这是算法书

Author: TlogyVine Version: 20241129b

目录

输入输出技巧

输入输出流的使用

流的问题

C++的“流”是“类”。

C++相较于 C 语言新加入的流都是类。比如说，cin 就是一个类。

C++还有一些非常好用的流。比如说，头文件<sstream> 的 stringstream 就非常好用，智能化程度较高，以下是演示：（someData 代表任意编码的字符串）

```
// 实例化：
stringstream ass;
// 输入：
ass << someData;
// 输出：
ass >> someData;
```

但是 stringstream 有一些无关紧要的缺陷，例如：不能通过中文的空格进行单词划分。这是程序开发的一个坑，不过我们一般输入的空格都是英文的空格，而且算法竞赛中也不会刻意用中文空格来卡玩家，所以问题不大，了解就好。

选择 cin 还是 scanf ？

相信大家都会用 cin 输入数据，请不要轻易唾弃它，通常情况下它可以极大地减少代码量。
cin 对 string 的输入很有帮助。不过你也可以选择用 scanf 输入 string，示例如下：

```
char ch[20];
string str;
scanf("%s", ch);
str = ch;
```

若使用了 cin 的程序超时，你可以尝试在程序开头写：

```
ios::sync_with_stdio(0);
cin.tie(0); cout.tie(0);
```

```

```

这是用来取消流同步的语句，因为 cin 每次输入数据的时候都要和 stdin 缓冲区同步数据，这可以提高 cin 的输入速度。

另外我想说的是，程序要在 scanf()函数和 cin 两种输入方式之间二选一。尽量不要在同一个程序里面混用，否则在需要取消流同步的时候下会遇上麻烦，需要把所有输入改为 cin 或者直接放弃 cin 用 scanf()。这种情况下无论选择哪一个，都比较耗费心思。

所以一开始就要决定，到底是全用方便的 cin，还是全用效率高的 scanf()。

智能的流

以下是 string 输出流的一个经典使用案例：

```
#define to_string(num) my_to_string(num)
string my_to_string(int num) {
    ostringstream oss;
    oss << num;
    return oss.str();
}
```

若你的编译器不支持 to_string 函数（如 DevCPP 5.40），那么你将要亲自编写。使用了 define 宏定义，即使评判系统支持 to_string，to_string 也会被替换掉，从而避免重定义。

输入的方法

不推荐函数

不推荐的输入函数：（C 语言）

```
char* gets(char* str);
int getc(FILE* stream);
```

gets()可以被 fgets()和 scanf()代替。

getc()可以被 fgetc()和 getchar()代替。

不推荐的处理空白的函数：（C 语言）

```
fflush(stdin);
```

很简单的道理，这个函数可能失效！是过时的玩意。作者以前使用的时候就闹出过笑话，现在依然有心理阴影。

字符行输入

使用 fget 函数（C 语言）

```
// 准备一个 char 数组
char charArray[MAXN];
// 行输入（C 语言）
fgets(charArray, sizeof(charArray), stdin);
charArray[strcspn(charArray, "\n")] = 0;
```

以下是两个函数的原型：

```
char* fgets(char* str, int n, FILE * stream);

typedef unsigned long long size_t;
size_t strcspn(const char* s, const char* reject);
```

fgets()的作用是从输入缓冲区读取 n 个字符串，直到读取完成或遇到 '\n'（换行符）。

第一个参数是指向数组的指针，这个数组用来储存读入的字符串。
第二个参数是 int，决定读取多少个字符。
（当我们在这个位置写上 sizeof(charArray)，可防止数组溢出）
第三个参数是 C 语言文件指针，这里填上 stdin（输入缓冲区）即可。

strcspn()表示从字符串 s 的开头开始连续比对，遇到不存在于 reject 内的字符就返回这个字符前的字符数。在这里实现的效果是返回 '\n'（换行符）的位置，这样以后赋值语句可将读取的字符串的 '\n' 替换为 '\0'。

需要注意的是，替换 '\0'这一步很关键，否则程序极易出 BUG。

使用 getline 函数（C++）

```
// 准备一个 string
string string_variable;
// 行输入（C++）
getline(cin, string_variable);
```

空白符怎么处理

总结：　C 语言用　getchar()
　　C++用　　cin >> ws;

接下来我们讨论什么时候需要处理空白符。

情况 1：scanf 函数存在%c（C 语言）

一旦 scanf 内涉及%c，就应该在所有 scanf 后紧跟 getchar。

特殊情况可以省去，最典型的就是 scanf("%c", &ch);在程序开头，此时%c 的接收不会受空白符的影响，因此也没有写 getchar 的必要。

一般写法：

```
scanf("%d", &num);
getchar();

scanf("%c", &ch);
getchar();
```

特殊情况：

```
// 仅出现在开头且仅出现一次
scanf("%c", &ch);

scanf("%d", &num1);
scanf("%d", &num2);
scanf("%d", &num3);
```

情况 2：未知状态下的%c 读取（C 语言）

假设一种情景，你接手了一个程序，你要在程序的中间插入一个带 %c 的 scanf 函数，这时你就会面

对一个问题，如何消除 stdin（C 语言输入流）内剩余下来的空格？

方法也很简单，就是无论如何先接收符号，将其用来检测，假如是空白符就销毁；是非空白符就返回到输入流。“返回到输入流”这一步很关键，由 ungetc 实现，否则检测完以后的数据就被破坏了。

```
char ch;
while ((ch = getchar()) == ' ');
ungetc(ch, stdin);
```

（该代码要写在 scanf 之前）

情况 3：getline 与 cin 混用（C++）

getline 会读取空白符；cin 会跳过空白符直到再次遇到一个空白符截至，此时输入流会保留一个空白符。若 getline 与 cin 混用，getline 可能会读取到这个空白符。

为了防止 getline 读取到空白符，在混用情况下请使用 std::ws。这是一个操纵符，令流连续忽略空白符，直到非空白符截止。这实现了类似“情况 2”演示的代码的功能，以下将作演示。

```
string widget;
string str;

cin >> widget;
cout << "W:" << widget << endl;

cin >> ws;
getline(cin, str);
cout << "S:" << str << endl;
```

注意 cin >> ws 放置的位置，是紧贴在 getline 之前！！！

来个错误演示：

```
string widget;
string str;

cin >> widget;
cin >> ws;
cout << "W:" << widget << endl;
```

```
    getline(cin, str);
    cout << "S:" << str << endl;

    // 终端输入输出:
    // hello
    // world
    // W : hello
    // S : world
```

试着输入看看，发现了什么，答案是不是在输入第二行以后才给出？因为输入第一行数据以后卡在了 cin >> ws，系统在这里等待客户输入，这显然是我们不想看到的。

不定长输入

即使掌握了 C++的输入方式，学习 C 语言的输入方式仍然有重要意义。

简单的循环输入（C/C++）

```
    // 准备一个临时变量
    int temp;
    // 循环输入（C 语言）
    while (scanf("%d", &temp) != EOF);
    // 循环输入（C++）
    while (cin >> temp);
```

不过这种输入方式常常不是给人用的，而是给算法判定系统。使用它来输入的时候，结束输入需要使用 Ctrl+Z 组合键。

自动逐数据输入（C 语言）

代码片段

```
// 准备一个数组和一个指针
int array[MAXN];
int pa = 0;
// 进入循环
while (1) {
    // 用来检测的临时变量
    char ch;
    // 主体，三行最重要的代码
    while ((ch = getchar()) == ' ');
```

```
    if (ch == '\n') break;
    ungetc(ch, stdin);
    // 检测通过以后执行输入
    scanf_s("%d", &array[pa++]);
}
```

重点是三行主体代码，第一行消耗多余空格；第二行检测是否到末尾；第三行将临时数据返回缓冲区。

为什么要返回数据到缓冲区呢？程序能进行到第三行，就说明这个取出来的数据既不是空白字符也不是换行符，而是一个有用数据，如果不返回，那么就会丢失掉。

另外，这个输入方式还可以改进，使之具备异常检测能力：

```
// 异常判断的输入
int decision_input(int array[], int *pa) {
    while (1) {
        // 用来检测的临时变量
        char ch;
        // 主体
        while ((ch = getchar()) == ' ');
        if (ch != '-' && (ch > '9' || ch < '0')) return 1;
        if (ch == '\n') return 0;
        ungetc(ch, stdin);
        // 检测通过以后执行输入
        scanf("%d", &array[(*pa)++]);
    }
}
```

该函数在遇到异常符号时会中止读入并返回 1。

相较于基础版，主体代码内多写了一行 if 语句。

需要注意的地方是，指针要写成(*pa)++。因为优先级相同时，程序会通过结合性来确定执行顺序。正好解指针符号（*）和自加符号（++）的优先级相同，遵循右往左结合的规律。

*pa++会使指针地址先自加（pa++），然后才解指针，得到的指针很有可能是野指针。这显然是我们不想要的。

自动逐数据输入（C++）

```
// 异常判断的输入
bool decision_input(int array[], int& pa) {
    while (1) {
        // 对空白符和换行符的判断
        while (cin.peek() == ' ') cin.ignore();
        if (cin.peek() == '\n') return 0;
        // 输入，如果异常则返回 1
        if (!(cin >> array[pa])) return 1;
```

```
        pa++;
    }
}
```

C++也可以实现类似的逐数据输入，cin.peek()可以检测下一个字符是什么，就不用抽出数据检测完以后又返回去。cin.ignore()使 cin 忽略掉空白符。cin >> array[pa]在输入异常的情况下会返回 0，然后通过反转，可以被 if 语句捕捉到。

智能逐数据输入（C++）

```
// 准备一个流和一个 string
stringstream ss;
string line;
// 准备一个数组和一个指针
int array[MAXN];
int pa = 0;

// 智能写入
getline(cin, line);
ss << line;
while (ss >> array[pa]) pa++;
```

作者超级推荐的方法，简洁优雅。

还有一些拓展组件也值得一学：

```
// 清空内容并重置状态
// str(" ")将内容清除，不重置状态。
// clear（）将状态重置，不清除内容。
ss.str("");
ss.clear();

// 清除前部分所有空白符
ss >> ws;

// 判断是否发生错误
if (ss.fail());
// 判断是否为严重的、通常无法恢复的错误
if (ss.bad());
```

输出的方法

流输出（C++）

首先介绍一种逼格很高的打印方法。通过输出流迭代器，把容器（或数组）内的数据复制到 cout，以此来实现打印的效果。

作者觉得这是一种很好的思路。

```
// 利用 copy 函数转移数据到输入流 的打印
copy(container.begin(), container.end(), ostream_iterator<int>(cout, " "));
cout << endl;
```

