信息学奥林匹克初级教程
（内部资料）

编者

2016 年 1 月 于清华园

目录

第 1 天

拾遗及准备

作为起始章节，本章将简短地回顾一下 C 和 C++ 语言程序设计的基础知识，初步学习 STL 的使用。很多知识我们在暑假期间已经细致地探讨过，这里简略带过，如有遗忘还请**务必**复习之前的材料或一起交流。

¶ 1.1 C 语言程序

C 语言程序以 `main()` 函数为入口，编程面向过程，典型的 C 程序可以看做函数与变量的组合，通过一串确定的操作序列（算法）达成期待的功能。

下面列举了一些重要知识，请确保你牢牢地掌握了它们:

- 变量的基本属性
- 各种基本数据类型的数据范围和使用方法
- 整数和 `char` 类型的联系和区别
- 数组的声明与初始化，字符数组
- 变量作用域

- `scanf()`、`printf()`、`gets()`、`puts()` 等的使用方法
- 补码规则与位运算
- 循环结构
- 函数与函数的传值调用机制
- 数学库函数、字符串库函数
- 宏定义与宏替换
- `const` 和 `static` 说明符的含义
- 指针的概念、指针与数组的联系
- 结构体、链表的概念和使用方法
- 基本排序算法
- 递归的概念与基本的递归设计

正常状态是看到这些条目的时候感觉到非常自然，把编程学作为骑自行车一样的底层技能，当然，“大致了解” 不代表不犯错误，还是请保持警惕，避免因基本知识的全局崩溃。

练习 1.1.1 (NOIP 2007 提高组初赛). 阅读下面的代码，给出其输出结果。

```
1  #include <stdio.h>
2
3  void fun(int *a, int *b) {
4      int *k;
5      k = a; a = b; b = k;
6  }
7
8  int main() {
9      int a = 3, b = 6, *x = &a, *y = &b;
10     fun(x, y);
11     printf("No.1: %d,%d ", a, b);
12     fun(&a, &b);
13     printf("No.2: %d,%d\n", a, b);
14     return 0;
15 }
```

输出结果: _____

练习 1.1.2. 写出所有你知道的 C 程序头文件文件名，回忆为什么要包含它们。

¶ 1.2 C++ 语言程序

1.2.1 从 C 到 C++

C++ 是 C 语言的一个超集，可以近似地说[1]，所有 C 语言代码都可以作为 C++ 代码通过编译。C++ 和 C 风格非常相似，你会发现转换到 C++ 竟是如此简单。当然，我们现阶段用到的，只是 C++ 的冰山一小角而已。下面一段代码是最基本的 C++ 程序。

代码 1.2.1: C++ 版 Hello world

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main() {
6     cout << "Hello, world." << endl;
7     return 0;
8 }
```

我们从下面几个方面分析它与之前程序的区别：

<iostream> C++ 的头文件只有文件名，没有扩展名.h，<iostream> 是像 <stdio.h> 一样的标准输入输出头文件，而 C++ 的输入输出都是“流”。需要注意的是，如果需要使用 C 的函数，则需要包含相应的头文件，规则是将原来头文件的.h 去掉，并在开头增加字母 c，例如要想使用 printf()，则需要包含 <cstdio>。

using namespace 这一指令告诉编译器下面的代码使用某一命名空间。命名空间（namespace）表示标识符（identifier）的可见范围。例如 cout 定义在 std 命名空间里，如果没有这句话，你将需要通过 std::cout 来使用标准输出流。通过命名空间，你可以创建另一个 cout，通过不同的命名空间区分开它们。因此这样做避免了大型项目中的命名冲突，现阶段我们不需要搞命名空间，每段程序前都加上这句话就好了。

cout 可以将它理解为 C++ 里的 printf()，但是它和 printf() 原理和使用方法都不一样。之前我们需要类型说明符如%s 来说明要输出的变量的类型，而现在这步操作由重载的 operator<< 完成，一切都是自动识别的。

endl 可以将它理解为 C++ 里的'\n'，注意它只适用于输出流。

[1]一些语法细节可能有冲突，这时可以通过 extern "C" {...} 的方式解决

1.2.2 C++ 中的输入输出

在 C++ 中变量的输入也很轻松，通过重载好的 `operator>>` 识别变量类型的操作依然可以自动完成。下面一段代码演示 C++ 中的变量输入。

代码 1.2.2: C++ 的输入方式

```
1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  int main(int argc, const char * argv[]) {
7      int a;
8      double b;
9      string c;
10     cin >> a >> b >> c;
11     cout << a << b << c << endl << c.length();
12     return 0;
13 }
```

我们再来看看这段程序中的新鲜事物：

string 要使用系统自带的字符串类型，需要包含 `<string>` 头文件，它是对老式字符数组的一种封装，提供相关方法如 `length()`（获取字符串长度）、`operator+`（相当于 C 语言中 `strcat()`）等等，并且，其空间的申请和释放是自动完成的。它可以从 C 风格的字符串中构造出来，也可以转化成 C 风格的字符串，通过方法 `c_str()` 实现。

cin 预先定义好的标准输入流，用法与 `cout` 相似（注意插入运算符 `>>` 和送出运算符 `<<` 的方向不要搞反）。`cin` 与 `scanf()` 各有优劣，具体情况具体分析，怎么简单怎么来。注意本例中 `c` 的输入以空白字符结束，`a`、`b`、`c` 输出时三者之间也没有空白字符隔开，建议自己尝试一下。

.length() 这种函数调用方式在 C 语言中不曾有过，事实上，`string` 是一个结构体[2]，其中不仅有 `char` 数组等数据，还定义了一些函数，称之为“方法”，访问数据通过“`.`”运算符，访问方法也是一样。这些方法共享结构体的一个实例对象中的数据，它们对外不可见，也就是一种“封装”，程序员仿佛创造了一个“对象”，交给它一些数据和一些任务，而至于它如何存、如何做则并不太关心，这就是“面向对象编程”的基本思想。全面展开可以铺出好几章来不过没有用，知道这样调用方法就好了。

[2]这样说是通俗但不严谨的

记得曾经一个微软工程师来作报告时讲“一切用户的输入都是邪恶的”，每年 NOIP 也都不乏“邪恶”的输入，这时就需要一些技巧。C++ 中输入输出靠流实现，有流我们就可以去缓冲，我们可以把一整行先读成字符串，把字符串转成流，然后从这个流里读需要的东西，这适用于以行分界的输入；甚至，我们可以在这个流中“动一些手脚”以获得更安全的输入。而这些操作，都通过定义在 `<sstream>` 中的 `stringstream` 完成。

`stringstream` 的方法和字符串很相似，可以从字符串中构造出来，如果 `s` 是一个字符串，那么 `stringstream ss(s);` 一句话就创建了一个从 `s` 构造出的流，接下来使用 `ss` 的方式和使用 `cin` 的方式基本相同。

另外，如果需要多次使用同一个流，必须调用 `clear()` 方法（字符串 `string` 和即将讲到的很多容器也都有此方法），建议将其声明为局部变量。

1.2.3 动态内存分配

关于堆上的内存分配，C++ 的处理方式和 C 也不一样，它对 C 的底层函数进行了封装以提高稳定性和安全性。还记得 `malloc()`、`free()` 的作用和用法吗？在 C++ 里它们变成了下面这样：

代码 1.2.3: 动态内存分配

```
1  #include <iostream>
2
3  typedef int ElemType;
4
5  struct Array {
6      Array(int len): size(len), base(new ElemType [len]) {}
7      ~Array() {
8          delete [] base;
9      }
10     ElemType & operator[](int i) {
11         return base[i];
12     }
13     ElemType *base;
14     int size;
15 };
16
17 int main(int argc, const char * argv[]) {
18     int tot;
19     std::cin >> tot;
20     Array array(tot);
```

```
21      for (int i(0); i < tot; i++)
22          std::cin >> array[i];
23      for (int i(0); i < array.size; i++)
24          std::cout << array[i] << ' ';
25      return 0;
26  }
```

事实上 `malloc()` 和 `free()` 仍然是可以用的，用法也和以前相同，只要你包含了 `<cstdlib>` 头文件，但是由于 **C++ 禁止隐式类型转换**，用 `new` 和 `delete` 相比之下更方便。上面代码能理解最好，不能理解的话知道 `new` 运算符可以申请一个对象的内存空间赋值给指向该对象的指针，`delete` 可以释放指针指向的对象的内存，带有 `[]` 的 `new` 和 `delete` 可以操作一个数组，就可以了。下面是详细解释：

构造函数 构造函数（Constructor），是一种特殊的方法。主要用来在创建对象时初始化对象，即为对象成员变量赋初始值，总与 `new` 运算符一起使用在创建对象的语句中。事实上 `operator new` 即调用了构造函数（如果程序员没写，编译器会提供一个默认的不带参数的构造函数）。

初始化列表 构造函数函数名与类名相同，不写返回值，冒号后接的是初始化列表，表示直接构造这些对象。

析构函数 析构函数（Destructor），与构造函数相反，当对象脱离其作用域时自动执行，执行清理操作。它不带参数也不能被重载。事实上，`operator delete` 即调用了析构函数，如果程序员没写，编译器也会提供一个默认的析构函数。

引用类型 引用（Reference），就是某一变量（目标）的一个别名，对引用的操作与对变量直接操作完全一样。函数可以使用引用作参数，也可以返回引用，注意代码1.2.3 中 `ElemType &` 不是取地址，而是返回一个引用，这样才允许 `std::cin` 像写入数组一样直接写入该函数的返回值。引用和指针相似但不一样（相当于编译器帮我们进行了取内容操作）。

循环计数器 注意代码1.2.3 中两个 `for` 循环都用了名为 `i` 的 `int` 型变量作计数器，并且它们都可以在 `for` 循环的括号中声明，作用域延伸到该 `for` 循环结束。

C++ 语言博大精深，如果有更感兴趣的东西，欢迎随时讨论。

练习 1.2.1 (NOIP 2007 提高组初赛). 格雷码是对十进制数的一种二进制编码。编码顺序与相应的十进制数的大小不一致。其特点是：对于两个相邻的十进制数，对应的两个格雷码只有一个二进制位不同。另外，最大数与最小数之间也仅有一个二进制位不同，如果把每个二进制的位看作一个开关，则将一个数变为相邻的另一个数，只须改动一个开关。因此，格雷码广泛用

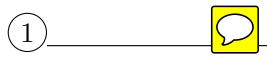
于信号处理、数–模转换等领域。以 *4* 位二进制数为例，格雷码编码如下：

十进制数	格雷码	十进制数	格雷码
0	0000	8	1100
1	0001	9	1101
2	0011	10	1111
3	0010	11	1110
4	0110	12	1010
5	0111	13	1011
6	0101	14	1001
7	0100	15	1000

下面程序的任务是：由键盘输入二进制数的位数 n（$n < 16$），再输入一个十进制数 m（$0 \leqslant m < 2n$），然后输出对应于 m 的格雷码（共 n 位，用数组 `gr[]` 存放）。

为了将程序补充完整，你必须认真分析上表的规律，特别是对格雷码固定的某一位，从哪个十进制数起，由 *0* 变为 *1*，或由 *1* 变为 *0*。

