node {
    int x, y;

    // 成员函数结构体重载，定义在结构体内
    node operator+(const node &T) {
        return {this->x + T.x, this->y + T.y};
    }
};

// 非成员函数运算符重载，定义在结构体外
node operator+(const node &A, const node &B) {
    return {A.x + B.x, A.y + B.y};
}
```

1.4.4.3 常见运算符重载场景

重载运算符的一个常见应用是，将重载了 < 运算符的结构体作为自定义比较函数传入优先队列等 STL 容器中。

priority_queue 模版类有三个模版参数，元素类型，容器类型，比较函数。其中后两个都

可以省略，默认容器为 vector，默认算子为 less：即小的往后排，大的往前排，这样的话就是大的先出队，小的后出队。

首先如果我们使用默认的 less 比较容器，那么在结构体中必须有重载的“<”运算符。

```
1. struct node {
2.     int l, r;
3.
4.     // 优先队列中的 l 较大者优先出队
5.     bool operator<(const node &T) const {
6.         return l < T.l;
7.     }
8. };
```

1.4.5 例题

旗鼓相当的对手 - 加强版（洛谷 P5741）

现有 $N(N \leq 1000)$ 名同学参加了期末考试，并且获得了每名同学的信息：姓名（不超过 8 个字符的字符串，没有空格）、语文、数学、英语成绩（均为不超过 150 的自然数）。如果某对学生 $< i,j >$ 的每一科成绩的分差都不大于 5，且总分分差不大于 10，那么这对学生就是“旗鼓相当的对手”。现在我们想知道这些同学中，哪些是“旗鼓相当的对手”？请输出他们的姓名。

所有人的姓名是按照字典序给出的，输出时也应该按照字典序输出所有对手组合。也就是说，这对组合的第一个名字的字典序应该小于第二个；如果两个组合中第一个名字不一样，则第一个名字字典序小的先输出；如果两个组合的第一个名字一样但第二个名字不同，则第二个名字字典序小的先输出。

【输入格式】

第一行输入一个正整数 N，表示学生个数。

第二行开始，往下 N 行，对于每一行首先先输入一个字符串表示学生姓名，再输入三个自然数表示语文、数学、英语的成绩。均用空格相隔。

【输出格式】

输出若干行，每行两个以空格隔开的字符串，表示一组旗鼓相当的对手。注意题目描述中的输出格式。

【分析】

定义一个结构体，分别存储当前学生的语文成绩、数学成绩、英语成绩、姓名和总分。然后两重循环，枚举任意两对同学。如果二位旗鼓相当，则输出两者姓名，注意比较字典序输出。

```
1. #include <iostream>
2. using namespace std;
```

```cpp
const int maxn = 10010;

struct node {
    string name;        // 姓名
    int c, m, e, sum;   // 语文，数学，英语，总成绩
} a[maxn];

int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> a[i].name >> a[i].c >> a[i].m >> a[i].e;
        a[i].sum = a[i].c + a[i].m + a[i].e;
    }
    for (int i = 1; i <= n; i++) {
        for (int j = i + 1; j <= n; j++) {
            if (abs(a[i].c - a[j].c) <= 5 && abs(a[i].m - a[j].m) <= 5 &&
                abs(a[i].e - a[j].e) <= 5 &&
                abs(a[i].sum - a[j].sum) <= 10) {   // 判断是否符合旗鼓相当的对手
                if (a[i].name > a[j].name) {        // 判断字典序
                    cout << a[j].name << " " << a[i].name << "\n";
                } else {
                    cout << a[i].name << " " << a[j].name << "\n";
                }
            }
        }
    }
    return 0;
}
```


2 算法基础

2.1 位运算

bit 是度量信息的基本单位，包含 0 和 1 两种状态，计算机的各种运算归根结底是一个个 bit 之间的运算，位运算是直接操作计算机中的 bit 进行的运算。使用位运算可以提高程序运行的时空效率。

下面讲解的位运算中 0/1 均为二进制表示下的数。

2.1.1 位运算符

1. & (与运算、and)

   两个位都是 1 时，结果才为 1，否则为 0，如：

```
  1 0 0 1 1
& 1 1 0 0 1
-----------
  1 0 0 0 1
```

常用性质：

1) 任何数和 int 类型最大值 0x7fffffff 作与运算得到的都是这个数本身。
2) a为任意整数，b为2^n时，求$a\%b$可以使用与运算$a\&(b-1)$代替提高效率。

2. | (或运算、or)

   两个位都是 0 时，结果才为 0，否则为 1，如：

```
  1 0 0 1 1
| 1 1 0 0 1
-----------
  1 1 0 1 1
```

常用性质：

1) $a|1$,奇数不变，偶数加一。
2) 任何数和 0 作或运算得到的都是这个数本身。

3. ^ (异或运算、xor)

   两个位相同则为 0，不同则为 1，如：

```
  1 0 0 1 1
^ 1 1 0 0 1
-----------
  0 1 0 1 0
```

常用性质：

1) 任何数和 0 做异或都等于它本身，任何数和它本身异或都等于 0。
2) xor运算的逆运算是它本身，也就是说两次异或同一个数最后结果不变，即$(a\ xor\ b)\ xor\ b\ =\ a$。
3) 异或运算满足交换律和结合律。
4) 偶数异或 1 相当于加一；奇数异或 1 相当于减一。因此“0 和 1”，“2 和 3”，“4 和 5”……关于异或 1 构成成对变换，这一性质经常用于图论邻接表中边的存储。

4. ~(取反运算、not)

   0 变为 1，1 变为 0，如：

```
  ~  1 0 0 1 1
  ------------
     0 1 1 0 0
```

5. <<(左移运算)

   向左进行移位操作，高位丢弃，低位补 0，如：

```
1. int a = 8;
2. a << 3;
3. // 移位前：0000 0000 0000 0000 0000 0000 0000 1000
4. // 移位后：0000 0000 0000 0000 0000 0000 0100 0000
```

常用性质：

1) $(1 << 31) - 1 = 0x7fffffff = int$ 最大值

6. >>(右移运算)

   向右进行移位操作，对无符号数，高位补 0，对于有符号数，高位补符号位，如：

```
1. int a = 8;
2. a >> 3;
3. // 移位前：0000 0000 0000 0000 0000 0000 0000 1000
4. // 移位后：0000 0000 0000 0000 0000 0000 0000 0001
5. int a = -8;
6. a >> 3;
7. // 移位前：1111 1111 1111 1111 1111 1111 1111 1000
8. // 移位后：1111 1111 1111 1111 1111 1111 1111 1111
```

常用性质

1) 使用右移运算代替除法运算可以使程序效率大大提高。一般经常用>> 1来代替除以 2 操作。

2.1.2 运算符优先级

一般来说位运算的优先级都不高，因此在写代码时，不确定的情况下位运算都尽量用括号括起来保证运算正确。

部分常用运算符优先级表如下所示，优先级越小越高。

优先级	运算符	结合律
1	后缀运算符：[] () . -> ++ --	从左到右
2	一元运算符（前缀）：++ -- ! ~ + - * & sizeof	从右到左
3	类型转换运算符：(类型名称)	从右到左
4	乘除法运算符：* / %	从左到右

5	加减法运算符：+ -	从左到右
6	移位运算符：<< >>	从左到右
7	关系运算符：<<= >>=	从左到右
8	相等运算符：== !=	从左到右
9	位运算符 AND：&	从左到右
10	位运算符 XOR：^	从左到右
11	位运算符 OR：|	从左到右

2.1.3 位运算应用

1. 二进制状态压缩

通过位运算可以实现状态压缩，即用二进制下的操作表示本该用布尔数组表示的状态空间，这种方式直接提升程序的时空效率。

操作	运算
取出整数n在二进制表示下的第k位	$(n >> k)\&1$
取出整数n在二进制表示下的第$0 \sim k-1$位（后k位）	$n\&((1 << k)-1)$
把整数n在二进制表示下的第k位取反	$n\ xor\ (1 << k)$
对整数n在二进制表示下的第k位赋值 1	$n\|(1 << k)$
对整数n在二进制表示下的第k位赋值 0	$n\&(\sim(1 << k))$

2. 位运算实现乘除法

数a向右移n位，相当于将a除以2^n；数a向左移一位，相当于将a乘以2^n。

```
int a;
a >> n;   // 相当于 a * 2^n
a << n;   // 相当于 a / 2^n
```

3. 位运算交换两数

位操作交换两数可以不需要第三个临时变量。

```
void swap(int &a, int &b) {
    a ^= b;
    b ^= a;
    a ^= b;
}

/*
操作解释：
第一步：a ^= b ---> a = (a^b);
```

```
10. 第二步：b ^= a ---> b = b^(a^b) ---> b = (b^b)^a = a
11. 第三步：a ^= b ---> a = (a^b)^a = (a^a)^b = b
12. */
```

4. 位运算判断奇偶数

计算机内数的存储都是二进制比特位。只要根据数的最后一位 0 还是 1 来决定即可，为 0 就是偶数，为 1 就是奇数。

```
1. if (a & 1) {
2.     // 奇数
3. } else {
4.     // 偶数
5. }
```

5. 位运算取负

将正数变成负数，负数变成正数。

```
1. int reversal(int a) {
2.     return ~a + 1;
3. }
```

6. 位运算求绝对值

整数的绝对值是其本身，负数的绝对值正好可以对其进行取反加一求得，即我们首先判断其符号位（整数右移31位得到0，负数右移31位得到−1,即 0xffffffff），然后根据符号进行相应的操作。

上面的操作可以进行优化，可以将$i == 0$的条件判断语句去掉。我们都知道符号位 i 只有两种情况，即$i = 0$为正，$i = -1$为负。对于任何数与 0 异或都会保持不变，与−1即 0xffffffff 进行异或就相当于对此数进行取反,因此可以将上面三目元算符转换为$((a\text{^}i) - i)$，即整数时a与 0 异或得到本身，再减去 0，负数时与 0xffffffff 异或将a进行取反，然后在加上 1，即减去$i(i = -1)$。

```
1. int abs1(int a) {
2.     int i = a >> 31;
3.     return i == 0 ? a : (~a + 1);
4. }
5.
6. int abs2(int a) {
7.     int i = a >> 31;
8.     return ((a ^ i) - i);
9. }
```

7. 位操作统计二进制中 1 的个数

统计二进制 1 的个数可以分别获取每个二进制位，计数完成后右移，重复此操作即可。还有一种方法，只要a不为 0 就令$a = a \,\&\, (a - 1)$，计数即可。

```
1. int binary_count1(int a) {
2.     int count = 0;
```

```
    while (a) {
        if (a & 1) count++;
        a >>= 1;
    }
    return count;
}

int binary_count2(int a) {
    int count = 0;
    while (a) {
        a = a & (a - 1);
        count++;
    }
    return count;
}
```

8. $lowbit$运算

$lowbit(n)$定义为非负整数n在二进制表示下“最低位的 1 及其后所有的 0”构成的数值。例如$n = 12$的二进制表示为 1100，则$lowbit(n) = (100)_2 = 4$。

设$n > 0$，n的第k位是1，第$0 \sim k-1$位都是 0。

为了实现$lowbit$运算，先把n取反，此时第k位变为 0，第$0 \sim k-1$位都是 1。再令$n = n+1$，此时因为进位，第k位变为 1，第$0 \sim k-1$位都是 0。

在上面的取反加 1 操作后，n的第$k+1$到最高位恰好与原来相反，所以$n\&(\sim n+1)$仅有第k位为 1，其余位都是 0,。而在补码表示下，$\sim n = -1-n$，因此：

$$lowbit(n) = n\&(\sim n+1) = n\&(-n)$$

2.1.4 例题

起床困难综合症（AcWing 998）

atm 终于来到了 drd 所在的地方，准备与其展开艰苦卓绝的战斗。drd 有着十分特殊的技能，他的防御战线能够使用一定的运算来改变他受到的伤害。具体说来，drd 的防御战线由 n 扇防御门组成。每扇防御门包括一个运算 op 和一个参数 t，其中运算一定是 OR, XOR, AND 中的一种，参数则一定为非负整数。如果还未通过防御门时攻击力为 x，则其通过这扇防御门后攻击力将变为 $x\ op\ t$。最终 drd 受到的伤害为对方初始攻击力 x 依次经过所有 n 扇防御门后转变得到的攻击力。由于 atm 水平有限，他的初始攻击力只能为 0 到 m 之间的一个整数（即他的初始攻击力只能在 $0,1,\ldots,m$ 中任选，但在通过防御门之后的攻击力不受 m 的限制）。

为了节省体力，atm 希望通过选择合适的初始攻击力使得他的攻击能让 drd 受到最大的伤害，请你帮他计算一下，他的一次攻击最多能使 drd 受到多少伤害。

【输入格式】

第 1 行包含 2 个整数，依次为 $n, m(2 \le n \le 10^5, 0 \le m \le 10^9)$，表示 drd 有 n 扇防御门，atm 的初始攻击力为 0 到 m 之间的整数。

接下来 n 行，依次表示每一扇防御门。每行包括一个字符串 op 和一个非负整数 $t(0 \le t \le 10^9)$，两者由一个空格隔开，且 op 在前，t 在后，op 表示该防御门所对应的操作，t 表示对应的参数。

【输出格式】

输出一个整数，表示 atm 的一次攻击最多使 drd 受到多少伤害。

【分析】

首先明确题目要求：我们要找到一个在区间$[0, m]$的初始攻击力经过一系列给定的位运算操作后，结果最大。观察范围发现m很大，无法通过枚举初始攻击力来解决问题。

经过本节位运算的学习，可以了解，在做位运算时，实际上是每一位分别进行运算，然后二进制顺序不变“拼起来”形成答案。这启发我们，能否把每一位分开考虑？显然，分开做的好处是，哪怕是一个$n = 10^9$范围的数，转化为二进制的位数最多是$\log_2 n$，这样就非常小了。

设初始攻击力为x，把x和t_i都按二进制展开，因为t_i都是确定的，而x的每一位是不定的，我们先不考虑t_i的二进制展开长度。设x展开后有p位，那么$(x) = b_{p-1}, b_{p-2}, \ldots, b_1, b_0$(这样展开的依据是左边为高位，右边为低位，事实上计算机中也是这样存储的)。最终答案的二进制位数并不能得知，但是x影响的最多就是答案的前p位。每一位和若干个给定的 0/1 比特做运算，因此独立的考虑每一位取 0/1 是正确的，每一位都按最优来取，答案就是最优的。

需要注意的是我们还有一个上界m，假如最后的答案每一位都取1，那么最终的初始攻击力是$2^p - 1$，但是m可能小于该值。那么就是在第i位取 1 时，考虑当前攻击力是否超过m，如果超过了就只能退而求其次取 0。当求出了初始攻击力，再按题目给定思路就可以直接求出答案。

解决本题的按位独立考虑的思想，在解决位运算问题中相当重要。

```
#include <math.h>
#include <iostream>
using namespace std;
const int maxn = 1e5 + 10;

struct node {
    int op;   // 1 -- OR; 2 -- XOR; 3 -- AND
    int t;
} opt[maxn];

int n, m;
string s;

```

```cpp
int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        cin >> s >> opt[i].t;
        if (s[0] == 'O')
            opt[i].op = 1;
        else if (s[0] == 'X')
            opt[i].op = 2;
        else
            opt[i].op = 3;
    }
    int maxBit = log2(m), val = 0;
    for (int i = maxBit; i >= 0; i--) {
        int b0 = 0, b1 = 1;
        for (int j = 1; j <= n; j++) {
            if (opt[j].op == 1) {
                b0 |= ((opt[j].t >> i) & 1);
                b1 |= ((opt[j].t >> i) & 1);
            } else if (opt[j].op == 2) {
                b0 ^= ((opt[j].t >> i) & 1);
                b1 ^= ((opt[j].t >> i) & 1);
            } else {
                b0 &= ((opt[j].t >> i) & 1);
                b1 &= ((opt[j].t >> i) & 1);
            }
        }
        if (b1 > b0 && val + (1 << i) <= m) val += 1 << i;
    }
    for (int j = 1; j <= n; j++) {
        if (opt[j].op == 1)
            val |= opt[j].t;
        else if (opt[j].op == 2)
            val ^= opt[j].t;
        else
            val &= opt[j].t;
    }
    cout << val << endl;
    return 0;
}
```


Cat（2019 ACM-ICPC 徐州现场赛/计蒜客 42540）

商店有10^{18}个物品，每个物品的价值恰好为 i。有T次询问，每次给出一个区间$[L,R]$内的物品。钱包有S元，但是我们每次只能买连续的物品，需要支付的价值为这几个连续的物品价值异或的结果。对于每个询问，我们最多能买多少物品。

【输入格式】

第一行输入一个正整数$T(1 \le T \le 5 \times 10^5)$代表询问的次数。

接下来T行，每一行包括三个整数$L, R, S(1 \le L, R \le 10^{18}, 0 \le S \le 2 \times 10^{18})$，代表此次询问的物品区间$[L, R]$，手里有$S$元。

【输出格式】

对于每一个询问输出一个整数，代表能购买物品的最大数目。如果一个物品也无法购买，输出-1。

【分析】

根据异或运算性质的第四条，再加上自己使用朴素程序运行自定义区间内的数，可以发现这样一个规律：每个偶数和与它相邻且大于它的那个奇数异或之后结果一定为 1。再根据异或运算的交换律和结合律；任何数和它本身异或之后为 0，任何数和 0 异或值不变。

然后这道题就差不多是处理各种情况了，当区间长度超过 4 之后一定能把$4k$的数异或成 0。然后就是分情况考虑，区间长度小于 4 直接暴力，大于 4 按左区间和右区间奇偶性分四个情况讨论。

本题的网上提交源可能暂时无法提交，主要是学习解决位运算问题的思想。因为本题是一个找规律题，代码不再放出。

2.2 时空复杂度分析

时间复杂度和空间复杂度是衡量一个算法效率的重要标准。

同一个算法在不同的计算机上运行的速度会有一定的差别，并且实际运行速度难以在理论上进行计算，实际去测量又比较麻烦，所以我们通常考虑的不是算法运行的实际用时，而是算法运行所需要进行的基本操作的数量。

在普通的计算机上，加减乘除、访问变量、给变量赋值等都可以看作基本操作。对基本操作的计数或是估测可以作为评判算法用时的指标。

在算法竞赛中，通过在线评测系统 OnlineJudge 来评测题目，大部分性能不错的 OJ，对于 C/C++来说，每秒处理运算的次数大概为10^8左右（Java 和 Python 运算效率略低，因此大多数题目对 Java 和 Python 的时间限制是 C/C++的 1.5 倍）。不同题目的时间规定和空间规定各不相同，因此需要选手对代码有一个大致的运行时间和占用内存的估算。

2.2.1 时间复杂度

2.2.1.1 定义

衡量一个算法的快慢，一定要考虑数据规模的大小。所谓数据规模，一般指输入数字的

数目、输入中给出的图的点数与边数等等。一般来说，数据规模越大，算法的用时就越长。而在算法竞赛中，我们衡量一个算法的效率时，最重要的不是看它在某个数据规模下的用时，而是看它的用时随数据规模而增长的趋势，即**时间复杂度**。

时间复杂度又分为几种，例如：

1. 最坏时间复杂度，即每个输入规模下用时最长的输入对应的时间复杂度。在算法竞赛中，由于输入可以在给定的数据范围内任意给定，我们为保证算法能够通过某个数据范围内的任何数据，一般考虑最坏时间复杂度。

2. 平均（期望）时间复杂度，即每个输入规模下所有可能输入对应用时的平均值的复杂度（随机输入下期望用时的复杂度）。

2.2.1.2 渐进符号

渐进符号是函数的阶的规范描述。简单来说，渐进符号忽略了一个函数中增长较慢的部分以及各项的系数（在时间复杂度相关分析中，系数一般被称作“常数”），而保留了可以用来表明该函数增长趋势的重要部分。

通常做题时我们只需要研究函数的渐进上界，使用O符号，因为我们关注的通常是程序用时的上界，而不关心其用时的下界。

$f(n)=O(g(n))$，当且仅当$\exists c,n_0$，使得 $\forall n>n_0, 0<f(n)<c\cdot g(n)$。

多个函数复杂度相加时取最大的那个作为时间复杂度即可。常见时间复杂度由小到大依次为：

$$O(1)<O(\log_2 n)<O(n)<O(n\log n)<O(n^2)<O(n^3)<\cdots<O(2^n)<O(n!)$$

2.2.1.3 均摊复杂度

算法往往是会对内存中的数据进行修改的，而同一个算法的多次执行，就会通过对数据的修改而互相影响。

例如快速排序中的“按大小分类”操作，单次执行的最坏时间复杂度，看似是 $O(n)$ 的，但是由于快排的分治过程，先前的“分类”操作每次都减小了数组长度，所以实际的总复杂度 $O(n\,log\,n)$，分摊在每一次“分类”操作上，是 $O(\log n)$。

多次操作的总复杂度除以操作次数，就是这种操作的均摊复杂度。

2.2.1.4 例子

1. 复杂度$O(1)$系列

当操作次数是常数级别，那么时间复杂度就是$O(1)$。

```
int N = 10000;
long long sum = 0;
for (int i = 1; i <= N; i++) {
    sum += i;
}
```

上述代码中，如果 N 的大小不被看作输入规模，那么这段代码的时间复杂度就是 $O(1)$，哪怕N过大也是代表常数过大。进行时间复杂度计算时，哪些变量被视作输入规模是很重要的，而所有和输入规模无关的量都被视作常量，计算复杂度时可当作 1 来处理。

2. 复杂度$O(\log_2 n)$系列

分析如下两个函数的时间复杂度：