循环输出（C++）

然后来介绍一种，比较常用的方法。这种方法的时候是 C++11 加入的，所以暂且叫它为现代迭代。另外紧接提供了一种常规的迭代方法。

```
// 利用现代迭代读取数据 的打印
for (auto& ele : container) {
    printf("%d ", ele);
}
printf("\n");

// 利用迭代器读取数据 的打印
for (auto i = container.begin(); i != container.end(); i++) {
    printf("%d ", *i);
}
printf("\n");
```

精细化输出（C++）

实际刷题的时候可能会被要求末尾不输出多余的空格，这时就要精细的处理了。具体就是在内部加入一个判断是否到末尾的语句。

```
// 在末尾不打印多余空格 的写法
for (auto i = container.begin(); i != container.end();) {
    printf("%d", *i);
    if (i++ != container.end()) {
        printf(" ");
    }
```

```
    }
    printf("\n");
```

迭代器与对象

说个高阶的C++知识，尖括号内的内容要在编译阶段就确认。所以尖括号内填“非对象”。
如priority_queue<int, vector<int>, decltype(cmp)>,
cmp虽是变量，但是decltype(cmp)表示的是“函数的类型”

迭代器辅助函数

有四个函数：
prev(it)
next(it)
advance(it, n)
distance(first, last)

以下是解释：
pr~：返回前一个迭代器
ne~：返回后一个迭代器
ad~：让迭代器前进n个位置 或后退-n个位置
di~：计算两个迭代器的距离

迭代器分支

记得声明：#include<iterator>

接下来的示例，将会用“T”代指“容器的类”

容器自带迭代器

一共有四个：
T::reverse
T::reverse_iterator
T::const_iterator
T::const_reverse_iterator

只有构造函数没有工厂函数。
值得注意的是，逆向迭代器的实际位置和逻辑位置不同。

			pos		

begin(v1)	v2	v3	v4	v5	end
rend	v1	v2	v3	v4	rbegin(v5)

将反向迭代器赋值给正向迭代器（或正到负），将输出不同结果！

安插迭代器

insert_iterator<T>
工厂函数：inserter(c, c.begin())
工厂函数：inserter<container>(c, c.begin())

front_insert_iterator<T>
back_insert_iterator<T>
工厂函数：back_inserter(c)
工厂函数：back_inserter<container>(c)

流类迭代器

流迭代器的操作很少很简单。以下结论来自网友“双子座断点”。

istream_iterator<T> iit(cin)

假设 p 是一个输入流迭代器，则其只能进行 ++p、p++、*p 操作，
同时输入迭代器之间也只能使用 == 和 != 运算符。

输入流迭代器的底层是通过重载 ++ 运算符实现的，该运算符内部会调用 operator >>读取数据。也就是说，假设 iit 为输入流迭代器，则只需要执行 ++iit 或者 iit++，即可读取一个指定类型的元素。

ostream_iterator<T> oit(cout)

假设 p 为一个输出迭代器，则它能执行 ++p、p++、*p=t 以及 *p++=t 等类似操作。

输出迭代器底层是通过重载赋值（=）运算符实现的，即借助该运算符，每个赋值给输出流迭代器的元素都会被写入到指定的输出流中。

可调用对象

一共有 6 种：

1. 普通函数
2. 类静态函数
3. 类成员函数
4. 函数指针
5. 仿函数（struct operator）
6. 匿名函数（lambda）

普通函数、类函数和函数指针

如何使用外部指针调用类成员函数？

```
// 假设实例是 ms
MyStruct ms;

// 编写一个指向 MyStruct 内部的指针类型
typedef void(MyStruct::* SP)();

// 实例化一个指针，并将指针绑定到 fun 函数
SP p = &MyStruct::fun;

// 通过实例调用该指针：
(ms.*p)();
```

函数名字与指针赋值

类成员函数的名子较为特殊，可以理解为“标签”而不是“地址”

```
void fun_a() {}

struct MyStruct {
    static void fun_b() {}
    void fun_c() {}
};

typedef void(*P)();
typedef void(MyStruct::*SP)();

int main() {
    P p1 = fun_a;
    P p2 = MyStruct::fun_b;
    SP p3 = &MyStruct::fun_c;

    /* other codes...*/
}
```

显然，只有“类成员函数”取地址需要 &符号。但这时还未实例化类对象，所以不可能是成员的“地址”，有个简单直接的证明：

```
    // *p 并不能调用函数！
    // 以下例子可以证明*p 得到的是 p

    // 显然这个过程并不会执行函数里面的内容
    p1 = *(*p1);
    p1 = *p1;
    p1 = p1;
    p2 = *(*p2);
    p2 = *p2;
    p2 = p2;

    // 这样写才真正调用了函数
    p1();
    p2();

    // 但是 p3 会报错，这就表明了 p3 有本质上的不同
    // 报错信息："*" 的操作数必须是指针，但它具有类型 "SP"
    p3 = *p3;
```

补充：可以注意到“普通函数”和“类静态函数”非常相似。这是因为“类静态函数”虽然属于某个类，但是“类静态函数”的位置是全局的。需要特别注意的是“类静态函数”不能使用”指向类内部的指针类型”（如上例的 SP），会报错。

致命的隐式 this 指针

三种函数中，只有“类成员函数”会有这个问题。在外部指针调用类成员函数时，会隐式传入 this 指针。当这个 this 的类型与原形函数不匹配，就会报错。

举个例子，本人想创建一个不会被修改的实例，例如：

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

struct MyStruct {
    void fun(int num) {}
};

int main() {
    const MyStruct ms;
    ms.fun(64);
```

```
    return 0;
}
```

然后就报错了，为什么？报错上说，对象含有与成员 函数“MyStruct::fun”不兼容的类型限定符，对象类型是：const MyStruct。

这是因为传回的 this 指针是有类型的，而且类型不对。解决方法也是有的，就是对原型函数 fun 进行修改，加上 const 标签：

```
    void fun(int num) const {}
```

仿函数与匿名函数

仿函数的常见写法

下列实例无实际含义

```
struct cmp {
    bool operator()(Node& a, Node&b) {
        return 0;
    }
};
```

匿名函数的写法

方括号内填写捕获方式和范围，下文称“捕获”暂且不考虑，那么完整写法是：

```
function<ReturnType(EleType)> fun = [](EleType num) -> ReturnType {};
```

（注：　ReturnType　返回的类型
　　EleType　参数列表的元素的类型
　　fun　一个假设的函数 ）

是不是很长很冗余？

没事我们一般不这么写,以下才是我们常见的写法：

模板一

```
function<ReturnType(EleType)> fun = [](EleType num){};
```

模板二

```
auto fun = [](EleType num) -> ReturnType {};
```

参数列表可以多填，但是 EleType 的数量和类型要和参数列表一一对应。

“捕获”可填“=”和“&”，分别代表对全局变量按“赋值传参”和“引用传参”。你也可以填变量。填了变量以后，将只能捕获对应的元素，不能捕获全局变量。

仿函数重要性质

仿函数是模仿“函数”的“类”。由类内部重载“函数调用符号（小括号）”实现。

仿函数有两种调用方式：

1. func()(n, m)
2. func(n, m)

第一种意味着 func 是一个“struct 模板”，func()是对这个模板的“实例化”，然后下一个小括号才是真正的调用。第二州意味着 func 是一个“仿函数对象”（已经被实例化），所以直接使用小括号即可调用函数。

作者评价：不知为何 C++官方的“类、结构体、模板等”的名称不以大写字母开头。“模板”与“对象”的名字真是不容易区分，如果是我们去编写，建议以大写字母开头，例如：

	错误写法	正确写法
类	class node{};	class Node{};
结构体	struct func{};	struct Func{};
模板	template<typename t> class plus{};	template<typename T> class Plus{};

但如果只是图个方便，而且自己心知肚明，那也无妨。

1. struct cmp{};
2. using ll = long long;
3. ...

这样也是可以的。

可调用对象的封装

记得声明：#include<functional>

可调用对象的封装类型有：

1. uniOp（一元操作）
2. binOp（二元操作）
3. uniPd(一元谓词)
4. binPd(二元谓词)
5. cmp（二元比较器）

简单来讲：

“N 元操作”是对元素的某种修改。

“N 元谓词”是使用元素，但是不修改。

“比较器”应该算是谓词的一种

封装器

std::function（多态函数封装器）

通常作为“类型”出现，比如 function<void(int, int)>

对此 AI 解释道：

> `std::function` 是 C++标准库中的一个模板类，它是一个通用的函数指针容器，可以存储任何类型的可调用对象，比如函数、方法甚至 Lambda 表达式。它的主要优点是可以动态绑定，即在运行时确定调用哪个函数，这使得函数指针的使用更加灵活。

它的强大之处在于它可以统一表示“任何类型的可调用对象”，用 function<()>来包装。

对于：

```
int(*fp)(int, int) = func;              // 函数指针
Func ff;                                // 仿函数
auto f1 = [](int a, int b)->int {};     // 匿名函数
```

可写：

```
function<int(int, int)> f1 = fp;
function<int(int, int)> f2 = ff;
function<int(int, int)> f3 = f1;
```

bind（函数重绑定）

一般形式：

```
auto newCallable = bind(callable, arg_list);
```

arg_list 的占位符这样写：std::placeholders::_[填写数字]
作者评价：极其脑残设计，一点都不好记

仿函数预制菜

都是“template 模板”意味着全部都未“实例化”。
直接调用的方式，如：plus<int>()(10, 1) 别忘记了它要两个括号。plus<int>()只是创建一个实例，后面的(10, 1)才是函数的调用，直接写 plus<int>(10, 1)将会导致报错。

以下除了 negate\logical_not\bit_not 都是二元的，即 binOp

算数仿函数

```
plus <T>();
minus <T>();
multiplies <T>();
divides <T>();
modulus <T>();
negate <T>();
```

关系仿函数

```
equal_to <T>();
not_equal_to <T>();
greater <T>();
greater_equal <T>();
less <T>();
less_equal <T>();
```

逻辑仿函数

