

1 常用容器

1.1 字符串 string

1.1.1 常用方法及函数

(1) 构造

构造函数: `string(长度, 初值)`

```
string s1;           // 构造字符串, 为空
string s2 = "awa!";  // 构造字符串, 并赋值awa!
string s3(10, '6');  // 构造字符串, 通过构造函数构造为6666666666
```

(2) 输入输出

C++

```
string s;
cin >> s;
cout << s;
# 希望保留空格, 可以使用getline函数
getline(cin, s1);
# 格式化输入函数sscanf()
sscanf(s, "%d %d %d", &year, &month, &day);
```

C

```
string s;
char buf[100];
scanf("%s", &buf);
s = buf;
printf("%s", s.c_str());
```

(3) 查询字符串信息、索引

```
cout << s1.empty() << endl;  // s 为空返回 true, 否则返回 false
cout << s1.size() << endl;   // 返回 s 中字符个数, 不包含空字符
cout << s1.length() << endl;  // 作用同上
```

(4) 拼接、比较等操作

```
s1+s2           // 返回 s1 和 s2 拼接后的结果。加号两边至少有一个 string 对象, 不能都是
s1 == s2        // 如果 s1 和 s2 中的元素完全相等则它们相等, 区分大小写
<, <=, >, >=   // 利用字符的字典序进行比较, 区分大小写
```

(5) ctype 头文件(判断字符类型：大/小写字母、标点、数字等)

```
isalnum(c) // 当是字母或数字时为真
isalpha(c) // 当是字母时为真
isdigit(c) // 当是数字是为真
islower(c) // 当是小写字母时为真
isupper(c) // 当是大写字母时为真
isspace(c) // 当是空白（空格、回车、换行、制表符等）时为真
isxdigit(c) // 当是16进制数字是为真
ispunct(c) // 当是标点符号时为真（即c不是 控制字符、数字、字母、可打印空白 中的一种）
isprint(c) // 当时可打印字符时为真（即c是空格或具有可见形式）
isgraph(c) // 当不是空格但可打印时为真
iscntrl(c) // 当是控制字符时为真
tolower(c) // 若c是大写字母，转换为小写输出，否则原样输出
toupper(c) // 类似上面的
```

(6) 修改 string 的操作

在 pos 之前插入 args 指定的字符。pos是一个下标或者迭代器。接受下标的版本返回一个指向 s 的引用；接受迭代器的版本返回一个指向第一个插入字符的迭代器

```
s.insert(pos, args)
// 在 s 的位置 0 之前插入 s2 的拷贝
s.insert(0, s2)
```

删除从 pos 开始的 len 个字符。如果 len 省略，则删除 pos 开始的后面所有字符。返回一个指向 s 的引用

```
s.erase(pos, len)
```

将 s 中的字符替换为 args 指定的字符。返回一个指向 s 的引用。

```
s.assign(args)
```

将 args 追加到 s。返回一个指向 s 的引用。args 不能是单引号字符，若是单个字符则必须用双引号表示。如，可以是 s.append("A") 但不能是 s.append('A')

```
s.append(args)
```

将 s 中范围为 range 内的字符替换为 args 指定的字符。range 或者是一个下标或长度，或者是一对指向 s 的迭代器。返回一个指向 s 的引用。

```
s.replace(range, args)
// 从位置 3 开始，删除 6 个字符，并插入 "aaa"。删除插入的字符数量不必相等
s.replace(3, 6, "aaa")
```

(7) string 搜索操作

搜索操作返回指定字符出现的下标，如果未找到返回 npos

```
s.find(args)    // 查找 s 中 args 第一次出现的位置  
s.rfind(args)   // 查找 s 中 args 最后一次出现的位置
```

在 s 中查找 args 中任何一个字符 最早/最晚 出现的位置

```
s.find_first_of(args) // 在 s 中查找 args 中任何一个字符最早出现的位置  
s.find_last_of(args)  // 在 s 中查找 args 中任何一个字符最晚出现的位置  
例如：  
string s1 = "nice to meet you~";  
cout << s1.find_first_of("mey") << endl; // 输出结果为 3, 'e' 出现的最早
```