```
1  #include <iostream>
2  #include <iomanip>
3
4  int main() {
5      int bound = 1, m, n, i, j, b, p, gr[15];
6      std::cout << "Input n, m:" << std::endl;
7      std::cin >> n >> m;
8      for (i = 1; i <= n; i++) bound = ____1____;
9      if (m < 0 || m >= bound) {
10         std::cout << "Data error!" << std::endl;
11         ____2____;
12     }
13     b = 1;
14     for (i = 1; i <= n; i++) {
15         p = 0; b = b * 2;
16         for (____3____; j <= m; j++)
17             if (____4____)
18                 p = 1 - p;
19         gr[i] = p;
20     }
21     for (i = n; ____5____)
22         std::cout << std::setw(1) << gr[i];
23     std::cout << std::endl;
24     return 0;
25 }
```



答案：①
②
③
④
⑤

C++ 相比于 C 语言增加了很多黑科技（虽然这里都没提到），而且并未影响 C++ 的运行效率，同时它对 C 也高度兼容，因此之后的题目和示例代码均采用 C++ 撰写，我们也约定对于每段程序都有的部分（如 `using namespace`）可能略去，还请同学们不要忘记。

¶ 1.3 STL 入门

如果只是为了让代码变得更简洁和漂亮而费这么大力气学 C++，那真是浪费情怀了... 这一小节将对 STL 进行简要介绍，STL 被认为是 NOIP 系列竞赛“不公平”的地方，因为 C 和 Pascal 语言中并没有类似 STL 的东西可以用，而 STL 又是如此强大，省去了很多不必要的麻烦。因此 NOI 发布的标准竞赛环境中规定禁止选手使用 STL，不过私下问一些过来人呢，又说无所谓反正他用了。我的想法是，先学，但也掌握自己实现的方法，见机行事。

定义 1.3.1. *STL = Standard Template Library*，标准模板库，惠普实验室开发的一系列软件的统称。它是由 *Alexander Stepanov*、*Meng Lee* 和 *David R Musser* 在惠普实验室工作时所开发出来的。由于实用性非常强，后被纳入 *C++* 标准中。

STL 主要由算法 (algorithm)、容器 (container) 和迭代器 (iterator) 三部分组成。

算法 算法部分提供了比较、交换、查找、遍历操作、复制、修改、移除、反转、排序、合并等算法。

容器 容器部分提供了向量 (vector)、列表 (list)、双端队列 (deque)、集合 (set)、栈 (stack)、队列 (queue)、映射 (map) 等经典数据结构。

迭代器 迭代器部分是 STL 的基础部分，向上支撑算法和容器部分，并在算法和容器之间起到桥梁的作用。

为了使用 STL 我们先从“泛型编程”说起。

定义 1.3.2. 泛型程序设计是程序设计语言的一种风格或范式。允许程序员在强类型程序设计

语言中编写代码时使用一些以后才指定的类型，在实例化时（*instantiate*）作为参数指明这些类型。

泛型可以做很多别人做不了的事情，很多黑科技也必须基于泛型实现。它的好处是容器与具体数据类型相互抽离，使作为“盒子”的容器可以不关心自己将来将盛放什么东西，专心将自己的方法搞好搞明确。C++ 通过模板支持泛型程序设计，而定义1.3.2 中的参数通过类似下面的方式指定：`std::vector<int> vec;` 这句话声明了一个向量对象，而向量中的元素数据类型是整型 `int`，如果类/结构引入了多个模板参数，这些参数则需要在尖括号中一一指定，逗号隔开。

STL 提供了很多种容器，我们常用的有向量 vector、集合 set、映射 map、栈 stack、队列 queue 等，常用的算法有排序和查找等，本节只介绍四种，剩下的以后会提到，也可以登录 `http://www.cplusplus.com/reference/` 到官方参考文档中深入学习。

`vector` 头文件：`<vector>`；向量，简单理解为变长数组，由于已经做好了封装，在预先不确定数组大小的情况下非常有用，动态扩充和动态删除都是自动完成的，需要注意的是，如果频繁地扩充和删除，则效率可能会降低。方法主要有：

- `vector()`：构造容量与长度均为 0 的空向量。
- `vector(size_type n, const T& value = T())`：构造容量与长度均为 n 的向量，而且各个元素的值为 `value`。
- `vector(vector<T>& x)`：构造向量，复制向量 x 的内容。结果向量的容量与长度相等。
- `vector(InputIterator first, InputIterator last)`：构造向量，复制从 first 开始并且在 last 之前的元素。
- `size_type size() const`：返回向量的长度。
- `bool empty()`：如果向量的长度为 0，则返回 true；否则返回 false。
- `void reserve(size_type n)`：预订向量的容量。如果向量原来的容量大于或等于 n，则容量不变；否则，容量变为 n。同时，向量的长度不变。
- `void resize (size_type n, T c=T())`：将向量的长度设置为 n。如果原来的长度小于 n，则超出原来长度的元素初始化为 c，原有元素的值不变。如果向量原来的容量大于或等于 n，则容量不变；否则，容量变为 n。

- `void clear()`：清空向量的内容，即向量长度将变为 0，但是向量的容量不变。
- `reference at(size_type n)` 和 `reference operator[](size_type n)`：返回下标为 n 的元素的引用。
- `reference front()`：返回第一个元素的引用。
- `reference back()`：返回最后一个元素的引用。
- `void assign(size_type n, const T& u)`：将向量的长度变为 n，并将各个元素的值均设置为 u。如果向量原来的容量小于 n，则向量的容量也变为 n；否则，向量的容量不变。
- `void assign(InputIterator first, InputIterator last)`：将迭代器从 first 到 last 界定的向量长度及元素值复制给当前向量。如果向量原来的容量小于新长度，则向量的容量也变为新长度；否则，向量的容量不变。
- `vector<T>& operator=(const vector<T>& x)`：复制 x 的内容。如果原来的容量已经超过向量 x 的长度，则容量不变。
- `void swap(vector<T>& x)`：将当前向量与向量 x 进行交换，包括内容与容量。
- `iterator insert(iterator position, const T& x)`：在指定的位置之前插入新元素 x。返回新元素所对应的迭代器。
- `void insert(iterator position, size_type n, const T& x)`：在指定的位置之前插入 n 个新元素 x。
- `void insert(iterator position, InputIterator first, InputIterator last)`：在指定的位置之前插入从 first 开始并且在 last 之前的元素。
- `void push_back(T& u)`：在向量的末尾添加新元素 u。此外，`vector` 事实上也提供了 `push_front(T& u)` 的操作，但这种操作必然重新分配内存，效率很低。
- `void pop_back()`：删除最后一个元素。
- `iterator erase(iterator position)`：删除迭代器 position 所对应的元素。该函数返回 position 之后的元素所对应的迭代器（! 一定要注意删除函数是有可能改变内存中数组基地址的，因此删除之后必须接收返回的新迭代器继续访问）。

- `iterator erase(iterator first, iterator last)`：删除从 first 开始并且在 last 之前的元素。该函数返回紧接着被删除元素之后的元素所对应的迭代器。

- `bool operator<(const vector<T>& x, const vector<T>& y)`：比较向量 x 与 y 之间大小。从头到尾逐个比较 x 与 y 的每个元素。如果所有元素都相等，则 x 与 y 相等；否则，由第一个不相等的元素之间的大小关系决定 x 与 y 之间大小。如果 x 与 y 的元素个数不同且长度小的向量的所有元素与另一个向量的对应元素依次相等，则认为长度小的向量小于长度大的向量（其他关系运算符也都有定义不再重复）。

`list` 头文件：`<list>`；链表，之前已经详细地介绍过机理，与 `vector` 不一样的是链表中的元素在内存里的位置是终身不变的，这对于存储大型、复杂、难以移动的元素有利，链表在任意位置插入和删除都很快，但索引和查找的效率较低。方法主要有：

- `void remove(const value_type& val)`：析构所有与 `val` 相等的元素并更改表的大小。

- 其他成员方法与 `vector` 基本相同，不再重复，但 `list` 没有重载 `operator[]`。

`set` 头文件：`<set>`；集合，用来标识不同元素的关键数据，关键字的值常简称为键值。在集合 set 中，相等键值的元素只能有一个。集合内部元素是有序的，用迭代器遍历将得到由小到大的序列，内部以树的方式存储，不支持下标运算符。方法主要有：

- `pair<iterator, bool> insert(const T& x)`：不妨设返回的数据对为 p。如果元素 x 不在集合中，则插入新元素 x，且返回值 `p.first` 为新元素所对应的迭代器，`p.second` 为 `true`；否则，直接返回的值 `p.first` 为在集合中元素 x 所对应的迭代器，且 `p.second` 为 `false`。另两种插入操作与 `vector` 相似，操作上将只判断输入迭代器的有效性，保证集合的性质不变。

- `size_type count(const key_type& x) const`：如果在集合中存在关键字为 x 的元素，则返回 1；否则，返回 0。

- `iterator find(const Key& k)`：如果存在键值为 k 的元素，则返回该元素所对应的迭代器，否则返回在最后一个元素之后的迭代器（与 `end()` 返回结果相同）。

- `iterator lower_bound(const Key& k)`：返回第一个大于或等于给定键值 k 的元素所对应的迭代器。如果相应的元素不存在，则返回在最后一个元素之后的迭代器。

- `iterator upper_bound(const Key& k)`：返回第一个大于给定键值 k 的元素所对应的迭代器。如果相应的元素不存在，则返回在最后一个元素之后的迭代器。
- `size_type erase(const key_type& k)`：删除键值为 k 的元素。返回实际被删除元素的个数。另两种删除操作与 `vector` 相同。
- 另一些方法，如构造函数、拷贝构造函数、赋值运算符、`swap(set& s)`、`size()`、`empty()`、`clear()` 与前面介绍过的 `vector` 的成员方法功能和接口都相同，不再重复。

map 头文件：`<map>`；映射，就是从键（key）到值（value）的映射，因为重载了 `operator[]`，它有点像“高级版”数组。方法主要有：

- `mapped_type& operator[](const key_type& k)`：查询键 k，如果存在，返回对应的值的引用，如果不存在，用默认构造函数构造值，插入到映射中，返回刚刚构造的值，该函数的调用与 `(*((insert(make_pair(k,mapped_type()))).first)).second` 等价。这也就是说，`map` 相当于保存有 `pair<KeyType, ValueType>` 类型的特殊集合。
- 另一些成员方法与 `set` 基本相同，不再重复。注意如果使用 `insert` 方法，则参数需要由 `make_pair` 或 `std::pair` 的构造函数构造出来，返回的迭代器也要取内容然后访问 `first` 和 `second`。

迭代器 头文件：`<iterator>` 不过只要使用了 STL 容器该头文件已经包含。迭代器，是对指针的一种封装，但这一步封装使 STL 异常强大，所有容器的迭代器接口都一致，方法基本一致（`++` 表示移动到后继，`--`表示移动到前驱），外部不需要关注容器内部的实现细节，从而使“算法”独立出来不需要重复编写，代码重构也很简单，更换容器甚至不需要改变太多代码。以 `vector` 为例，容器中的方法如下：

- `iterator begin()`：返回第一个元素所对应的迭代器。
- `iterator end()`：返回在最后一个元素之后的迭代器。（! 一定要理解准确，STL 的所有区间都是左闭右开区间，end() 返回的迭代器永远是不应该被访问的，想想看，这样做有什么好处?）
- `reverse_iterator rbegin()`：返回逆序的第一个元素所对应的迭代器。
- `reverse_iterator rend()`：返回逆序的在最后一个元素之后的迭代器。

- 事实上对于 const 限定的容器，还有 cbegin()、cend()、crbegin()、crend() 四个方法，但竞赛中用不到，如果没有必要，请不要加 const 限定符。

算法 头文件：<algorithm>；算法库中提供了很多有用的算法并且效率一般很高，例如 C 语言中我们曾经用宏定义实现过的 min 和 max 在算法库中都通过模板实现，直接使用即可。下面列举一些通用的算法以供使用（下面所有的区间都是指左闭右开区间，激动人心的是，这些模板函数不仅支持各种迭代器，传给它们原始的指针也是可以的）：

- void swap(T& a, T& b)：交换两个变量（注意参数是引用），T 需要是能够拷贝构造，支持复制运算的类型。
- ForwardIterator max_element(ForwardIterator first, ForwardIterator last)：顺序查找区间内最大元素。
- ForwardIterator min_element(ForwardIterator first, ForwardIterator last)：顺序查找区间内最小元素。
- void sort(RandomAccessIterator first, RandomAccessIterator last)：排序，最常用算法之一，待排序数据类型需要支持 operator<，或者可以通过构造仿函数作为第三个参数传递给 sort，STL 中涉及元素比较的算法都重载了两个版本，默认调用 operator<，如果提供了仿函数则调用仿函数类的 operator()。
- OutputIterator copy(InputIterator first, InputIterator last, OutputIterator result)：区间拷贝，result 指向目标起点，返回拷贝后的目标终点（which points to the element following the last element copied）。注意该函数不检查目标区域有效性，需要预留好空间。
- void fill(ForwardIterator first, ForwardIterator last, const T& val)：区间赋值，区间内所有元素赋值为 val。
- void generate(ForwardIterator first, ForwardIterator last, Generator gen)：区间赋值，但赋的值是通过对 gen() 的调用实现，它可以是一个函数，也可以是一个仿函数。
- Function for_each(InputIterator first, InputIterator last, Function fn)：对区间内所有元素应用一个函数/仿函数 fn。
- InputIterator find(InputIterator first, InputIterator last, const T& val)：在区间内顺序查找第一个与 val 相等的元素并返回指向它的迭

代器，如果不存在，返回 last。比较操作通过 operator== 实现。

- difference_type count(InputIterator first, InputIterator last, const T& val)：返回区间内与 val 相等的元素个数，比较操作同上。
- ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2)：在第一个区间内查找对应第二个区间的第一个匹配，返回其起点的迭代器，如果第二个区间为空，返回值未定义。
- ForwardIterator remove(ForwardIterator first, ForwardIterator last, const T& val)：将区间内所有与 val 相等的元素移除。
- void replace(ForwardIterator first, ForwardIterator last, const T& old_value, const T& new_value)：将区间内所有与 val 相等的元素替换为 new_value。
- void reverse(BidirectionalIterator first, BidirectionalIterator last)：将一个区间反转。

STL 性能很强，速度很快，值得信赖，如果发现程序效率低，一般应认为是自己的问题（自己写 STL 里的容器没见过比 STL 更快的），应重新衡量算法设计的策略。事实上通过通用的算法已经可以求解很多问题，通用算法实现起来也不难，但是直接调用这些算法可以节省大量时间，这在考试的考查机制之下非常重要。不过，需要强调的是，不论你使用库里的什么东西，一定要先做测试，一方面，库也可能有 Bug，另一方面，使用上的细节也可能出现问题，望请引起重视。

例题 1.3.1. 输入一些单词，找出所有满足如下条件的单词：该单词不能通过字母重排，得到输入文本中的另外一个单词，重排过程中字母不分大小写。输入单词以空格隔开，输入 # 表示结束（结束符不算单词），输出所有满足上述条件的单词，输出时保留输入中的大小写，按字典序进行排列，由小到大逐行输出。

样例输入：