```
logical_not <T>();
logical_and <T>();
logical_or <T>();
```

位运算仿函数
bit_and <T>(); bit_or <T>(); bit_xor <T>(); bit_not <T>();

散装文件知识

简单科普：

修改器（Modifiers）
会改变容器状态的函数，它们通常涉及到容器内部元素的添加、删除或替换。
操作（Operations）
不会改变容器状态的函数，但可能会对容器中的元素进行排序、筛选或修改。

头文件

本章节要介绍的头文件：

（一）算法头文件：<algorithm> <numeric> <limits>
（二）与 C 语言有关的头文件：<cmath> <cstdio> <cstdlib> <cstring>
（三）会了最好，不会不影响竞赛的头文件：<iomanip> <bitset> <random>
（四）容易遗忘的头文件：<fstream> <list> <string>

本书其它章节有介绍：

（五）输入输出技巧：<iostream> <sstream>
（六）迭代器与泛型：<functional> <iterator>

本书不介绍：（不介绍这些头文件的原因各式各样（o ˚▽ ˚)o 见谅！）

<bits/stdc++.h> <map> <queue> <set> <stack> <tuple>
<unordered_map> <unordered_set> <utility> <vector> <wchar>

算法头文件（STL）

本节着重讲解三个算法头文件

ALGORITHM（包含：STL 算法）

注： beg 起始迭代器；end 末迭代器；dest 指向目的容器的迭代器

序列操作	
排序	**sort**(beg, end) **stable_sort**(beg, end) **next_permutation**(beg, end) **prev_permutation**(beg, end)
归并	**merge**(beg1, end1, beg2, end2, dest) **inplace_merge**(beg, mid, end)
去重	**unique**(beg, end) **unique_copy**(beg, end, dest)
划分	nth_element(beg, mid, end) **partition**(beg, end, uniPd) **stable_partition**(beg, end, uniPd)

比对操作		
查找 **（顺序）**	find(beg, end, val) find_if(beg, end, uniPd) search_n(beg, end, count, val) find_end(beg1, end1, beg2, beg2) search(beg1, end1, beg2, beg2) 作者评价：这 TM 就是一坨答辩。傻逼命名	查找值 查找值 查找值（连续 n 次出现） 查找子串（最后一次出现的位置） 查找子串（第一次出现的位置）
查找 **（二分）**	binary_search(beg, end, val) **equal_range**(beg, end, val) **upper_bound**(beg, end, val) **lower_bound**(beg, end, val)	
查询 **（顺序）**	**count**(beg, end, val) **count_if**(beg, end, uniPd) **min_element**(beg, end) **max_element**(beg, end) **min**(val1, val2) **max**(val1, val2) equal(beg1, end1, beg2)	

复制粘贴	
转移	swap(val1, val2) **reverse**(beg, end) **reverse_copy**(beg, end, dest) **fill**(beg, end, val) **fill_n**(dest, num, val) **copy**(beg, end, val) **copy_n**(beg, num, val) **copy_if**(beg, end, dest, uniPd) for_each(beg, end, uniOp) **transform**(beg, end, dest, uniOp) **transform**(beg1, end1, beg2, dest, binOp)

对于一些函数的解析

permutation 的原理：

对于：12344321

1. 逆向遍历找到第一个非连续递增（从右向左看）的数：ni
   逆向遍历找到第一个大于 ni 的数：nj
2. 交换 ni 和 nj
   12(3)4(4)321 -> 12(4)4(3)321
3. 翻转 ni 原位之后的序列

12(4)(43321) -> 12412334

4. 作特判

```
bool next_permutation(vector<int>& nums) {
    int len = nums.size();
    if (len == 0) return false;
    if (len == 1) return false;

    for (int i = len - 2; i >= 0; i--) {
        // 找到第一个非单调递增元素（从右向左看）
        if (nums[i] < nums[i+1]) {
            // 找到第一个大于 nums[i]的数
            int j = len - 1;
            while (nums[i] >= nums[j]) j--;
            swap(nums[i], nums[j]);
            reverse(nums.begin() + i + 1, nums.end());
            return true;
        }
    }
    reverse(nums.begin(), nums.end());
    return false;
}

bool prev_permutation(vector<int>& nums) {
    int len = nums.size();
    if (len == 0) return false;
    if (len == 1) return false;

    for (int i = len - 2; i>=0; i--) {
        // 找到第一个非单调递减元素（从右向左看）
        if (nums[i] > nums[i+1]) {
            // 找到第一个小于 nums[i]的数
            int j = len - 1;
            while (nums[i] <= nums[j]) j--;
            swap(nums[i], nums[j]);
            reverse(nums.begin() + i + 1, nums.end());
            return true;
        }
    }
    reverse(nums.begin(), nums.end());
    return false;
}
```

equal_range 的原理：

equal_range 返回的是左闭右开区间，迭代器二元组（pair）。实际上是分别调用了 upper_bound 和 lower_bound，这点从源代码里面可以得知。如此一想，就能理解这两个函数的返回值分别指向哪里了。

NUMERIC（包含：STL 数学运算）

有以下五个函数：

```
accumulate(beg, end, init, [op])
inner_product(beg1, end1, beg2, init, [op1], [op2])
adjacent_difference(beg, end, dest)
partial_sum(beg, end, dest)
iota(beg, end, val)
```

以下是解释：

in~：第一个 op 表示 beg1 到 end1 之间的运算，默认 plus<int>()

in~：第二个 op 表示 beg1 与 beg2 之间的运算，默认 multiplies<int>()

io~：公差为 1，起始为 val 的递增数列

LIMITS（包含：限制类）

有以下两个函数：

```
numeric_limits<T>::max()
numeric_limits<T>::min()
```

如果想使用宏定义最大最小值，就使用<climits>。

在 MAX 或 MIN 前加上“内置类型”，例如：SHRT_MAX，LLONG_MAX。

作者评价：short 和 longLong 的宏定义就是“脑残设计”，非要扣掉个字母

C 风格头文件（C/C++）

本节着重讲解 <cmath> 和 <cstdlib> 的内容

CMATH（包含：数学函数）

注：该头文件较为死板，每个函数都是固定的参数类型和返回类型。（C 语言没有模板概念导致的）

```
double sin(double n);
double cos(double n);
double tan(double n);
double asin(double n);
double acos(double n);
double atan(double n);
```

- 采用的是“弧度制”

- 特别有：4 * atan(1) = acos(-1) = π

```
double exp(double n);
double log(double n);
```

- log2(n)是 log(n)/log(2)，这个简单数学原理大家应该都懂
- e 约等于 2.718281828459045，double 的精度够容纳 15 位。

如果是当场推理 e，公式在这：

```
double res = 1.0;
int n = 0x7fffffff;
double t = 1 + 1.0 / n;
for (int i = 0; i < n - 1; i++) {
    res *= t;
}
```

作者评价：可怜的程序跑断腿了，才勉强精确到小数点后 8 位。哈哈
最后，记得把值拷贝下来使用

```
double pow(double a, double n);
double hypot(double x, double y);
```

```
int abs(int n);
double fabs(double n);
double floor(double n);
double ceil(double n);
double round(double n);
```

CSTDLIB（包含：转换函数）

注：该头文件并未提供 std::string 的转换方式，而是 CTypeStr 的

const char*转换

<cstdlib> 的内容比 <stdlib.h> 的内容多，而且补充了一些漏掉的功能

```
int atoi(cstr);
double atof(cstr);
long atol(cstr);
long long atoll(cstr);

float strtof(cstr, &endptr);
double strtod(cstr, &endptr);
long strtol(cstr, &endptr, radix);
long long strtoll(cstr, &endptr, radix);
unsigned long strtoul(cstr, &endptr, radix);
unsigned long long strtoull(cstr, &endptr, radix);
```

注：cstr：“C 语言风格的字符串”的指针，即 const char*
endptr：一个指针，char*，执行完毕后会指向字符串的末尾
radix：指定进制，类型是 int，比如填 10

评价：这里有一个比较坑的，atof 返回的是 double

动态分配（C 语言）

创建一个堆对象：(*T)malloc(num * sizeof(T))
释放堆对象：free(T)

（C++使用者建议使用 new 和 delete）

终端悬停（C/C++）

代码：system(“pause”);

便捷工具（C/C++）

作者评价：这是对于 C 语言来说的，对 C++来说一点也不方便

bsearch(key, base, num, size, cmp)：

1. key: 要查找的目标元素，可以是一个指向目标值的指针。
2. base: “已经排序”的数组的第一个元素的指针。
3. num: 数组中的元素个数。
4. size: 数组内每个元素的大小。
5. cmp: 一个指向比较函数的指针。

```
// C 语言的“比较函数 cmp”的写法极度死板
int cmp(const void* a, const void* b){
    转换部分：
    类型指针 pa = (类型指针)a;
    类型指针 pa = (类型指针)a;

    比较部分：
    如果第一个元素小于第二个，返回负数
    大于则返回正数
    相等则返回零
}
```

以下对 bsearch 的使用作演示

```
// 演示
#include <iostream>
#include <cstdlib>
using namespace std;
```

```
int num[10] = { 1, 2, 3, 3, 4, 7, 8, 10, 10, 11 };
int val = 8;
void* p;
int* ip;

int cmp(const void* a, const void* b) {
    const int* pa = (const int*)a;
    const int* pb = (const int*)b;
    if (*pa < *pb) return -1;
    if (*pa > *pb) return 1;
    return 0;
}

int main() {
    p = bsearch(&val, num, 10, sizeof(int), cmp);
    ip = (int*)p;

    if (p != NULL) {
        printf("%d\n", *ip);
        printf("%d\n", ip - num);
    }
    return 0;
}
```

qsort(base, num, size, cmp):

1. base: 数组的第一个元素的指针。
2. num: 数组中的元素个数。
3. size: 数组内每个元素的大小。
4. cmp: 一个指向比较函数的指针。

注：对比 bsearch 函数就是少了参数 key。

文件读写（STL）

本节先带读者了解 <cstdio>，然后过度到 <fstream>

文件读写，无非三个过程：

1. 打开文件
2. 操作文件
3. 关闭文件

简单读写（C 语言）

```
#define _CRT_SECURE_NO_WARNINGS
```

取消安全模式。这意味着可以使用“标准函数”，而非“安全函数”。
不过，接下来只介绍安全模式函数：

打开

```
errno_t fopen_s(
    FILE** _Stream,
    const char* _FileName,
    const char* _Mode
);
```

关闭（没有安全模式）

```
int fclose(FILE* _Stream);
```

errno_t：返回 0 表示打开成功，返回 1 表示异常。实质是 int
_Stream：任意 FILE 指针的取址
_FileName：文件路径
_Mode：模式

模式一览	含义
r、w、a	读写追加

注：不要字母大写，程序会崩溃

读取

```
size_t fread_s(
    void* _Buffer,
    size_t _BufferSize,
    size_t _ElementSize,
    size_t _ElementCount,
    FILE* _Stream
);
```

写入（没有安全模式）