在 s 中查找 第一个/最后一个 不在 args 中的字符的位置

```
s.find_first_not_of(args) // 查找 s 中 第一个不在 args 中的字符的位置  
s.find_last_not_of(args)  // 查找 s 中 最后一个不在 args 中的字符的位置  
例如：  
string s1 = "nice to meet you~";  
cout << s1.find_first_not_of("nop") << endl; // 输出结果为 1, 'i' 不在 "nop" 里
```

(8) 数值与字符串互转 (C++11)

源	目的	函数
int / long long / float / double / long double	string	to_string()
string	int	stoi()
string	long long	stoll()
string	float	stof()
string	double	stod()
string	long double	stold()

(9) 字符串反转

```
string s2 = "12345";    // 初始化一个字符串  
reverse(s2.begin(), s2.end()); // 反转 string 定义的字符串 s2  
cout << s2 << endl; // 输出 54321
```

(10) 提取字符串

从索引 pos 开始，提取连续的 n 个字符，包括 pos 位置的字符。

```
string ss = s.substr(pos, n)
```

(11) 大小写转换

通过STL的transform算法配合的toupper和tolower来实现该功能

```
transform(s.begin(),s.end(),s.begin(),::tolower);
```

1.1.2 注意事项

尾接字符串一定要用 +=

string 的 += 运算符，将会在原字符串原地尾接字符串。而 + 了再 = 赋值，会先生成一个临时变量，在复制给 string。

通常字符串长度可以很长，如果使用 + 字符串很容易就 TLE 了。

```
// 优化前: 15139ms
string s;
for (int i = 0; i < 5e5; i++)
    s = s + "a";

// 优化后: < 1ms (计时器显示0)
string s;
for (int i = 0; i < 5e5; i++)
    s += "a";
```

1.2 向量 [vector]

1.2.1 常用方法

构造

`vector<类型> arr(长度, [初值])`

时间复杂度: $O(n)$

常用的一维和二维数组构造示例，高维也是一样的（就是会有点长）。

```
vector<int> arr;           // 构造int数组
vector<int> arr(100);      // 构造初始长100的int数组
vector<int> arr(100, 1);   // 构造初始长100的int数组，初值为1
vector<vector<int>> mat(100, vector<int> ()); // 构造初始100行，不指定列数的二维数组
vector<vector<int>> mat(100, vector<int> (666, -1)); // 构造初始100行，初始666列的二维数组，初值为-1
```

构造二维数组的奇葩写法，千万别用：

```
vector<int> arr[100];      // 正确，构造初始100行，不指定列数的二维数组，可用于链式前向星存图
vector<int> arr[100](100, 1); // 语法错误！
vector<int> arr(100, 1)[100]; // 语法错误！
vector<int> arr[100] {{100, 1}, {100, 1}}; // 正确但奇葩，使用列表初始化
```

尾接 & 尾删

- `.push_back(元素)`: 在 vector 尾接一个元素, 数组长度 $+1$.
- `.pop_back()`: 删除 vector 尾部的一个元素, 数组长度 -1 .

时间复杂度: 均摊 $O(1)$

```
// init: arr = []
arr.push_back(1);
// after: arr = [1]
arr.push_back(2);
// after: arr = [1, 2]
arr.pop_back();
// after: arr = [1]
arr.pop_back();
// after: arr = []
```

中括号运算符

和一般数组一样的作用

时间复杂度: $O(1)$

获取长度

`.size()`

获取当前 vector 的长度

时间复杂度: $O(1)$

```
for (int i = 0; i < arr.size(); i++)
    cout << a[i] << endl;
```

清空

`.clear()`

清空 vector

时间复杂度: $O(n)$

判空

`.empty()`

如果是空返回 `true` 反之返回 `false`.

时间复杂度: $O(1)$

改变长度

`.resize(新长度, [默认值])`

修改 vector 的长度

- 如果是缩短, 则删除多余的值
- 如果是扩大, 且指定了默认值, 则新元素均为默认值 (旧元素不变)

时间复杂度： $O(n)$

1.2.2 适用情形

一般情况 `vector` 可以替换掉普通数组，除非该题卡常。

有些情况普通数组没法解决： $n \times m$ 的矩阵， $1 \leq n, m \leq 10^6$ 且 $n \times m \leq 10^6$

- 如果用普通数组 `int mat[1000010][1000010]`，浪费内存，会导致 MLE。
- 如果使用 `vector<vector<int>> mat(n + 10, vector<int> (m + 10))`，完美解决该问题。