```
ladder came tape soon leader acme RIDE lone Dreis peat ScAlE orb eye
Rides dealer NotE derail LaCeS drIed noel dire Disk mace Rob dries
#
```

样例输出：

```
Disk
```

```
NotE
derail
drIed
eye
ladder
soon
```

题目分析:

这貌似是字谜爱好者常玩的游戏，一般的人脑运算方式可能是“枚举”，读所有的单词，枚举所有的排列，然后去重？好吧说到这里复杂度已经无法想象了... 事实上如果每个单词的所有排列都能确定地生成一种形式，那么问题迎刃而解。这种形式怎么找呢？按题意重排过程中不分大小写，那么干脆全小写，排列有很多种，但是按从小到大的只有一种，于是我们得出了字符串归一化的方式。

归一化之后，只要我们能构建从字符串到“出现次数”的映射，就可以轻松地判断哪些单词满足条件，为此，map 非常合适。输入输出直接用系统的 string 类型即可，数组可以用 vector 保存，排序算法库里有，于是这个题目可很轻松地写出来。

参考题解:

```
1  #include <iostream>
2  #include <string>
3  #include <map>
4  #include <cctype>
5  #include <vector>
6  #include <algorithm>
7
8  using namespace std;
9
10 inline void toLower(char &ch) {
11     ch = tolower(ch);
12 }
13
14 inline void output(const string &s) {
15     cout << s << endl;
16 }
17
18 string normalize(const string &s) {
19     string ans(s);
20     for_each(ans.begin(), ans.end(), toLower);
21     sort(ans.begin(), ans.end());
22     return ans;
23 }
24
```

```
25 int main() {
26     string s;
27     vector<string> words;
28     map<string, int> freq;
29     while (cin >> s) {
30         if (s[0] == '#') break;
31         words.push_back(s);
32         freq[normalize(s)]++;
33     }
34     vector<string> ans;
35     ans.reserve(words.size());
36     for (vector<string>::iterator it = words.begin(); it != words
           .end(); it++)
37         if (freq[normalize(*it)] == 1)
38             ans.push_back(*it);
39     sort(ans.begin(), ans.end());
40     for_each(ans.begin(), ans.end(), output);
41     return 0;
42 }
```

上面演示了容器的基本使用方式，注意 `vector<string>::iterator` 一句，`iterator` 是在 `vector<string>` 里利用 `typedef` 定义好的数据类型，对其他的容器使用方式也相似。用不用 `<algorithm>` 看个人习惯，其实有了 *lambda* 表达式之后是最爽的，但是竞赛环境比较古老，所以咱们就不讲了 =。=

练习 1.3.1. 现从左到右有 n 个木块，编号为 $0 \sim n-1$，请你编写一个程序，模拟以下 *4* 种操作（下面的 a 和 b 都是木块编号，且 a 和 b 在同一堆的指令非法，应当忽略）

- *move a onto b*: 把 a 和 b 上方的木块全部归位，然后把 a 摞在 b 的上面。
- *move a over b*: 把 a 上方的木块全部归位，然后把 a 放在 b 所在木块堆的顶部。
- *pile a onto b*: 把 b 上方的木块全部归位，然后把 a 及上面的木块整体摞在 b 的上面。
- *pile a over b*: 把 a 及上面的木块整体摞在 b 所在木块堆的顶部。

输入首先给出木块块数，然后有若干指令，直到 *"quit"* 表示输入结束。所有操作结束后，输出每个位置的木块列表，按照从底部到顶部的顺序排列。

样例输入:

```
10
move 9 onto 1
move 8 over 1
```

```
move 7 over 1
move 6 over 1
pile 8 over 6
pile 8 over 5
move 2 over 1
move 4 over 9
quit
```

样例输出:

```
0: 0
1: 1 9 2 4
2:
3: 3
4:
5: 5 8 7 6
6:
7:
8:
9:
```

¶ 1.4 小结

时间紧，任务重，一天学完 C++ 也情非所愿，很多东西看书看不懂，看课件看不懂，多写写就好了。C++ 标准库里函数命名都是很讲究的，搞清楚命名方式，当想要某种功能时从名字猜也基本可以猜到，实在不行，可以阅读头文件里的代码，总能发现需要的东西。希望每个人都能在 C++ 中找到自己喜欢的部分!

另外，这里为了铺垫简单说下时间复杂度，严谨的证明不大有必要，可以直观地 “感觉” 程序是否可能超时，代入题目数据范围的上界，假设时间限制为 1 秒，可以参考下表判断:

表 1.1: 1s 可以处理的循环次数

1000000	游刃有余
10000000	勉勉强强
100000000	很悬，仅限循环体非常简单的情况

¶ 1.5 参考习题

习题 1.1. 阅读代码写结果:

```
#include <iostream>

using namespace std;

int main() {
    int i, p[5], q[5], x, y = 20;
    for (i = 0; i <= 4; i++)
        cin >> p[i];
    q[0] = (p[0] + p[1]) + (p[2] + p[3] + p[4]) / 7;
    q[1] = p[0] + p[1] / ((p[2] + p[3]) / p[4]);
    q[2] = p[0] * p[1] / p[2];
    q[3] = q[0] * q[1];
    q[4] = q[1] + q[2] + q[3];
    x = (q[0] + q[4] + 2) - p[(q[3] + 3) % 4];
    if (x > 10)
        y += (q[1] * 100 - q[3]) / (p[p[4] % 3] * 5);
    else
        y += 20 + (q[2] * 100 - q[3]) / (p[p[4] % 3] * 5);
    cout << x << "," << y << endl;
}
```

输入: 6 6 5 5 3

输出:

习题 1.2. 阅读代码写结果:

```
#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;

int main() {
    int a1[51] = {0};
    int i, j, t, t2, n = 50;
    for (i = 2; i <= sqrt(n); i++)
        if (a1[i] == 0) {
            t2 = n / i;
            for(j = 2; j <= t2; j++)
                a1[i * j] = 1;
        }
```

```
16      t = 0;
17      for (i = 2; i <= n; i++)
18          if(a1[i] == 0) {
19              cout << setw(4) << i;
20              t++;
21              if (t % 10 == 0)
22                  cout << endl;
23          }
24      cout << endl;
25      return 0;
26  }
```

输出：

习题 1.3 (正方形). 有 n 行 n 列（$2 \leqslant n \leqslant 9$）的小黑点，还有 m 条线段连接其中的一些小黑点。试统计这些线段连成了多少个正方形（每种边长分别统计）。

行从上到下编号为 $1 \sim n$，列从左到右编号为 $1 \sim n$，边用 H_{ij} 和 V_{ij} 表示，分别代表从（i, j）到（i, $j+1$）和从（i, j）到（$i+1$, j）的边。

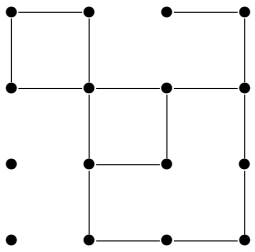


图 1.1: 正方形

如图*1.1*所示，最左边的线段将表示为 V_{11}，图中包含两个边长为 *1* 和一个边长为 *2* 的正方形。输入多行，第一行为点维数 n，第二行为连线总数 m，后 m 行为连线信息。输出多行，为所有尺寸的正方形，每行两个整数 a 和 b 表示有 a 个边长为 b 的正方形。如果没有，输出 None。

样例输入：

```
3
H 1 1
H 2 1
V 2 1
```

样例输出：

```
None
```

习题 1.4 (特别困的学生). 课堂上有 n 个学生（$n \leqslant 10$），每个学生都有一个“睡眠–清醒”周

期，其中第 i 个学生醒 a_i 分钟后睡 b_i 分钟，然后重复 $(1 \leqslant a_i, b_i \leqslant 5)$，初始时第 i 个学生处在他的周期的第 c_i 分钟。每个学生在临睡前会察看全班睡觉人数是否严格大于清醒人数，只有这个条件满足时才敢睡觉，否则就坚持听课 a_i 分钟后再次检查这个条件。问经过多长时间后全班第一次达到都清醒的状态？（图*1.2*给出了样例输入输出对应的各时刻状态）

输入多行，第一行为学生总数 n，后面 n 行每行三个整数 a，b，c 如题所述。

输出一行，为所求时间点，如果并不存在“全部清醒”的时刻，输出 -1。

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
		×	×	×	×			×	×	×	×						
×	×	×	×	×		×	×	×	×	×		×	×	×	×	×	
×	×	×		×	×	×	×		×	×	×	×					

图 1.2: 3 个学生每个时刻的行为，× 表示睡觉

样例输入：

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

样例输出：

```
18
```

习题 1.5 (洪水). 有一个 $m \times n$ $(1 \leqslant m, n < 30)$ 的网格，每个格子是边长 *10* 米的正方形，网格四周是无限大的墙壁。输入每个格子的海拔高度，以及网格内雨水的总体积，输出水位的海拔高度以及有多少百分比的区域有水（即高度严格低于水平面）。

输入多行，第一行为 m 和 n，接下来 m 行每行 n 个整数表示各方格海拔，正数表示高于海平面，负数表示低于海平面，区域描述结束后还有一行，一个整数表示区域内水的总体积。

输出最后水平面基于海平面的高度和被淹没百分比（保留至小数点后两位），格式见样例。

样例输入：

```
3 3
25 37 45
51 12 34
94 83 27
10000
```

样例输出：

```
Water level is 46.67 meters.
66.67 percent of the region is under water.
```

第 2 天

简单的数值计算问题

数学，特别是数论与计算机科学有着紧密的联系，所以也常被选作题材。本章将介绍一些基础的数学方法，这一部分大家比我强，欢迎随时发现可以应用数论知识的题目，把窍门分享给大家。另外，专门的数学问题自然由数学竞赛考察，信息学更偏重“建模”的过程，请在例题和习题中予以关注。

¶ 2.1 Euclid 算法

为了求两个数的最大公约数[1]，我们马上能想到的方式是从最小数开始试除这两个数，能同时整除（即% 运算返回 0）即为最大公约数，显然这个算法的时间复杂度是不能忍受的，为了进行优化引入欧几里德算法（辗转相除法）。

设 $gcd(a,b)$ 是返回自然数 a 和 b 的最大公约数的函数，记 a 除以 b 得到的商和余数分别为 p 和 q，由于 $a = b \times p + q$，所以 $gcd(b,q)$ 既能整除 a 又能整除 b，也就能整除 $gcd(a,b)$；反之，由于 $q = a - b \times p$，同理 $gcd(a,b)$ 能整除 $gcd(b,q)$，因此得到公式 $gcd(a,b) = gcd(b,q)$。代码如下：

[1] a 和 b 的最大公约数 g 是 a 和 b 的线性和中的最小正整数，即所有形如 $ua + vb$（其中 u 和 v 是整数）的数中的最小正整数。

代码 2.1.1: 辗转相除法