```
size_t fwrite(
    const void* _Buffer,
    size_t _ElementSize,
    size_t _ElementCount,
    FILE* _Stream
);
```

_Buffer：缓冲区指针（缓冲区实际上就是个数组）

以上两个函数一般用于读取写入“字符 char”

一种简单思路：

其实到这，如果是熟练的 C++使用者就无敌了

std::string 的“C 风格构造函数”
std::string 的“赋值成员函数 assign”
std::stringstream 的“C 风格构造函数
std::stringstream 的“流操作符 <<”
...

都可以将读取到的“C 风格字符串”转换为“C++风格字符串”从而转换为 C++的问题。
利用“智能流　类”可以轻松地解决问题，如：分割、转换类型...

总之就是，虽然吾不会“C++的文件流读取”和“C 语言的其它读取函数”。
但是，吾能提取最基础的 char，所以除了多占用一些缓存数组以外，吾就是无敌的！

简单读写（C++）

C++主要使用输入输出流，以下以〈fstream〉为中心进行讲解

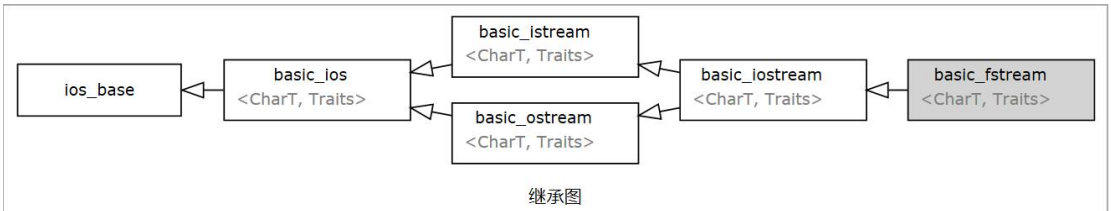
简单介绍：

fstream 是 basic_fstream<char> 的一个别名

basic_fstream 继承自：

basic_ios	提供输入输出流的基础功能。
basic_filebuf	提供文件缓冲区类，用于实际的文件读写操作
basic_istream	提供输入流的功能。
basic_ostream	提供输出流的功能

参考文档如此写道：



继承图

构造函数：

fstream()：默认构造函数，不打开任何文件。
fstream(const char* filename, ios_base::openmode mode)

文件操作：

open(const char* filename, ios_base::openmode mode)
close()

read(char* buf, streamsize n)

write(const char* buf, streamsize n)

getline(char* buf, streamsize n)　　最多读取 n-1 个字符，自动添加一个\0 作为行尾。

puts(const char* s)　　将字符串 s 写入文件，并在末尾添加一个换行符。

get(char& c)

put(char c)

位置控制：

seekg(pos_type pos)　　将输入文件指针移动到 pos 指定的位置。

seekg(off_type off, ios_base::seekdir dir)　相对于 dir 指定的位置移动 off 个单位。

tellg()　　返回当前输入文件指针的位置。

状态检查：

eof()：检查是否到达文件末尾。

fail()：检查上一次操作是否失败。

bad()：检查是否发生严重的文件错误。

good()：检查文件流是否处于良好状态。

格式化：

sync()：同步文件缓冲区和文件本身。

flush()：清空输出缓冲区，但不关闭文件。

数据打印（STL）

本节着重讲解 <iomanip> 、<bitset> 和 <random> 的内容

BITSET（包含：二进制显示类）

构造函数	bitset<n> bs;　　// bs 有 n 位，每位都为 0 bitset<n> bs(num);　　// bs 是 unsigned long 型 num 的一个副本 bitset<n> bs(str);　　// bs 是 string 对象 str 中含有的位串的副本 bitset<n> bs(str, pos, n);　// bs 是 s 中从位置 pos 开始的 n 个位的副本 注：创建 bitset 的时候，设定长度不足会导致截断，而不是回滚
get 函数	bs.any()　　是否存在为 1 的位 bs.none()　　是否不存在为 1 的位 bs.test(pos)　在 pos 处的位是否为 1 bs.count()　　为 1 的位的数量 bs.size()　　位的数量 bs[pos]　　实际返回一个引用类
set 函数	bs.set()　　把所有位写为 1 bs.set(pos)　把 pos 处的位写为 1 bs.reset()　　把所有位写为 0 bs.reset(pos)　把 pos 处的位写为 0 bs.flip()　　把所有位反转 bs.flip(pos)　把 pos 处的位反转
转换函数	bs.to_string() bs.to_ulong()

	bs.to_ullong()

IOMANIP（包含：输出控制）

自动重置的控件（只有一个）：

setw(整型 setWidth)　　设置位宽（默认：向右对齐）

一次编写，一直有效的函数：

setiosflags(iosBaseFunction)　　设置标志
setprecision(整型 digit)　　设置精度
setfill(字符型 ch)　　设置填充

进制标志：

ios_base::dec
ios_base::oct
ios_base::hex

setiosflags 配置：

ios::fixed：设置浮点数以固定的小数位数显示。
ios::scientific：设置浮点数以科学计数法表示。
ios::left：输出左对齐。
ios::right：输出右对齐。
ios::skipws：忽略前导空格。
ios::uppercase：在以科学计数法输出 E 与十六进制输出 X 时以大写输出，否则小写。
ios::lowercase：在以科学计数法输出 e 与十六进制输出 x 时以小写输出，否则大写。
ios::showpoint：强制显示小数点。
ios::showpos：输出正数时显示"+"号。

RANDOM（包含：梅森缠绕器）

<—— 旧方法（cstdlib） ——>

种子：srand((unsigned int)time(NULL))
生成数字：rand()

<—— 新方法（random）——>

种子：

使用随机设备：random_device rd;
使用时间：time(NULL)

随机数生成器：

使用梅森缠绕器：mt19937 gen(/* 这里填写种子 */);

分布算法函数：

均匀分布（整型）：uniform_int_distribution<T> uid(first: T, last: T) // 闭区间
均匀分布（实数）：uniform_real_distribution<T> urd(first: T, last: T) // 左闭右开
正态分布：normal_distribution<> nd(mean: double, stddev: double)
伯努利分布：bernoulli_distribution bd(probability: double)

字符（STL）

本节着重讲解 <string> 的内容，同时与 <cstring> 进行比较。

C++风格字符串

我们常用的 std::string 实际上是一个对 std::basic_string 的特化版本
std::basic_string 是一个模板类，可以用于创建不同字符类型的字符串

类型	定义
std::string	std::basic_string<char>
std::wstring	std::basic_string<wchar_t>
std::u8string (C++20)	std::basic_string<char8_t>
std::u16string (C++11)	std::basic_string<char16_t>
std::u32string (C++11)	std::basic_string<char32_t>

省略“pmr 名字域”的内容（目前没必要知道，用不上）

元素访问	front()：访问字符串的第一个字符。 back()：访问字符串的最后一个字符。 c_str()：返回一个指向 null-terminated C 风格的字符数组的指针。
迭代器	begin()、cbegin()：返回指向字符串开始的迭代器。 end()、cend()：返回指向字符串末尾的迭代器。 rbegin()、crbegin()：返回指向字符串开始的逆向迭代器。 rend()、crend()：返回指向字符串末尾的逆向迭代器。
容量	empty()：检查字符串是否为空。 size()、length()：返回字符串中的字符数。 reserve()：预留足够的存储空间。 capacity()：返回当前对象分配的存储空间能保存的字符数量。
修改器	clear()：清除字符串内容。 insert()：在指定位置插入字符或字符串。 erase()：移除字符串中的一部分。 push_back()：在字符串末尾追加一个字符。 pop_back()：（C++11）移除字符串末尾的字符。 append()：在字符串末尾追加字符或字符串。 resize()：更改字符串存储的字符数。 swap()：交换两个字符串的内容。
操作	compare()：比较两个字符串。 replace()：替换字符串中的指定部分。 substr()：返回字符串的子串。 copy()：复制字符串到新的字符数组。
查找	find()：在字符串中查找字符或子串。 rfind()：在字符串中查找子串的最后一次出现。

	find_first_of()：查找任何字符的首次出现。 find_first_not_of()：查找任何字符的首次缺失。 find_last_of()：查找任何字符的最后一次出现。 find_last_not_of()：查找任何字符的最后一次缺失。
常量	npos：一个静态成员常量，用于表示“未找到”或“无位置”。也可以是最大可能长度

C 风格字符串

作者评价：较为死板，但同时又很严谨。全部摆在一起比较，几乎是一团糟。

1. 从参数列表就可以看出，不需要修改的数组的指针都是 const 类型。
2. C 语言没有布尔类型（bool），所以判断大小全由 int 代替，左小为负。

和 C++比较相似的是，对于统计数量采用的都是 size_t

容量	`size_t strlen(const char* str);`

修改器	`char* strcpy(char* dest, const char* src);`
	`char* strncpy(char* dest, const char* src, size_t num);`
	`char* strcat(char* dest, const char* src);`
	`char* strncat(char* dest, const char* src, size_t num);`

操作	`int strcmp(const char* str1, const char* str2);` `int strncmp(const char* str1, const char* str2, size_t num);`

查找 1)	`size_t strspn(const char* str, const char* charset);`
	`size_t strcspn(const char* str, const char* charset);`
	`const char* strpbrk(const char* str, const char* charset);`

strspn: str 开始处连续出现的字符中，包含在 charset 中的字符的个数
strcspn: str 开始处连续出现的字符中，不包含在 charset 中的字符的个数
strpbrk: str 中第一次出现的任何 charset 中的字符，并返回指向该字符的指针
总结：从效果上 strcspn 和 strpbrk 更像
实际：变量 charset 不是真的集合，查询时间复杂度 O(nm)

查找 2)	`const char* strchr(const char* str, int val);` `const char* strstr(const char* str, const char* substr);`

链表（STL）

本节着重讲解 <list> 里面的成员函数。

元素访问	front()：返回列表第一个元素的引用。 back()：返回列表最后一个元素的引用。
迭代	begin()：返回一个迭代器，指向列表的第一个元素。

器	cbegin()：（C++11）返回一个常量迭代器，指向列表的第一个元素。 end()：返回一个迭代器，指向列表末尾的下一个位置。 cend()：（C++11）返回一个常量迭代器，指向列表末尾的下一个位置。 rbegin()：返回一个逆向迭代器，指向列表的最后一个元素。 crbegin()：（C++11）返回一个常量逆向迭代器，指向列表的最后一个元素。
容量	empty()：检查列表是否为空。如果列表为空，返回 true；否则返回 false。 size()：返回列表中的元素数量。
修改器	clear()：清空容器，移除所有元素。 insert()：在容器中插入一个或多个元素。 emplace()：在容器中就地构造一个新元素。 erase()：从容器中移除一个或多个元素。 push_back()/emplace_back()：在容器末尾添加一个元素。 pop_back()：移除容器末尾的元素。 push_front()/emplace_front()：在容器开头添加一个元素。 pop_front()：移除容器开头的元素。 resize()：改变容器的大小。 swap()：交换两个容器的内容。
操作	merge()：合并两个容器，通常用于有序容器。 splice()：将一个容器中的元素或一段元素移动到另一个容器。 remove()/remove_if()：从容器中移除满足特定条件的元素，但不改变容器的容量。 reverse()：反转容器中的元素顺序。 unique()：移除容器中连续重复的元素。 sort()：对容器中的元素进行排序。

正则表达式

记得写上 #include<regex>

作者评价：　　要区分匹配的是单个字符串，还是长句（多个字符串组合）
　　否则说，你把长句传入单字符匹配函数，结果死活匹配不出来

普遍语法

“匹配模式”是多个“元素+[量词]”的关系组合
以下表格记不住没关系，多看几遍就行

基础篇（简单占位）

. - 除换行符以外的所有字符。
^ - 字符串开头。
$ - 字符串结尾。
\d,\w,\s - 匹配数字、字符、空格。
\D,\W,\S - 匹配非数字、非字符、非空格。

进阶篇（选择）

[abc] - 匹配 a、b 或 c 中的一个字母。
[a-z] - 匹配 a 到 z 中的一个字母。
[^abc] - 匹配除了 a、b 或 c 中的其他字母。
aa|bb - 匹配 aa 或 bb。

进阶篇（量词）

? - 0 次或 1 次匹配。
* - 匹配 0 次或多次。
+ - 匹配 1 次或多次。
{n} - 匹配 n 次。
{n,} - 匹配 n 次以上。
{m,n} - 最少 m 次，最多 n 次匹配。

然后是高级篇，说实话，作者本人也没搞懂。不会可以选择跳过。

高级篇（模式）

(expr) - 捕获 expr 子模式,以\1 使用它。
(?:expr) - 忽略捕获的子模式。
(?=expr) - 正向预查模式 expr。
(?!expr) - 负向预查模式 expr。

语法优先级

优先级，从上往下，依次下降

\	转义符
() (?:) (?=) []	圆括号和方括号
* + ? {n} {n,} {n,m}	限定符
^ $ \char char	定位点和序列
|	或

三个算法函数

regex_match 用 regex 模式串匹配整个 basic_string 主串
regex_search 用 regex 模式串匹配 basic_string 主串中的子串

regex_replace　　替换匹配成功的 basic_string 主串中的子串

以下是简单的实战演示：

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

int main() {
    // 主串
    string str = "Hello World";
    // 模式串
    string pt;   // pattern
    regex re;    // a_regex
    // 格式串
    string fmt("DEBUG");  // format
    char cfmt[5] = "BUG"; // cstyle_format
    // sm：固定(const_iterator)
    // mr：可变(iterator)
    smatch sm;   // a_smatch
    match_results<string::iterator> mr;// a_match_results
    // 输出迭代器
    string ostr; //output_string
    back_insert_iterator<string> bii = back_inserter(ostr);

    /*————————————————————————————————————————————————————————*/

    // 匹配整个 string
    pt = string{ "He.{9}" };
    re = regex(pt);
    cout << regex_match(str, re) << endl;
    cout << regex_match(str, sm, re) << endl;
    cout << regex_match(str.begin(), str.end(), re) << endl;
    cout << regex_match(str.begin(), str.end(), mr, re) << endl;

    // 匹配 string 的子串
    pt = string{ "l{1,2}\\d" };
    re = regex(pt);
    cout << regex_search(str, re) << endl;
    cout << regex_search(str, sm, re) << endl;
    cout << regex_search(str.begin(), str.end(), re) << endl;
    cout << regex_search(str.begin(), str.end(), mr, re) << endl;
```

```
    // 替换匹配部分
    pt = string{ "1{1,2}\\d" };
    re = regex(pt);

    cout << regex_replace(str, re, fmt) << endl;
    cout << regex_replace(str, re, cfmt) << endl;
    regex_replace(bii, str.begin(), str.end(), re, fmt);
    cout << ostr << endl;
    regex_replace(bii, str.begin(), str.end(), re, cfmt);
    cout << ostr << endl;

    return 0;
}
```