```
int count(int n) {
    int cnt = 0;
    while (n) {
        if (n & 1) cnt++;
        n >>= 1;
    }
    return cnt;
}

int binary_Search(int l, int r, int v) {
    while (l <= r) {
        int mid = (l + r) >> 1;
        if (v == a[mid]) return mid;
        if (v > a[mid])
            l = mid + 1;
        else
            r = mid - 1;
    }
    return -1;
}
```


第一个 count 函数中，其功能是计算数n二进制位中 1 的个数。一个正整数总存在$2^p \le n \le 2^q, p \ge q \ge 0$,显然对于一个正整数划分为二进制之后，遍历所有二进制位的时间复杂度为$O(\log n)$。

第二个函数是对数组a中范围$[l,r]$的区域寻找是否存在一个数等于v。代码首先找到区间的中点位置，判断该位置的数是否等于v，该操作使得区间每次减半，设n为区间长度，则时间复杂度为$O(\log n)$。

3. 复杂度$O(n)$系列

```
int n;
long long sum = 0;
for (int i = 1; i <= n; i++) {
```

```
    sum += i;
}
```

上述代码时间复杂度显然为$O(n)$。有一些较为复杂的情况，在 n 的循环内部存在一些操作，但通过计数统计得出这些操作的次数之和可视为常数，那么时间复杂度仍然是$O(n)$不变。

4. 复杂度$O(n\log n)$系列

```
void func(int n) {
    bool vis[maxn];
    for (int i = 2; i <= n; i++) {
        for (int j = i; j <= n; j += i) {
            //...
        }
    }
}
```

通过观察上述 func 函数代码不难看出，总的循环次数为：$\frac{n}{2}+\frac{n}{3}+\frac{n}{4}+\cdots+1$，根据调和级数公式得：$\frac{n}{2}+\frac{n}{3}+\frac{n}{4}+\cdots+1\approx n\sum_{k=1}^{n}\frac{1}{k}\approx n\ln n$，因此时间复杂度为$O(n\log n)$。

5. 复杂度$O(2^n)$系列

```
int f(int n) {
    return n > 2 ? f(n - 1) + f(n - 2) : 1;
}
```

对于每个$f(n)$，都需要递归到$f(n-1)$和$f(n-2)$，在纸上展开后，发现构成了一棵二叉树，其中树的高度为n，因此近似的将时间复杂度估算为$O(2^n)$。

2.2.2 空间复杂度

空间复杂度是对一个算法在运行过程中临时占用存储空间大小的量度。在算法竞赛中，空间复杂度也是在做题时衡量算法是否可以正确通过的要素之一。

一个算法的空间复杂度只考虑在运行过程中为局部变量分配的存储空间的大小，它包括为参数表中形参变量分配的存储空间和为在函数体中定义的局部变量分配的存储空间两个部分。如对于递归算法来说，一般都比较简短，算法本身所占用的存储空间较少，但运行时需要一个附加堆栈，从而占用较多的临时工作单元；若写成非递归算法，一般可能比较长，算法本身占用的存储空间较多，但运行时将可能需要较少的存储单元。

计算机存储空间的计算单位有，bit()、Byte()、KB、MB、GB 等，其中 bit 是最小单位。换算关系为：1 byte = 8 bit；1 KB = 1024 Byte = 2^10 Byte；1 MB = 1024 KB；1 GB = 1024 MB。

C++中基本数据类型对应的字节数分别为：

1. char：1 Byte;

2. short int：2 Byte;
3. int：4 Byte;
4. unsigned int：4 Byte;
5. long：4 Byte;
6. unsigned long：4 Byte;
7. long long：8 Byte;
8. float：4 Byte;
9. double：8 Byte。

2.3 排序

2.3.1 选择排序

选择排序是一种简单直观的排序算法。它的工作原理是每次找出第 i 小的元素（也就是当前待排序数组中最小的元素），然后将这个元素与数组第 i 个位置上的元素交换。

由于交换两个元素位置操作的存在，选择排序是一种不稳定的排序算法。

选择排序的最优时间复杂度、平均时间复杂度和最坏时间复杂度均为 $O(n^2)$。

```
// 对 n 个数，区间范围[1,n]的数组 a 进行选择排序
void selection_sort(int* a, int n) {
    for (int i = 1; i < n; i++) {
        int k = i;
        for (int j = i + 1; j <= n; j++) {
            if (a[j] < a[k]) {
                k = j;
            }
        }
        swap(a[i], a[k]);   // 交换数组中的两个元素
    }
}
```

2.3.2 冒泡排序

冒泡排序是一种简单的排序算法。由于在算法的执行过程中，较小的元素像是气泡般慢慢“浮”到数列的顶端，故叫做冒泡排序。

它的工作原理是每次检查相邻两个元素，如果前面的元素与后面的元素满足给定的排序条件，就将相邻两个元素交换。当没有相邻的元素需要交换时，排序就完成了。经过 i 次扫描后，数列的末尾 i 项必然是最大的 i 项，因此冒泡排序最多需要扫描$n-1$遍数组就能

完成排序。

冒泡排序是一种稳定的排序算法。在序列完全有序时，冒泡排序只需遍历一遍数组，不用执行任何交换操作，时间复杂度为 $O(n)$。在最坏情况下，冒泡排序要执行$\frac{n(n-1)}{2}$次交换操作，时间复杂度为$O(n^2)$。因此，冒泡排序的平均时间复杂度为$O(n^2)$。

```
1. // 对 n 个数，区间范围[1,n]的数组 a 进行冒泡排序
2. void bubble_sort(int* a, int n) {
3.     bool ok = true;
4.     while (ok) {
5.         ok = false;
6.         for (int i = 1; i < n; i++) {
7.             if (a[i] > a[i + 1]) {
8.                 ok = true;
9.                 swap(a[i], a[i + 1]);
10.            }
11.        }
12.    }
13. }
```

2.3.3 插入排序

插入排序是一种简单直观的排序算法。它的工作原理为将待排列元素划分为“已排序”和“未排序”两部分，每次从“未排序”的元素中选择一个插入到“已排序”的元素中的正确位置。

插入排序是一种稳定的排序算法。插入排序的最优时间复杂度为$O(n)$，在数列几乎有序时效率很高。插入排序的最坏时间复杂度和平均时间复杂度都为 $O(n^2)$。

```
1. // 对 n 个数，区间范围[1,n]的数组 a 进行插入排序
2. void insertion_sort(int* a, int n) {
3.     // [1, i]视为已排序的部分；[i + 1, n]视为未排序部分
4.     for (int i = 1; i <= n; i++) {
5.         int temp = a[i];
6.         int j = i - 1;
7.         while (j > 0 && a[j] > temp) {
8.             a[j + 1] = a[j];
9.             j--;
10.        }
11.        a[j + 1] = temp;
12.    }
13. }
```

2.3.4 快速排序

快速排序，简称“快排”，是一种被广泛运用的排序算法。其工作原理是通过分治的方式来将一个数组排序。

快速排序的基本思想是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。最坏时间复杂度$O(n^2)$，平均时间复杂度$O(n\log n)$，一般来说快速排序是所有排序中最快的。

设当前区间为$[x,y]$，基准数为k，基准数下标为x，左右指针分别为i,j，主要步骤为：

1. j从右开始遍历直到找到一个大于k的数。
2. i从左开始遍历直到找到一个小于k的数。
3. 若$i<j$则交换$a[i],a[j]$，重复上述过程直到$i==j$。
4. 递归处理区间$[x,i-1]$、$[i+1,y]$。

当左、右两个部分各数据排序完成后，整个数组的排序也就完成了。

注意：因为我们默认最左边为基准数，就应该先从**右边**开始检索。

```
// 对区间范围[x,y]的数组 a 进行快速排序
void quick_sort(int* a, int x, int y) {
    if (x >= y) return;
    int i = x, j = y, k = a[x];
    while (i != j) {
        while (a[j] >= k && i < j) j--;
        while (a[i] <= k && i < j) i++;
        if (i < j) swap(a[i], a[j]);
    }
    swap(a[i], a[x]);
    quick_sort(a, x, i - 1);
    quick_sort(a, i + 1, y);
}
```

快速排序应用——快速选择问题：给定n个整数的数组a和一个正整数m，输出该序列的第m大的元素。

不难发现，在快速排序中我们每次选择的分界值，其下标正好对应排序后的位置，那么我们直接类似二分搜索那样找，时间复杂度$O(n)$。

```
int quick_search(int* a, int x, int y, int m) {
    if (x == y) return a[x];
    int i = x, j = y, k = a[x];
    while (i != j) {
        while (a[j] >= k && i < j) j--;
        while (a[i] <= k && i < j) i++;
```

```
        if (i < j) swap(a[i], a[j]);
    }
    swap(a[i], a[x]);
    if (i == m) return a[i];
    if (i < m)
        return quick_search(a, i + 1, y, m);
    else
        return quick_search(a, x, i - 1, m);
}
```

2.3.5 归并排序

归并排序是高效的基于比较的稳定排序算法。

归并排序基于分治思想将数组分段排序后合并，时间复杂度在最优、最坏与平均情况下均为 $O(n\log n)$，空间复杂度为$O(n)$。归并排序一般使用与原数组等长的辅助数组，还可以只使用$O(1)$的辅助空间。

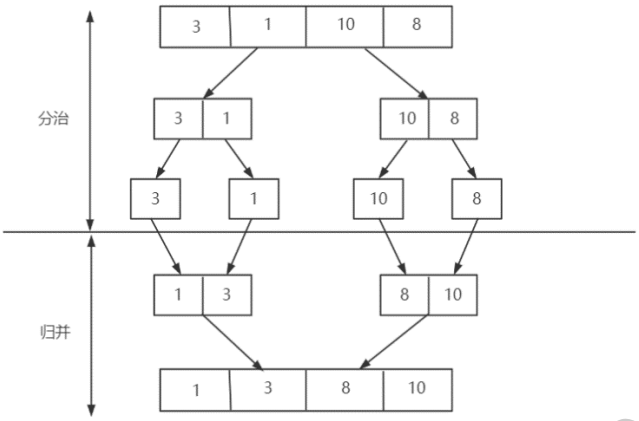
2.3.5.1 归并

如果a,b数组是已经排序后的两个数组，最终我们需要的是将它们合并到c数组。设置两个指针i,j分别指向a,b数组的第一个元素，每次比较a_i,b_j，较小的那个添加进c数组，并移动对应指针前进一步，重复该步骤直到某个指针移出对应数组的范围。最后如果还有一个指针还在对应数组范围内，那么只需要移动它之后的元素都添加到c数组。最后的c数组一定是排好序的数组。

```
void merge(int* a, int* b, int* c, int n, int m) {
    int i, j, k;
    i = j = k = 0;
    while (i < n && j < m) {
        if (a[i] < b[j])
            c[k++] = a[i++];
        else
            c[k++] = b[j++];
    }
    while (i < n) c[k++] = a[i++];
    while (j < m) c[k++] = b[j++];
}
```

2.3.5.2 归并排序

归并排序是建立在归并操作上的一种有效的排序算法,该算法是采用分治法的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列。归并排序是一种稳定的排序方法，时间复杂度$O(n\log n)$。归并排序的简单示例如下图所示。



按照归并的思想，显然需要一个辅助数组t，在归并的时候，将上面归并的几个判断条件简化，即：

1. 只要有一个序列非空就要继续合并。
2. 如果第二个为空，则复制左半部分$a[p]$。如果两个都非空，因为我们要将较小的排在前面，那么就要当$a[p] \le a[q]$时复制$a[p]$。
3. 否则复制$a[q]$。

```
1. // 对区间范围[x,y)的数组 a 进行归并排序
2. void merge_sort(int* a, int* t, int x, int y) {
3.     if (y - x > 1) {
4.         int mid = x + (y - x) / 2;
5.         int p = x, q = mid, i = x;
6.         merge_sort(a, t, x, mid);
7.         merge_sort(a, t, mid, y);
8.         while (p < mid || q < y) {
9.             if (q >= y || (p < mid && a[p] <= a[q]))
10.                 t[i++] = a[p++];
11.             else
12.                 t[i++] = a[q++];
13.         }
14.         /*上面实际是这一部分的简写
15.         while (p < mid && q < y) {
16.             if (a[p] <= a[q])
17.                 t[i++] = a[p++];
18.             else
```

```
19.                 t[i++] = a[q++];
20.         }
21.         while (p < mid) t[i++] = a[p++];
22.         while (q < y) t[i++] = a[q++];
23.         */
24.         for (int i = x; i < y; i++) a[i] = t[i];   // 从辅助数组复制到原数组
25.     }
26. }
```

2.3.5.3 逆序对

对于一个数组a若 $i > j$ 且 $a[i] < a[j]$ 那么称$a[i]$与$a[j]$是一个逆序对，其实逆序对就是求每个数前面有多少个比它大的数。

归并排序可以顺带完成逆序对数的计算：由于合并操作是从小到大进行的，当右边的$a[q]$复制到$t[i]$中时，左边还没来得及复制到t的数就是比$a[q]$大的数，因此在累加器中加上左边元素个数$m-p$即可。

尽管在归并时归并多次，但是每次归并后都能保证同一区间前面的数比它小，后面的元素就不需考虑。

2.3.6 桶排序

桶排序是一种较为简单直观的排序算法，适用于待排序数据值域较大但分布比较均匀的情况。

桶排序的步骤如下所示：

1. 设置一个定量的数组当作空桶；
2. 遍历序列，并将元素一个个放到对应的桶中；
3. 对每个不是空的桶进行排序；
4. 从不是空的桶里把元素再放回原来的序列中。

如果使用稳定的内层排序，并且将元素插入桶中时不改变元素间的相对顺序，那么桶排序就是一种稳定的排序算法。由于每块元素不多，一般使用插入排序。此时桶排序是一种稳定的排序算法。

桶排序的平均时间复杂度为$O(n + n^2/k + k)$（将值域平均分成 n 块 + 排序 + 重新合并元素）。最坏时间复杂度为 $O(n^2)$。

```
1. const int N = 100010;
2.
3. int n, w, a[N];
4. vector<int> bucket[N];
```

```
5.
6. void insertion_sort(vector<int>& A) {
7.     for (int i = 1; i < A.size(); ++i) {
8.         int key = A[i];
9.         int j = i - 1;
10.        while (j >= 0 && A[j] > key) {
11.            A[j + 1] = A[j];
12.            --j;
13.        }
14.        A[j + 1] = key;
15.    }
16. }
17.
18. void bucket_sort() {
19.    int bucket_size = w / n + 1;
20.    for (int i = 0; i < n; ++i) {
21.        bucket[i].clear();
22.    }
23.    for (int i = 1; i <= n; ++i) {
24.        bucket[a[i] / bucket_size].push_back(a[i]);
25.    }
26.    int p = 0;
27.    for (int i = 0; i < n; ++i) {
28.        insertion_sort(bucket[i]);
29.        for (int j = 0; j < bucket[i].size(); ++j) {
30.            a[++p] = bucket[i][j];
31.        }
32.    }
33. }
```


2.3.7 基数排序

基数排序是一种非比较型的排序算法，最早用于解决卡片排序的问题。

基数排序的工作原理是将待排序的元素拆分为 k 个关键字（比较两个元素时，先比较第一关键字，如果相同再比较第二关键字……），然后先对第 k 关键字进行稳定排序，再对第 $k-1$ 关键字进行稳定排序，再对第 $k-2$ 关键字进行稳定排序……最后对第一关键字进行稳定排序，这样就完成了对整个待排序序列的稳定排序。因此，基数排序是一种稳定的排序算法。

对于整数来说，基数排序的基本思想是：将所有待比较数值统一为同样的数位长度，数位较短的数前面补零。然后将整数按位数切割成不同的数字，然后从最低位开始，依次进行一次排序。这样从最低位排序一直到最高位排序完成以后，数列就变成一个有序序列。

一般来说，如果每个关键字的值域都不大，就可以使用**计数排序**（计数排序的工作原理是使用一个额外的数组，其中第i个元素是待排序数组中值等于i的元素的个数，然后遍历新数组即可轻松将所有元素排到正确的位置）作为内层排序，此时的复杂度为$O(kn+\sum_{i=1}^{k} w_i)$，其中 w_i 为第 i 关键字的值域大小。如果关键字值域很大，就可以直接使用基于比较的 $O(nk\log n)$ 排序而无需使用基数排序了。

基数排序的空间复杂度为$O(k+n)$。

基数排序的模板如下：

```
const int N = 100010;
const int W = 100010;
const int K = 100;

int n, w[K], k, cnt[W];

struct Element {
    int key[K];

    bool operator<(const Element& y) const {
        // 两个元素的比较流程
        for (int i = 1; i <= k; ++i) {
            if (key[i] == y.key[i]) continue;
            return key[i] < y.key[i];
        }
        return false;
    }
} a[N], b[N];

void counting_sort(int p) {
    memset(cnt, 0, sizeof(cnt));
    for (int i = 1; i <= n; ++i) ++cnt[a[i].key[p]];
    for (int i = 1; i <= w[p]; ++i) cnt[i] += cnt[i - 1];
    // 为保证排序的稳定性，此处循环 i 应从 n 到 1
    // 即当两元素关键字的值相同时，原先排在后面的元素在排序后仍应排在后面
    for (int i = n; i >= 1; --i) b[cnt[a[i].key[p]]--] = a[i];
    memcpy(a, b, sizeof(a));
}

void radix_sort() {
    for (int i = k; i >= 1; --i) {
        // 借助计数排序完成对关键字的排序
        counting_sort(i);
    }
}
```

对整数序列基数排序的代码如下：

```
// 对长度为 n，区间范围[0,n)的数组 a 进行基数排序
void radix_sort(int n, int* a) {
    int* b = new int[n];   // 临时空间
    int* cnt = new int[1 << 8];
    int mask = (1 << 8) - 1;
    int *x = a, *y = b;
    for (int i = 0; i < 32; i += 8) {
        for (int j = 0; j != (1 << 8); ++j) cnt[j] = 0;
        for (int j = 0; j != n; ++j) ++cnt[x[j] >> i & mask];
        for (int sum = 0, j = 0; j != (1 << 8); ++j) {
            // 等价于 std::exclusive_scan(cnt, cnt + (1 << 8), cnt, 0);
            sum += cnt[j], cnt[j] = sum - cnt[j];
        }
        for (int j = 0; j != n; ++j) y[cnt[x[j] >> i & mask]++] = x[j];
        std::swap(x, y);
    }
    delete[] cnt;
    delete[] b;
}
```

2.3.8 堆排序

堆排序是指利用二叉堆这种数据结构所设计的一种排序算法，堆排序的适用数据结构为数组。

堆排序的本质是建立在堆上的选择排序。首先建立大顶堆，然后将堆顶的元素取出，作为最大值，与数组尾部的元素交换，并维持残余堆的性质；之后将堆顶的元素取出，作为次大值，与数组倒数第二位元素交换，并维持残余堆的性质；以此类推，在第 $n-1$ 次操作后，整个数组就完成了排序。

为了方便下述代码使用了 STL 中的 priority_queue。

```
// 对长度为 n，区间范围[1,n]的数组 a 进行堆排序
void heap_sort(int* a, int n) {
    // 创建小根堆
    priority_queue<int, vector<int>, greater<int>> q;
    for (int i = 1; i <= n; i++) q.push(a[i]);
    for (int i = 1; i <= n; i++) {
        if (!q.empty()) {
            a[i] = q.top();
            q.pop();
        }
    }
```

```
12. }
```

2.3.9 希尔排序

希尔排序也称为缩小增量排序法，是插入排序的一种改进版本。希尔排序以它的发明者希尔命名。

希尔排序对不相邻的记录进行比较和移动：

1. 将待排序序列分为若干子序列（每个子序列的元素在原始数组中间距相同）；
2. 对这些子序列进行插入排序；
3. 减小每个子序列中元素之间的间距，重复上述过程直至间距减少为 1。

希尔排序是一种不稳定的排序算法。希尔排序的最优时间复杂度为$O(n)$。希尔排序的平均时间复杂度和最坏时间复杂度与间距序列的选取（就是间距如何减小到 1）有关，比如“间距每次除以 3”的希尔排序的时间复杂度是$O(n^{3/2})$。已知最好的最坏时间复杂度为$O(n\log^2 n)$。希尔排序的空间复杂度为$O(1)$。

```
1. // 对长度为 n，区间范围[0,n)的数组 a 进行希尔排序
2. void shell_sort(int* a, int n) {
3.     int h = 1;
4.     while (h < n / 3) {
5.         h = 3 * h + 1;
6.     }
7.     while (h >= 1) {
8.         for (int i = h; i < n; i++) {
9.             for (int j = i; j >= h && a[j] < a[j - h]; j -= h) {
10.                swap(a[j], a[j - h]);
11.            }
12.        }
13.        h = h / 3;
14.    }
15. }
```

2.3.10 例题

拼数（洛谷 P1012）

设有 n个正整数 $a_1, a_2, \ldots, a_n$，将它们联接成一排，使得相邻数字首尾相接后组成一个最大的整数。

【输入格式】

第一行有一个正整数n，表示数字的个数。

第二行有n个整数，表示给出的 n个整数 a_i。

【输出格式】

一个正整数，表示最大的整数。

【分析】

本题主要是证明一个想法，即把字典序最大的放在前面最优。思路和二进制类似，十进制中越高的位对答案产生的影响越大，那么考虑两个整数a,b，若字典序$a>b$，那么一定存在某一位$a_i>b_i$，此时不管他们前后接多少数，a放在b前面总是比b放在a前面更优，因此将数组转化为字符串处理字典序，从大到小排序然后把字典序大的放在前面即可。

为了方便，下面程序使用了 algorithm 头文件下的 std::sort()，可以自行提前了解。对排序掌握不熟练的同学仍然建议手写排序。