另外，`vector` 的数据储存在堆空间中，不会爆栈。

1.2.3 注意事项

提前指定长度

如果长度已经确定，那么应当直接在构造函数指定长度，而不是一个一个 `.push_back()`。因为 `vector` 额外内存耗尽后的重分配是有时间开销的，直接指定长度就不会出现重分配了。

```
// 优化前：522ms
vector<int> a;
for (int i = 0; i < 1e8; i++)
    a.push_back(i);
// 优化后：259ms
vector<int> a(1e8);
for (int i = 0; i < a.size(); i++)
    a[i] = i;
```

当心 `size_t` 溢出

`vector` 获取长度的方法 `.size()` 返回值类型为 `size_t`，通常 OJ 平台使用的是 32 位编译器（有些平台例如 cf 可选 64 位），那么该类型范围为 $[0, 2^{32})$ 。

```
vector<int> a(65536);
long long a = a.size() * a.size(); // 直接溢出变成0了
```

1.3 栈 [stack](#)

1.3.1 常用方法

作用	用法	示例
构造	<code>stack<类型> stk</code>	<code>stack<int> stk;</code>
进栈	<code>.push(元素)</code>	<code>stk.push(1);</code>
出栈	<code>.pop()</code>	<code>stk.pop();</code>
取栈顶	<code>.top()</code>	<code>int a = stk.top();</code>
查看大小 / 清空 / 判空	略	略

1.3.2 适用情形

如果不卡常的话，就可以直接用它而不需要手写栈了。

另外，vector 也可以当栈用，vector 的 `.back()` 取尾部元素，就相当于取栈顶，`.push_back()` 相当于进栈，`.pop_back()` 相当于出栈。

1.3.3 注意事项

不可访问内部元素！下面都是错误用法

```
for (int i = 0; i < stk.size(); i++)
    cout << stk[i] << endl;
for (auto ele : stk)
    cout << stk << endl;
```

1.4 队列 [queue](#)

1.4.1 常用方法

作用	用法	示例
构造	<code>queue<类型> que</code>	<code>queue<int> que;</code>
进队	<code>.push(元素)</code>	<code>que.push(1);</code>
出队	<code>.pop()</code>	<code>que.pop();</code>
取队首	<code>.front()</code>	<code>int a = que.front();</code>
取队尾	<code>.back()</code>	<code>int a = que.back();</code>
查看大小 / 清空 / 判空	略	略

1.4.2 适用情形

如果不卡常的话，就可以直接用它而不需要手写队列了。

1.4.3 注意事项

不可访问内部元素！下面都是错误用法

```
for (int i = 0; i < que.size(); i++)
    cout << que[i] << endl;
for (auto ele : que)
    cout << ele << endl;
```

1.5 优先队列 [priority_queue](#)

```
#include <queue>
```

提供常数时间的最大元素查找，对数时间的插入与提取，底层原理是二叉堆。

1.5.1 常用方法

构造

```
priority_queue<类型, 容器, 比较器> pque
```

- 类型：要储存的数据类型
- 容器：储存数据的底层容器，默认为 `vector<类型>`，竞赛中保持默认即可
- 比较器：比较大小使用的比较器，默认为 `less<类型>`，可自定义

```
priority_queue<int> pque1; // 储存int的大顶堆
priority_queue<int, vector<int>, greater<int>> pque2; // 储存int的小顶堆
```

对于需要自定义比较器的情况，涉及一些初学时容易看迷糊的语法（重载小括号运算符 / lambda 表达式），在此就不展开讲了。如果想要了解，可以查阅 `cppreference` 中的代码示例。

其他

作用	用法	示例
进堆	<code>.push(元素)</code>	<code>que.push(1);</code>
出堆	<code>.pop()</code>	<code>que.pop();</code>
取堆顶	<code>.top()</code>	<code>int a = que.top();</code>
查看大小 / 判空	略	略

进出队复杂度 $O(\log n)$ ，取堆顶 $O(1)$ 。

1.5.2 适用情形

持续维护元素的有序性：每次向队列插入大小不定的元素，或者每次从队列里取出大小最小/最大的元素，元素数量 n ，插入操作数量 k 。

- 每次插入后进行快速排序： $k \cdot n \log n$
- 使用优先队列维护： $k \cdot \log n$

1.5.3 注意事项

仅堆顶可读

只可访问堆顶，其他元素都无法读取到。下面是错误用法：

```
cout << pque[1] << endl;
```

所有元素不可写

堆中所有元素是不可修改的。下面是错误用法：

```
pque[1] = 2;
pque.top() = 1;
```


如果你恰好要修改的是堆顶元素，那么是可以完成的：

```
int tp = pqe.top();
pqe.pop();
pqe.push(tp + 1);
```