图论

图的储存

边集数组

主体

```
// 边集数组
struct Node {
    int u;
    int v;
    int w;
} edge[MAXE];
// 当前边数
int cnt = 0;
```

功能函数

```
// 新插入一条边
void add(int u, int v, int w) {
    edge[cnt].u = u;
    edge[cnt].v = v;
    edge[cnt].w = w;
    cnt++;
}
```

STL 邻接表

主体

```
// 节点结构体
struct Node {
    int v;  //下一个节点
    int w;  //权重
};

// STL 邻接表（动态）
vector<list<Node>> g;
// STL 邻接表（静态）
list<Node> g[MAXV];
```

功能函数

```
// 重载构造函数（可选，写在 struct Node 内部）
Node(int _v, int _w) : v(_v), w(_w) {}
```

链式前向星

主体

```
// 节点结构体 & 半边集数组
struct Node {
    int v;
    int w;
    int next;  // 下一个子节点
} edge[MAXE];

// 头节点数组
int head[MAXV];

// 当前边数
// 指向半边级数组空位
int cnt = 0;
```

功能函数

```
// 添加一条边
void add(int u, int v, int w) {
    edge[cnt].v = v;
```

```
    edge[cnt].w = w;
    edge[cnt].next = head[u];
    head[u] = cnt++;
}
```

最短路径

目前你需要学会的最短路径算法有 5 个

1. BFS
2. Floyd
3. Dijkstra
4. Bellman-Ford
5. SPFA

BFS 在其它章节会讲；Floyd 不实用；SPFA 比 Bellman-Ford 难
所以本书只教 Dijkstra 和 Bellman-Ford

路径保存方法

前驱节点

```
list<int> pre[MAXV];
```

路径集

回溯过程中得到的一条最短路径不一定是答案，这个具体看题目要求。回溯以后可以选择更新答案 ans，也可以把这条路径添加到 result 中。

```
vector<int> path;

// 第一种选择（直接更新答案）
vector<int> ans;
// 第二种选择（添加多个答案）
vector<vector<int>> result;
```

回溯

```
// 回溯法
```

```
void DFS(int s) {
    // 确定这个点是不是源点
    if (pre[s].empty()) {
        /* 使用 for 循环遍历最小路径 */
        /* 在遍历的过程动态修改（可选）*/
        return;
    }
    // 回溯
    for (auto& node : pre[s]) {
        path.push_back(node);
        DFS(node);
        path.pop_back();
    }
}
```

Dijkstra

简单距离（邻接矩阵）

前置

```
int n;
int g[MAXV][MAXV];
int dis[MAXV];
bool vis[MAXV];
```

主体

```
// 迪杰斯特拉
void Dijkstra(int s) {
    // 初始化数组
    fill(dis, dis + MAXV, INF);
    fill(vis, vis + MAXV, false);
    // 初始化 dis 起点为 0
    dis[s] = 0;
    for (int i = 0; i < n; i++) {
        int u = -1;
        int MIN = INF;
        for (int j = 0; j < n; j++) {
            if (!vis[j] && dis[j]!=INF && dis[j] < MIN) {
                u = j;
                MIN = dis[j];
            }
        }
        // 找不到小于 INF 的 dis[u]，说明剩下的节点不连通
```

```
        if (u == -1) return;
        vis[u] = true;
        for (int v = 0; v < n; v++) {
            if (!vis[v] && g[u][v] != INF && dis[v] > dis[u] + g[u][v]) {
                dis[v] = dis[u] + g[u][v];
            }
        }
    }
}
```

补充

时间复杂度：O(n^2)

前驱节点记录

修改位置：for（for if for）

```
    for (int v = 0; v < n; v++) {
        if (!vis[v]) {
            if (dis[v] > dis[u] + g[u][v]) {
                dis[v] = dis[u] + g[u][v];
                pre[v].clear();
                pre[v].push_back(u);
            }
            else if (dis[v] == dis[u] + g[u][v]) {
                pre[v].push_back(u);
            }
        }
    }
```

Bellman-Ford

简单距离（STL 邻接表）

前置

一个 STL 邻接表；

```
int n;
int dis[MAXV];
```

主体

```
// 贝尔曼福德（STL 邻接表）
bool Bellman_Ford(int s) {
```

```
    fill(dis, dis + MAXV, INF);
    dis[s] = 0;
    for (int i = 1; i < n; i++) {
        bool flag = false;
        for (int u = 0; u < n; u++) {
            for (auto& ele : g[u]) {
                if (dis[ele.v] > dis[u] + ele.w) {
                    dis[ele.v] = dis[u] + ele.w;
                    flag = true;
                }
            }
        }
        // 没有可以松弛的边，提前退出
        if (!flag) {
            return false;
        }
    }
    // 再执行一边，检测是否有负权环
    for (int u = 0; u < n; u++) {
        for (auto& ele : g[u]) {
            if (dis[ele.v] > dis[u] + ele.w) {
                return true;
            }
        }
    }
    return false;
}
```

补充

无

简单距离（边集数组）

前置

一个边集数组：

```
int n;
int dis[MAXV];
```

主体

```
// 贝尔曼福德（边集数组）
bool Bellman_Ford(int s) {
    fill(dis, dis + MAXV, INF);
```

```
    dis[s] = 0;
    for (int i = 1; i < n; i++) {
        bool flag = false;
        for (int j = 0; j < cnt; j++) {
            if (dis[edge[j].v] > dis[edge[j].u] + edge[j].w) {
                dis[edge[j].v] = dis[edge[j].u] + edge[j].w;
                flag = true;
            }
        }
        // 没有可以松弛的边，提前退出
        if (!flag) {
            return false;
        }
    }
    // 再执行一边，检测是否有负权环
    for (int i = 0; i < cnt; i++) {
        if (dis[edge[i].v] > dis[edge[i].u] + edge[i].w) {
            return true;
        }
    }

    return false;
}
```

补充

无

动态规划

章节前言：

这是反人类的算法。

动态规划的简单思路是从前到后推导数据。但是实际很少人能想出题解，这是很正常的情况，需要多练。

按方向来分，动态规划可以分为“自上而下”和“自下而上”。“自上而下”主要依靠递归，另外可以附加记忆答案的哈希表。“自下而上”主要依靠迭代。作者推荐大家主要学习自下而上的动态规划，因为自下而上的代码通常更为简短。