```
#include <algorithm>
#include <iostream>
using namespace std;

string s[200];

bool cmp(string s1, string s2) {
    int len = min(s1.size(), s2.size());
    for (int i = 0; i < len; i++) {
        if (s1[i] != s2[i]) return s1[i] > s2[i];
    }
    if (s1.size() < s2.size()) return s1[0] > s2[len];
    return s1[len] > s2[0];
}

int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++) cin >> s[i];
    sort(s + 1, s + 1 + n, cmp);
    string ans = "";
    for (int i = 1; i <= n; i++) ans += s[i];
    cout << ans << "\n";
    return 0;
}
```


Ultra-QuickSort（POJ 2299）

给定一个长度为n的序列a，如果只允许进行比较和交换相邻两个数的操作，求至少需要多少次交换才能把a从小到大排序。

【输入格式】

输入包含若干个测试案例。每个测试用例都以一行开始，包含一个整数$n(n<500{,}000)$，即输入序列的长度。接下来的n行中的每一行都包含一个整数$0\le a[i]\le 999{,}999{,}999$，即第

i个输入序列元素。输入被一个长度为$n = 0$的序列所终止，这个序列不能被处理。

【输出格式】

对于每个输入序列，输出一行，其中包含一个整数op，即对给定输入序列进行排序所需的最小交换操作数。

【分析】

只通过比较和交换相邻两个数值的排序方法，实际上就是冒泡排序。在排序过程中每找到一对大小颠倒的相邻数值，把它们交换，就会使整个序列的逆序对个数减少 1。最终排好序后逆序对个数显然为 0，所以对a进行冒泡排序需要的最少交换次数就是序列a中逆序对的个数。我们直接使用归并排序求出a的逆序对数就是本题的答案。

```cpp
#include <iostream>
using namespace std;
const int maxn = 5e5 + 10;
int a[maxn], t[maxn];
long long cnt = 0;

void merge_sort(int x, int y) {
    if (y - x > 1) {
        int mid = x + (y - x) / 2;
        int p = x, q = mid, i = x;
        merge_sort(x, mid);
        merge_sort(mid, y);
        while (p < mid || q < y) {
            if (q >= y || (p < mid && a[p] <= a[q]))
                t[i++] = a[p++];
            else {
                t[i++] = a[q++];
                cnt += mid - p;
            }
        }
        for (int i = x; i < y; i++) a[i] = t[i];
    }
}

int main() {
    int n;
    while (~scanf("%d", &n)) {
        if (n == 0) break;
        for (int i = 0; i < n; i++) scanf("%d", &a[i]);
        cnt = 0;
        merge_sort(0, n);
        printf("%lld\n", cnt);
    }
```

```
34.
35.     return 0;
36. }
```

2.4 模拟

模拟就是用计算机来模拟题目中要求的操作。模拟题目通常具有码量大、操作多、思路繁复的特点，但是模拟题一般侧重考察代码能力而不是算法能力，在难度上并不会太高。由于它码量大，经常会出现难以查错的情况，如果在考试中写错是相当浪费时间的。

写模拟题时，遵循以下的建议有可能会提升做题速度：

- 在动手写代码之前，在草纸上尽可能地写好要实现的流程。
- 在代码中，尽量把每个部分模块化，写成函数、结构体或类。
- 对于一些可能重复用到的概念，可以统一转化，方便处理：如，某题给你 "YY-MM-DD 时：分" 把它抽取到一个函数，处理成秒，会减少概念混淆。
- 调试时分块调试。模块化的好处就是可以方便的单独调某一部分。
- 写代码的时候一定要思路清晰，不要想到什么写什么，要按照落在纸上的步骤写。

实际上，上述步骤在解决其它类型的题目时也是很有帮助的，因此模拟题考察的能力是算法竞赛的基础能力之一。

2.4.1 高精度

在 C++中，int 整数的最大值为$2^{31}-1\approx 2\times 10^9$，long long 整数的最大值为$2^{63}-1\approx 9\times 10^{18}$。还有一种整数数据类型叫__int128（等价于__int128_t），它是 GCC 提供的扩展，可以当作 128 位整数使用，其表示的最大值为$2^{127}-1\approx 10^{38}$。编译器的 gcc 是不支持__int128 这种数据类型的，比如在 codeblocks 16.01/Dev C++是无法编译的，但是提交到大部分 OJ 上是可以编译且能用的。C/C++的 IO 是不认识__int128 这种数据类型的，因此要自己实现 IO。除了 IO 之外，基本数据类型支持的运算__int128 均支持。

```
1. // 输入__int128 整数
2. __int128 read() {
3.     __int128 x = 0, f = 1;
4.     char ch = getchar();
5.     while (ch < '0' || ch > '9') {
6.         if (ch == '-') f = -1;
7.         ch = getchar();
8.     }
9.     while (ch >= '0' && ch <= '9') {
10.        x = x * 10 + ch - '0';
11.        ch = getchar();
```

```
    }
    return x * f;
}

// 输出__int128 整数
void write(__int128 x) {
    if (x < 0) putchar('-'), x = -x;
    if (x > 9) write(x / 10);
    putchar(x % 10 + '0');
}
```

如果需要更多位数的整数，那么就需要使用高精度（又被称为大数）。高精度的实现有很多种，常见的是使用字符数组或者 C++ string 来表示大数，还有一些实现思路是用数组来表示大数。下面的例子将用整数数组来实现大数，本例中的基本运算的对象均为非负整数，如果支持负数运算需要加入一些特判，如有需要可以手动实现一个 BigInt 结构体或类的模板，方便后续使用。可以使用“压位”来减少空间复杂度，具体实现可以课后研究。

下述重载的运算符函数必须写在结构体内否则会报错。

2.4.1.1 结构体和构造函数

注意需要从后向前存，运算时从前向后运算，输出时从后向前输出。

```
const int maxn = 1005;

struct BigInt {
    int a[maxn];
    int len;

    BigInt() {
        len = 1;
        memset(a, 0, sizeof a);
    }

    BigInt(char *s) {
        len = strlen(s);
        int cnt = 0;
        for (int i = len - 1; i >= 0; i--) a[cnt++] = s[i] - '0';
    }

    BigInt(string s) {
        len = s.size();
        int cnt = 0;
        for (int i = len - 1; i >= 0; i--) a[cnt++] = s[i] - '0';
```

```
    }

    BigInt &operator=(const BigInt &T) {   // 重载赋值运算符，大数之间进行赋值运算
        len = T.len;
        memset(a, 0, sizeof(a));
        for (int i = 0; i < len; i++) a[i] = T.a[i];
        return *this;
    }

    void println() {
        for (int i = len - 1; i >= 0; i--) printf("%d", a[i]);
        puts("");
    }

    void print() {
        for (int i = len - 1; i >= 0; i--) printf("%d", a[i]);
    }
};
```

2.4.1.2 逻辑比较

```
bool operator>(const BigInt &T) const {
    if (len > T.len) return 1;
    if (len < T.len) return 0;
    int cur = len - 1;
    while (a[cur] == T.a[cur] && cur >= 0) cur--;
    if (cur >= 0 && a[cur] > T.a[cur])
        return 1;
    else
        return 0;
}

bool operator<(const BigInt &T) const {
    if (len < T.len) return 1;
    if (len > T.len) return 0;
    int cur = len - 1;
    while (a[cur] == T.a[cur] && cur >= 0) cur--;
    if (cur >= 0 && a[cur] < T.a[cur])
        return 1;
    else
        return 0;
}

```

```
bool operator==(const BigInt &T) const {
    if (len != T.len) return 0;
    for (int i = len - 1; i >= 0; i--)
        if (a[i] != T.a[i]) return 0;
    return 1;
}
```

2.4.1.3 大数加法

```
BigInt operator+(const BigInt &T) const {
    BigInt ret = T;
    int maxlen = len > T.len ? len : T.len;   // 取长度最长的那个
    for (int i = 0; i < maxlen; i++) {
        ret.a[i] += a[i];
        if (ret.a[i] >= 10) {
            ret.a[i + 1]++;
            ret.a[i] -= 10;
        }
    }
    ret.len = ret.a[maxlen] > 0 ? maxlen + 1 : maxlen;
    return ret;
}
```

2.4.1.4 大数减法

```
BigInt operator-(const BigInt &T) const {
    BigInt t1, t2;
    bool flag;
    if (*this > T) {   // 取得最大数并标记首位是否为负
        t1 = *this, t2 = T;
        flag = 0;
    } else {
        t1 = T, t2 = *this;
        flag = 1;
    }
    int maxlen = t1.len;
    for (int i = 0; i < maxlen; i++) {
        if (t1.a[i] < t2.a[i]) {   // 需要向前借位
            int j = i + 1;
            while (t1.a[j] == 0) j++;
            t1.a[j--]--;
            while (j > i) t1.a[j--] = 9;
```

```
            t1.a[i] += 10 - t2.a[i];
        } else
            t1.a[i] -= t2.a[i];
    }
    while (t1.a[maxlen - 1] == 0 && t1.len > 1) {   // 看看从最高位开始是否被借位变为 0
        t1.len--;
        maxlen--;
    }
    if (flag) t1.a[maxlen - 1] = -t1.a[maxlen - 1];   // 处理首位的正负
    return t1;
}
```

2.4.1.5 大数乘法

```
BigInt operator*(const BigInt &T) const {
    BigInt ans;
    for (int i = 0; i < len; i++) {
        for (int j = 0; j < T.len; j++) {
            ans.a[i + j] += a[i] * T.a[j] % 10;
            ans.a[i + j + 1] += a[i] * T.a[j] / 10;
        }
    }
    for (int i = 0; i < len + T.len; i++) {
        ans.a[i + 1] += ans.a[i] / 10;
        ans.a[i] %= 10;
    }
    ans.len = len + T.len;
    while (ans.a[ans.len - 1] == 0 && ans.len > 1) ans.len--;
    return ans;
}
```

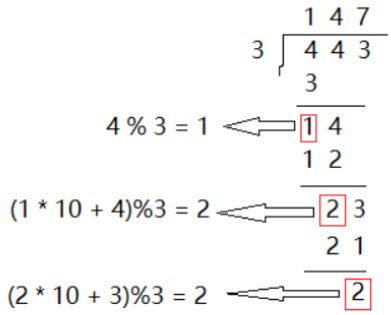
2.4.1.6 大数除法

```
BigInt operator/(const int &b) const {
    BigInt ret;
    int down = 0;
    for (int i = len - 1; i >= 0; i--) {
        ret.a[i] = (a[i] + down * 10) / b;
        down = a[i] + down * 10 - ret.a[i] * b;
    }
    ret.len = len;
    while (ret.a[ret.len - 1] == 0 && ret.len > 1) ret.len--;
```

```
10.     return ret;
11. }
```

2.4.1.7 大数取模

以 443 % 3 为例，我们可以发现实际上需要的是对每一位依次取模，并加上来自高位的余数乘以十。




```
1. int operator%(const int &b) const {
2.     int ret = 0;
3.     for (int i = len - 1; i >= 0; i--) ret = (ret * 10 + a[i]) % b;
4.     return ret;
5. }
```

2.4.2 例题

帮贡排序（洛谷 P1786）

目前帮派内共最多有一位帮主，两位副帮主，两位护法，四位长老，七位堂主，二十五名精英，帮众若干。

现在 absi2011 要对帮派内几乎所有人的职位全部调整一番。他发现这是个很难的事情。于是要求你帮他调整。他给你每个人的以下数据：他的名字（长度不会超过 30），他的原来职位，他的帮贡，他的等级。

他要给帮贡最多的护法的职位，其次长老，以此类推。可是，乐斗的显示并不按帮贡排序而按职位和等级排序。他要你求出最后乐斗显示的列表(在他调整过职位后)：职位第一关键字，等级第二关键字。

注意：absi2011 无权调整帮主、副帮主的职位，包括他自己的。他按原来的顺序给你（所以，等级相同的，原来靠前的现在也要靠前，因为经验高低的原因，但此处为了简单点省去经验。）

【输入格式】

第一行一个正整数 $n(3 \leq n \leq 110)$，表示星月家园内帮友的人数。下面 n 行每行两个字符串两个整数，表示每个人的名字、职位、帮贡（小于等于10^9的非负整数）、等级（范围为$[1,150]$）。

【输出格式】

一共输出 n 行，每行包括排序后乐斗显示的名字、职位、等级。

【分析】

由题意，我们可以总结出一下本题的做法：首先所有信息输入后排序，排序方式为：先按帮贡排序；若帮贡一样，则按输入顺序排列。再重新编好职位后排输出顺序，也就是职位内的排名，排序方式为：先按现在的职位排序；若职位相同，再按等级排序；如果恰好等级也相同，则按输入顺序排列。

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
using namespace std;
const int maxn = 1e6 + 10;

struct node {
    string name;
    string pos;
    int x, y, pre;

    node() {}

    node(string s1, string s2, int a, int b, int c) {
        name = s1, pos = s2, x = a, y = b, pre = c;
    }
};

int cnt[5] = {2, 4, 7, 25, 200};
string d[5] = {"HuFa", "ZhangLao", "TangZhu", "JingYing", "BangZhong"};

node a[200];
vector<node> res, ans;

bool cmp1(node &p, node &q) {
    if (p.x == q.x)
        return p.pre < q.pre;
    else
        return p.x > q.x;
}

```

```
33. int getNum(string s) {
34.     if (s == "BangZhu")
35.         return 7;
36.     else if (s == "FuBangZhu")
37.         return 6;
38.     else if (s == "HuFa")
39.         return 5;
40.     else if (s == "ZhangLao")
41.         return 4;
42.     else if (s == "TangZhu")
43.         return 3;
44.     else if (s == "JingYing")
45.         return 2;
46.     else
47.         return 1;
48. }
49.
50. bool cmp2(node p, node q) {
51.     if (getNum(p.pos) == getNum(q.pos)) return p.y == q.y ? p.pre < q.pre : p.y >
q.y;
52.     return getNum(p.pos) > getNum(q.pos);
53. }
54.
55. int main() {
56.     int n;
57.     cin >> n;
58.     for (int i = 1; i <= n; i++) {
59.         cin >> a[i].name >> a[i].pos >> a[i].x >> a[i].y;
60.         a[i].pre = i;
61.         if (a[i].pos == "BangZhu" || a[i].pos == "FuBangZhu")
62.             ans.push_back(a[i]);
63.         else
64.             res.push_back(a[i]);
65.     }
66.     sort(res.begin(), res.end(), cmp1);
67.     int k = 0;
68.     for (auto i : res) {
69.         i.pos = d[k];
70.         ans.push_back(i);
71.         cnt[k]--;
72.         if (cnt[k] == 0) k++;
73.     }
74.     sort(ans.begin(), ans.end(), cmp2);
75.     for (auto i : ans) {
```

```
76.         cout << i.name << " " << i.pos << " " << i.y << "\n";
77.     }
78.     return 0;
79. }
```

插松枝（PTA 团队程序设计天梯赛练习集 L2-041）

人造松枝加工场的工人需要将各种尺寸的塑料松针插到松枝干上，做成大大小小的松枝。他们的工作流程是这样的：

每人手边有一只小盒子，初始状态为空。每人面前有用不完的松枝干和一个推送器，每次推送一片随机型号的松针片。

工人首先捡起一根空的松枝干，从小盒子里摸出最上面的一片松针 —— 如果小盒子是空的，就从推送器上取一片松针。将这片松针插到枝干的最下面。工人在插后面的松针时，需要保证，每一步插到一根非空松枝干上的松针片，不能比前一步插上的松针片大。如果小盒子中最上面的松针满足要求，就取之插好；否则去推送器上取一片。如果推送器上拿到的仍然不满足要求，就把拿到的这片堆放到小盒子里，继续去推送器上取下一片。注意这里假设小盒子里的松针片是按放入的顺序堆叠起来的，工人每次只能取出最上面（即最后放入）的一片。

当下列三种情况之一发生时，工人会结束手里的松枝制作，开始做下一个：

（1）小盒子已经满了，但推送器上取到的松针仍然不满足要求。此时将手中的松枝放到成品篮里，推送器上取到的松针压回推送器，开始下一根松枝的制作。

（2）小盒子中最上面的松针不满足要求，但推送器上已经没有松针了。此时将手中的松枝放到成品篮里，开始下一根松枝的制作。

（3）手中的松枝干上已经插满了松针，将之放到成品篮里，开始下一根松枝的制作。

现在给定推送器上顺序传过来的 N 片松针的大小，以及小盒子和松枝的容量，请你编写程序自动列出每根成品松枝的信息。

【输入格式】

输入在第一行中给出 3 个正整数：$N(\leq 10^3)$，为推送器上松针片的数量；$M(\leq 20)$为小盒子能存放的松针片的最大数量；$K(\leq 5)$为一根松枝干上能插的松针片的最大数量。

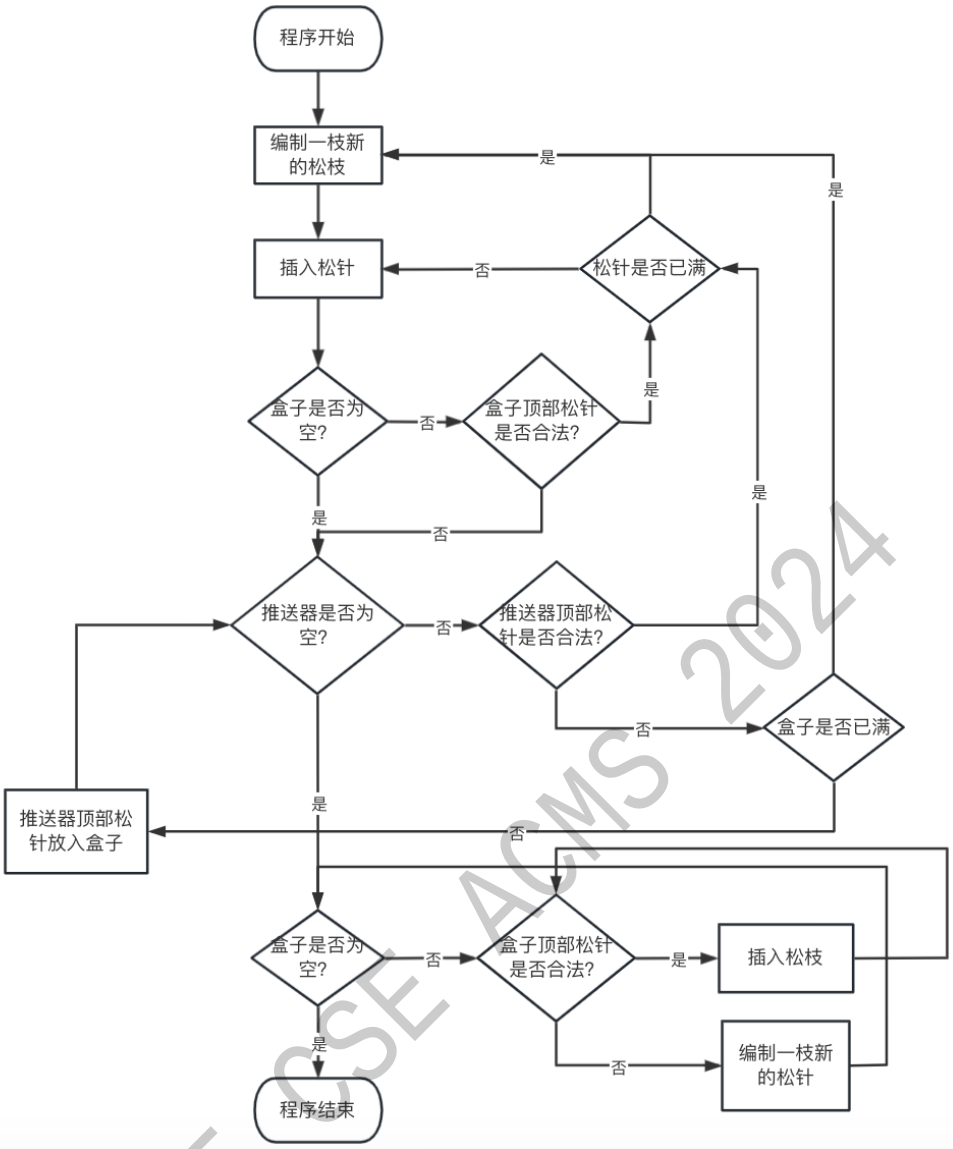
随后一行给出 N 个不超过 100 的正整数，为推送器上顺序推出的松针片的大小。

【输出格式】

每支松枝成品的信息占一行，顺序给出自底向上每片松针的大小。数字间以 1 个空格分隔，行首尾不得有多余空格。

【分析】

本题主要考察模拟的能力，如果能按照题意构造出如下流程图，按照流程图编码、调试会使得解题和 debug 效率大大提高。



如下代码有以下优点：能使用函数的代码段就使用函数，不去嵌套太多的 if-else 语句；对关键变量和函数进行注释，提高 code 效率；松针已满的判断在每次插入松枝后进行；推送器为空后的操作另写独立函数执行；这样极大的减少了编码的逻辑复杂性。

```
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;
const int maxn = 2e5 + 10;

int N, M, K;   // N: 推送器上松针的数目；M: 小盒子的容量；K：松枝的容量
vector<int> machine;   // 推送器
vector<int> box;       // 小盒子
vector<int> s;         // 松枝数组