1.6 集合 set

提供对数时间的插入、删除、查找的集合数据结构。底层原理是红黑树。

集合三要素	解释	set	multiset	unordered_set
确定性	一个元素要么在集合中，要么不在	✓	✓	✓
互异性	一个元素仅可以在集合中出现一次	✓	✗ (任意次)	✓
无序性	集合中的元素是没有顺序的	✗ (从小到大)	✗ (从小到大)	✓

1.6.1 常用方法

构造

`set<类型, 比较器> st`

- 类型：要储存的数据类型
- 比较器：比较大小使用的比较器，默认为 `less<类型>`，可自定义

```
set<int> st1; // 储存int的集合（从小到大）
set<int, greater<int>> st2; // 储存int的集合（从大到小）
```

对于需要自定义比较器的情况，涉及一些初学时容易看迷糊的语法（重载小括号运算符 / lambda 表达式），在此就不展开讲了。

遍历

可使用迭代器进行遍历：

```
for (set<int>::iterator it = st.begin(); it != st.end(); ++it)
    cout << *it << endl;
```

基于范围的循环（C++ 11）：

```
for (auto &ele : st)
    cout << ele << endl;
```

其他

作用	用法	示例
插入元素	<code>.insert(元素)</code>	<code>st.insert(1);</code>
删除元素	<code>.erase(元素)</code>	<code>st.erase(2);</code>
查找元素	<code>.find(元素)</code>	<code>auto it = st.find(1);</code>
判断元素是否存在	<code>.count(元素)</code>	<code>st.count(3);</code>
查看大小 / 清空 / 判空	略	略

增删查时间复杂度均为 $O(\log n)$

1.6.2 适用情形

- 元素去重: $[1,1,3,2,4,4] \rightarrow [1,2,3,4]$
- 维护顺序: $[1,5,3,7,9] \rightarrow [1,3,5,7,9]$
- 元素是否出现过: 元素大小 $[-10^{18}, 10^{18}]$, 元素数量 10^6 , vis 数组无法实现, 通过 set 可以完成。

1.6.3 注意事项

不存在下标索引

set 虽说可遍历, 但仅可使用迭代器进行遍历, 它不存在下标这一概念, 无法通过下标访问到数据。下面是错误用法:

```
cout << st[0] << endl;
```

元素只读

set 的迭代器取到的元素是只读的 (因为是 const 迭代器), 不可修改其值。如果要改, 需要先 erase 再 insert. 下面是错误用法:

```
cout << *st.begin() << endl; // 正确。可读。
*st.begin() = 1;             // 错误! 不可写!
```

不可用迭代器计算下标

set 的迭代器不能像 vector 一样相减得到下标。下面是错误用法:

```
auto it = st.find(2); // 正确, 返回2所在位置的迭代器。
int idx = it - st.begin(); // 错误! 不可相减得到下标。
```

1.7 映射 [map](#)

```
#include <map>
```

提供对数时间的有序键值对结构。底层原理是红黑树。

性质	解释	map	multimap	unordered_map
互异性	一个键仅可以在映射中出现一次	✓	✗ (任意次)	✓
无序性	键是没有顺序的	✗ (从小到大)	✗ (从小到大)	✓

1.7.1 常用方法

构造

`map<键类型, 值类型, 比较器> mp`

- 键类型：要储存键的数据类型
- 值类型：要储存值的数据类型
- 比较器：键比较大小使用的比较器，默认为 `less<类型>`，可自定义

```
map<int, int> mp1; // int->int 的映射（键从小到大）
map<int, int, greater<int>> st2; // int->int 的映射（键从大到小）
```

对于需要自定义比较器的情况，涉及一些初学时容易看迷糊的语法（重载小括号运算符 / lambda 表达式），在此就不展开讲了。

遍历

可使用迭代器进行遍历：

```
for (map<int, int>::iterator it = mp.begin(); it != mp.end(); ++it)
    cout << it->first << ' ' << it->second << endl;
```