按类型来分，主要有 5 种，“01 背包”、“完全背包”、“股票问题”、“子序列问题”、“其它线性 DP”。

本书专注于，介绍模板的使用和效果，会更少地介绍基本原理，所以有什么想不清楚的最好去翻一翻其它书籍。这里我推荐《代码随想录》，这是我学习动态规划的主要工具。

DP 数组如何降维？

动态规划中，若下一层的数据的初始状态，由上层数据复制粘贴得到，那么可以实现 dp 数组的降维。

降维 dp 数组具有重要优势，占用空间小，编写方便。特别是，基于降维 dp 数组的“01 背包”和“完全背包”的模板十分容易记忆。接下来展示的“背包问题”就直接采用降维 dp 数组了，不再介绍原来的版本。

示例

`// 二维数组（一种转移方程）`
`dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]);`
`// 一维数组（转化后的结果）`
`dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);`

01 背包

最大价值

主体

```
// 01 背包
int maxValue(vector<int>& value, vector<int>& weight, int bagSize) {
    // 确定 dp 数组
    vector<int> dp(bagSize + 1, 0);
    // 先物品后背包。背包从后往前遍历
    for (int i = 0; i < weight.size(); i++) {
        for (int j = bagSize; j >= weight[i]; j--) {
            dp[j] = max(dp[j], dp[j-weight[i]]+value[i]);
        }
    }
    return dp[bagSize];
}
```

填充方法数

主体

```
// 01 背包
int findWay(vector<int>& nums, int bagSize) {
    // 确定 dp 数组
    vector<int> dp(bagSize + 1, 0);
    // 初始化
```

```
    dp[0] = 1;
    // 先物品后背包。背包从后往前遍历
    for (int i = 0; i < nums.size(); i++) {
        for (int j = bagSize; j >= nums[i]; j--) {
            dp[j] += dp[j - nums[i]];
        }
    }
    return dp[bagSize];
}
```

完全背包

最大价值

与“01 背包：最大价值”的区别在于换了第二层 for 循环的遍历方向。

主体

```
// 完全背包：最大价值
int maxValue(vector<int>& value, vector<int>& weight, int bagSize) {
    // 确定 dp 数组
    vector<int> dp(bagSize + 1, 0);
    // 先物品后背包。背包从前往后遍历
    for (int i = 0; i < weight.size(); i++) {
        for (int j = weight[i]; j <= bagSize; j++) {
            dp[j] = max(dp[j], dp[j-weight[i]]+value[i]);
        }
    }
    return dp[bagSize];
}
```

还可以简单交换内外层。以下是演示。

主体

```
// 完全背包：最大价值
int findValue(vector<int>& value, vector<int>& weight, int bagSize) {
    // 确定 dp 数组
    vector<int> dp(bagSize + 1, 0);
    // 初始化
    // 先物品后背包。背包从前往后遍历
    for (int j = 0; j <= bagSize; j++) {
        for (int i = 0; i < weight.size(); i++) {
            if (j - weight[i] >= 0) dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
```

```
        }
    }
    return dp[bagSize];
}
```

可以看到，因为把第一层调到了内部，所以指针 j 无法享受指针 i 的“照顾”。要在内层加上一个 if 语句来防止数组越界访问。

完全背包的填充有“排列”和“组合”之分。

在“完全背包：最大价值”的算法中，我们可以知道，交换两层 for 循环是不会改变输出结果的。但是在这里就会有明显的区别。

先物品后容量，这样的选择是稳定的。比如说有一个物品栏：[V,I,N,E]，我们假定选出的物品按先后顺序排列，那么就不会选出类似[N,V]这样的列表。因为必然先有 V 再有 N。

先容量后物品，这样的选择是不稳定的。我们同样假设物品栏是[V,I,N,E]，选出的物品按先后顺序排列。你会发现对物品的一遍 for 循环扫描以后，还可以再来一遍。这样物品栏上的靠前物品就可以在后来的扫描中被选上，从而产生诸如[V,N,N],[N,V,N]这样的元素类型和数量相同的列表。

综上，我们可以知道，“组合”和“排列”的区分在于，物品的扫描（或者说“遍历”）是否会多次从头开始。

只遍历一次，那就是稳定的，得到的结果是“组合”数；

多次从头开始，那就是不稳定的，得到的结果是“排列”数。

读者初次阅读可能会被绕晕。

为什么稳定代表“组合”呢？我们可以这样想，有答案：[V,I,N]、[I,V,N]、[V,N,I]...但是我们只算其中一个，那就是按物品栏相对顺序来存放的[V,I,N]。我们忽略了后者，把它们看作了同一个答案，而这就是“组合”的含义。

“排列”同理可推测，这里不再赘述。

填充方法数：排列

主体

```
// 完全背包：排列
int findWay(vector<int>& nums, int bagSize) {
    // 确定 dp 数组
    vector<int> dp(bagSize + 1, 0);
    // 初始化
    dp[0] = 1;
    // 先容量后物品
    for (int i = 0; i <= bagSize; i++) {
        for (int j = 0; j < nums.size(); j++) {
            if (i - nums[j] >= 0) dp[i] += dp[i-nums[j]];
```

```
        }
    }
    return dp[bagSize];
}
```

填充方法数：组合

主体

```
// 完全背包：组合
int findWay(vector<int>& nums, int bagSize) {
    // 确定 dp 数组
    vector<int> dp(bagSize + 1, 0);
    // 初始化
    dp[0] = 1;
    // 先物品后容量
    for (int i = 0; i < nums.size(); i++) {
        for (int j = nums[i]; j <= bagSize; j++) {
            dp[j] += dp[j - nums[i]];
        }
    }
    return dp[bagSize];
}
```

区分“相同”的概念。

排列组合有些需要注意的地方，“价值相同的元素”和“同一个元素”的概念是不一样的。以下拿排列作讲解：

若背包容量是 2。物品栏存在“价值相同的元素”：[1,1']，那么统计的排列是：[1,1']，[1',1]，[1',1']，[1,1]，输出 4。物品栏存在的是“同一个元素”：[1]，那么统计的排列就是[1,1]，输出 1。

股票问题

至多购一张

主体

```
// 股票问题（一）
int maxProfit1(vector<int>& prices) {
    int len = prices.size();
    if (len == 0) return 0;
    vector<vector<int>> dp(len, vector<int>(2, 0));
    dp[0][0] -= prices[0];
    for (int i = 1; i < len; i++) {
```

```
            dp[i][0] = max(dp[i - 1][0], -prices[i]);
            dp[i][1] = max(dp[i - 1][1], prices[i] + dp[i - 1][0]);
        }
        return dp[len - 1][1];
}
```

这里的 dp[i][0]和 dp[i][1]分别代表在第 i+1 天“持有股票”和“不持有股票”所能获得的最大利润。所以说，“股票问题”也是“状态问题”，求某件物在某处的某个状态的最佳效果。

```
// 股票问题（一，压缩）
int maxProfit(vector<int>& prices) {
    int len = prices.size();
    if (len == 0) return 0;
    vector<vector<int>> dp(2, vector<int>(2, 0));
    dp[0][0] -= prices[0];
    for (int i = 1; i < len; i++) {
        dp[i % 2][0] = max(dp[(i - 1) % 2][0], -prices[i]);
        dp[i % 2][1] = max(dp[(i - 1) % 2][1], prices[i] + dp[(i - 1) % 2][0]);
    }
    return dp[(len - 1) % 2][1];
}
```

压缩的编写方式是减少 dp 数组的空间为 4，然后令每个 dp 数组的第一个空“%2”。但是由于压缩版不容易直接写出来，所以作者推荐先记未压缩版。

同时限持一

唯一不同：

```
dp[i][0] = max(dp[i - 1][0], -prices[i]);
```

改为了：

```
dp[i][0] = max(dp[i - 1][0], dp[i - 1][1] - prices[i]);
```

即在第一条 dp 状态转移方程的“-prices[i]”前加上“dp[i - 1][1]”。

主体

```
// 股票问题（二）
int maxProfit(vector<int>& prices) {
    int len = prices.size();
    if (len == 0) return 0;
    vector<vector<int>> dp(len, vector<int>(2, 0));
    dp[0][0] -= prices[0];
```

```
    for (int i = 1; i < len; i++) {
        dp[i][0] = max(dp[i - 1][0], dp[i - 1][1] - prices[i]);
        dp[i][1] = max(dp[i - 1][1], dp[i - 1][0] + prices[i]);
    }
    return dp[len - 1][1];
}
```

同时限持一，至多购 N 张

以限购 2 张为例，那么 dp 数组可以表示为：

dp[i][0] 不操作
dp[i][1] 第一次持有
dp[i][2] 第一次不持有
dp[i][3] 第二次持有
dp[i][4] 第二次不持有

主体

```
// 股票问题（三）
int maxProfit(vector<int>& prices) {
    int len = prices.size();
    if (len == 0) return 0;
    vector<vector<int>> dp(len, vector<int>(5, 0));
    dp[0][1] = -prices[0];
    dp[0][3] = -prices[0];
    for (int i = 1; i < len; i++) {
        dp[i][0] = dp[i - 1][0];
        dp[i][1] = max(dp[i - 1][1], dp[i - 1][0] - prices[i]);
        dp[i][2] = max(dp[i - 1][2], dp[i - 1][1] + prices[i]);
        dp[i][3] = max(dp[i - 1][3], dp[i - 1][2] - prices[i]);
        dp[i][4] = max(dp[i - 1][4], dp[i - 1][3] + prices[i]);
    }
    return dp[len - 1][4];
}
```

矩阵快速幂

左矩阵一行行地读取，右矩阵一列列地读取。
通过简单的线性代数知识即可实现如下转换：

```
[T(n+1)]   [1,1,1][T(n  )]
[T(n  )] = [1,0,0][T(n-1)]
[T(n-1)]   [0,1,0][T(n-2)]
```

```cpp
// 力扣 1137
class Solution {
private:
    using LL = long long;
public:
    vector<vector<LL>> mul(
        vector<vector<LL>>& a,
        vector<vector<LL>>& b) {
        vector<vector<LL>> res(b.size(), vector<LL>(a.size(), 0));
        for (auto i = 0; i < a.size(); i++) {
            for (auto j = 0; j < b[0].size(); j++) {
                for (auto k = 0; k < a[0].size(); k++) {
                    res[i][j] += a[i][k] * b[k][j];
                }
            }
        }
        return res;
    }
    int tribonacci(int n) {
        if (n == 0) return 0;
        if (n == 1 || n == 2) return 1;
        vector<vector<LL>> ans = {
            {1},{1},{0}
        };
        vector<vector<LL>> mat = {
            {1,1,1},
            {1,0,0},
            {0,1,0}
        };
        int k = n - 2;
        while (k != 0) {
            if (k & 1) ans = mul(mat, ans);
            mat = mul(mat, mat);
            k >>= 1;
        }
        return ans[0][0];
    }
};
```