```

```
13. // 打印松枝
14. void print() {
15.     if (!s.size()) return;
16.     for (int i = 0; i < s.size(); i++) printf("%d%c", s[i], i == s.size() - 1 ?
'\n' : ' ');
17.     s.clear();
18. }
19.
20. // 是否能插入松枝
21. bool insertS(int x) {
22.     if (s.empty()) {
23.         s.push_back(x);
24.         return true;
25.     }
26.     if (s.back() >= x) {
27.         s.push_back(x);
28.         return true;
29.     }
30.     return false;
31. }
32.
33. // 是否能插入盒子
34. bool insertBox(int x) {
35.     if (box.size() < M) {
36.         box.push_back(x);
37.         return true;
38.     }
39.     return false;
40. }
41.
42. // 推送器为空后，只需要判断盒子
43. void onlyBoxOp() {
44.     print();
45.     while (!box.empty()) {
46.         while (!box.empty() && insertS(box.back())) {
47.             box.pop_back();
48.             if (s.size() == K) print();
49.         }
50.         print();
51.     }
52. }
53.
54. void work() {
55.     // 无限的松枝
```

```
    while (1) {
        // 推送器&盒子均为空，结束程序
        if (box.empty() && machine.empty()) break;
        if (!box.empty() && insertS(box.back())) {// 盒子不为空且可以从盒子中成功插入
            box.pop_back();
            if (s.size() == K) {
                print();
                continue;
            }
        } else {
            if (machine.empty()) {
                onlyBoxOp();
                break;
            }
            if (insertS(machine.back())) {
                machine.pop_back();
                if (s.size() == K) {
                    print();
                    continue;
                }
            } else {
                if (insertBox(machine.back()))
                    machine.pop_back();
                else
                    print();
            }
        }
    }
    print();
}

int main() {
    scanf("%d%d%d", &N, &M, &K);
    for (int i = 1, x; i <= N; i++) {
        scanf("%d", &x);
        machine.push_back(x);
    }
    reverse(machine.begin(), machine.end());
    work();
    return 0;
}
```

2.5 递归&分治

2.5.1 递归

递归，在数学和计算机科学中是指在函数的定义中使用函数自身的方法，在计算机科学中还额外指一种通过重复将问题分解为同类的子问题而解决问题的方法。

递归的基本思想是某个函数直接或者间接地调用自身，这样原问题的求解就转换为了许多性质相同但是规模更小的子问题。求解时只需要关注如何把原问题划分成符合条件的子问题，而不需要过分关注这个子问题是如何被解决的。

递归代码最重要的两个特征：结束条件和自我调用。自我调用是在解决子问题，而结束条件定义了最简子问题的答案。

```
1. int func(传入数值) {
2.     if (终止条件) return 最小子问题解;
3.     return func(缩小规模);
4. }
```

递归的优点有：

1. 结构清晰，可读性强。
2. 练习分析问题的结构。当发现问题可以被分解成相同结构的小问题时，递归写多了就能敏锐发现这个特点，进而高效解决问题。

递归的缺点有：在程序执行中，递归是利用堆栈来实现的。每当进入一个函数调用，栈就会增加一层栈帧，每次函数返回，栈就会减少一层栈帧。而栈不是无限大的，当递归层数过多时，就会造成**栈溢出**的后果。比如斐波那契数列的递归写法极大降低了算法的时空效率。

2.5.2 分治

分治就是把一个复杂的问题分成两个或更多的相同或相似的子问题，直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。

分治算法的大概的流程可以分为三步：

1. 分解原问题为结构相同的子问题。
2. 分解到某个容易求解的边界之后，进行递归求解。
3. 将子问题的解合并成原问题的解。

分治法能解决的问题一般有如下特征：

- 该问题的规模缩小到一定的程度就可以容易地解决。
- 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质，利用该问题分解出的子问题的解可以合并为该问题的解。

- 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。

以归并排序为例。假设归并排序的函数为 merge_sort，完成有序数组归并的函数为 merge。要解决的问题是：对传入的一个数组排序。这个问题可以分解，给一个数组排序等于给该数组的左右两半分别排序，然后合并成一个数组。

```
1. void merge_sort(一个数组) {
2.     if (可以很容易处理) return;
3.     merge_sort(左半个数组);
4.     merge_sort(右半个数组);
5.     merge(左半个数组, 右半个数组);
6. }
```

注意到，merge_sort 与二叉树的后序遍历模板极其相似。因为分治算法的思路是分解→解决（触底）→合并（回溯），先左右分解，再处理合并，回溯就是在退栈，即相当于后序遍历。

分治和递归的区别为：递归是一种编程技巧，一种解决问题的思维方式；分治算法很大程度上是基于递归的，解决更具体问题的算法思想。

2.5.3 例题

外星密码（洛谷 P1928）

有了防护伞，并不能完全避免 2012 的灾难。地球防卫小队决定去求助外星种族的帮助。经过很长时间的努力，小队终于收到了外星生命的回信。但是外星人发过来的却是一串密码。只有解开密码，才能知道外星人给的准确回复。解开密码的第一道工序就是解压缩密码，外星人对于连续的若干个相同的子串 X 会压缩为 [DX] 的形式（D 是一个整数且 $1 \le D \le 99$），比如说字符串 CBCBCBCB 就压缩为 [4CB] 或者[2[2CB]]，类似于后面这种压缩之后再压缩的称为二重压缩。如果是 [2[2[2CB]]] 则是三重的。现在我们给你外星人发送的密码，请你对其进行解压缩。

【输入格式】

输入一行，一个长度不超过 20000 的字符串，表示外星人发送的密码。题目保证不超过十层压缩。

【输出格式】

输出一行，一个字符串，表示解压缩后的结果。

【分析】

使用递归算法，在读入这个字符串之后，找出被压缩的内容，再对被压缩的那个字符串实行解压缩操作。

以 SAD[2MLE[2TLE[2RE[2WA]]]] 为例分析本题。首先是找到被压缩的部分：[2MLE[2TLE[2RE[2WA]]]]，从此处开始递归。对这个部分进行解压，找到被压缩的部分：

[2TLE[2RE[2WA]]]；再对这个部分进行解压，找到被压缩的部分：[2RE[2WA]]；再对这个部分进行解压，找到被压缩的部分：[2WA]。接着开始一层一层跳出递归，对每个部分进行解压并加到前一个字符串的末尾。递归结束，密码破译完毕。

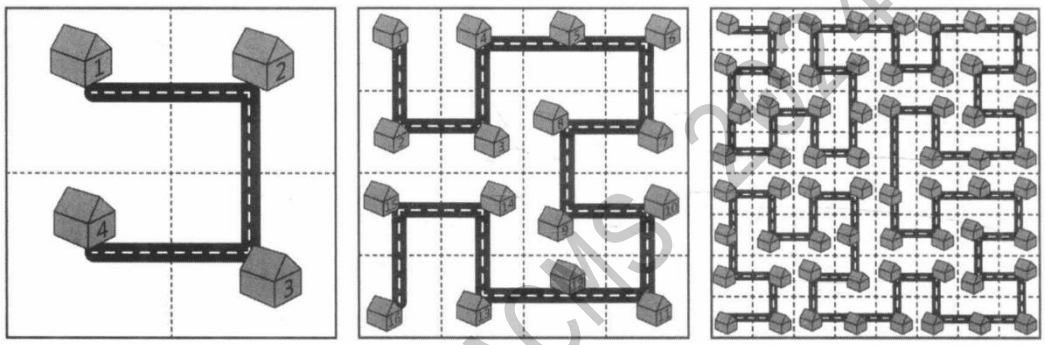
解决递归题目最重要的是：千万不要跳进递归函数里面企图探究更多细节，否则就会陷入无穷的细节无法自拔。

```cpp
#include <iostream>
#include <stack>
using namespace std;
const int maxn = 2e5 + 10;

string dfs(string s) {
    string ans = "";
    int k = 0;
    for (int i = 0; i < s.size(); i++) {
        if (s[i] == '[') {
            stack<char> st;
            int j;
            for (j = i; j < s.size(); j++) {
                if (s[j] == '[')
                    st.push(s[j]);
                else if (s[j] == ']') {
                    st.pop();
                    if (st.empty()) break;
                }
            }
            ans += dfs(s.substr(i + 1, j - i - 1));
            i = j;
        } else {
            if (s[i] >= '0' && s[i] <= '9')
                k = k * 10 + s[i] - '0';
            else
                ans += s[i];
        }
    }
    if (k == 0) k = 1;
    string res = "";
    for (int i = 0; i < k; i++) res += ans;
    return res;
}

// CD[2AB[3XY]C]

int main() {
```

```
    string s;
    cin >> s;
    cout << dfs(s) << "\n";
    return 0;
}
```

分形之城（AcWing 98）

城市扩建的规划是个令人头疼的大问题。规划师设计了一个极其复杂的方案：当城市规模扩大之后，把与原来城市结构一样的区域复制或旋转 90 度之后建设在原来的城市周围（详细地说，将原来的城市复制一遍放在原城市的上方，将顺时针旋转 90 度后的城市放在原城市的左上方，将逆时针旋转 90 度后的城市放在原城市的左方），再用道路将四部分首尾连接起来，如下图所示。



等级 1　　等级 2　　等级 3

容易看出，扩建后的城市的各个房屋仍然由一条道路连接。定义 N 级城市为拥有2^{2N}座房屋的城市。对于任意等级的城市，从左上角开始沿着唯一的道路走，依次为房屋标号，就能够得到每间房屋的编号了。住在其中两间房屋里的两人想知道，如果城市发展到了一定等级，他俩各自所处的房屋之间的直线距离是多少。你可以认为图中的每个格子都是边长为 10 米的正方形，房屋均位于每个格子的中心点上。

【输入格式】

第一行输入正整数 $n(1 \le n \le 1000)$，表示测试数据的数目。

以下 n行，输入 n 组测试数据，每组一行，包括三个整数 $N, A, B(1 \le N \le 31, 1 \le A, B \le 2^{2N})$，表示城市等级以及两个街区的编号，整数之间用空格隔开。

【输出格式】

一共输出 n 行数据，每行对应一组测试数据的输出结果，结果四舍五入到整数。

【分析】

这是著名的通过一定规律无限包含自身的“分形”图。为了计算方便，我们把题目中房屋的编号都减去 1，即从 0 开始编号，并把 A 和 B 也都减掉 1。

本题关键是要解决：求编号为 M 的房屋（从 0 开始编号）在N级城市中的位置。把该问题记为 $calc(N, M)$，本题就是求 $calc(N, A)$与 $calc(N, B)$ 的距离。

不难看出，$N(N > 1)$级城市由四座 $N-1$级城市组成，其中左上的 $N-1$ 级城市顺时

针旋转了 90 度，左下的 $N-1$ 级城市逆时针旋转了 90 度。进一步观察，当这四座 $N-1$ 级城市首尾相接后，左上、左下的 $N-1$ 级城市的房屋编号顺序各自发生了颠倒，这相当于左上、左下两座城市发生了“水平翻转”。读者可以根据题目中等级 1 和等级 2 的两幅图验证上述结论。

在求解 $calc(N,M)$ 时，因为 $N-1$ 级城市有 $2^{2N}-2$ 座房屋，所以我们先递归求解 $calc(N-1, M\ mod\ 2^{2N}-2)$，记求出的位置为 (x,y)，其中 x 为行号，y 为列号，从 0 开始编号。再根据 $\frac{M}{2^{2N}}-2$ 的大小，很容易确定编号为 M 的房屋处于四座 $N-1$ 级城市中的哪一座。

1. 若处于左上的 $N-1$ 级城市中，则需要把 (x,y) 所在的 $N-1$ 级城市顺时针旋转 90 度，坐标变为 $(y, 2^{N-1}-x-1)$，再水平翻转，坐标最终变为 (y,x)。这就是该房屋在 N 级城市中的位置。

2. 若处于右上的 $N-1$ 级城市中，则该房屋在 N 级城市中的位置应为 $(x, y+2^{N-1})$。

3. 若处于右下的 $N-1$ 级城市中，则该房屋在 N 级城市中的位置应为 $(x+2^{N-1}, y+2^{N-1})$。

4. 若处于左下的 $N-1$ 级城市中，则需要把 (x,y) 所在的 $N-1$ 级城市逆时针旋转 90 度再水平翻转，坐标变为 $(2^{N-1}-y-1, 2^{N-1}-x-1)$。在 N 级城市中的位置还要把行号再加 2^{N-1}，最终得到 $(2^N-y-1, 2^{N-1}-x-1)$。

```cpp
#include <cmath>
#include <iostream>
using namespace std;
typedef long long ll;

pair<ll, ll> calc(ll n, ll m) {
    if (n == 0) return make_pair(0, 0);
    ll len = 1LL << (n - 1), cnt = 1LL << (2 * n - 2);
    pair<ll, ll> pos = calc(n - 1, m % cnt);
    ll x = pos.first, y = pos.second;
    ll z = m / cnt;
    if (z == 0) return make_pair(y, x);
    if (z == 1) return make_pair(x, y + len);
    if (z == 2) return make_pair(x + len, y + len);
    return make_pair(2 * len - 1 - y, len - 1 - x);
}

int main() {
    int t;
    scanf("%d", &t);
    while (t--) {
        ll n, a, b;
```

```
23.         scanf("%lld%lld%lld", &n, &a, &b);
24.         pair<ll, ll> x = calc(n, a - 1);
25.         pair<ll, ll> y = calc(n, b - 1);
26.         ll dx = x.first - y.first, dy = x.second - y.second;
27.         double ans = (sqrt(double(dx * dx + dy * dy)) * 10);
28.         printf("%0.lf\n", ans);
29.     }
30.     return 0;
31. }
```

2.6 递推

对一个待求解的问题，当它局限在某处边界、某个小范围或者某种特殊情形下时，其答案往往是已知的。如果能将该解答的应用场景扩大到原问题的状态空间，并且扩展过程的每个步骤具有相似性，就可以考虑使用递推（或者递归）求解。

以已知的“问题边界”为起点向“原问题”正向推导的扩展方式就是递推。很多时候推导路线难以确定，这时以“原问题”为起点尝试寻找把状态空间缩小到已知“问题边界”的路线，再通过该路线反向回溯的遍历方式就是上一节讲到的递归。

递推和数学联系紧密，比如许多数列都有递推公式，这种数列称之为递推序列；递推还和动态规划息息相关。

费解的开关（AcWing 95）

你玩过“拉灯”游戏吗？25 盏灯排成一个 5×5 的方形。每一个灯都有一个开关，游戏者可以改变它的状态。每一步，游戏者可以改变某一个灯的状态。游戏者改变一个灯的状态会产生连锁反应：和这个灯上下左右相邻的灯也要相应地改变其状态。我们用数字 1 表示一盏开着的灯，用数字 0 表示关着的灯。

给定一些游戏的初始状态，编写程序判断游戏者是否可能在 6 步以内使所有的灯都变亮。

【输入格式】

第一行输入正整数 n，代表数据中共有 n 个待解决的游戏初始状态。

以下若干行数据分为 n 组，每组数据有 5 行，每行 5 个字符。每组数据描述了一个游戏的初始状态。各组数据间用一个空行分隔。

【输出格式】

一共输出 n 行数据，每行有一个小于等于 6 的整数，它表示对于输入数据中对应的游戏状态最少需要几步才能使所有灯变亮。

对于某一个游戏初始状态，若 6 步以内无法使所有灯变亮，则输出 −1。

【分析】

在上述规则的 01 矩阵的点击游戏中，很容易发现三个性质:

1. 每个位置至多只会被点击一次。

2. 若固定了第一行(不能再改变第一行)，则满足题意的点击方案至多只有 1 种。其原因是:当第i行某一位为 1 时，若前i行已被固定，只能点击第$i+1$行该位置上的数字才能使第i行的这一位变成 0。从上到下按行使用归纳法可得上述结论。

3. 点击的先后顺序不影响最终结果。

于是，我们不妨先考虑第一行如何点击。在枚举第一行的点击方法($2^5 = 32$种)后，就可以认为第一行“固定不动”，再考虑第 2~5 行如何点击。而按照上述性质 2,此时第 2~5 行的点击方案是确定的一从第一行开始递推，当第i行某一位为 1 时，点击第$i+1$行该位置上的数字。若到达第n行时不全为 0，说明这种点击方式不合法。在所有合法的点击方式中取点击次数最少的就是答案。对第一行的 32 次枚举涵盖了该问题的整个状态空间，因此该做法是正确的。

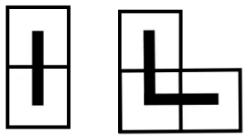
对于第一行点击方法的枚举，可以采用位运算的方式，枚举0~31这 32 个 5 位二进制数，若二进制数的第$k(0 \le k < 5)$位为 1，就点击 01 矩阵第 1 行第$k+1$列的数字。

```
#include <cstring>
#include <iostream>
using namespace std;

int a[10][10], b[10][10];
const int dx[] = {1, -1, 0, 0}, dy[] = {0, 0, 1, -1};

bool judge() {
    for (int i = 1; i <= 5; i++)
        for (int j = 1; j <= 5; j++) {
            if (b[i][j] == 0) return 0;
        }
    return 1;
}

bool check(int x, int y) {
    return x >= 1 && x <= 5 && y >= 1 && y <= 5;
}

void click(int r, int c) {
    b[r][c] ^= 1;
    for (int i = 0; i < 4; i++) {
        int x = r + dx[i], y = c + dy[i];
        if (check(x, y)) {
            b[x][y] ^= 1;
        }
    }
}
```

```
29.
30. int main() {
31.     int n;
32.     string str;
33.     cin >> n;
34.     while (n--) {
35.         for (int i = 1; i <= 5; i++) {
36.             cin >> str;
37.             for (int j = 1; j <= 5; j++) {
38.                 a[i][j] = str[j - 1] - '0';
39.             }
40.         }
41.         int up = 1 << 5, ans = 1e9;
42.         for (int s = 0; s < up; s++) {
43.             memcpy(b, a, sizeof a);
44.             int cnt = 0;
45.             for (int k = 0; k <= s; k++) {
46.                 if (s & (1 << k)) {
47.                     cnt++;
48.                     click(1, k + 1);
49.                 }
50.             }
51.             for (int i = 1; i <= 4; i++) {
52.                 for (int j = 1; j <= 5; j++) {
53.                     if (b[i][j] == 0) {
54.                         cnt++;
55.                         click(i + 1, j);
56.                     }
57.                 }
58.             }
59.             if (judge()) ans = min(ans, cnt);
60.         }
61.         if (ans <= 6)
62.             cout << ans << "\n";
63.         else
64.             cout << -1 << "\n";
65.     }
66.     return 0;
67. }
```

覆盖墙壁（洛谷 P1990）

给出一个$2*n$的墙壁，规定只能用两种可以任意旋转的砖头覆盖，砖头的形状如下：



求使用这两种砖头覆盖墙壁的方案数对 10000 取模的结果。

【输入格式】

第一行输入正整数 $n(1 \le n \le 10^6)$，表示墙壁的长。

【输出格式】

输出覆盖方法对 10000 取模的结果。

【分析】

这种题目主要是推出递推式，虽然是二维的，但是定义状态时仍可以定义 $f[i]$ 表示填充满前面的 $2*i$ 个格子的方案数，假设当前要填第 i 列，那么可以尝试一些放置方法，然后从前面已经得出的答案中递推求出：

（1）假设竖着放置 $1*2$ 的砖块在第 i 列，那么 $f[i] += f[i-1]$。

（2）假设最后两列横着放两个 $1*2$ 的砖块，那么 $f[i] += f[i-2]$。

（3）假设最后放置一个丁字砖块(这个砖块有两种放置方法，因此最后这个累加的答案乘二），此时会得到一个凸出，为了消除这个凸出，可以再放置一个丁字块，这时 $f[i] += f[i-3]$；还可以先放一个 $1*2$，然后再放置一个丁字块，这时 $f[i] += f[i-4]$；仔细观察，还可以上下交错放两个 $1*2$，然后放一个丁字块补齐凸出，这时 $f[i] += f[i-5]$...以此类推可以发现，这种放置方式最终能放置的方案总数为 $\sum_0^{i-3} f[i]$。

综上，$f[i] = f[i-1] + f[i-2] + 2*\sum_0^{i-3} f[i]$，初始化为 $f[0]=1, f[1]=1, f[2]=2$。

```
#include <iostream>
using namespace std;
typedef long long ll;
const int Mod = 10000;
const int maxn = 1e6 + 10;

ll f[maxn], sum[maxn];

int main() {
    int n;
    cin >> n;
    f[0] = 1, f[1] = 1, f[2] = 2;
    sum[0] = 1, sum[1] = 2, sum[2] = 4;
    for (int i = 3; i <= n; i++) {
        f[i] = (f[i] + f[i - 1]) % Mod;
        f[i] = (f[i] + f[i - 2]) % Mod;
        f[i] = (f[i] + sum[i - 3] * 2) % Mod;
```

```
        sum[i] = (sum[i - 1] + f[i]) % Mod;
    }
    cout << f[n] << endl;
    return 0;
}
```