基于范围的循环（C++ 11）：

```
for (auto &pr : mp)
    cout << pr.first << ' ' << pr.second << endl;
```

结构化绑定 + 基于范围的循环（C++17）：

```
for (auto &[key, val] : mp)
    cout << key << ' ' << val << endl;
```

其他

作用	用法	示例
增 / 改 / 查元素	中括号	<code>mp[1] = 2;</code>
查元素（返回迭代器）	<code>.find(元素)</code>	<code>auto it = mp.find(1);</code>
删除元素	<code>.erase(元素)</code>	<code>mp.erase(2);</code>
判断元素是否存在	<code>.count(元素)</code>	<code>mp.count(3);</code>

作用	用法	示例
查看大小 / 清空 / 判空	略	略

增删改查时间复杂度均为 $O(\log n)$

1.7.2 适用情形

需要维护映射的场景可以使用：输入若干字符串，统计每种字符串的出现次数。(`map<string, int> mp`)

1.7.3 注意事项

中括号访问时默认值

如果使用中括号访问 `map` 时对应的键不存在，那么会新增这个键，并且值为默认值，因此中括号会影响键的存在性。

```
map<char, int> mp;
cout << mp.count('a') << endl; // 0
mp['a'];                      // 即使什么都没做，此时mp['a']=0已经插入了
cout << mp.count('a') << endl; // 1
cout << mp['a'] << endl;       // 0
```

不可用迭代器计算下标

`map` 的迭代器不能像 `vector` 一样相减得到下标。下面是错误用法：

```
auto it = mp.find('a'); // 正确，返回a所在位置的迭代器。
int idx = it - mp.begin(); // 错误！不可相减得到下标。
```

1.8 二元组 pair

1.8.1 常用方法

构造

`pair<第一个值类型, 第二个值类型> pr`

- 第一个值类型：要储存的第一个值的数据类型
- 第二个值类型：要储存的第二个值的数据类型

```
pair<int, int> p1;
pair<int, long long> p2;
pair<char, int> p3;
// ...
```

赋值

老式

```
pair<int, char> pr = make_pair(1, 'a');
```

列表构造 C++11

```
pair<int, char> pr = {1, 'a'};
```

取值

直接取值

- 取第一个值: `.first`
- 取第二个值: `.second`

```
pair<int, char> pr = {1, 'a'};  
int awa = pr.first;  
char bwb = pr.second;
```

结构化绑定 C++17

```
pair<int, char> pr = {1, 'a'};  
auto &[awa, bwb] = pr;
```

判同

直接用 `==` 运算符

```
pair<int, int> p1 = {1, 2};  
pair<int, int> p2 = {1, 3};  
if (p1 == p2) { ... } // false
```

1.8.2 适用场景

所有需要二元组的场景均可使用，效率和自己定义结构体差不多。

2 迭代器

2.1 迭代器用法

对于 vector 容器，它的迭代器功能比较完整，以它举例：

- `.begin()`：头迭代器
- `.end()`：尾迭代器，指向的是最后一个元素**再后面一位**
- `.rbegin()`：反向头迭代器
- `.rend()`：反向尾迭代器
- 迭代器 `+` 整型：将迭代器向后移动
- 迭代器 `-` 整型：将迭代器向前移动

- 迭代器 `++`：将迭代器向后移动 1 位
- 迭代器 `--`：将迭代器向前移动 1 位
- 迭代器 `-` 迭代器：两个迭代器的距离
- `prev(it)`：返回 `it` 的前一个迭代器
- `next(it)`：返回 `it` 的后一个迭代器
- 迭代器与指针相似，如果对它使用解引用运算符，即 `*it`，就能取到对应值了