更标准的写法是：

```cpp
// 力扣 1137
class Solution {
private:
```

```
    using LL = long long;
public:
    int tribonacci(int n) {
        if (n == 0) {
            return 0;
        }
        if (n <= 2) {
            return 1;
        }
        vector<vector<long>> q = { {1, 1, 1}, {1, 0, 0}, {0, 1, 0} };
        vector<vector<long>> res = pow(q, n);
        return res[2][0] + res[2][1];
    }

    vector<vector<long>> pow(vector<vector<long>>& a, long n) {
        vector<vector<long>> ret = { {1, 0, 0}, {0, 1, 0}, {0, 0, 1} };
        while (n > 0) {
            if ((n & 1) == 1) {
                ret = multiply(ret, a);
            }
            n >>= 1;
            a = multiply(a, a);
        }
        return ret;
    }

    vector<vector<long>> multiply(vector<vector<long>>& a, vector<vector<long>>& b) {
        vector<vector<long>> c(3, vector<long>(3));
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                c[i][j] = a[i][0] * b[0][j] + a[i][1] * b[1][j] + a[i][2] * b[2][j];
            }
        }
        return c;
    }
};
```

为何返回的是 return res[0][2]?

按照方法二的公式

列矩阵【T(n+2)，T(n+1)，T(n)】=n 个 M 矩阵相乘，

再与列矩阵【T(2),T(1),T(0)】(1,1,0)相乘，

最后得到的是三行一列的矩阵，T(n)应该等于 res[2][0]*T(2)+res[2][1]*T(1)+res[2][2]*T(0)

带入 T(2),T(1),T(0)的值(1,1,0)，最后 T(n)=res[2][0]+res[2][1];

列矩阵【T(n)，T(n-1)，T(n-2)】=n 个 M 矩阵相乘，再与列矩阵【T(0),T(-1),T(-2)】(0,0,1)相乘，

于是乎，T(n)等于 res[0][0]*T(0)+res[0][1]*T(-1)+res[0][2]*T(-2)
带入 T(0),T(-1),T(-2)的值(0,0,1)，最后 T(n)=res[0][2]；
至于 T(-1),T(-2)为何为 0 和 1，是因为
T(-1)=T(2)-T(1)-T(0)=1-1-0=0
T(-2)=T(1)-T(0)-T(-1)=1-0-0=1

区间信息维护与查询

RMQ 区间查询

接下来介绍三种，ST 算法、线段树、树状数组。

算法\时间复杂度	创建	查询	修改
ST 算法	O(nlogn)	O(1)	不支持
线段树	O(nlogn)	O(logn)	O(logn)
树状数组	O(nlogn)	O(logn)	O(logn)

总结：

ST 算法：适合查询频繁且不需要修改的场景。
线段树：适合查询和修改都频繁的场景，支持更复杂的查询，如区间和、区间最大值等。
树状数组：适合前缀和查询和单点更新频繁的场景，不支持区间修改。

ST 算法

这里演示的是查询给定区域的最大值和最小值的差。

由于 F[j][i]表示的是 j 到 j+2^i-1 之间的区间，所以当 i 设定为 20 时可以覆盖 1 到 5e5 的范围，设定为 32 时可覆盖到 1 到 2e9 的范围。

前置

```
// 倍增表
int Fmax[MAXN][32];
int Fmin[MAXN][32];
// 准备一个数组和一个变量
// 范围从 1 到 n
int a[MAXN], n;
```

创建

```
// 创建稀疏表
void ST_create() {
    for (int i = 1; i <= n; i++) {
        Fmax[i][0] = a[i];
        Fmin[i][0] = a[i];
    }
    int k = log2(n);
    // 倍率
    for (int i = 1; i <= k; i++) {
        // 动态规划
        for (int j = 1; j + (1 << i) <= n - 1; j++) {
            Fmax[j][i] = max(Fmax[j][i - 1], Fmax[j + (1 << (i - 1))][i - 1]);
            Fmin[j][i] = min(Fmin[j][i - 1], Fmin[j + (1 << (i - 1))][i - 1]);
        }
    }
}
```

查询

```
int RMQ(int left, int right) {
    int k = log2(right - left + 1);
    int m1 = max(Fmax[left][k], Fmax[right - (1 << k) + 1][k]);
    int m2 = min(Fmin[left][k], Fmin[right - (1 << k) + 1][k]);
    return m1 - m2;
}
```

线段树

这里演示的是查询给定区域的元素和。

主要学习 4 个函数，创建、查询、下压、修改（标记）。

如果是静态查询，那么只要“创建”和“查询”函数；如果要加入“区间修改”功能，那么需要 4 个函数，这时“下压”会成为“查询”和“修改”函数的前置。

创建和查询

前置

```
// 准备一个数组
// 范围从 1 到 n
int a[MAXN];
// 静态数组
```

```
struct Node {
    int l, r;
    int sum;
    int lz;
}tree[MAXN * 4];
```

创建

```
//创建
void build(int i, int l, int r) {
    tree[i].l = l;
    tree[i].r = r;
    if (l == r) {
        tree[i].sum = a[l];
        return;
    }
    int mid = l + r >> 1;
    build(i * 2, l, mid);
    build(i * 2 + 1, mid + 1, r);
    tree[i].sum = tree[i * 2].sum + tree[i * 2 + 1].sum;
}
```

采用了后序遍历，先划分，然后递归从小到大确定每个区间的值。函数的结构是，给区间变量赋值，接下来赋值 sum 数据。

查询

```
// 区间查询
int search(int i, int l, int r) {
    if (tree[i].l >= l && tree[i].r <= r) return tree[i].sum;
    if (tree[i].l > r || tree[i].r < l) return 0;
    push_down(i);
    int s = 0;
    if (tree[i * 2].r >= l) s += search(i * 2, l, r);
    if (tree[i * 2 + 1].l <= r) s += search(i * 2 + 1, l, r);
    return s;
}
```

如果没有修改的必要，那么可以把函数里面的 push_down 函数删去。关于 push_down 函数，接下来会讲。

区间更新

下压

```
// 下压
void push_down(int i) {
    if (tree[i].lz != 0) {
        int l = i * 2;
        int r = i * 2 + 1;
        tree[l].lz += tree[i].lz;
        tree[r].lz += tree[i].lz;
        tree[l].sum += tree[i].lz * (tree[l].r - tree[l].l + 1);
        tree[r].sum += tree[i].lz * (tree[r].r - tree[r].l + 1);
        tree[i].lz = 0;
    }
}
```

区间标记

```
// 标记
void modify(int i, int l, int r, int val) {
    if (tree[i].l >= l && tree[i].r <= r) {
        tree[i].sum += val * (tree[i].r - tree[i].l + 1);
        tree[i].lz += val;
        return;
    }
    push_down(i);
    if (tree[i * 2].r >= l) modify(i * 2, l, r, val);
    if (tree[i * 2 + 1].l <= r) modify(i * 2 + 1, l, r, val);
    tree[i].sum = tree[i * 2].sum + tree[i * 2 + 1].sum;
}
```

找到合适的区间，标记起来。

关于下压函数，我有一个技巧，那就是在递归之前下压。

单查询（可被代替）

遵循一个原则，位置在哪边就进入哪个区间。抗逆性不如 search 函数。至少 search 函数越界依然可以发挥作用。而且特别麻烦的是，每次查询，都要把 ans 初始化为 0。

单点查询

```
// 单点查询的结果
int ans = 0;
```

```
// 单点查询
void query(int i, int pos) {
    ans += sign[i].sum;
    if (sign[i].l == sign[i].r) return;
    int mid = sign[i].l + sign[i].r >> 1;
    if (pos <= mid) query(i * 2, pos);
    else query(i * 2 + 1, pos);
}
```

树状数组

前置部分

前置

```
// 普通数组和树状数组
// 范围从 1 到 n
int a[MAXN];
int c[MAXN];
int n;
```

前驱后继区间计算

```
// 计算跳跃距离
int lowBit(int i) {
    return (-i) & i;
}
```

简单来讲，跳跃的距离是当前位置的二进制下，从低位到高位第一个 1 出现的位置的值。前驱指的是左侧兄弟节点，后继指的是右侧父节点，前驱会往左走，后继会往右走。

更新与初始化

更新

```
// 更新某个位置
void add(int i, int val) {
    for (; i <= n; i += lowBit(i)) {
        c[i] += val;
    }
}
```

初始化

```
// 初始化
for (int i = 1; i <= n; i++) {
    add(i, a[i]);
}
```

区间查询

区间查询

```
// 查询区间
int search(int l, int r) {
    return sum(r) - sum(l-1);
}
```

“树状数组”比“线段树”的“区间查询”的抗逆性弱一点，一旦区间查询反过来就会导致 bug。作者觉得，这里的树状数组的 bug 和前缀表的 bug 有点像。

LCA 最近祖先

DFS 暴力解

这是作者最早接触的 LCA 算法题型，来自：力扣 236.二叉树的最近公共祖先

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        // 如果其中一个节点是祖先，
        // 那么所有路径只有这个节点的返回。
        // 如果这个节点是两节点外的祖先，
        // 那么它的 left 和 right 应当不返回 nullptr
        if (root == nullptr || root == p || root == q) {
            return root;
```

```
        }
        TreeNode* left = lowestCommonAncestor(root->left, p, q);
        TreeNode* right = lowestCommonAncestor(root->right, p, q);
        if (left && right) {
            return root;
        }
        return left ? left : right;
    }
};
```

查询的时间复杂度是 O(n)，创建时间大概是 0，因为它直接使用了原链树。这也是本书的 LCA 问题的模板中唯一对链树的解法。

树上倍增法

注意，这里的 ST 算法与 RMQ 问题的 ST 算法的含义不同，前者表示的是倍增以后所到达的位置，后者表示区间内的最值。

前置

```
int F[MAXN][32];
int D[MAXN];
int n, k;
```

接下来要初始化，初始化的前提是已获得 f[j][0]，即每个节点的父节点。如果您还不知道怎么获得 f[j][0]，这里提供了一种基于 DFS 的思路：

前置