```
1 int gcd(int a, int b) {
2     return b == 0 ? a : gcd(b, a % b);
3 }
```

递归非常高效，是否需要考虑栈溢出呢？可以证明该函数递归层数不超过 $4.785 \lg N + 1.6723$，哪怕取 `unsigned long long` 的最大值，递归层数也没有超过 100，完全没有问题。

例题 2.1.1. 给出一个这样的除法表达式：$X_1/X_2/\cdots/X_K$，其中 X_i 和 K 是正整数且 $X_i \leqslant 10^9$，$K \leqslant 10000$，表达式应按照从左向右的顺序求值，但也可以在表达式中嵌入括号以改变计算顺序，例如表达式 1/2/1/2 值为 1/4 但 (1/2)/(1/2) 值为 1。

输入 $X_1, X_2, \cdots, X_K$，判断是否可以通过添加括号，使表达式的值为整数。

题目分析：

暴力枚举？哈哈哈那还是太 *naive* 了根本没法做好不好...

由于表达式不管运算顺序如何一定可以写成 A/B 这样的形式，其中 A 表示其中一些 X_i 的乘积，B 表示其他数的乘积，并且，不管如何更改运算顺序，X_2 永远是 B 的一员。

于是我们总是想构造这样一个“最简”的形式，如果连它都不能是整数，那就不可能有整数的形式了，要找的形式显然要求分母里的因子个数越少越好，幸运的是，我们可以给出使除了 X_2 的数都放在分子位置的构造，满足

$$E = \frac{X_1 X_3 \cdots X_K}{X_2}$$

于是问题就简化成了判断 E 是不是整数，很简单嘛，直接算就好了？然而对于题里的数据范围，直接算肯定要越界，于是想到高精度（大整数乘除），但是写起来巨麻烦，不用高精度的话，我们还可以应用唯一分解定理——

$$X_2 = p_1^{a_1} p_2^{a_2} \cdots p_n^{a_n}$$

然后依次判断每个 $p_i^{a_i}$ 是否是 $X_1 X_3 \cdots X_K$ 的约数，即将分子也分解为各个 $p_j^{a_j}$ 的乘积，依次判断 a_i 是否不小于 a_j。这样不需要高精度，但需要素数算法和分解算法，依然很麻烦。不过已经想到这了的话，岂不可以直接对 E 进行约分吗？下面给出题解的伪代码描述——

Require: $x[1 \ldots k]$

1: $x[2] \leftarrow x[2] \,/\, \mathrm{GCD}(x[2], x[1])$
2: **for** $i \leftarrow 3$ **to** k **do**
3: $\quad x[2] \leftarrow x[2] \,/\, \mathrm{GCD}(x[2], x[i])$
4: **end for**

5: **return** $x[2] == 1$

这是本书中第一次出现伪代码，代码是直接能编译的，伪代码是用于描述算法的，可以大大缩减篇幅和提高可读性，基本含义在单词中都有体现，在伪代码里以缩进表示层次结构。

练习 2.1.1. 我们定义横纵坐标均为整数的点为“格点”，给定平面上的两个格点 $P_1 = (x_1, y_1)$ 和 $P_2 = (x_2, y_2)$，求线段 P_1P_2 上除端点外的格点数。($-10^9 \leqslant x_1, x_2, y_1, y_2 \leqslant 10^9$)

根据最大公约数的定义可以知道，一定存在整数对 (x, y) 使得 $ax + by = \gcd(a, b)$ 成立，下面引入扩展欧几里德算法求这对整数。事实上，假设已经求得了一对 (x', y')，满足

$$bx' + (a \bmod b)y' = \gcd(a, b)$$

那么代入 $a \bmod b = a - \lfloor a/b \rfloor \times b$（除法是向下取整）可得

$$ay' + b(x' - \lfloor a/b \rfloor \times y') = \gcd(a, b)$$

于是我们得到了递归的扩展欧几里德算法，伪代码如下：

Algorithm 2.1 Extended-Euclid 算法

```
1: function EXTENDED-EUCLID(a, b)
2:     if b == 0 then
3:         return (a, 1, 0)
4:     else
5:         (d', x', y') ← EXTENDED-EUCLID(b, a mod b)
6:         return (d', y', x' − ⌊a/b⌋y')
7:     end if
8: end function
```

伪代码中返回了三元组，但 C++ 函数只能返回一个值，要返回两个可以用 `std::pair<>`，三个以上就要自己写结构体了，不过我们也可以通过引用解决这个问题，参考代码如下：

代码 2.1.2: 扩展欧几里德算法

```
1 int extgcd(int a, int b, int &x, int &y) {
2     int d = a;
3     if (b) {
4         d = extgcd(b, a % b, y, x);
5         y -= (a / b) * x;
6     }
7     else {
8         x = 1;
```

```
9           y = 0;
10      }
11      return d;
12  }
```

注意 $b = 0$ 时由于 $a \times 1 + b \times 0 = a$ 需要特判，函数返回 $\gcd(a, b)$。该函数递归的方式与 gcd 相似，复杂度相同，可以证明返回结果满足 $|x| \leqslant b$ 且 $|y| \leqslant a$。

练习 2.1.2. 双六是一种桌上游戏。一个双六上面有向前向后无限延续的格子，每个格子上都写有整数。其中 0 号格子是起点，1 号格子是终点。有这样一种骰子，它只有 $a, b, -a, -b$ 四个整数，所以根据 a 和 b 的值的不同，有可能无法到达终点。

$\cdots$	-4	-3	-2	-1	0	1	2	3	4	$\cdots$

图 2.1: 双六

给定 a 和 b $(1 \leqslant a, b \leqslant 10^9)$，掷出四个整数各多少次可以到达终点呢？如果解不唯一，输出任意一组，如果无解，输出 -1。

样例输入：

```
4 11
```

样例输出：

```
3 0 0 1
```

¶ 2.2 同余与模算术

我们将 a 除以 m 所得的余数记作 $a \bmod m$，规定 $0 \leqslant a \bmod m \leqslant m - 1$，注意 C++ 里的`%`运算符略微不同，$m$ 为负时允许返回负数，但 m 不能为 0。我们将 a 和 b 除以 m 后所得的余数相等记作 $a \equiv b \pmod m$，它充要于 "$a - b$ 可以整除 n"。需要特别注意模运算之前的运算有越界的可能，中间计算建议使用 `long long` 类型。

2.2.1 基本的模运算

模运算存在下面的公式：

$$(a + b) \bmod n = ((a \bmod n) + (b \bmod n)) \bmod n$$
$$(a - b) \bmod n = ((a \bmod n) - (b \bmod n) + n) \bmod n$$
$$ab \bmod n = (a \bmod n)(b \bmod n) \bmod n$$

假设 $a \equiv c \pmod m$ 和 $b \equiv d \pmod m$ 成立，那么有模运算律：

$$\begin{aligned} a+b &\equiv c+d \pmod m \\ a-b &\equiv c-d \pmod m \\ a\times b &\equiv c\times d \pmod m \end{aligned}$$

练习 2.2.1. 输入正整数 n 和 m，输出 $n \bmod m$，其中 $n \leqslant 10^{100}$，$m \leqslant 10^9$。

练习 2.2.2. 输入正整数 a、n 和 m，输出 $a^n \bmod m$，其中 $a, n, m \leqslant 10^9$。分析其复杂度。

2.2.2 快速幂运算

练习2.2.2也称为模取幂，是许多素数测试程序和 RSA 公钥加密系统中的基本运算，为了优雅地解决它，人们考虑用“反复平方”的方式进行优化。事实上，对于要求的幂 x^n，当 n 为偶数时有 $x^n = (x^2)^{n/2}$，当 n 为奇数时有 $x^n = (x^2)^{n/2} \times x$，都可以转换至 $n/2$ 的情况，故称之为“反复平方”法。算法描述如下：

Algorithm 2.2 快速幂运算-反复平方法

Require: the binary representation of b - $<b_k, b_{k-1}, \cdots, b_0>$
Ensure: $n \neq 0$

```
1: function MODULAR-EXPONENTIATION(a, b, n)                ▷ 计算 a^b mod n
2:     d ← 1
3:     for i ← 0 to k do
4:         if b_i == 1 then
5:             d ← (d · a) mod n
6:         end if
7:         a ← a^2 mod n
8:     end for
9:     return d
10: end function
```

以前学过的进制转换告诉我们，假设 $b_{c_i}(1 \leqslant i \leqslant j)$ 值为 1，则有 $b = 2^{c_1} + 2^{c_2} + \cdots + 2^{c_j}$，也即 $a^b = a^{2^{c_1}} a^{2^{c_2}} \cdots a^{2^{c_j}}$。反复平方法就是就是一直把 a 平方，当 b_k 值为 1 时将上一步的结果，也就是 a^{2^k} 提取到结果里。

C++ 的位运算规则与 C 相同，可以将 b 逐位右移，b_i 就是每一时刻的末位，可以用逻辑与（`&`）提取出来，终止条件可以通过 $b > 0$ 判定。

练习 2.2.3. 我们把对任意的 $1 < x < n$ 都有 $x^n \equiv x \pmod n$ 成立的合数 n 称为 *Carmichael*

Number。对于给定的整数 n，判断它是不是 *Carmichael Number*。其中 $2 < n < 65000$。

2.2.3 模线性方程（组）

现在来考虑求解下面方程的问题（其中 $a, n > 0$）：

$$ax \equiv b \pmod{n}$$

对于上面方程，当且仅当 $d \mid b$ 时它对于未知量 x 有解且有 d 个不同的解，这里 $d = \gcd(a, n)$。

定理 2.2.1. 令 $d = \gcd(a, n)$，假设对某些整数 x' 和 y'，有 $d = ax' + ny'$，如果 $d \mid b$，则方程 $ax \equiv b \pmod{n}$ 有一个解的值为 x_0，这里 $x_0 = x'(b/d) \bmod n$。

证明. 由于 $ax' \equiv d \pmod{n}$，有

$$\begin{aligned} ax_0 &\equiv ax'(b/d) \pmod{n} \\ &\equiv d(b/d) \pmod{n} \\ &\equiv b \pmod{n} \end{aligned}$$

因此 x_0 是 $ax \equiv b \pmod{n}$ 的一个解。 ■

定理 2.2.2. 假设方程 $ax \equiv b \pmod{n}$ 有解（即 $d \mid b$，这里 $d = \gcd(a, n)$），且 x_0 是该方程任意一个解，那么该方程对模 n 恰有 d 个不同的解，分别为 $x_i = x_0 + i(n/d)$，这里 $i = 0, 1, \cdots, d-1$，也即方程的解 $x \equiv x_0 \pmod{n/d}$。

证明. 因为 $n/d > 0$ 并且对于 $i = 0, 1, \cdots, d-1$，有 $0 \leqslant i(n/d) < n$，所以对模 n，值 $x_0, x_1, \cdots, x_{d-1}$ 都是不相同的。因为 x_0 是 $ax \equiv b \pmod{n}$ 的一个解，故有 $ax_0 \bmod n \equiv b \pmod{n}$，因此，对 $i = 0, 1, \cdots, d-1$，有

$$\begin{aligned} ax_i \bmod n &= a(x_0 + i(n/d)) \bmod n \\ &= (ax_0 + ain/d) \bmod n \\ &= ax_0 \bmod n \\ &\equiv b \pmod{n} \end{aligned}$$

那么 $ax_i \equiv b \pmod{n}$，则 x_i 也是一个解，由于方程 $ax \equiv b \pmod{n}$ 恰有 d 个解，因此 $x_0, x_1, \cdots, x_{d-1}$ 是方程的全部解。 ■

现在已经为求解方程 $ax \equiv b \pmod{n}$ 完成了数学上的必要准备，下面算法求解该方程，它返回二元组 (x_0, d)。

Algorithm 2.3 模线性方程求解

Ensure: $a, n \in \mathbb{Z}^+$, $b \in \mathbb{Z}$