对于其他容器，由于其结构特性，上面的功能不一定都有（例如 `set` 的迭代器是不能相减求距离的）

2.2 常见问题

`.end()` 和 `.rend()` 指向的位置是无意义的值

对于一个长度为 10 的数组：`for (int i = 0; i < 10; i++)`，第 10 位是不可访问的

对于一个长度为 10 的容器：`for (auto it = a.begin(); it != a.end(); ++it)`，`.end` 是不可访问的

不同容器的迭代器功能可能不一样

迭代器细化的话有正向、反向、双向，每个容器的迭代器支持的运算符也可能不同，因此不同容器的迭代器细节很有可能是不一样的。

删除操作时需要警惕

建议：如无必要，别用迭代器操作容器。（遍历与访问没关系）

3 常用函数

3.1 `lower_bound()` / `upper_bound()`

在**已升序排序**的元素中，应用二分查找检索指定元素，返回对应元素迭代器位置。**找不到则返回尾迭代器。**

- `lower_bound()`：寻找 $\geq x$ 的第一个元素的位置
- `upper_bound()`：寻找 $> x$ 的第一个元素的位置

怎么找 $\leq x$ / $< x$ 的第一个元素呢？

- $> x$ 的第一个元素的前一个元素（如果有）便是 $\leq x$ 的第一个元素
- $\geq x$ 的第一个元素的前一个元素（如果有）便是 $< x$ 的第一个元素

返回的是迭代器，如何转成下标索引呢？减去头迭代器即可。

用法示例

```
template< class ForwardIt, class T >
ForwardIt lower_bound( ForwardIt first, ForwardIt last, const T& value );
```

```
vector<int> arr{0, 1, 1, 1, 8, 9, 9};
vector<int>::iterator it = lower_bound(arr.begin(), arr.end(), 7);
int idx = it - arr.begin();
// idx = 4
```

我们通常写成一行：

```
vector<int> arr{0, 1, 1, 1, 8, 9, 9};
idx = lower_bound(arr.begin(), arr.end(), 7) - arr.begin(); // 4
idx = lower_bound(arr.begin(), arr.end(), 8) - arr.begin(); // 4
idx = upper_bound(arr.begin(), arr.end(), 7) - arr.begin(); // 4
idx = upper_bound(arr.begin(), arr.end(), 8) - arr.begin(); // 5
```

3.2 reverse()

反转一个可迭代对象的元素顺序

用法示例

```
template< class BidirIt >
void reverse( BidirIt first, BidirIt last );
```

3.3 unique()

消除数组的重复**相邻**元素，数组长度不变，但是有效数据缩短，返回的是有效数据位置的结尾迭代器。

例如：\$[1,1,4,5,1,4]\to[1,4,5,1,4,\underline{?}]\$，下划线位置为返回的迭代器指向。

```
template< class ForwardIt >
ForwardIt unique( ForwardIt first, ForwardIt last );
```

用法示例

单独使用 unique 并不能达成去重效果，因为它只消除**相邻**的重复元素。但是如果序列有序，那么它就能去重了。

但是它去重后，序列尾部会产生一些无效数据：\$[1,1,2,4,4,4,5]\to[1,2,4,5,\underline{?,?,?}]\$，为了删掉这些无效数据，我们需要结合 erase。

最终，给 vector 去重的写法便是：

```
vector<int> arr{1, 2, 1, 4, 5, 4, 4};
sort(arr.begin(), arr.end());
arr.erase(unique(arr.begin(), arr.end()), arr.end());
```

3.4 数学函数

所有函数参数均支持 `int` / `long long` / `float` / `double` / `long double`

注意事项 由于浮点误差，有些的数学函数的行为可能与预期不符，导致 WA。如果你的操作数都是整型，那么用下面的写法会更稳妥。

$$\lfloor \frac{a}{b} \rfloor$$

- 别用: `floor(1.0 * a / b)`
- 要用: `a / b`

$$\lceil \frac{a}{b} \rceil$$

- 别用: `ceil(1.0 * a / b)`
- 要用: `(a + b - 1) / b`

$$\lfloor \sqrt{a} \rfloor$$

- 别用: `(int) sqrt(a)`
- 要用: 二分查找

$$a^b$$

- 别用: `pow(a, b)`
- 要用: 快速幂

$$\lfloor \log_2 a \rfloor$$

- 别用: `log2(a)`
- 要用: `__lg` (不规范, 但是这是竞赛) / `bit_width` (C++20 可用)

3.5 `gcd()` / `lcm()`

(C++17) 返回最大公因数 / 最小公倍数

```
int x = gcd(8, 12); // 4
int y = lcm(8, 12); // 24
```

如果不是 C++17, 但是是 GNU 编译器 (g++) , 那么可以用内置函数 `__gcd()` .

当然, `gcd` / `lcm` 函数也挺好写, 直接写也行 (欧几里得算法) :

```
int gcd(int a, int b)
{
    if (!b)
        return a;
    return gcd(b, a % b);
}

int lcm(int a, int b)
{
    return a / gcd(a, b) * b;
}
```