2.7 枚举子集

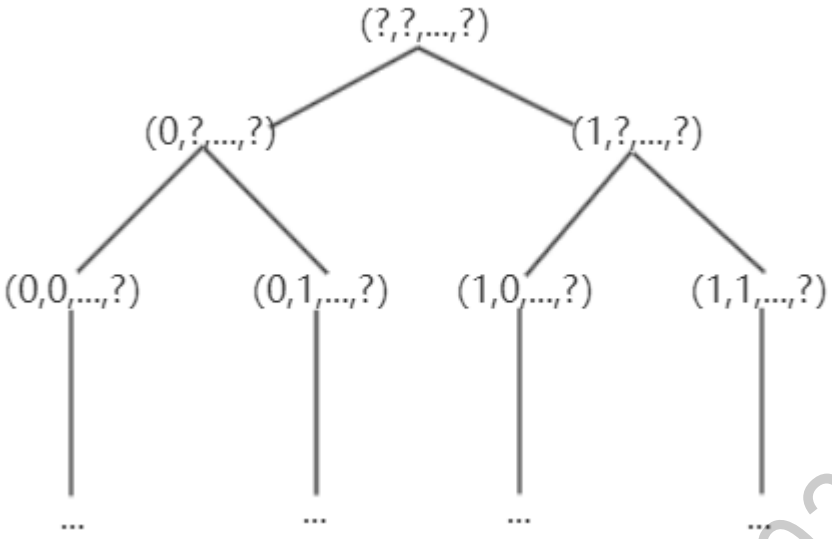
2.7.1 增量构造法

思路：设初始ans集合为空，每次从枚举当前集合最小值以后的元素，将当前枚举到的一个元素放到ans集合中，然后继续递归构造子集。这样保证每次的ans集合一定是没有出现过的。时间复杂度为$O(2^n)$。

```
int a[maxn], ans[maxn];
int n;

void print_subset(int cur) {   // cur 初始输入 1
    for (int i = 1; i < cur; i++) printf("%d%c", ans[i], i == cur - 1 ? '\n' : ' ');
    int s = ans[cur - 1] + 1;   // 确定当前最小元素
    for (int i = s; i <= n; i++) {
        ans[cur] = i;
        print_subset(cur + 1);
    }
}
```

2.7.2 位向量法

思路：对于大小为n集合中的所有元素，设置一个布尔数组$vis[i]$代表集合的第i位是否在当前子集。由于必须当所有元素确定是否存在于子集才能得到该子集，因此必须在布尔数组n个位置都确定后才可以输出子集。构成的解答树如下：

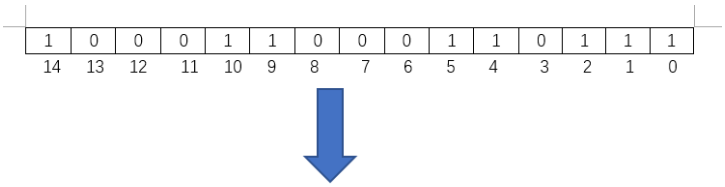



```
1. bool vis[maxn];
2. int a[maxn];
3. int n;
4.
5. void print_subset(int cur) {
6.     if (cur == n + 1) {
7.         for (int i = 1; i <= n; i++)
8.             if (vis[i]) printf("%d ", a[i]);
9.         printf("\n");
10.        return;
11.    }
12.    vis[cur] = 1;
13.    print_subset(cur + 1);
14.    vis[cur] = 0;
15.    print_subset(cur + 1);
16. }
```


2.7.3 二进制法

观察位向量法不难发现每一个元素是否选取的标准是布尔数组的值，考虑到二进制的每一位也是 0/1，因为计算机底层的运算就是比特之间的运算，使用二进制优化可以极大提高算法的时空效率。

二进制从右向左（从低位到高位）第 i 位的大小是2^i，那么集合{0,1,2,4,5,9,10,14}就可以被表示为 100011000110111，如下图所示：



对于大小为n的集合，从 0 枚举到2^{n-1}，可以发现，刚好每个子集对应的 0/1 串都出现了一次，那么只需要枚举就可以遍历所有的子集。

```
1. void print_set(int s) {
2.     for (int i = 0; i < n; i++)
3.         if (s & (1 << i)) printf("%d ", i);
4.     printf("\n");
5. }
6.
7. void print_subset(int n) {
8.     for (int i = 0; i < (1 << n); i++) print_subset(i);
9. }
```

2.8 前缀和与差分

2.8.1 前缀和

前缀和可以简单理解为“数列a的前n项的和”，前缀和S可以表示为$S[i]=\sum_{j=1}^{i}a[j]$。前缀和是一种重要的预处理方式，能大大降低查询的时间复杂度。初始化前缀和的方式很简单：

```
1. for(int i = 1; i <= n; i++) {
2.     S[i] = S[i - 1] + a[i];
3. }
```

前缀和最主要的性质为：数列a中任何一个区间$[l,r]$的和，都可以被表示为：$sum(l,r)=\sum_{i=l}^{r}a_i=S[r]-S[l-1]$。

同理在二维矩阵中，也可以求出前缀和。对于n行m列的二维数组b，设前缀和$sum[i][j]=\sum_{i=1}^{n}\sum_{j=1}^{m}b[i][j]$。这里前缀和数组的构建利用了容斥定理。类似一维前缀和的方法去求二维前缀和：$sum[i][j]=sum[i-1][j]+sum[i][j-1]+b[i][j]$，会发现和预期结果不一样，是因为$sum[i][j]$表示以$(1,1)$为左上角，$(i,j)$为右下角的子矩阵，上面的递推式显然多加了一个$sum[i-1][j-1]$，因此正确的递推式为：

$$sum[i][j]=sum[i-1][j]+sum[i][j-1]-sum[i-1][j-1]+b[i][j]$$

```
1. for (int i = 1; i <= n; i++) {
```

```
    for (int j = 1; j <= m; j++) {
        sum[i][j] = sum[i - 1][j] + sum[i][j - 1] - sum[i - 1][j - 1] + b[i][j];
    }
}
```

得到了二维前缀和，如何求一个左上角为(x_1,y_1)，右下角为(x_2,y_2)的子矩阵元素的和？还是利用二维前缀和的性质：**$sum[i][j]$表示以$(1,1)$为左上角，(i,j)为右下角的子矩阵。**因此不难得出：$sum[x_2][y_2]-sum[x_1-1][y_2]-sum[x_2][y_1-1]+sum[x_1-1][y_1-1]$。

最大加权矩形（洛谷 P1719）

给定一个 $n\times n$ 矩阵。要求矩阵中最大加权矩形，即矩阵的每一个元素都有一权值，权值定义在整数集上。从中找一矩形，矩形大小无限制，是其中包含的所有元素的和最大 。矩阵的每个元素属于 $[-127,127]$。

【输入格式】

第一行输入正整数 $n(1\le n\le 120)$，接下来是n行n列的矩阵。

【输出格式】

输出最大子矩阵的和。

【分析】

在学习本题之前应该掌握最大子段和问题：对于一个线性的序列，如何找到一段区间的数，使得其和最大。

考虑暴力的做法，如果枚举矩阵的两个行坐标和两个列坐标，这样能在时间复杂度$O(n^4)$内解决本题。但这个复杂度稍大，尝试优化。数据范围使得$O(n^3)$复杂度做法绰绰有余，这里一个关键的优化思路是：**二维变一维**。

也就是说，把矩阵的每一列，从第 1 行开始逐行累加到下一行，这样矩阵的每一个位置(i,j)代表的是$\sum_{k=1}^{i}a[k][j]$，即进行了列压缩。这样我们在枚举时只需要枚举两个行坐标，然后是利用最大字段在垂直的“压缩序列”中寻找最大字段和。不难发现这样可以和朴素算法一样枚举到每一个子矩阵，并且优化了时间复杂度。

```
#include <iostream>
using namespace std;

int a[150][150], sum[150][150];

int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            cin >> a[i][j];
            sum[i][j] = sum[i - 1][j] + a[i][j];
        }
    }
```

```
15.
16.     int ans = -1e9;
17.     for (int i = 0; i < n; i++) {
18.         for (int j = i + 1; j <= n; j++) {
19.             int res = 0, pre_min = 0;
20.             for (int k = 1; k <= n; k++) {
21.                 res = res + sum[j][k] - sum[i][k];
22.                 ans = max(ans, res - pre_min);
23.                 pre_min = min(pre_min, res);
24.             }
25.         }
26.     }
27.     cout << ans << endl;
28.     return 0;
29. }
```

2.8.2 差分

差分即相邻两个数的差，给定下标从 1 开始的数组a我们能得到其差分数组$d[i]=a[i]-a[i-1],a[0]=0$。

差分与前缀和的区别：前缀和求的是$sum[i]=\sum_{j=1}^{i}a[j]$，如果使用前缀和表示某个位置的元素，那么$a[i]=sum[i]-sum[i-1]$。对比差分，我们发现差分和前缀和在预处理和求和的运算刚好是一对互逆运算，“差分序列”的“前缀和序列”就是原序列，“前缀和序列”的“差分序列”也是原序列。

差分序列的重要应用为：当我们把序列a的区间$[l,r]$上每个元素都加上x，其差分序列 d 的变化为$d[l]$加x，$d[r+1]$减 x，其他位置不变。在很多题目中，利用差分可以把原序列上的“区间操作”转化为差分序列上的“单点操作”进行计算，降低求解难度。

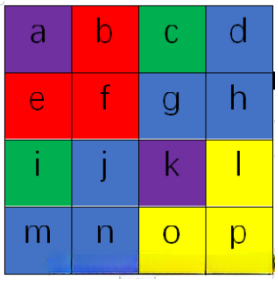
同理二维前缀和，也存在二维数组的差分，即二维差分。类似二维前缀和，我们可以推导出二维差分的递推式：$d[i][j]=a[i][j]-a[i][j-1]-a[i-1][j]+a[i-1][j-1]$。

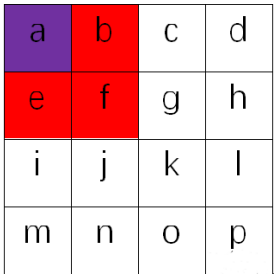
```
1. for (int i = 1; i <= n; i++) {
2.     for (int j = 1; j <= m; j++) {
3.         cin >> a[i][j];
4.         d[i][j] = a[i][j] - a[i - 1][j] - a[i][j - 1] + a[i - 1][j - 1];
5.         // 对比前缀和：sum[i][j] = a[i][j] + sum[i - 1][j] + sum[i][j - 1] - sum[i -
1][j - 1];
6.     }
7. }
```

而对于原矩阵某个位置的值，等价于以这个元素为右下角，原矩阵左上角为左上角的子矩阵中所有元素的和，也就是说对差分数组d求二维前缀和。

一维差分可以方便对一维序列进行区间加法，那么二维差分可以方便对子矩阵进行加法。

差分矩阵中改变一个数影响的是以该元素为左上角，原矩阵的右下角(n,m)为右下角形成子矩阵区域。类似于一维差分，二维差分可以进行快速子矩阵加法，当我们需要对如下左图的红色区域加上一个数p时，只需要$d[1][1]+p$。





产生的影响为整个矩阵，为了消除其他位置的影响，我们需要将上述右图白色区域的影响消除，也就是要更新$d[3][1],d[1][3],d[3][3]$，蓝色区域需要减去p即绿色位置减去p，而黄色区域多减了，需要再加上p，即所有紫色区域加上p。

```
1. void add(int i, int j, int x, int y, int val) {
2.     d[i][j] += val, d[x + 1][y + 1] += val;
3.     d[x + 1][j] -= val, d[i][y + 1] -= val;
4. }
```

如何由差分矩阵求得原矩阵？类似于一维差分，当我们需要求得原矩阵时，不可能对于每一个元素和左上角形成的子矩阵求和，这样时间复杂度太高，实际上我们可以使用二维循环，对于第一列，手算一下不难得出是正确的，然后对于每个元素左上角 2 * 2 的子矩阵，除了当前元素的位置其余三个位置都是原矩阵的元素，那么我们对上述求差分的过程反着来即可。

$$a[i][j]=d[i][j]+a[i-1][j]+a[i][j-1]-a[i-1][j-1] \Leftrightarrow a'[i][j]=d[i][j]+d[i-1][j]+d[i][j-1]-d[i-1][j-1]$$

这样当我们对差分矩阵动态修改之后就可以$O(nm)$得到新的矩阵了。

Monitor（HDU 6514）

给出一个$n*m$的麦田（左下角为(1,1)，右上角(n,m)），其中有 p 个矩形区域安装了监控，接下来有 q 个贼想偷某个矩形范围内的庄稼，问监控能否拍到所有的贼。

【输入格式】

本题有多个测试用例。

每个测试用例的第一行包括两个正整数$n,m(1\le n,1\le m,1\le n*m\le 10^7)$。

第二行包括一个正整数$p(1\le p\le 10^6)$代表监控的数量。接下来p行，每行四个正整数$x_1,y_1,x_2,y_2(1\le x_1,x_2\le n,1\le y_1,y_2\le m)$，代表监控范围矩阵的左下角和右上角。

接下来一行包括一个正整数$q(1\le q\le 10^6)$代表贼的数量。然后q行，每行四个正整数$x_1,y_1,x_2,y_2(1\le x_1,x_2\le n,1\le y_1,y_2\le m)$，代表贼偷窃子矩阵的左下角和右上角。

【输出格式】

对于每个测试用例应该输出 q 行。每行包括一个字符串 YES 或 NO，代表监控能否拍到所有的贼。

【分析】

首先我们需要将有监控的区域都置为 1，也就是每个监控区域的所有元素都加一，这时不难想到二维差分，有的元素可能被多个监控重复覆盖，那么差分矩阵还原时需要将大于 1 的元素置为 1。然后对还原的矩阵求前缀和，只需要看贼偷东西的矩阵的元素和是否为矩阵面积。

题目有两大坑点：

（1）题目给的坐标是左下角为(1,1)，右上角(n,m)的矩阵，但是计算机存储的矩阵是左上角为(1,1)，右下角(n,m)。要么坐标转化一下，要么将矩阵颠倒一下。

（2）矩阵的范围是 $n*m \le 10^7$的动态二维数组，无法通过数组存，要么使用 vector 套 vector，要么就使用一维数组模拟二维数组，将下标转化一下。

```
#include <iostream>
#include <vector>
using namespace std;
typedef long long ll;

const int maxn = 1e7 + 10;

void add(vector<vector<int>> &d, int i, int j, int x, int y, int val) {
    d[i][j] += val, d[x + 1][y + 1] += val;
    d[i][y + 1] -= val, d[x + 1][j] -= val;
}

int getSum(vector<vector<int>> &d, int i, int j, int x, int y) {
    return d[x][y] - d[i - 1][y] - d[x][j - 1] + d[i - 1][j - 1];
}

int main() {
    int n, m, p, q;
    int lx, ly, rx, ry;
    while (scanf("%d%d", &n, &m) != EOF) {
        vector<vector<int>> d(n + 2, vector<int>(m + 2));
        scanf("%d", &p);
        while (p--) {
            scanf("%d%d%d%d", &lx, &ly, &rx, &ry);
            lx = n + 1 - lx, rx = n + 1 - rx;
            swap(lx, rx);
            add(d, lx, ly, rx, ry, 1);
        }
        for (int i = 1; i <= n; i++) {
```

```
            for (int j = 1; j <= m; j++) {
                d[i][j] = d[i][j] + d[i - 1][j] + d[i][j - 1] - d[i - 1][j - 1];
            }
        }
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                if (d[i][j]) d[i][j] = 1;
            }
        }
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                d[i][j] = d[i][j] + d[i - 1][j] + d[i][j - 1] - d[i - 1][j - 1];
            }
        }
        scanf("%d", &q);
        while (q--) {
            scanf("%d%d%d%d", &lx, &ly, &rx, &ry);
            lx = n + 1 - lx, rx = n + 1 - rx;
            swap(lx, rx);
            int ans = getSum(d, lx, ly, rx, ry);
            if (ans == (rx - lx + 1) * (ry - ly + 1))
                puts("YES");
            else
                puts("NO");
        }
    }
    return 0;
}
```

[Poetize6] IncDec Sequence（洛谷 P4552）

给定一个长度为 n 的数列$\{a_1, a_2, \ldots, a_n\}$，每次可以选择一个区间$[l, r]$，使这个区间内的数都加 1 或者都减 1。

请问至少需要多少次操作才能使数列中的所有数都一样，并求出在保证最少次数的前提下，最终得到的数列有多少种。

【输入格式】

第一行一个正整数 $n(n \leq 10^5)$。

接下来 n 行,每行一个整数,第 $i+1$ 行的整数表示 $a_i(0 \leq a_i \leq 2^{31})$。

【输出格式】

第一行输出最少操作次数。

第二行输出最终能得到多少种结果。

【分析】

求出 a 的差分序列 b，其中$b_1 = a_1$，$b_i = a_i - a_{i-1}(2 \leq i \leq n)$，令$b_{n+1} = 0$。题目对

序列a的操作，相当于每次可以选出$b_1, b_2, \ldots, b_{n+1}$中的任意两个数，一个加 1，另一个减 1。目标是把 $b_2, b_3, \ldots, b_n$变为全零。最终得到的数列 a 就是由 n 个 b_1构成的。

从$b_1, b_2, \ldots, b_{n+1}$中任选两个数的方法可分为四类:

1. 选 b_i 和 b_j，其中$2 \le i, j \le n$。这种操作会改变 $b_2, b_3, \ldots, b_n$ 中两个数的值。应该在保证 b_i 和 b_j 一正一负的前提下，尽量多地采取这种操作，更快地接近目标。

2. 选 b_1 和 b_j，其中$2 \le j \le n$。

3. 选 b_j 和 b_{n+1}，其中$2 \le i \le n$。

4. 选 b_1 和 b_{n+1}。这种情况没有意义，因为它不会改变 $b_2, b_3, \ldots, b_n$ 的值，相当于浪费了一次操作，一定不是最优解。

设 $b_2, b_3, \ldots, b_n$ 中正数总和为p，负数总和的绝对值为 q。首先以正负数配对的方式尽量执行第 1 类操作，可执行$\min(p, q)$ 次。剩余$|p - q|$个未配对，每个可以选与b_1 或b_{n+1}配对，即执行第 2 或 3 类操作，共需$|p - q|$次。

综上所述，最少操作次数为$\min(p, q) + |p - q| = \max(p, q)$次。根据$|p - q|$次第 2、3 类操作的选择情况，能产生$|p - q| + 1$种不同的$b_1$的值，即最终得到的序列$a$可能有$|p - q| + 1$种。

```cpp
#include <iostream>
using namespace std;
typedef long long ll;
const int maxn = 1e5 + 10;

ll a[maxn], b[maxn];

int main() {
    int n;
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) scanf("%lld", &a[i]);
    ll q = 0, p = 0;
    for (int i = 2; i <= n; i++) {
        b[i] = a[i] - a[i - 1];
        if (b[i] >= 0)
            q += b[i];
        else
            p += abs(b[i]);
    }
    ll ans = min(q, p) + abs(q - p);
    printf("%lld\n%lld\n", ans, abs(q - p) + 1);
    return 0;
}
```

2.9 离散化

离散化本质上可以看成是一种哈希，其保证数据在哈希以后仍然保持原来的全/偏序关系。

通俗地讲就是当有些数据因为本身很大或者类型不支持，自身无法作为数组的下标来方便地处理，而影响最终结果的只有元素之间的相对大小关系时，我们可以将原来的数据按照从大到小编号来处理问题，即离散化。用来离散化的可以是大整数、浮点数、字符串等等。

1. 去重离散化

给定一个包含重复元素数组，而我们用到其中的每一个数即可，这时可以使用 C++自带的 unique 函数进行离散化，但要在使用前将无序的数组排序，使得重复的元素相邻。（PS：这里的去除并非真正意义的删除，而是将重复的元素放到容器的末尾，返回值是去重之后的尾地址）

2. 映射离散化

映射离散化通常采用给数组每个数映射一个 id 的方法得到一个新的数组。例如一个四个数的数组$a = \{10^4, 10^6, 10^8, 10^{10}\}$，如果给每个数一个映射将$a$数组转化为$b = \{1,2,3,4\}$，新的数组虽然与原数组无关，但是数组元素之间的相对大小关系是相同的。

常见的映射离散化还可以使用map等数据结构。

在归并排序一节中我们提到了逆序对的概念，使用树状数组解决逆序对问题，可以视为离散化的一个较好的案例，感兴趣者可以课下查阅，也可以在学习过树状数组后自行思考。

2.10 贪心

贪心算法，是用计算机来模拟一个贪心的人做出决策的过程。程序每一步行动总是按某种指标选取最优的操作，总是只考虑眼前的情况，并不考虑以后可能造成的影响。因此，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。

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。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. 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。