```
function MODULAR-LINEAR-EQUATION-SOLVER(a, b, n)
    (d, x', y') ← EXTENDED-EUCLID(a, n)
    if d | b then
        return (x'⌊b/d⌋ mod ⌊n/d⌋, d)
    else
        print "No solutions."
    end if
end function
```

如果方程 $ax \equiv 1 \pmod{n}$ 有解，那么我们定义 x 为 a 对模 n 的**乘法逆元**，记作 a^{-1}，逆元有和倒数相似的性质。根据定理2.2.2，当且仅当 $\gcd(a, n) = 1$ 时方程 $ax \equiv 1 \pmod{n}$ 有唯一解，否则无解。这个问题相当于算法2.3中 $b = 1$ 的情况。

事实上，由于

$$ax \equiv 1 \pmod{n} \Leftrightarrow \gcd(a, n) = 1 = ax + ny$$

运用扩展欧几里德算法即可高效地得到 a 对模 n 的乘法逆元。

例题 2.2.1 (NOIP 2012 提高组 Day 2)**.** 求关于 x 的同余方程 $ax \equiv 1 \pmod{b}$ 的最小正整数解。其中 $2 \leqslant a, b \leqslant 2,000,000,000$。

输入只有一行，包含两个正整数 a，b，用一个空格隔开。

输出只有一行，包含一个正整数 x_0，即最小正整数解。输入数据保证一定有解。

样例输入:

```
3 10
```

样例输出:

```
7
```

题目分析:

没学过数论这题估计药丸，学过数论应该感到这是 *2012* 年最水的一题，直接求就好了，但是注意要求的是最小正整数解，扩展欧几里德算法可能求出负值，直接返回 x 只能通过四个测试点，需要进行处理。简单的已经考过，估计下次看到考数论的题，外面肯定包装了好几层了...

参考代码:

```
1 int mod_inverse(int a, int n) {
2     int x, y;
3     extgcd(a, n, x, y);
```

```
4 |     return (n + x % n) % n;
5 | }
```

练习 2.2.4 (C Looooops). 给定一个 C 风格的循环语句：

```
for (variable = A; variable != B; variable += C)
    statement;
```

编译器想知道对于 k 位无符号整数类型（$0 \leqslant varible < 2^k$）的一组 A，B 和 C，循环体将执行多少次?

输入四个整数 A、B、C、k，其中 $1 \leqslant k \leqslant 32$，$0 \leqslant A, B, C < 2^k$，输出循环体执行次数，如果给定循环是死循环，输出 FOREVER。

样例输入：

```
3 7 2 16
```

样例输出：

```
2
```

求解模线性方程相当于把形如 $ax \equiv b \pmod{n}$ 的方程转化到形如 $x \equiv x_0 \pmod{n}$ 的方程，那么对于模线性方程组如何求呢，是否可以把一组方程也合并成这种形式? 为了解决这个问题，首先介绍著名的中国剩余定理（又称孙子定理）。

定理 2.2.3 (中国剩余定理). 令 $n = n_1 n_2 \cdots n_k$，其中因子 n_i 两两互斥，记 $a_i = a \bmod n_i$，则存在双射

$$a \leftrightarrow (a_1, a_2, \cdots, a_k)$$

也就是说，如果 $a \leftrightarrow (a_1, a_2, \cdots, a_k)$ 且 $b \leftrightarrow (b_1, b_2, \cdots, b_k)$，那么

$$\begin{aligned}(a+b) \bmod n &\leftrightarrow ((a_1+b_1) \bmod n_1, \cdots, (a_k+b_k) \bmod n_k)\\ (a-b) \bmod n &\leftrightarrow ((a_1-b_1) \bmod n_1, \cdots, (a_k-b_k) \bmod n_k)\\ (ab) \bmod n &\leftrightarrow ((a_1 b_1) \bmod n_1, \cdots, (a_k b_k) \bmod n_k)\end{aligned}$$

证明. 从 a 转换为 $(a_1, a_2, \cdots, a_k)$ 执行 k 次模运算即可，下面进行逆向转换。

首先定义 $m_i = n/n_i$（$i = 1, 2, \cdots, k$，下同），于是 m_i 是除了 n_i 外的所有 n_j 的乘积。再定义

$$c_i = m_i (m_i^{-1} \bmod n_i) \tag{2.1}$$

由于 m_i 和 n_i 互质，所以 $m_i^{-1} \bmod n_i$ 保证存在，所以式子2.1总是良定义的。据此可以得到计算 a 的方式：

$$a \equiv (a_1 c_1 + a_2 c_2 + \cdots + a_k c_k) \pmod{n} \tag{2.2}$$

下证等式2.2能保证 $a \equiv a_i \pmod{n_i}$。注意到，如果 $j \neq i$，则 $m_j \equiv 0 \pmod{n_i}$，这意味着 $c_j \equiv m_j \equiv 0 \pmod{n_i}$。另外，由等式2.1知，$c_i \equiv 1 \pmod{n_i}$。因此得到一组对应关系：

$$c_i \leftrightarrow (0, 0, \cdots, 0, 1, 0, \cdots, 0)$$

这是一个除了第 i 个坐标为 1 外坐标均为 0 的向量，相当于这种表示的 “基”。对所有 i，有

$$\begin{aligned} a &\equiv a_i c_i \pmod{n_i} \\ &\equiv a_i m_i (m_i^{-1} \bmod n_i) \pmod{n_i} \\ &\equiv a_i \pmod{n_i} \end{aligned}$$

因此逆向变换依然保证 $a \equiv a_i \pmod{n_i}$，所以这种关系是一一对应。
再根据 $x \bmod n_i = (x \bmod n) \bmod n_i$ 和模运算规则可直接推出定理中的公式组。 ■

推论 2.2.3.1. 如果 $n_1, n_2, \cdots, n_k$ 两两互质，且 $n = n_1 n_2 \cdots n_k$，则对任意整数 $a_1, a_2, \cdots, a_k$，关于未知量 x 的联立方程组 $x \equiv a_i \pmod{n_i}$（其中 $i = 1, 2, \cdots, k$）对模 n 有唯一解。

推论 2.2.3.2. 如果 $n_1, n_2, \cdots, n_k$ 两两互质，且 $n = n_1 n_2 \cdots n_k$，则对所有整数 x 和 a，$x \equiv a \pmod{n_i}$（其中 $i = 1, 2, \cdots, k$）当且仅当 $x \equiv a \pmod{n}$。

中国南北朝时期（公元 5 世纪）的数学著作《孙子算经》卷下第二十六题，叫做 “物不知数”，原文是 “有物不知其数，三三数之剩二，五五数之剩三，七七数之剩二。问物几何？”。下面以此为例直观地理解一下中国剩余定理：

首先有线性同余方程组 $\begin{cases} x \equiv 2 \pmod{3} \\ x \equiv 3 \pmod{5} \\ x \equiv 2 \pmod{7} \end{cases}$，所有 n_j 互质，则 $n = 3 \times 5 \times 7 = 105$。

那么 $m_1 = 35$，$m_1^{-1} \pmod{3} = 2$，$c_1 = 70 = 2 \times 35 \equiv \begin{cases} 1 \pmod{3} \\ 0 \pmod{5} \\ 0 \pmod{7} \end{cases}$

$m_2 = 21$，$m_2^{-1} \pmod{5} = 1$，$c_2 = 21 = 1 \times 21 \equiv \begin{cases} 0 \pmod{3} \\ 1 \pmod{5} \\ 0 \pmod{7} \end{cases}$

$m_3 = 15$，$m_3^{-1} \pmod{7} = 1$，$c_3 = 15 = 1 \times 15 \equiv \begin{cases} 0 \pmod{3} \\ 0 \pmod{5} \\ 1 \pmod{7} \end{cases}$

最终得到 $233 = 2 \times 70 + 3 \times 21 + 2 \times 15 \equiv \begin{cases} 2 \times 1 + 3 \times 0 + 2 \times 0 \equiv 2 \pmod 3 \\ 2 \times 0 + 3 \times 1 + 2 \times 0 \equiv 3 \pmod 5 \\ 2 \times 0 + 3 \times 0 + 2 \times 1 \equiv 2 \pmod 7 \end{cases}$

于是通解为 $x = 233 + k \times 105,\ k \in \mathbb{Z}$.

上述算法已经可以解模线性方程组 $\begin{cases} x \equiv b_1 \pmod{n_1} \\ x \equiv b_2 \pmod{n_2} \\ \quad\vdots \\ x \equiv b_k \pmod{n_k} \end{cases}$，但操作比较麻烦。其实可以递推求解，记方程组解为 $x_0 \equiv b_0 \pmod{n_0}$，由 $\begin{cases} x_0 = b_0 + yn_0 \equiv x_i \pmod{n_i} \\ x_0 \equiv b_i \pmod{n_i} \end{cases}$ 联立得：

$$n_0 y \equiv (b_i - b_0) \pmod{n_i}$$

如果这个方程无解（$d \nmid (b_i - b_0)$）则方程组无解，反之更新 b_0 和 n_0，算法如下：

Algorithm 2.4 模线性方程组求解

Require: array $(b_1, b_2, \cdots, b_k)$ and $(n_1, n_2, \cdots, n_k)$