2.10.1 经典问题

1. 排队接水问题

问题描述：n个人在一个水龙头前排队接水，假如每个人接水的时间为 T_i，找出一个排队顺序，使得 n 个人的等待时间之和最少。

思路：每个人的等待时间为前面人的接水时间之和，排队越靠前被计算次数越多，因此越小的排在前面显然是最优的。

2. 选择不相交区间

问题描述：数轴上有 n 个开区间 (l_i, r_i)，从中选出尽量多的区间，使得选出的这些区间两两没有交点。

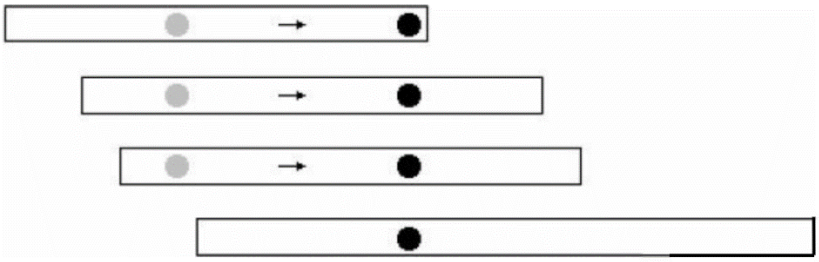
思路：若区间相互包含，即区间 x 完全包含 y。那么选择 x 是不划算的，因为 x 和 y最多只能选择一个，而我们要选择对其他区间影响最小的。接下来按照右端点 r 进行升序排序（之所以不按照左端点 l 排序，考虑一下三条线段 $[1,5], [2,3], [4,5]$ 的情况）。贪心策略是：一定要选第一个区间，把所有和区间 1 相交的区间排除在外，再选其后第一个可以选择的区间，循环进行。

3. 区间选点问题

问题描述：数轴上有 n 个闭区间 $[l_i, r_i]$，取尽量少的点，使得每个区间内都至少有一个点（不同区间内含的点可以是同一个）。

思路：考虑两个区间的情况，如果两个区间有相交区间，那么必定要在这个相交区间内取一点覆盖两个区间。如果三个区间有公共相交区间，同理要在这个相交区间内取一点覆盖三个区间。以此类推，于是我们可以维护一个相交的区间，如果下一条线段和该区间有交点，那么更新该区间为 $[max(l, l_i), min(r, r_i)]$；否则就新开一段区间等于下一条线段。为了使一段区间能覆盖尽可能多的点，同理上面的不相交区间，我们应该对所有区间按右端点 r_i 降序处理。

实际上没必要维护区间，只需要维护一个右端点就可以，我们发现排序后公共区间的左端点是不断向右收缩的，而如果有相交右端点是不会变的，于是可以只维护右端点。

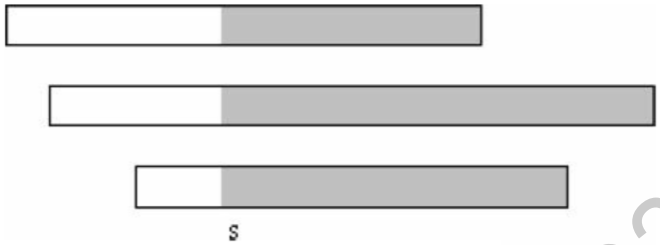


4. 区间覆盖问题

问题描述：数轴上有 n 个闭区间 $[l_i, r_i]$，选择尽量少的区间覆盖一条指定线段 $[s, t]$。

思路：先进行一次预处理，将每个区间在 $[s, t]$ 外的部分切除（因为它们的存在是毫无

意义的）。在预处理后，相互包含的区间中，较小的那个显然不应该考虑。把各区间按照左端点 l 从小到大排序。如果区间 1 的起点不是 s，无解（因为其他区间的起点更大，不可能覆盖到 s 点），否则选择起点在 s 的最长区间。选择此区间 $[l_i, r_i]$ 后，新的起点应该设置为 r_i，并且忽略所有区间在 r_i 之前的部分，就像预处理一样。之后选择剩余区间中有效长度最长的一个，如图，s 为当前有效起点（此前部分已被覆盖），则应该选择区间 2。重复以上过程直到覆盖了指定区间或者无解。



2.10.2 例题

防晒（AcWing 110）

有 C 头奶牛进行日光浴，第 i 头奶牛需要 $minSPF[i]$ 到 $maxSPF[i]$ 单位强度之间的阳光。每头奶牛在日光浴前必须涂防晒霜，防晒霜有 L 种，涂上第 i 种之后，身体接收到的阳光强度就会稳定为 $SPF[i]$，第 i 种防晒霜有 $cover[i]$ 瓶。

求最多可以满足多少头奶牛进行日光浴。

【输入格式】

第一行输入整数 C 和 L。$(1 \leq C, L \leq 2500)$

接下来的 C 行，按次序每行输入一头牛的 $minSPF$ 和 $maxSPF$ 值，即第 i 行输入 $minSPF[i]$ 和 $maxSPF[i]$。$(1 \leq minSPF, maxSPF \leq 1000)$

再接下来的 L 行，按次序每行输入一种防晒霜的 SPF 和 $cover$ 值，即第 i 行输入 $SPF[i]$ 和 $cover[i]$。$(1 \leq SPF \leq 1000)$

每行的数据之间用空格隔开。

【输出格式】

输出一个整数，代表最多可以满足奶牛日光浴的奶牛数目。

【分析】

按照$minSPF$递减的顺序把奶牛排序，依次考虑每头奶牛。

对于每头奶牛，扫描一遍所有的防晒霜，在这头奶牛能用(能用指的是该防晒霜的强度符合这头奶牛的范围，并且瓶数还有剩余)的防晒霜里找SPF值最大的使用。

以上算法的贪心策略是在满足条件的前提下每次选SPF最大的防晒霜。这个策略为什么是正确的呢?我们考虑这一步策略的作用范围扩展到后续其他奶牛之后产生的影响。每瓶防

晒霜是否可用，会被$minSPF$与 $maxSPF$两个条件限制。因为奶牛已被按照$minSPF$递减排序，所以每一个不低于当前奶牛$minSPF$值的防晒霜，都不会低于后面其他奶牛的 $minSPF$。也就是说，对于当前奶牛可用的任意两瓶防晒霜x,y，如果$SPF[x] < SPF[y]$，那么后面其他奶牛只可能出现“x,y都能用”“x,y都不能用”或者“x能用，y不能用”这三种情况之一。因此当前奶牛选择$maxSPF$较大的y去使用，对于整体问题的影响显然比选择$maxSPF$ 较小的x更好。

另外，每头奶牛对答案的贡献至多是 1。即使让当前这头奶牛放弃日光浴，留下防晒霜给后面的某一头奶牛用，对答案的贡献也不会变得更大。综上所述，尽量满足当前的奶牛，并选择SPF值尽量大的防晒霜是一个正确的贪心策略。

```
#include <algorithm>
#include <iostream>
using namespace std;
typedef long long ll;
const int maxn = 5005;

int n, m;

struct cow {
    int l, r;
} a[maxn];

struct node {
    int s, c;
} b[maxn];

bool cmp1(const cow &p, const cow &q) {
    return p.r == q.r ? p.l < q.l : p.r < q.r;
}

bool cmp2(const node &p, const node &q) {
    return p.s < q.s;
}

int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) cin >> a[i].l >> a[i].r;
    for (int i = 1; i <= m; i++) cin >> b[i].s >> b[i].c;
    sort(a + 1, a + 1 + n, cmp1);
    sort(b + 1, b + 1 + m, cmp2);
    int ans = 0;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
```

```
34.             if (b[j].s >= a[i].l && b[j].s <= a[i].r && b[j].c) {
35.                 ans++;
36.                 b[j].c--;
37.                 break;
38.             }
39.         }
40.     }
41.     cout << ans << "\n";
42.     return 0;
43. }
```

畜栏预定（AcWing 111）

有 N 头牛在畜栏中吃草。每个畜栏在同一时间段只能提供给一头牛吃草，所以可能会需要多个畜栏。

给定 N 头牛和每头牛开始吃草的时间 A 以及结束吃草的时间 B，每头牛在 $[A,B]$ 这一时间段内都会一直吃草。当两头牛的吃草区间存在交集时（包括端点），这两头牛不能被安排在同一个畜栏吃草。

求需要的最小畜栏数目和每头牛对应的畜栏方案。

【输入格式】

第 1 行：输入一个整数 $N(1 \leq N \leq 50000)$。

第 2 到 $N+1$ 行：第 $i+1$ 行输入第 i 头牛的开始吃草时间 A 以及结束吃草时间 B，数之间用空格隔开。$(1 \leq A,B \leq 100000)$。

【输出格式】

第 1 行：输出一个整数，代表所需最小畜栏数。

第 2 到 $N+1$ 行：第 $i+1$ 行输出第 i 头牛被安排到的畜栏编号，编号是从 1 开始的**连续**整数，只要方案合法即可。

【分析】

按照开始吃草的时间把牛排序。

维护一个数组S，记录当前每个畜栏安排进去的最后一头牛，最初没有畜栏。依次对于每头牛，扫描数组S，找到任意一个畜栏，满足当前的牛开始吃草的时间不早于畜栏中最后一头牛结束吃草的时间。如果这样的畜栏不存在，则为其新建一个畜栏。

这个贪心算法的时间复杂度是$O(N^2)$，请读者自行证明其正确性。我们可以用一个小根堆(STL priority_queue)维护每个畜栏最后一头牛结束吃草的时间，尝试把当前的牛安排在堆顶(结束时间最早)的畜栏中，时间复杂度可以降低到$O(N\log N)$。

```
1. #include <algorithm>
2. #include <iostream>
3. #include <queue>
4.
5. using namespace std;
```

```cpp
typedef long long ll;
const int maxn = 2e5 + 10;

struct node {
    int l, r, id;

    bool operator<(const node &p) const {
        return r >= p.r;
    }
} a[maxn];

bool cmp(node &p, node &q) {
    return p.l == q.l ? p.r < q.r : p.l < q.l;
}

priority_queue<node> q;

int ans[maxn];

int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> a[i].l >> a[i].r;
        a[i].id = i;
    }
    sort(a + 1, a + 1 + n, cmp);
    int cnt = 0;
    q.push({a[1].l, a[1].r, ++cnt});
    ans[a[1].id] = cnt;
    for (int i = 2; i <= n; i++) {
        node cur = q.top();
        if (cur.r < a[i].l) {
            q.pop();
            cur.r = a[i].r;
            ans[a[i].id] = cur.id;
            q.push(cur);
        } else {
            q.push({a[i].l, a[i].r, ++cnt});
            ans[a[i].id] = cnt;
        }
    }
    cout << cnt << "\n";
    for (int i = 1; i <= n; i++) cout << ans[i] << "\n";
```

```
    return 0;
}
```

国王游戏（AcWing 114）

恰逢 H 国国庆，国王邀请 n 位大臣来玩一个有奖游戏。

首先，他让每个大臣在左、右手上面分别写下一个整数，国王自己也在左、右手上各写一个整数。然后，让这 n 位大臣排成一排，国王站在队伍的最前面。排好队后，所有的大臣都会获得国王奖赏的若干金币，每位大臣获得的金币数分别是：排在该大臣前面的所有人的左手上的数的乘积除以他自己右手上的数，然后向下取整得到的结果。

国王不希望某一个大臣获得特别多的奖赏，所以他想请你帮他重新安排一下队伍的顺序，使得获得奖赏最多的大臣，所获奖赏尽可能的少。注意，国王的位置始终在队伍的最前面。

【输入格式】

第一行包含一个整数 $n(1 \le n \le 1000)$，表示大臣的人数。

第二行包含两个整数 a 和 b，之间用一个空格隔开，分别表示国王左手和右手上的整数。$(0 < a, b < 10000)$

接下来 n 行，每行包含两个整数 a 和 b，之间用一个空格隔开，分别表示每个大臣左手和右手上的整数。$(0 < a, b < 10000)$

【输出格式】

输出只有一行，包含一个整数，表示重新排列后的队伍中获奖赏最多的大臣所获得的金币数。

【分析】

按照每个大臣左、右手上的数的乘积从小到大排序，就是最优排队方案。这个贪心算法可以使用微扰(邻项交换)证明。

对于任意一种顺序,设n名大臣左、右手上的数分别是$A[1]\sim A[n]$与$B[1]\sim B[n]$,国王手里的数是$A[0]$和$B[0]$。

如果我们交换两个相邻的大臣i与$i+1$，在交换前这两个大臣获得的奖励是：

$$\frac{1}{B[i]} * \prod_{j=0}^{i-1} A[j] \text{与} \frac{1}{B[i+1]} * \prod_{j=0}^{i} A[j]。$$

交换之后这两个大臣获得的奖励是:

$$\frac{1}{B[i+1]} * \prod_{j=0}^{i-1} A[j] \text{与} \frac{A[i+1]}{B[i]} * \prod_{j=0}^{i} A[j]$$

其他大臣获得的奖励显然都不变，因此我们只需要比较上面两组式子最大值的变化。提取公因式$\prod_{j=0}^{i-1} A[j]$后，实际上需要比较下面两个式子的大小关系：

$$\max\left(\frac{1}{B[i]}, \frac{A[i]}{B[i+1]}\right), \max\left(\frac{1}{B[i+1]}, \frac{A[i+1]}{B[i]}\right)$$

两边同时乘上$B[i] * B[i+1]$，变为比较:

$$\max(B[i+1], A[i] * B[i]), \max(B[i], A[i+1] * B[i\ +1])$$

注意到大臣手上的数都是正整数，故$B[i+1] \le A[i+1] *\ B[i+1]$，且$A[i] * B[i] \ge B[i]$。

于是,当 $A[i]*B[i] \le A[i+1]*B[i+1]$ 时,左式≤右式,交换前更优。当$A[i]*B[j] \ge A[i+1]*B[i+1]$时，左式≥右式，交换后更优。也就是说，在任何局面下，减小逆序对数都不会造成整体结果变差，而增加逆序对数则不会使整体结果变好。

根据冒泡排序的知识，任何一个序列都能通过邻项交换的方式变为有序序列。故当逆序对数为 0，即按上述方案排序时就是最优策略。

最后需要注意本题需要使用高精度，因为代码过长省略高精度模板部分。

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>
using namespace std;
typedef long long ll;
const int maxn = 2e6 + 10;

int n;

struct node {
    int a, b;
} t[maxn];

bool cmp(node &p, node &q) {
    return p.a * p.b < q.a * q.b;
}

struct BigInt {
    //...
};

int main() {
    cin >> n;
    cin >> t[0].a >> t[0].b;
    for (int i = 1; i <= n; i++) cin >> t[i].a >> t[i].b;
    sort(t + 1, t + n + 1, cmp);
    BigInt res(t[0].a);
    BigInt ans(0);
    for (int i = 1; i <= n; i++) {
        BigInt cur = res;
        cur = cur / t[i].b;
        if (ans < cur) ans = cur;
        res = res * t[i].a;
    }
    ans.print();
    return 0;
```

```
37. }
```

2.11 二分&三分

2.11.1 二分查找

二分查找，也称折半搜索，是用来在一个有序数组中查找某一元素的算法。

二分的实现方法多种多样，但是其细节之处确实需要仔细考虑。对于整数域上的二分，需要注意终止边界、左右区间取舍时的开闭情况，避免漏掉答案或造成死循环;对于实数域上的二分，需要注意精度问题。

以使用二分在一个升序数组中查找一个数为例。它每次考察数组当前部分的中间元素，如果中间元素刚好是要找的，就结束搜索过程；如果中间元素小于所查找的值，那么左侧的只会更小，不会有所查找的元素，只需到右侧查找；如果中间元素大于所查找的值同理，只需到左侧查找。时间复杂度$O(logn)$。

- 中间位置mid除了$mid = (l + r) >> 1$，还能 $mid = l + ((r - l) >> 1)$，这样可以防止溢出。
- 循环结束的条件一般是 $l < r$，严格说法是当搜索区间为空时循环结束，两边都是闭区间那么就是 $l \leq r$。

```
1. int binary_Search(int *a, int l, int r, int v) {  // 在数组 a 区间[l,r]二分查找元素 v
2.     while (l <= r) {
3.         int mid = (l + r) >> 1;       // 取中间位置
4.         if (v == a[mid]) return mid;   // 查找成功，则返回所在位置
5.         if (v > a[mid])
6.             l = mid + 1;   // 从后半部分继续查找
7.         else
8.             r = mid - 1;   // 从前半部分继续查找
9.     }
10.    return -1;   // 元素不存在，返回-1
11. }
```

除了寻找有序序列中的元素之外，二分查找还有多个其他应用。

1. 二分查找后继：在单调递增序列a的区间$[l,r]$中找到大于等于x的最小的元素（即 x 或 x 的后继）。

```
1. int lower_bound(int *a, int l, int r, int x) {
2.     while (l < r) {
3.         int mid = (l + r) >> 1;
4.         if (a[mid] >= x)
5.             r = mid;
6.         else
```

```
            l = mid + 1;
        }
        return a[l];
    }
```

2. 二分查找前驱：在单调递增序列a的区间$[l,r]$中找到小于等于x的最大的元素（即 x 或 x 的前驱）。

```
int getPrev(int *a, int l, int r, int x) {
    while (l < r) {
        int mid = (l + r + 1) >> 1;
        if (a[mid] <= x)
            l = mid;
        else
            r = mid - 1;
    }
    return a[l];
}
```

在查找后继的代码中，若$a[mid] \geq x$，则根据序列 a 的单调性，mid 之后的数会更大，所以$\geq x$的最小的数不可能在 mid 之后,可行区间应该缩小为左半段。因为mid也可能是答案，故此时应取$r = mid$。同理，若$a[mid] < x$，取$l = mid + 1$。

在查找前驱的代码中，若$a[mid] \leq x$，则根据序列 a 的单调性，mid 之前的数会更小，所以$\leq x$的最大的数不可能在mid之前，可行区间应该缩小为右半段。因为mid也可能是答案，故此时应取$l = mid$。同理，若$a[mid] > x$，取$r = mid - 1$。

如上面两段代码所示，这种二分写法可能会有两种形式:

1) 缩小范围时，$r = mid$，$l = mid + 1$，取中间值时，$mid = (l + r) >> 1$。

2) 缩小范围时，$l = mid$，$r = mid - 1$，取中间值时，$mid = (l + r + 1) >> 1$。

如果不对mid的取法加以区分，假如第二段代码也采用$mid = (l + r) >> 1$，那么当$r - l$等于1时，就有$mid = (l + r) >> 1 = l$。接下来若进入$l = mid$分支，可行区间未缩小，造成死循环；若进入$r = mid - 1$分支，造成$l > r$，循环不能以$l = r$结束。因此对两个形式采用配套的mid取法是必要的。上面两段代码所示的两个形式共同组成了这种二分的实现方法。注意，我们在二分实现中采用了右移运算>> 1,而不是整数除法/2。这是因为右移运算是向下取整，而整数除法是向零取整，在二分值域包含负数时后者不能正常工作。

仔细分析这两种mid的取法，我们还发现：$mid = (l + r) >> 1$不会取到r这个值，$mid = (l + r + 1) >> 1$不会取到l这个值。我们可以利用这一性质来处理无解的情况，把最初的二分区间$[1,n]$分别扩大为$[1,n + 1]$和$[0,n]$，把a数组的一个越界的下标包含进来。如果最后二分终止于扩大后的这个越界下标上，则说明a中不存在所求的数。

总而言之，正确写出这种二分的流程是:

1. 通过分析具体问题，确定左右半段哪一个是可行区间，以及mid归属哪一半段。
2. 根据分析结果,选择“$r = mid, l = mid + 1, mid = (l + r) >> 1$”和“$l = mid, r =$

$mid-1$，$mid=(l+r+1)>>1$”两种形式之一。

3. 二分终止条件是$l==r$，该值就是答案所在位置。

使用的这种二分方法的优点是始终保持答案位于二分区间内，二分结束条件对应的值恰好在答案所处位置，还可以很自然地处理无解的情况，形式优美。唯一的缺点是由两种形式共同组成，需要认真考虑实际问题选择对应的形式。

也有其他的二分写法，采用“$l=mid+1,r=mid-1$，终止条件为$l>r$”或“$l=mid,r=mid$”来避免产生两种形式，但也相应地造成了丢失在mid点上的答案、二分结束时可行区间未缩小到确切答案等问题，需要额外加以处理。无论是那种形式，熟悉一种并使用基本不会出现大的问题。

C++ STL 中的 lower_bound 与 upper_bound 函数实现了在一个序列中二分查找某个整数x的后继，具体用法如下。

```
1. int idx1 = lower_bound(a, a + n, x) - a;  // 在范围[0, n)的数组 a 找到大于等于 x 的第一个下标
2. int idx2 = upper_bound(a + 1, a + n + 1, x) - a;  // 在范围[1, n]的数组 a 找到大于 x 的第一个下标
```

在实数域上二分较为简单，确定好所需的精度eps，以$l+eps<r$为循环条件，每次根据在 mid 上的判定选择$r=mid$或$l=mid$ 分支之一即可。一般需要保留k位小数时，则取$eps=10^{-(k+2)}$。

有时精度不容易确定或表示，就干脆采用循环固定次数的二分方法，也是一种相当不错的策略。这种方法得到的结果的精度通常比设置eps更高。