```
function MODULAR-LINEAR-EQUATION-GROUP-SOLVER
    b ← b_1, n ← n_1
    for i ← 2 to k do
        (y, d) ← MODULAR-LINEAR-EQUATION-SOLVER(n, b_i − b, n_i)
        b ← b + n · y
        n ← n · n_i/d
    end for
    return (b, n)
end function
```

练习 2.2.5. 选择 k 个不同的正整数 $a_1, a_2, \cdots, a_k$，对于一个非负整数 m，可以得到一组 $r_1, r_2, \cdots, r_k$，其中 $r_i = m \bmod a_i$。给定 k 组 (a_i, r_i)，求该组表示法能确定的最小正整数 m，如果不存在这样的 m，输出 -1。

样例输入：

```
2
8 7
11 9
```

样例输出：

```
31
```

¶ 2.3 素数基础算法

判断一个数是否是素数（质数）可以用试除法。如果 d 是 n 的约数，那么 n/d 也是 n 的约数（$n = d \times n/d$）。因此 $\min(d, n/d) \leqslant \sqrt{n}$，所以试除法只要检查 $2 \sim \sqrt{n}$ 的所有整数就够了。复杂度 $O(\sqrt{n})$，还可以接受。

虽然上述算法已经可以满足许多需求，但如果素性测试过于频繁，则有更高效的算法——Eratosthenes 筛法。先把 2 到 n 范围内的所有整数写下来，其中最小的数字 2 是素数，将表中所有 2 的倍数划掉，剩下最小的数字 3 也是素数，再把所有 3 的倍数划掉，反复操作，就能枚举出 n 以内的所有素数。算法如下：

Algorithm 2.5 Eratosthenes 筛法

Require: boolean array $x[2...n]$ initialized by true while $x[0]$ and $x[1]$ is false.

```
1:  procedure SIEVE(n)                                  ▷ 筛选 n 以内素数
2:      for i ← 2 to √n do
3:          if x[i] is true then
4:              j ← i^2
5:              repeat
6:                  x[j] ← false
7:                  j ← j + i
8:              until j > n
9:          end if
10:     end for
11: end procedure
```

筛选法复杂度仅为 $O(n \log\log n)$，非常可以接受。根据素数定理：

$$\pi(x) \sim \frac{x}{\ln x}$$

可以大致判断出来 n 以内有多少个素数，其中 $\pi(x)$ 表示不超过 x 的素数的个数。筛选法是典型的空间换时间，假设我们只需要得到区间 $[a, b)$ 内的素数且 a 和 b 很大，那么算法需要优化一下，如前所述，b 以内合数的最小质因数一定不超过 $\sqrt{b}$，如果我们有 $\sqrt{b}$ 以内的素数表的话，就可以把筛选法应用在 $[a, b)$ 上了，也即在 $[2, \sqrt{b})$ 中筛选的同时，也把倍数从 $[a, b)$ 的表中划去。

练习 2.3.1. 给出正整数 n 和 m，问区间 $[n, m]$ 内的“无平方因子”的数有多少个？整数 p 无平方因子，当且仅当不存在 $k > 1$，使得 p 是 k^2 的倍数。其中 $1 \leqslant n \leqslant m \leqslant 10^{12}$，$m - n \leqslant 10^7$。

¶ 2.4 高精度计算

高精度计算应该算初学者的一大难关，我也不想写其实，它相比于后面章节的内容，看起来技术含量没太高，但是对编程的基本功要求很高，非常非常容易出错，稍不小心就容易崩溃掉。高精度也可以当做一次 C++ 实战，毕竟用 C 写高精度要麻烦好多倍...

高精度计算的核心思想就是小学学的竖式计算，核心算法是进位处理，加减按位加减，乘法“卷积”，除法试除。写高精度计算应该兼顾执行效率和编程效率，尽可能用最快的速度写最高效的算法。同时应该根据题目写需要的高精度，比如题目中只可能出现正数、没有用到除法，等等。下面给出参考代码，`Integer` 是用法与 `int` 相似的大整数类，它支持正负数。

代码 2.4.1: 高精度示例

```
#include <iostream>
#include <string>
#include <sstream>
#include <iomanip>
#include <vector>

struct Integer {
    static const int SCALE      = 10000;
    static const int SCALELEN   = 4;
    std::vector<int> data;
    bool negative;
    Integer(): negative(false), data(1, 0) {}
    Integer(int n): negative(n < 0) {
        if (negative) n = -n;
        if (!n)
            data.push_back(0);
        else while (n) {
            data.push_back(n % SCALE);
            n /= SCALE;
        }
    }
    Integer(const std::string &s): negative(s[0] == '-') {
        int end = (int)s.length(), front = negative ? 1 : 0;
        for (int j = end; j > front; j -= SCALELEN) {
            int i = j - SCALELEN < front ? front : j - SCALELEN;
            data.push_back(std::stoi(s.substr(i, j - i)));
        }
    }
    Integer(const Integer &r): data(r.data), negative(r.negative)
        {}
```

```
30      Integer(std::vector<int>::const_iterator l, std::vector<int
           >::const_iterator r): data(l, r), negative(false) {
31          // assert(l < r);
32          trim();
33      }
34      inline Integer &operator=(const Integer & rhs) {
35          data = rhs.data;
36          negative = rhs.negative;
37          return *this;
38      }
39      inline operator std::string() const {
40          std::stringstream ss;
41          if (negative)
42              ss << '-';
43          ss << data.back();
44          for (int i = (int)data.size() - 2; i >= 0; i--)
45              ss << std::setfill('0') << std::setw(SCALELEN) <<
                   data.at(i);
46          return ss.str();
47      }
48      inline Integer operator-() const {
49          Integer ans = *this;
50          ans.negative = !ans.negative;
51          return ans;
52      }
53      inline Integer operator+() const {
54          Integer ans = *this;
55          ans.negative = false;
56          return ans;
57      }
58      inline Integer operator+(const Integer & rhs) const {
59          if (negative != rhs.negative)
60              return *this - (-rhs);
61          Integer ans = *this;
62          for (int j = 0; j < rhs.data.size(); j++) {
63              if (j < ans.data.size())
64                  ans.data.at(j) += rhs.data.at(j);
65              else ans.data.push_back(rhs.data.at(j));
66          }
67          ans.carry();
68          return ans;
69      }
70      inline Integer &operator+=(const Integer & rhs) {
71          *this = *this + rhs;
72          return *this;
```

```
73      }
74      inline Integer &operator++() {
75          data.front()++;
76          carry();
77          return *this;
78      }
79      inline Integer operator++(int) {
80          Integer temp(*this);
81          ++*this;
82          return temp;
83      }
84      inline Integer operator-(const Integer & rhs) const {
85          if (negative != rhs.negative)
86              return *this + (-rhs);
87          if (negative)
88              return +rhs + *this;
89          if (*this < rhs)
90              return -(rhs - *this);
91          Integer ans = *this;
92          for (int j = 0; j < rhs.data.size(); j++)
93              if (j < data.size())
94                  ans.data.at(j) -= rhs.data.at(j);
95          ans.carry();
96          ans.trim();
97          return negative ? +ans : ans;
98      }
99      inline Integer &operator-=(const Integer & rhs) {
100         *this = *this - rhs;
101         return *this;
102     }
103     inline Integer operator*(const Integer & rhs) const {
104         Integer ans;
105         ans.negative = negative != rhs.negative;
106         ans.data.resize(data.size() + rhs.data.size(), 0);
107         for (int i = 0; i < data.size(); i++) {
108             for (int j = 0; j < rhs.data.size(); j++)
109                 ans.data.at(i + j) += data.at(i) * rhs.data.at(j)
                        ;
110             ans.carry(i);
111         }
112         ans.trim();
113         return ans;
114     }
115     inline Integer &operator*=(const Integer & rhs) {
116         *this = *this * rhs;
```

```
117         return *this;
118     }
119     inline Integer operator/(const Integer & rhs) const {
120         // assert(rhs != 0);
121         Integer lhs = +*this, trhs = +rhs;
122         if (lhs < trhs) return (Integer)0;
123         Integer ans = divHelper(lhs, trhs);
124         ans.negative = ans.data.back() != 0 && negative != rhs.
               negative;
125         return ans;
126     }
127     inline Integer &operator/=(const Integer & rhs) {
128         *this = *this / rhs;
129         return *this;
130     }
131     inline Integer operator%(const Integer & rhs) const {
132         Integer ans, lhs = +*this, trhs = +rhs;
133         if (lhs < trhs) return *this;
134         divHelper(lhs, trhs);
135         return lhs;
136     }
137     inline Integer &operator%=(const Integer & rhs) {
138         *this = *this % rhs;
139         return *this;
140     }
141     inline bool operator<(const Integer & rhs) const {
142         if (negative != rhs.negative)
143             return negative > rhs.negative;
144         int size = (int)data.size();
145         if (size != rhs.data.size())
146             return negative ? size > rhs.data.size() : size < rhs
                   .data.size();
147         int index;
148         for (index = size - 1; index >= 0 && data.at(index) ==
               rhs.data.at(index); index--)
149             ;
150         return index != -1 && (negative ? data.at(index) > rhs.
               data.at(index) : data.at(index) < rhs.data.at(index))
               ;
151     }
152 private:
153     inline void carry(int pos = 0) {
154         int size = (int)data.size();
155         for (int i = pos; i < size - 1; i++) {
156             if (data.at(i) >= SCALE) {
```