```
 1. const double eps = 1e-8;
 2.
 3. // 写法一
 4. while (l + eps < r) {
 5.     double mid = (l + r) / 2;
 6.     if (check(mid))
 7.         l = mid;
 8.     else
 9.         r = mid;
10. }
11.
12. // 写法二
13. for (int i = 0; i < 100; i++) {
14.     if (check(mid))
15.         l = mid;
16.     else
17.         r = mid;
18. }
```

2.11.2 二分答案

一个宏观的最优化问题也可以抽象为函数，其“定义域”是该问题下的可行方案，对这些可行方案进行评估得到的数值构成函数的“值域”，最优解就是评估值最优的方案(不妨设评分越高越优)。假设最优解的评分是S，显然对于$\forall x > S$，都不存在一个合法的方案达到x分，否则就与S的最优性矛盾;而对于$\forall x \leq S$，一定存在一个合法的方案达到或超过x分，因为最优解就满足这个条件。这样问题的值域就具有一种特殊的单调性一在 S 的一侧合法、在 S 的另一侧不合法，就像一个在$(-\infty, S]$上值为 1，在$(S, +\infty)$上值为 0 的分段函数，可通过二分找到这个分界点 S。借助二分，我们把求最优解的问题，转化为给定一个值mid，判定是否存在一个可行方案评分达到mid的问题。

木材加工（洛谷 P2440）

木材厂有 n 根原木，现在想把这些木头切割成 k 段长度均为 l 的小段木头（木头有可能有剩余）。当然，我们希望得到的小段木头越长越好，请求出 l 的最大值。

木头长度的单位是 cm，原木的长度都是正整数，我们要求切割得到的小段木头的长度也是正整数。例如有两根原木长度分别为 11 和 21，要求切割成等长的 6 段，很明显能切割出来的小段木头长度最长为 5。

【输入格式】

第一行是两个正整数 $n, k(1 \leq n \leq 10^5, 1 \leq k \leq 10^8)$，分别表示原木的数量，需要得到的小段的数量。

接下来 n 行，每行一个正整数 $L_i(1 \leq L_i \leq 10^8)$，表示一根原木的长度。

【输出格式】

仅一行，即 l 的最大值。如果连 1cm 长的小段都切不出来，输出 0。

【分析】

很典型的二分答案题目。题目要求把木材切割成很多段，使得这个短段尽量的长。我们切割的段肯定是在$[0, \max\{L_i\}]$中，满足二分的有界性。由题意显然单调，满足单调性。所以可以使用二分答案求解。

二分的 check 函数很好实现。以当前的答案x为标准去切割这些木材。枚举所有木材，对于每一段木材其能分割的段数最大是L_i / x，设一个累加器cnt记录这个值，将cnt与k比较判断答案是否可行。如果发现能切够k段则去右半部分找更大的解，如果发现切不够k段就去左边找可行解。

```
1. #include <iostream>
2. using namespace std;
3. typedef long long ll;
4. const int maxn = 1e5 + 10;
5.
6. int n, k;
```

```
int a[maxn];

bool check(int x) {
    ll ans = 0;
    for (int i = 1; i <= n; i++) {
        ans += a[i] / x;
    }
    return ans >= k;
}

int main() {
    cin >> n >> k;
    for (int i = 1; i <= n; i++) cin >> a[i];
    int l = 1, r = 1e9, ans;
    while (l <= r) {
        int mid = (l + r) >> 1;
        if (check(mid))
            ans = mid, l = mid + 1;
        else
            r = mid - 1;
    }
    if (check(ans))
        cout << ans << endl;
    else
        cout << 0 << endl;
    return 0;
}
```


最佳牛围栏（AcWing 102）

农夫约翰的农场由 N 块田地组成，每块地里都有一定数量的牛，其数量不会少于 1 头，也不会超过 2000 头。

约翰希望用围栏将一部分连续的田地围起来，并使得围起来的区域内每块地包含的牛的数量的平均值达到最大。围起区域内至少需要包含 F 块地，其中 F 会在输入中给出。

在给定条件下，计算围起区域内每块地包含的牛的数量的平均值可能的最大值是多少。

【输入格式】

第一行输入整数 $N(1 \leq N \leq 100000)$ 和 $F(1 \leq F \leq N)$，数据间用空格隔开。

接下来 N 行，每行输入一个整数，第 $i+1$ 行输入的整数代表第 i 片区域内包含的牛的数目。

【输出格式】

输出一个整数，表示平均值的最大值乘以 1000 再向下取整之后得到的结果。

【分析】

二分答案，判定“是否存在一个长度不小于L的子段,平均数不小于二分的值”。如果把

数列中每个数都减去二分的值，就转化为判定“是否存在一个长度不小于L的子段，子段和非负”。下面我们着重来解决以下两个问题：

1. 求一个子段，它的和最大，没有“长度不小于L”这个限制。无长度限制的最大子段和问题是一个经典问题，只需$O(n)$扫描该数列，不断把新的数加入子段，当子段和变成负数时，把当前的整个子段清空。扫描过程中出现过的最大子段和即为所求。

2. 求一个子段，它的和最大，子段的长度不小于L。子段和可以转化为前缀和相减的形式，即设 sum_i表示$A_1 \sim A_i$的和，则有：

$$\max_{i-j\geq L}\{A_{j+1}+A_{j+2}+\cdots+A_i\} = \max_{L\leq i\leq n}\{sum_i - \min_{0\leq j\leq i-L}\{sum_j\}\}$$

仔细观察上面的式子可以发现，随着i的增长，j的取值范围$0 \sim i-L$每次只会增大 1。换言之，每次只会有一个新的取值进入 $\min\{sum_j\}$ 的候选集合，所以我们没有必要每次循环枚举j，只需要用一个变量记录当前最小值，每次与新的取值 sum_{i-L}取min就可以了。

解决了问题 2 之后，我们只需要看一下最大子段和是不是非负数，就可以确定二分上下界的变化范围了。

```
#include <iostream>
using namespace std;
const double eps = 1e-6;
const int maxn = 2e5 + 10;

int n, f;
int a[maxn];
double b[maxn];

bool check(double x) {
    for (int i = 1; i <= n; i++) b[i] = b[i - 1] + a[i] - x;
    double sum = 1e9, ans = -1e9;
    for (int i = f; i <= n; i++) {
        sum = min(sum, b[i - f]);
        ans = max(ans, b[i] - sum);
    }
    return ans >= 0;
}

int main() {
    cin >> n >> f;
    for (int i = 1; i <= n; i++) cin >> a[i];
    double l = 1e-6, r = 1e9;
    while (r - l > eps) {
        double mid = (l + r) / 2.0;
        if (check(mid))
            l = mid;
        else
```

```
            r = mid;
    }
    int ans = (l + eps) * 1000;
    printf("%d\n", ans);
    return 0;
}
```

2.11.3 三分

有一类函数被称为单峰函数，它们拥有唯一的极大值点，在极大值点左侧严格单调上升，右侧严格单调下降；或者拥有唯一的极小值点，在极小值点左侧严格单调下降，在极小值点右侧严格单调上升。为了避免混淆，我们也称后一种为单谷函数。对于单峰或单谷函数，我们可以通过三分法求其极值。

以单峰函数f为例，我们在函数定义域$[l,r]$上任取两个点$lmid$与$rmid$，把函数分为三段。

1. 若$f(lmid) < f(rmid)$，则$lmid$与$rmid$要么同时处于极大值点左侧(单调上升函数段)，要么处于极大值点两侧。无论哪种情况下，极大值点都在$lmid$右侧，可令$l = lmid$。
2. 同理，若$f(lmid) > f(rmid)$,则极大值点一定在 $rmid$ 左侧,可令$r = rmid$。

如果我们取$lmid$与$rmid$为三等分点，那么定义域范围每次缩小 1/3。如果我们取$lmid$与$rmid$ 在二等分点两侧极其接近的地方，那么定义域范围每次近似缩小 1/2。通过 log 级别的时间复杂度即可在指定精度下求出极值。这就是三分法。

注意到在介绍单峰函数时特别强调了“严格”单调性。若在三分过程中遇到$f(lmid) = f(rmid)$，当函数严格单调时，令$l = lmid$ 或$r = rmid$ 均可。如果函数不严格单调，即在函数中存在一段值相等的部分，那么无法判断定义域的左右边界如何缩小，三分法就不再适用。

确定三等分点$lmid$和$rmid$主要有两种不同的方法：

1) 确定区间中点作为左中点，接着确定右半区间的中点作为右中点

```
int midl = (l + r) / 2;
int midr = (midl + r) / 2;
```

2) 找到这段区间的三分之一长度，接着直接确定左右中点：

```
int midl = l + (r - l) / 3;
```

```
2. int midr = r - (r - l) / 3;
```

Strange fuction（HDU 2899）

现在有方程$f(x) = 6 * x^7 + 8 * x^6 + 7 * x^3 + 5 * x^2 - k * x(0 \leq x \leq 100)$，每次给出$k$，求该函数在[0,100]的最小值。

【输入格式】

第一行输入正整数 $T(1 \leq T \leq 100)$ 代表测试用例的数目。

接下来 T 行，每行输入一个整数$k(k \leq 10^{10})$。

【输出格式】

输出函数在值域[0,100]的最小值，四舍五入精确到小数点后四位。

【分析】

首先不难发现，在第一象限该函数一定是一个凹函数图像，那么很轻松就想到用三分去写。实际上还可以二分，对上述函数求导后，其导函数在第一象限一定是一个单调递增的直线，二分就是找导函数值为 0 的x。

```
1. #include <cmath>
2. #include <iostream>
3. using namespace std;
4. const double eps = 1e-8;
5. const int maxn = 2e5 + 10;
6.
7. double f(double x, double y) {
8.     return 6.0 * pow(x, 7.0) + 8.0 * pow(x, 6.0) + 7.0 * pow(x, 3.0) + 5.0 * pow(x,
2.0) - y * x;
9. }
10.
11. int dcmp(double d) {
12.     if (fabs(d) < eps) return 0;
13.     return d > 0 ? 1 : -1;
14. }
15.
16. double tri_search(double k) {
17.     double l = 0, r = 100.0, midl, midr;
18.     while (r - l > eps) {
19.         midl = l + (r - l) / 3, midr = r - (r - l) / 3;
20.         if (dcmp(f(midl, k) - f(midr, k)) < 0) {
21.             r = midr;
22.         } else
23.             l = midl;
24.     }
25.     return f(midl, k);
26. }
27.
```

```
28. int main() {
29.     int T;
30.     double y;
31.     scanf("%d", &T);
32.     while (T--) {
33.         scanf("%lf", &y);
34.         printf("%.4lf\n", tri_search(y));
35.     }
36.     return 0;
37. }
```

2.12 倍增

倍增,字面意思就是“成倍增长”。这是指我们在进行递推时,如果状态空间很大，通常的线性递推无法满足时间与空间复杂度的要求，那么我们可以通过成倍增长的方式，只递推状态空间中在 2 的整数次幂位置上的值作为代表。当需要其他位置上的值时，我们通过“**任意整数可以表示成若干个 2 的次幂项的和**”这一性质，使用之前求出的代表值拼成所需的值。所以使用倍增算法也要求我们递推的问题的状态空间关于 2 的次幂具有可划分性。

“倍增”与“二进制划分”两个思想互相结合，降低了求解很多问题的时间与空间复杂度。数论基础中的快速幂其实就是“倍增”与“二进制划分”思想的一种体现。在本节中，主要研究序列上的倍增问题，包括求解 RMQ（区间最值）问题的 ST 算法。关于求解最近公共祖先（LCA）等在树上的倍增应用，我们将在数据结构章节中进行探讨。

假设我们需要去解决这样一个问题：给定一个长度为N的数列 A，然后进行q次询问，每次给定一个整数T，求出最大的k，满足$\sum_{i=1}^{k} A[i] \le T$。你的算法必须是在线的（必须即时回答每一个询问，不能等待收到所有询问后再统一处理），假设$0 \le T \le \sum_{i=1}^{N} A[i]$。

最朴素的做法当然是从前向后枚举k，每次询问花费的时间与答案的大小有关，最坏情况下为 $O(q * N)$。

如果我们能够先花费$O(N)$的时间预处理A数组的前缀和数组S，就可以二分k的位置，比较$S[k]$与T的大小来确定二分上下界的变化，每次询问花费的时间都是$O(logn)$，总的时间复杂度$O(q * logn)$可以接受。这个算法在平均情况下表现很好，但是它的缺点是如果每次询问给定的整数T都非常小，造成答案k也非常小，那么该算法可能还不如从前向后枚举更优。

我们可以设计这样一种倍增算法:

1. 令$p = 1, k = 0, sum = 0$。
2. 比较“A数组中k之后的p个数的和”与T的关系，也就是说，如果$sum + S[k + p] - S[k] \le T$，则令$sum \mathrel{+}= S[k + p] - S[k], k \mathrel{+}= p, p \mathrel{*}= 2$，即累加上这$p$个数的和，然后把$p$的跨度增长一倍。如果$sum + S[k + p] - S[k] > T$，则令$p \mathrel{/}= 2$。
3. 重复上一步，直到p的值变为 0，此时k就是答案。

这个算法始终在答案大小的范围内实施“倍增”与“二进制划分”思想，通过若干长度为 2 的次幂的区间拼成最后的k，时间复杂度级别为答案的对数，能够应对T的各种大小情况。

天才 ACM（AcWing 109）

给定一个整数 M，对于任意一个整数集合 S，定义“校验值”如下：从集合 S 中取出 M 对数(即 $2\times M$ 个数，不能重复使用集合中的数，如果 S 中的整数不够 M 对，则取到不能取为止)，使得“每对数的差的平方”之和最大，这个最大值就称为集合 S 的“校验值”。

现在给定一个长度为 N 的数列 A 以及一个整数 T。我们要把 A 分成若干段，使得每一段的“校验值”都不超过 T。求最少需要分成几段。

【输入格式】

第一行输入整数 $K(1\le K\le 12)$，代表有 K 组测试数据。

对于每组测试数据，输入的第一行包含三个整数 $N,M,T(1\le N,M\le 500000,0\le T\le 10^{18})$ 。第二行包含 N 个整数，表示数列$A_1,A_2\dots A_N(0\le A_i\le 2^{20})$。

【输出格式】

对于每组测试数据，输出其答案，每个答案占一行。

【分析】

根据贪心，从第一个数开始能向右找的数作为第一段越多越好，然后以此类推。

对于一个子区间 a，如何求出校验值？每对数差的平方和最大，展开平方和公式，得到若 a_i*a_j 最小则差的平方和最大，联系排序不等式，每次取出最大值和最小值，累加平方和即可。

求出一段区间的校验值最少需要 $O(n\log n)$ 的复杂度。显然我们可以对每个起点的位置，在后缀中二分答案每次二分 $O(m\log m)$ （m为起点到二分的 mid 的距离）检查答案。看似可以通过本题，但是只是平均的情况下可以通过。考虑最坏的情况 $T=0$，那么每次都要检查 log 次，这样的时间复杂度是接近 $O(n^2\log n)$ 的。

采用如下的方式倍增：

1. 初始化 $p=1,R=L$

2. 求出 $[L,R+p]$ 这一段的校验值，若小于等于 T，则 $R+=p,p=p*2$；否则 $p=p/2$。

3. 重复上一步，直到 p 的值变为 0，此时的 R 即为所求。

使用倍增可以达到我们期望用二分达到的时间复杂度 $O(n\log^2 n)$。

此算法还可以优化，假设我们上次已经对区间 $[L,R]$ 排序，接下来要检测的新的一段是 $[R+1,R+p]$，那么可以只对新增部分排序然后归并，这样可以使总体时间复杂度降低到 $O(n\log n)$，可以完美通过本题。

```
1. #include <algorithm>
2. #include <iostream>
3. using namespace std;
```

```cpp
typedef long long ll;
const int maxn = 5e5 + 10;

int T, n, m;
int a[maxn], b[maxn], c[maxn];
ll val;

void merge(int l1, int r1, int l2, int r2) {
    int i = l1, j = l2, k = 0;
    while (i <= r1 && j <= r2) {
        if (a[i] <= b[j])
            c[k++] = a[i++];
        else
            c[k++] = b[j++];
    }
    while (i <= r1) c[k++] = a[i++];
    while (j <= r2) c[k++] = b[j++];
}

bool check(int l1, int r1, int l2, int r2) {
    int len = r2 - l2 + 1;
    for (int i = l2; i <= r2; i++) b[i - l2] = a[i];
    sort(b, b + len);
    merge(l1, r1, 0, len - 1);
    int i = 0, j = r2 - l1, k = 0;
    ll ans = 0;
    while (i < j && k < m) {
        ans += 1LL * (c[j] - c[i]) * (c[j] - c[i]);
        i++, j--, k++;
    }
    return ans <= val;
}

int main() {
    scanf("%d", &T);
    while (T--) {
        scanf("%d%d%lld", &n, &m, &val);
        for (int i = 1; i <= n; i++) scanf("%d", &a[i]);
        int ans = 0;
        for (int i = 1; i <= n; i++) {
            int p = 1, j = i;
            while (p) {
                if (j + p <= n && check(i, j, j + 1, j + p)) {
                    j += p;
```

```
48.                     for (int k = i; k <= j; k++) a[k] = c[k - i];
49.                     p <<= 1;
50.                 } else
51.                     p >>= 1;
52.             }
53.             i = j;
54.             ans++;
55.         }
56.         printf("%d\n", ans);
57.     }
58.     return 0;
59. }
```

ST 表（洛谷 P3865）

给定一个长度为 N 的数列，和 M 次询问，求出每一次询问的区间内数字的最大值。

【输入格式】

第一行包含两个整数 N,M，分别表示数列的长度和询问的个数。

第二行包含 N 个整数（记为 a_i，依次表示数列的第 i 项。

接下来 M 行，每行包含两个整数 l_i,r_i，表示查询的区间为 $[l_i,r_i]$。

【输出格式】

输出包含 M 行，每行一个整数，依次表示每一次询问的结果。

【分析】

本题属于 RMQ 问题(区间最值问题)。一个序列的子区间个数显然有$O(N^2)$个，根据倍增思想，我们首先在这个规模为$O(N^2)$的状态空间里选择一些 2 的整数次幂的位置作为代表值。

设$F[i,j]$表示数列 A 中下标在子区间$[i,i+2^j-1]$里的数的最大值，也就是从i开始的2^j个数的最大值。递推边界显然是$F[i,0]=A[i]$，即数列 A 在子区间$[i,i]$里的最大值。

在递推时，我们把子区间的长度成倍增长，有公式$F[i,j]=\max(F[i,j-1],F[i+2^j-1,j-1])$，即长度为 2 的子区间的最大值是左右两半长度为$2^j-1$的子区间的最大值中较大的一个。

当询问任意区间$[l,r]$的最值时，我们先计算出一个k，满足$2^k<r-l+1\le 2^{k+1}$，也就是使 2 的k次幂小于区间长度的前提下最大的k。那么“从l开始的2^k个数”和“以r结尾的2^k个数”这两段一定覆盖了整个区间$[l,r]$，这两段的最大值分别是$F[l,k]$和$F[r-2^k+1,k]$，二者中较大的那个就是整个区间$[l,r]$的最值。因为求的是最大值，所以这两段只要覆盖区间$[l,r]$即可，即使有重叠也没关系。

```
1. #include <cmath>
2. #include <iostream>
3. using namespace std;
4. const int maxn = 1e5 + 10;
```

```
 5.
 6. int n, m;
 7. int a[maxn], Log[maxn], d[maxn][20];
 8.
 9. void init() {
10.     for (int i = 2; i <= n; i++) Log[i] = Log[i / 2] + 1;
11.
12.     for (int i = 1; i <= n; i++) d[i][0] = a[i];
13.
14.     for (int j = 1; (1 << j) <= n; j++) {
15.         for (int i = 1; i + (1 << j - 1) <= n; i++) {
16.             d[i][j] = max(d[i][j - 1], d[i + (1 << j - 1)][j - 1]);
17.         }
18.     }
19. }
20.
21. int main() {
22.     int l, r;
23.     scanf("%d%d", &n, &m);
24.     for (int i = 1; i <= n; i++) scanf("%d", &a[i]);
25.     init();
26.     while (m--) {
27.         scanf("%d%d", &l, &r);
28.         int k = Log[r - l + 1];
29.         printf("%d\n", max(d[l][k], d[r - (1 << k) + 1][k]));
30.     }
31.     return 0;
32. }
```


2.13 双指针

双指针（又称为：尺取法、two pointers），是算法竞赛中一个常用的优化技巧，用来解决序列的区间问题。如果区间是单调的，也常常用二分法来求解，所以很多问题用尺取法和二分法都可以。

双指针是一种简单而又灵活的技巧和思想，单独使用可以轻松解决一些特定问题，和其他算法结合也能发挥多样的用处。双指针顾名思义，就是同时使用两个指针，在序列、链表结构上指向的是位置，在树、图结构中指向的是节点，通过或同向移动，或相向移动来维护、统计信息。

Floyd 判环法，又称龟兔赛跑算法，是一个可以在有限状态机、迭代函数或者链表上判断是否存在环，以及判断环的起点与长度的算法。其核心思想就是双指针，算法原理十分简洁：首先两个指针都指向链表的头部，令一个指针一次走一步，另一个指针一次走两步，如

果它们相遇了，证明有环，否则无环，时间复杂度 $O(n)$。

广义的来讲，在序列中使用两个指针枚举下标求解问题的形式，都算是双指针。但实际竞赛上双指针的应用大都在于根据“双指针”的单调性，将原本更高时间复杂度的算法优化为线性$O(n)$。常见的双指针题目类型有：

1. 在原序列的基础上，找到一个区间，使得这个区间满足一个性质，且这个性质在维护的过程是单调的。
2. 找到若干个数，使得这些数满足一个性质，且他们的最大值最小值之差最小（这种题目要排序后体现单调性）。

Subsequence（POJ 3061）

给出了一个由N个正整数$(10 < N < 100\,000)$，每个都小于或等于 10000 的序列，以及一个正整数$S(S < 100\,000\,000)$。写一个程序，找出序列中连续元素的子序列的最小长度，其总和大于或等于S。

【输入格式】

第一行是测试案例的数量。

对于每个测试用例，程序必须从第一行读出数字N和S，用一个空格隔开。序列的数字在测试案例的第二行给出，用空格隔开。

【输出格式】

对于每个案例，程序必须在输出文件的不同行中打印结果。

【分析】

如果暴力寻找所有子区间，复杂度太高。

本题属于双指针最经典的问题，在正整数序列中找到一个区间，使得区间的和满足大于等于某个数。考虑设置两个指针$l,r(l \le r)$，若此时区间(l,r)满足了题目的限制，当r右移时，不难发现如果要求出题目的最优解，l的位置一定不会再向左移动，满足了“两个指针”同向移动的单调性，这时l,r的移动收缩，看起来像是一个序列上的“滑动窗口”。

```
#include <iostream>
using namespace std;
const int maxn = 1e5 + 10;
int a[maxn];

int main() {
    int t, n, s;
    scanf("%d", &t);
    while (t--) {
        scanf("%d%d", &n, &s);
        for (int i = 0; i < n; i++) cin >> a[i];
        int l = 0, r = 0, ans = 1e9, sum = 0;
        while (l < n) {
            while (r < n && sum < s) sum += a[r++];
```

```
            if (sum < s) break;
            ans = min(ans, r - l);
            sum -= a[l++];
        }
        printf("%d\n", (ans == 1e9 ? 0 : ans));
    }
    return 0;
}
```

A-B 数对（洛谷 P1102）

给出一串正整数数列以及一个正整数 C，要求计算出所有满足 $A-B=C$ 的数对的个数（不同位置的数字一样的数对算不同的数对）。

【输入格式】

输入共两行。

第一行，两个正整数 $N,C(1\le N\le 2\times 10^5,1\le C<2^{30})$。

第二行，N 个正整数，作为要求处理的那串整数，每个数的范围是$[0,2^{30}]$。

【输出格式】

一行，表示该串正整数中包含的满足 $A-B=C$ 的数对的个数。

【分析】

本题有一些使用 map 等较为简单的解法，但我们考虑双指针的解法。

我们考虑题目要求求出所有$A-B=C$的数对，我们可以先将原数组排序，然后就会发现每个数A，对应的数B一定是一段连续的区间。

然后我们考虑如何去找到这个区间。显然是要找到这个连续区间的左端点和右端点，考虑到排序之后序列的有序性，我们枚举每个数，他们的左端点和右端点都是单调不降的，因此我们可以用双指针来维护这个东西。

```
#include <algorithm>
#include <iostream>
using namespace std;
typedef long long ll;
const int maxn = 2e5 + 10;

int a[maxn];

int main() {
    int n, k;
    cin >> n >> k;
    for (int i = 1; i <= n; i++) cin >> a[i];
    sort(a + 1, a + 1 + n);
    int l = 1, r1 = 1, r2 = 1;
    ll ans = 0;
    while (l <= n) {
```

```
        while (r1 <= n && a[r1] - a[l] < k) ++r1;
        while (r2 <= n && a[r2] - a[l] <= k) ++r2;
        if (a[r1] - a[l] < k) break;
        ans += r2 - r1;
        l++;
    }
    cout << ans << "\n";
    return 0;
}
```

逛画展（洛谷 P1638）

博览馆正在展出由世上最佳的 m 位画家所画的图画。

游客在购买门票时必须说明两个数字，a 和 b，代表他要看展览中的第 a 幅至第 b 幅画（包含 a,b）之间的所有图画，而门票的价钱就是一张图画一元。

Sept 希望入场后可以看到所有名师的图画。当然，他想最小化购买门票的价格。请求出他购买门票时应选择的 a,b，数据保证一定有解。若存在多组解，**输出 a 最小的那组**。

【输入格式】

第一行两个整数 $n,m(1 \le n \le 10^6)$，分别表示博览馆内的图画总数及这些图画是由多少位名师的画所绘画的。

第二行包含 n 个整数 $a_i(1 \le a_i \le m \le 2000$，代表画第 i 幅画的名师的编号。

【输出格式】

一行两个整数 a,b。

【分析】

题目要求，找到一个区间使得区间包含该序列所有种类的数。考虑暴力：枚举右端点r，贪心找到一个最优的左端点l使得区间$[l,r]$符合题目要求，判断区间长度是否更优，更新答案。但是这个复杂度太高，下面尝试使用双指针将这个暴力优化。

若区间$[l,r]$满足了题目的限制，当r右移时，不难发现如果要求出题目的最优解，l的位置一定不会再向左移动，满足了“双指针”的单调性。观察发现大师编号最多只有 2000 个，那么设置一个数组cnt记录当前两指针区间内每种数出现了多少次，当每种数都出现了至少一次时才符合题目要求。于是使用一个变量sum记录有多少种数。当两个指针移动时，更新cnt和sum，$sum = m$时更新答案。

```
#include <iostream>

using namespace std;
const int maxn = 1e6 + 10;

int a[maxn], cnt[2021];

int main() {
    int n, m;
```

```
10.     scanf("%d%d", &n, &m);
11.     for (int i = 1; i <= n; i++) scanf("%d", &a[i]);
12.     int l = 1, r = 0;
13.     int sum = 0, L, R, ans = 1e9;
14.     while (l <= n) {
15.         while (r < n && sum < m) {
16.             if (!cnt[a[r + 1]]) sum++;
17.             cnt[a[++r]]++;
18.         }
19.         if (sum == m && r - l + 1 < ans) {
20.             ans = r - l + 1;
21.             L = l, R = r;
22.         }
23.         if (--cnt[a[l++]] == 0) sum--;
24.     }
25.     printf("%d %d\n", L, R);
26.     return 0;
27. }
```

2.14 对拍

本节介绍随机数据的生成方法与对拍测试方法。本节介绍一种 C++随机数产生器，根据题目要求构造各种规模的输入数据，用于对自己编写的程序进行检测。同时，读者也将学习编写简单的脚本，自动化、批量化运行“数据生成程序”和两份不同的“问题求解程序”，并对两份程序的输出结果进行比对我们把这种过程称为“对拍”。

随机数据生成与对拍可用于以下场景:

（1）在无法获得实时评测反馈的比赛中，思考并实现了一个“高分解法”，但实在不会证明自己的结论，或者不能确保自己编写的程序是否完全正确。

这种情况下，建议读者酌情分配一些时间，额外编写一份随机数据生成程序、一份用朴素算法求解的程序（通常朴素解法时间复杂度高，但实现简单，不易出错）。然后把“高分解法”与“朴素解法”在小数据范围内进行对拍，看二者的输出结果是否始终保持一致。

（2）在平时解题时，自己编写的程序无法在 Online Judge 上取得 Accepted 结果，调试很久仍未发现错误，并且不能下载到题目的测试数据，或者虽然能下载到测试数据,但发生错误的数据规模过大，不容易进行调试。

这时，可以编写一个随机数据生成程序，再编写一个使用朴素算法的程序（或者直接在网络上搜索其他人的 AC 程序），与自己的“错误解法”对拍。我们可以适当调整随机数据的规模，控制在易于人工演算和调试的范围内。虽然数据越小，出错概率越低，但是“对拍”脚本能够批量化执行，在成千上万次检测中，一般总能找到一个造成错误的小规模数据。

（3）有一个不错的构思，自己出了一道题目。

此时当然需要生成一些测试数据，并且需要用“对拍”来检测自己编写的“标准程序”的正确性。不过，除了随机数据外，通常还需要增加一些特殊构造的数据，保证测试数据的全面性。

2.14.1 随机数据生成

头文件 cstdlib(stdlib.h)包含 rand 和 srand 两个函数，以及 RAND_MAX 常量。RAND_MAX 是一个不小于 32767 的整数常量，它的值与操作系统环境、编译器环境有关。一般来说，在 Windows 系统中为 32767，在类 Unix 系统中为 2147483647。

rand 函数返回一个 0~RAND_MAX 之间的随机整数 int。

srand(seed)函数接受 unsigned int 类型的参数 seed，以 seed 为“随机种子”。rand 函数基于线性同余递推式生成随机数，“随机种子”相当于计算线性同余时的一个初始参数，感兴趣的读者可以查阅相关资料。若不执行 srand 函数，则种子默认为 1。

当种子确定后，接下来产生的随机数列就是固定的，所以这种随机方法也被称为“伪随机”。因此，一般在随机数据生成程序 main 函数的开头，用当前系统时间作为随机种子。

头文件 ctime(time.h)包含 time 函数，调用 time(0)可以返回从 1970 年 1 月 1 日 0 时 0 分 0 秒(Unix 纪元)到现在的秒数。执行 srand((unsigned)time(0)) 即可初始化随机种子。

有了 rand 函数，随机生成整数序列将十分方便；如果要生成实数，可以先生成一个较大的整数，然后除以 10 的次幂。下面的程序可作为随机数据生成器的模板。

```
1. int random(int n) {
2.     return (long long)rand() * rand() % n;
3. }
```

如果需要更多的要求，比如生成 long long 范围内的整数。可以按照字符数组等方式生成整数，可以自己实现自定义随机数模板方便使用。

n个点$n-1$条边的连通无向图称为树，这里提供了生成一棵带权树的简单实现。在后续学习过并查集之后，使用并查集也可以方便的生成一棵树。

```
1. // 随机生成一棵带权树
2. void randTree(int val_range) {
3.     for (int i = 2; i <= n; i++) {
4.         // 从 2~n 之间的每个点向 1~i-1 之间的点随机连一条边
5.         int fa = random(i - 1) + 1;
6.         int val = random(val_range) + 1;
7.         printf("%d %d %d", fa, i, val);
8.     }
9. }
```

随机生成一张 n 个点 m 条边的无向图，图中不存在重边、自环，且必须连通，保证$5 \leq n \leq m \leq n*(n-1)/4 \leq 10^6$

```cpp
pair<int, int> e[1000005];     // 保存数据
map<pair<int, int>, bool> h;   // 防止重边

void randGraph(int n, int m) {
    // 先生成一棵树，保证连通
    for (int i = 1; i < n; i++) {
        int fa = random(i) + 1;
        e[i] = make_pair(fa, i + 1);
        h[e[i]] = h[make_pair(i + 1, fa)] = 1;
    }
    // 再生成剩余的 m-n+1 条边
    for (int i = n; i <= m; i++) {
        int x, y;
        do {
            x = random(n) + 1;
            y = random(n) + 1;
        } while (x == y || h[make_pair(x, y)]);
        e[i] = make_pair(x, y);
        h[e[i]] = h[make_pair(y, x)] = 1;
    }
    // 随机打乱并输出
    random_shuffle(e + 1, e + m + 1);
    for (int i = 1; i <= m; i++) printf("%d %d\n", e[i].first, e[i].second);
}
```


在树、图结构中，有三类特殊数据常用于对程序进行极端状况下的效率测试:

（1）链形数据——有很长的直径。就是把N个节点用$N-1$条边连成一条长度为$N-1$的“链”。这种数据会造成很大的递归深度，也是点分治等算法需要特别注意的数据。

（2）菊花形数据——有度数很大的节点。以 1 号节点为中心，2~N号节点都用一条边与 1 号节点相连，最终 1 号节点的度数为$N-1$。这种数据画出来形似一朵“菊花”，缩点等图论算法若处理不当，复杂度容易在这种数据上退化。

（3）蒲公英形数据。即链形和菊花形数据的综合。令树的一部分构成链，一部分构成菊花，再把两部分连接。

在以上三种数据的基础上，再添加少量随机的边，即可得到一张既包含局部特殊结构、又不失一般性和多样性的图。课下自己尝试生成相关的数据。

2.14.2 对拍

假设我们已经编写好了三个程序:

1. 自己编写的“正解”，即准备提交评测的程序，名为 sol.cpp。该程序从文件 data.in 中读取输入数据，并输出答案到文件 data.out 中。

2. 自己编写的“朴素解法”程序，名为 bf.cpp。该程序从文件 data.in 中读取输入数据，并输出答案到文件 data.ans 中。
3. 自己编写的随机数据生成程序，名为 random.cpp。该程序输出随机数据到文件 data.in 中。

依次编译这三个程序，得到三个可执行文件，例如在 Windows 系统下得到 sol.exe，bf.exe 和 random.exe。现在，我们需要编写一个 C++语言的“对拍脚本”，循环执行以下过程:

1. 运行随机数据生成器 random。
2. 运行“正解”程序 sol。
3. 运行“朴素解法”程序 bf。
4. 比对文件 data.out 和 data.ans 的内容是否一致。

头文件 cstdlib(stdlib.h)中提供了一个函数 system，它接受一个字符串参数，并把该字符串作为系统命令执行。例如代码 system("C:\\random.exe")执行 C 盘根目录下的 random.exe 文件。

Windows 系统命令 fc 或类 Unix 系统命令 diff 可以执行文件比对的工作，它们接受两个文件路径，比较二者是否一致。若一致，返回 0，否则返回非零值。如下为一个 Windows 系统下的对拍程序：

```cpp
#include <cstdio>
#include <cstdlib>
#include <ctime>

int main() {
    for (int T = 1; T <= 10000; T++) {
        //
        system("C:\\random.exe");
        // 当前程序已经运行的 CPU 时间，Windows 下单位 ms，Unix 下单位 s
        double st = clock();
        system("C:\\sol.exe");
        double ed = clock();
        system("C:\\bf.exe");
        if (system("fc C:\\data.out C:\\data.ans")) {
            puts("Wrong Answer!");
            // 程序立刻退出，此时 data.in 即为发生错误的数据
            return 0;
        } else {
            printf("Accepted, 测试点 #%d, 用时 %.0lfms\n", T, ed - st);
        }
    }
    return 0;
}
```

2.15 常数优化

卡常数，又称底层常数优化，是信息学竞赛中一种针对程序基本操作进行空间或时间上优化的行为，与时间复杂度或剪枝有别。 例如，同样复杂度的递归和递推，显然递推做法常数更小。虽然程序渐进时间复杂度可以接受，但是由于实现/算法本身的时间常数因子较大，使得无法在 OI/ACM-ICPC 等算法竞赛规定的时限内运行结束。

2.15.1 读入优化

常见读入方法耗时从小到大为：fread 读入 < 快速读入 <scanf 读入 <cin 读入。一般来说使用快速读入即可解决算法竞赛中读入卡常。快速读入是一种手写函数读入：

```
int read() {
    int x = 0, f = 1;
    char ch = getchar();
    while (ch < '0' || ch > '9') {
        if (ch == '-') f = -1;
        ch = getchar();
    }
    while (ch >= '0' && ch <= '9') {
        x = (x << 1) + (x << 3) + (ch ^ 48);
        ch = getchar();
    }
    return x * f;
}
```

还有一种优化版本的快速读入，和原来对比取代了 getchar()，更快的原因是因为他整段读取，把输入流中整段都存储与处理，操作更加连续，直到数组处理完毕再继续输入。

```
static char buf[100000], *pa = buf, *pb = buf;
#define gc pa == pb && (pb = (pa = buf) + fread(buf, 1, 100000, stdin), pa == pb) ? EOF : *pa++

inline int read() {
    register int x(0);
    register char c(gc);
    while (c < '0' || c > '9') c = gc;
    while (c >= '0' && c <= '9') x = (x << 1) + (x << 3) + (c ^ 48), c = gc;
    return x;
}
```

通常来说，在题目没有卡快速读入时，cin 操作比 scanf 要慢，但是可以经过一系列操作，使得 cin 的读入速度不比 scanf 差太多。

关闭同步/解除绑定。std::ios::sync_with_stdio(false)函数是一个“是否兼容 stdio”的开关，C++ 为了兼容 C，保证程序在使用了 printf 和 std::cout 的时候不发生混乱，将输出流绑到了一起。同步的输出流是线程安全的。这其实是 C++ 为了兼容而采取的保守措施，也是使 cin/cout 速度较慢的主要原因。我们可以在进行 IO 操作之前将 stdio 解除绑定，但是在这样做之后要注意不能同时使用 std::cin 和 scanf，也不能同时使用 std::cout 和 printf，但是可以同时使用 std::cin 和 printf，也可以同时使用 scanf 和 std::cout。

tie 函数。tie 是将两个 stream 绑定的函数，空参数的话返回当前的输出流指针。在默认的情况下 std::cin 绑定的是 std::cout，每次执行 << 操作符的时候都要调用 flush() 来清理 stream buffer，这样会增加 IO 负担。可以通过 std::cin.tie(0)来解除 std::cin 与 std::cout 的绑定，进一步加快执行效率。

2.15.2 输出优化

常见输出方法耗时从小到大为：fwrite 输出 < 快速输出 <printf 输出 <cout 输出。一般来说使用快速输出即可解决算法竞赛中输出卡常。快速输出如下。

```
1. void write(int a) {
2.     if (a < 0) putchar('-'), a = -a;
3.     if (a >= 10) write(a / 10);
4.     putchar(a % 10 + 48);
5. }
```

如上输入优化所述，关闭同步/解除绑定和 cout.tie(0)也可以加快 cout 的输出速度。在 cout 输出回车时，std::endl 使得输出效率下降，因为 std::endl 会首先在输出流的缓冲区插入一个换行符\n，然后强制刷新输出流的缓冲区；而\n 只是简单的在输出流的缓冲区中插入换行符，因此为了提高程序执行效率，建议养成输出\n 换行的习惯。

2.15.3 register 优化

register 的意思是“寄存器”，即把循环变量放到寄存器里面。其实很多情况下，其优化效果并不明显，在 int 中的优化效果相对比较明显，而如果是在 STL（标准模板库）的迭代器中使用，效果将会十分明显，同理，其原因是需要寄存的内容较多。

它的作用是声明一个反复使用的局部变量，不在内存中开辟空间而使用寄存器。使用方式为：

```
1. for (register int i = 0; i <= n; i++) cout << i << ' ';
```

不过开了 O2 优化的程序会将大多数适合寄存器的内容放入寄存器，所以手动的 register 就几乎没有优化了。

2.15.4 inline 优化

如果一个简单函数调用次数很多，属于瓶颈，调用时的传参可能会大大影响程序效率。这种情况下可以把这个函数声明时使用 inline 关键字，inline 声明的函数称为内联函数，可以减少跳转和传参的代价。

实际上，就是把整个函数进程放入一个地址中，按一定的顺序运行。引入内联函数实际上就是为了解决这一问题，减少了函数的跳转过程，同时整体的连贯性比较强，也就优化了整体时间。同时，他避免了函数调用多余的进出栈操作，进行进一步优化，而且简单的程序反复调用产生的瓶颈效应会导致时间大大增加，此时牺牲一定空间将这个函数声明为内联函数，就可以达到很好的优化效果。

2.15.5 吸氧优化

“吸氧”是指在程序代码块中的如下语句：

```
1. #pragma GCC optimize (1)
2. #pragma GCC optimize (2)
3. #pragma GCC optimize (3)
```

其实这就是在网上常见的“吸氧”。O0 代表不做任何优化，这是默认的编译选项。O1 优化会消耗少多的编译时间，它主要对代码的分支，常量以及表达式等进行优化。O2 会尝试更多的寄存器级的优化以及指令级的优化，它会在编译期间占用更多的内存和编译时间。O3 在 O2 的基础上进行更多的优化，例如使用伪寄存器网络，普通函数的内联，以及针对循环的更多优化。

这种优化一般来说了解即可，在做题时一般用不到此种优化，如果程序执行超时优先考虑算法或者代码问题，而不需要“为了优化而优化”。

2.15.6 其他优化

1. 二进制优化。在位运算章节大部分有所提及。
2. 手写 STL 优化。对于同样地数据结构，STL 封装的数据结构一般来说常数稍大，可以考虑手写 STL 卡常。
3. 玄学优化。有兴趣自行了解，竞赛一般不作要求。

2.16 拟阵

拟阵（Matroid），最开始由 Hassler Whitney 于 1935 年提出，不算是一个新颖的技术，

但是由于其在组合数学，算法，代数，拓扑，加密等领域有着许许多多的应用，直到现在，还是大家非常乐于研究的工具。熟悉的算法例如 Kruskal 最小生成树算法，匈牙利算法等等，都可以被拟阵解释，甚至应用在别的问题上。

通过线性代数我们了解到“线性独立”的概念，既是我们有一组同维度的向量$v_1 v_2 v_3 \dots v_n$，如果他们中的任意一个都不能被其他的向量线性表出，那么他们便是线性独立的。拟阵就是将这独立的概念扩展在其他组合问题（尤其是图论）上，从而用类似线性代数的方式来研究算法问题，事实证明，这样的做法是非常成功的。

拟阵定义：给定一个全集U，一个U上的集族$\mathcal{F}$被称作拟阵，记作$\mathcal{M} = (U, \mathcal{F})$，如果集族$\mathcal{F}$满足：

- $\emptyset \in \mathcal{F}$
- 如果$A \in \mathcal{F}, B \subseteq A$，那么$B \in \mathcal{F}$
- 如果$A \in \mathcal{F}, B \in \mathcal{F}$且$|A| < |B|$，那么存在$b \in B \backslash A$使得$A \cup \{b\} \in \mathcal{F}$

U中的元素被叫做边，而$\mathcal{F}$中的集合$S \in \mathcal{F}$叫做独立集。第二条叫做拟阵的遗传性质，第三条叫做拟阵的交换性质。这三条又被叫做拟阵公理。往往，交换性质是考察一个集族是不是拟阵的关键，因为前两个性质是很容易满足的。