```
157                 data.at(i + 1) += data.at(i) / SCALE;
158                 data.at(i) %= SCALE;
159             } else if (data.at(i) < 0) {
160                 data.at(i) += (-data.at(i) / SCALE + 1) * SCALE;
161                 data.at(i + 1) -= -data.at(i) / SCALE + 1;
162             }
163         }
164         if (data.back() >= SCALE) {
165             data.push_back(data.back() / SCALE);
166             data.at(size - 1) %= SCALE;
167         }
168     }
169     inline int trim(int pos = 0) {
170         int ans = 0;
171         for (int i = (int)data.size() - 1; i > pos && data.at(i)
               == 0; i--, ans++)
172             data.pop_back();
173         if (data.back() == 0)
174             negative = false;
175         return ans;
176     }
177     static inline Integer divHelper(Integer & lhs, const Integer
           & rhs) {
178         Integer ans;
179         int it = int(lhs.data.size() - rhs.data.size());
180         while (it >= 0) {
181             ans *= SCALE;
182             int times = lhs.data.back() / rhs.data.back();
183             Integer temp(lhs.data.begin() + it, lhs.data.end());
184             while (temp - rhs * times < 0)
185                 times--;
186             while (!(temp - rhs * times < rhs))
187                 times++;
188             temp = rhs * times;
189             std::vector<int>::iterator jt = lhs.data.begin() + it
                   ;
190             for (int j = 0; j < temp.data.size(); j++, jt++)
191                 *jt -= temp.data.at(j);
192             ans += times;
193             lhs.carry(it);
194             lhs.trim(it);
195             it--;
196         }
197         return ans;
198     }
```

```
199 | };
```

在我自己的机器上测试，1K 位以内的所有运算都可以迅速跑出来，应对竞赛足够用了。重要的事情说三遍：不要抄代码！不要抄代码！不要抄代码！下面分几个角度作简单阐释，具体的可以当面交流，如果发现代码有 bug 请务必指出！

存储 存储方式有很多种，一种是直接存成字符串，好处是输入输出极其简单，坏处是运算起来非常麻烦而且转换来转换去效率低下；一种是十进制按位存储，好处是输入输出、计算都很自然，坏处是存储消耗大以及运算效率低（你可以通过更改 `SCALE` 和 `SCALELEN` 至 10 和 1 来对比运行效率）；还有一种是十六进制存储，这样最大限度利用存储空间和提升效率（因为从底层来看位运算比数值运算高速得多），虽然输入输出很麻烦和耗时。综合来看，建议采用万进制按位存储。

构造和析构 `std::vector<Ty>` 还是值得信赖的，所以我们的核心数组用它实现，这样就不用手动析构了。而且它对于进位等处理非常灵活。注意构造函数中使用了 `std::stoi()` 和 `substr()` 等函数操作字符串。

进位 网上很多代码都是把进位写在加减乘运算里的，其实划分出来更清晰一些，参见 `carry()` 函数；低位如果溢出数制则进位，低于 0 则借位，注意最高位需要特殊判断一下。

比较 参见小于号运算符重载，如果两数符号不同、位数不等可以直接得出结论，否则自高位向低位比较，有不相等即分出高下。

加减 逐位加减即可，运算完一次进位、一次去除前导零（对于减法）。如果需要处理正负数，我们令加加号只处理同号数字，异号则交由减法，令减号只处理同号正数且左操作数不小于右操作数，异号或同号负数交给加法，左操作数小则交换运算次序，从而避免繁冗的逻辑和代码。

乘法 模拟竖式乘法，注意形如 $9999\,9999 \times 9999\,9999$ 的算式由于卷积的累积，中间数值可能越界，所以每一趟都进一次位。乘法有可能得出 0，所以也需要去除前导零。

除法 除法算是最麻烦的高精度了，这里还是按照竖式除法的试除来写，注意在每次试除时，平凡的做法是一次次调用减法，但对于 $9999\,9999 \,/\, 1$ 这样的算式而言效率低下，这里先求最高位倍数再做微调，可以提升速度。

取模 取模是在除法的同时完成的，所以除法写在了 `divHelper()` 函数里。可以通过返回值或传进去的引用分别得到商和余数。

const 限定符 代码里给出了 `const` 限定符的标准用法，我的建议是，要么都用，要么完全

不用，用错了的 `const` 会一直不让你通过编译。

其他 `inline` 说明符建议编译器将函数内联化编译，普通函数只有一套二进制代码，在每个调用点都要压入内存然后执行，内联的函数在编译期直接在调用点展开，效率更高，但编译器有权忽略此项建议。`private` 限定符限制在类/结构外不能调用其后函数，事实上我们也不希望用户去调用后几个函数。

练习 2.4.1 (NOIP 2012 提高组 Day 1)**.** 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。

输入多行，第一行包含一个整数 n，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。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。

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。

样例输入：

```
3
1 1
2 3
7 4
4 6
```

样例输出：

```
2
```

¶ 2.5 小结

后续还会有其他数值计算问题的介绍，强烈建议编程到 OJ 上验证结果，事先根据题目的数据范围估计一下 `int`、`long long` 等是否足以容纳输入输出和中间值，一定要谨慎。另外，伪代码的表示可能会引起一段时间的不适，毕竟存在“翻译”成自己的理解再“翻译”成 C++ 代码的过程，事实上这样的过程走了几次之后相应的算法也就记住了，加油吧！

¶ 2.6 参考习题

习题 2.1. 阅读代码写结果：

```
#include <cstdio>
#include <cstdlib>

int a[50];

void work(int p, int r) {
    if (p < r) {
        int i = p - 1, j, temp;
        for (j = p; j < r; j++)
            if (a[j] >= a[r]) {
                i++;
                temp = a[i]; a[i] = a[j]; a[j] = temp;
            }
        temp = a[i + 1]; a[i + 1] = a[r]; a[r] = temp;
        work(p, i);
        work(i + 2, r);
    }
}

int main() {
    int n, i, sum = 0;
    scanf("%d", &n);
    for (i = 0; i < n; i++)
        scanf("%d", &(a[i]));
    work(0, n - 1);
    for (i = 0; i < n - 1; i++)
        sum += abs(a[i + 1] - a[i]);
    printf("%d\n", sum);
    return 0;
}
```

输入：`10 23 435 12 345 3123 43 456 12 32 -100`

输出：_____

习题 2.2. 阅读代码写结果：

```
#include<cstdio>
#include<cstring>

int main() {
    char str[60];
```

```
6       int len, i, j, chr[26];
7       char mmin = 'z';
8       scanf("%s", str);
9       len = (int)strlen(str);
10      for (i = len - 1; i >= 1; i--)
11          if (str[i - 1] < str[i])
12              break;
13      if (i == 0) {
14          printf("No result!\n");
15          return 0;
16      }
17      for (j = 0; j < i - 1; j++)
18          putchar(str[j]);
19      memset(chr, 0, sizeof(chr));
20      for (j = i; j < len; j++) {
21          if (str[j] > str[i - 1] && str[j] < mmin)
22              mmin = str[j];
23          chr[str[j] - 'a']++;
24      }
25      chr[mmin - 'a']--;
26      chr[str[i - 1] - 'a']++;
27      putchar(mmin);
28      for(i = 0; i < 26; i++)
29          for(j = 0; j < chr[i]; j++)
30              putchar(i + 'a');
31      putchar('\n');
32      return 0;
33  }
```

输入：zzyzcccbbbaaa

输出：_____

习题 2.3. 有两个向量 $v_1 = (x_1, x_2, \cdots, x_n)$ 和 $v_2 = (y_1, y_2, \cdots, y_n)$，允许任意交换 v_1 和 v_2 各自的分量的顺序，试计算 v_1 和 v_2 的内积 $x_1y_1 + \cdots + x_ny_n$ 的最小值。

输入 3 行，第一行为向量维数 n，后两行每行 n 个整数，且 $1 \leqslant n \leqslant 800$，$-100000 \leqslant x_i, y_i \leqslant 100000$。输出所求值。

习题 2.4. 两只青蛙在网上相识了，它们聊得很开心，于是觉得很有必要见一面。它们很高兴地发现它们住在同一条纬度线上，于是它们约定各自朝西跳，直到碰面为止。可是它们出发之前忘记了一件很重要的事情，既没有问清楚对方的特征，也没有约定见面的具体位置。不过青蛙们都是很乐观的，它们觉得只要一直朝着某个方向跳下去，总能碰到对方的。但是除非这两只青蛙在同一时间跳到同一点上，不然是永远都不可能碰面的。为了帮助这两只乐观的青蛙，你被要求写一个程序来判断这两只青蛙是否能够碰面，会在什么时候碰面。

我们把这两只青蛙分别叫做青蛙 A 和青蛙 B，并且规定纬度线上东经 0 度处为原点，由东往西为正方向，单位长度 1 米，这样我们就得到了一条首尾相接的数轴。设青蛙 A 的出发点坐标是 x，青蛙 B 的出发点坐标是 y。青蛙 A 一次能跳 m 米，青蛙 B 一次能跳 n 米，两只青蛙跳一次所花费的时间相同。纬度线总长 L 米。现在要你求出它们跳了几次以后才会碰面。

输入只包括一行 5 个整数 x, y, m, n, L，其中 $x \neq y < 2000000000$，$0 < m, n < 2000000000$，$0 < L < 2100000000$。

输出碰面所需要的跳跃次数，如果永远不可能碰面则输出 `Impossible`。

样例输入：

```
1 2 3 4 5
```

样例输出：

```
4
```

习题 2.5 (X-factor Chains)**.** 对于一个正整数 X，其一个长度为 m 的因子链是这样一串整数：

$$1 = X_0, X_1, X_2, \cdots, X_m = X$$

并且满足 $X_i < X_{i+1}$ 且 $X_i \mid X_{i+1}$。

输入正整数 X ($X \leqslant 2^{20}$)，输出因子链最大链长以及最长因子链的条数。

样例输入：

```
10
```

样例输出：

```
2 2
```

List of Algorithms