```
// 子节点链表
list<int> C[MAXN];
```

父节点与深度获取

```
// 获取树的必要信息
void DFS(int pre, int num, int deep) {
    F[num][0] = pre;
    D[num] = deep;
    for (auto& ele : C[num]) {
        DFS(num, ele, deep + 1);
    }
}
```

初始化

```
// 建表
void ST_create() {
    for (int i = 1; i <= k; i++) {
        for (int j = 1; j <= n; j++) {
            F[j][i] = F[F[j][i - 1]][i - 1];
        }
    }
}
```

查询的时候要用到数组 D[MAXN]，如果还未获取可参照上文的“父节点与深度获取”。

查询

```
// 查询
int LCA(int x, int y) {
    // 使得 y 离祖宗更远
    if (D[x] > D[y]) {
        swap(x, y);
    }
    // 从 y 出发，往上走
    for (int i = k; i >= 0; i--) {
        if (D[F[y][i]] >= D[x]) {
            y = F[y][i];
        }
    }
    // 发现祖宗是 x
    if (x == y) return x;
    // 一起向上走
    for (int i = k; i >= 0; i--) {
        if (F[x][i] != F[y][i]) {
            x = F[x][i];
            y = F[y][i];
        }
    }
    return F[x][0];
}
```

对了，不要忘记计算 k 的值（k = log2(n);）。写的时候要注意，不要重复定义，否则会出现赋值给局部变量 k，而不改变全局变量 k 的情况。以下是实机演示：

```
int main() {
```

```
    int root = 8;
    n = 16;
    k = log2(n);

    C[8] = list<int>{ 5, 4, 1 };
    C[5] = list<int>{ 9 };
    C[4] = list<int>{ 6, 10 };
    C[1] = list<int>{ 14, 13 };
    C[6] = list<int>{ 15, 7 };
    C[10] = list<int>{ 11, 16, 2 };
    C[16] = list<int>{ 3, 12 };

    DFS(root, root, 1);
    // 第一个空填 root 或 0，推荐填 root。
    // 第二个空填随你便
    ST_create();
    cout << LCA(5, 1) << endl;
    cout << LCA(8, 1) << endl;
    cout << LCA(6, 14) << endl;
    cout << LCA(10, 15) << endl;
    cout << LCA(15, 13) << endl;
    cout << LCA(3, 2) << endl;
    cout << LCA(15, 7) << endl;
    cout << LCA(3, 12) << endl;
    //输出应为 8 8 8 4 8 10 6 16
    return 0;
}
```

在线 RMQ 算法

该算法通过查询 DFS 遍历的路径，找到两个点最早出现的位置。祖先就在这两个位置的区间内。显然祖先就是这个区间内深度最小的元素。

对于这个区间，想要快速查询，我们自然而然地想到，这不就是求区间最值吗，最值对应的元素就是我们想要的元素。

所以这里作者使用 RMQ 的 ST 算法，注意这里的 ST 算法有所不同。要处理映射关系，我们的 F[][]主要是为元素服务的，而不是深度，所以 D[]只是我们用来判断的工具，接下来你将会注意到这点。

前置

```
// 属性
list<int> C[MAXN];
int D[MAXN];
// 访问
vector<int> seq = { 0 };
```

```
bool vis[MAXN];
int pos[MAXN];
```

这里给 seq 先垫了一个 0 用来占位，使得元素由 1 到 len。这里的 len 是有效数据段的长度，也是 seq.size()-1。稍后，会给出 len 全局变量。

获取欧拉序列、深度哈希表

```
void DFS(int num, int deep) {
    vis[num] = true;
    pos[num] = seq.size();
    seq.push_back(num);
    D[num] = deep;
    for (auto& ele : C[num]) {
        if (!vis[ele]) {
            DFS(ele, deep + 1);
            seq.push_back(num);
        }
    }
}
```

ST 算法的前置

```
// 下标对应 seq 的下标，记录对应 seq 的数字
int F[MAXN<<2][20];
int len;
```

实际上作者没有仔细计算过欧拉序列 F 数组要开多长，只是粗略推测了一下。DFS 遍历从宏观上来讲不是每个元素都要走一个来回吗？考虑到中转节点要写入不止两次，开 2 倍空间长度一定不够用，所以应当开 4 倍空间。

ST 算法

```
void ST_create() {
    len = seq.size();
    for (int i = 1; i < len; i++) {
        F[i][0] = seq[i];
    }
    int k = log2(len);
    for (int i = 1; i <= k; i++) {
        for (int j = 1; j + (1<<i)-1 <= len; j++) {
            if (D[F[j][i - 1]] < D[F[j + (1 << (i - 1))][i - 1]]) {
```

```
                F[j][i] = F[j][i - 1];
            }
            else {
                F[j][i] = F[j + (1 << (i - 1))][i - 1];
            }
        }
    }
}
int RMQ(int l, int r) {
    int k = log2(r - l + 1);
    if (D[F[l][k]] < D[F[r - (1 << k) + 1][k]]) {
        return F[l][k];
    }
    else {
        return F[r - (1 << k) + 1][k];
    }
}
```

查询

```
int LCA(int x, int y) {
    int l = pos[x];
    int r = pos[y];
    if (l > r) {
        swap(l, r);
    }
    return seq[RMQ(l, r)];
}
```

以下是作者在研究的时候用的演示：

```
int main() {
    C[1] = list<int>{ 2,3 };
    C[2] = list<int>{ 4,5 };
    C[4] = list<int>{ 6 };
    C[5] = list<int>{ 7 };
    C[6] = list<int>{ 8,9 };
    DFS(1, 1);
    for (auto& ele : seq) {
        cout << ele << " ";
    } cout << endl;
    for (auto& ele : seq) {
```

```
        cout << D[ele] << " ";
    } cout << endl;
    ST_create();
    cout << LCA(2, 3) << endl;
    cout << LCA(1, 1) << endl;
    cout << LCA(6, 7) << endl;
    cout << LCA(6, 9) << endl;
    cout << LCA(8, 3) << endl;
    // 输出应为：
    // 0 1 2 4 6 8 6 9 6 4 2 5 7 5 2 1 3 1
    // 0 1 2 3 4 5 4 5 4 3 2 3 4 3 2 1 2 1
    // 1 1 2 6 1
    return 0;
}
```

世界就是树

树的遍历

BFS 扩散

队列的节点

```
struct Node {
    int x;
    int y;
    int z;
    Node(int a, int b, int c) :
        x(a), y(b), z(c) {}
};
```

注：低于三维时，可以不用编写“队列的节点”

扩散字典

```
int X[6] = { 1, 0, 0, -1, 0, 0 };
int Y[6] = { 0, 1, 0, 0, -1, 0 };
int Z[6] = { 0 ,0, 1, 0, 0, -1 };
```

节点合法性判断

```
bool judge(int x, int y, int z) {
    if (x >= n || x < 0) return false;
    if (y >= m || y < 0) return false;
    if (z >= l || z < 0) return false;
    if (v[x][y][z] == 1) return false;
    if (g[x][y][z] == 0) return false;
    return true;
}
```

广度优先遍历

```
// 统计有效体积（可选）
int Volume = 0;
// 广度优先队列
void BFS(int x, int y, int z) {
    if (judge(x, y, z)) {
        int nowVolume = 0;

        queue<Node> q;
        q.push(Node(x, y, z));
        v[x][y][z] = 1;

        while (!q.empty()) {
            Node now = q.front();
            q.pop();
            nowVolume++;

            for (int i = 0; i < 6; i++) {
                int nowX = now.x + X[i];
                int nowY = now.y + Y[i];
                int nowZ = now.z + Z[i];

                if (judge(nowX, nowY, nowZ)) {
                    q.push(Node(nowX, nowY, nowZ));
                    v[nowX][nowY][nowZ] = 1;
                }
            }
        }
        // 保存大于阈值的体积（可选）
        if (nowVolume >= t) {
            Volume += nowVolume;
        }
```

```
    }
}
```

DFS 回溯

模板一

```
void DFS(int s, int index, int deep) {
    path.push_back(s);
    if (deep == /* 这里填入一个目标深度 */) {
        path.pop_back();
        return;
    }
    for (int i = index; i < arr.size(); i++) {
        DFS(arr[i], i, deep + 1);
    }
    path.pop_back();
}
```

模板二

```
void DFS(int index, int deep) {
    if (deep == /* 这里填入一个目标深度 */) {
        return;
    }
    for (int i = index; i < arr.size(); i++) {
        path.push_back(arr[i]);
        DFS(i, deep + 1);
        path.pop_back();
    }
}
```

经过观察可以知道第二种会在进入 if 语句的时候少第一个元素。

树的运算

集合运算（STL）

本文介绍四种

填空：

第一至第四：填写迭代器（指针也可以）
第五：填写输出的容器的迭代器或指针
最后的可选空：填写元素之间的比较器

注意事项：

1. multiset 也可以使用，但是基于哈希表的 unordered_set 不行。

2. 第五个空的迭代器或指针对应的输出容器，内存必须充足，否则运行以后程序崩溃。
   可填 ostream_iterator<int>(cout, " ")，表示打印；
   或者填 inserter 迭代器，可以写入容器；
   .....更多的用法，需要使用者去挖掘。

3. 算法会消除容器间的重复项，但是对输入容器自身重复的元素不会消除。
   例如：　[1,2,2,4]和[1,2,5]求并集，得[1,2,2,4,5]。
   可以看到第一个集合里面的两个 2 并未去重

主体

```
// 迭代器（每次算法运算完以后，迭代器会被改变，需要刷新）
multiset<int>::iterator beg1 = str1.begin();
multiset<int>::iterator end1 = str1.end();
multiset<int>::iterator beg2 = str2.begin();
multiset<int>::iterator end2 = str2.end();

set_union                   (beg1, end1, beg2, end2, dest);
set_intersection            (beg1, end1, beg2, end2, dest);
set_difference              (beg1, end1, beg2, end2, dest);
set_symmetric_difference    (beg1, end1, beg2, end2, dest);
```

实际上这些集合算法，不是和 set 绑定的，可以用于其它容器。但是有必须明确的是，这些容器内部元素的排序必须是有序的，否则运行以后程序崩溃。

补充示范（数组）

```
int num1[4] = { 0, 1, 3, 4 };
int num2[3] = { 2, 3, 4 };
int num3[10];
set_union(num1, num1+4, num2, num2+3, num3);
```

关系树

并查集：测连通性的树
KD 树：测最近点的树

并查集

前置

```
int f[MAXN];
```

主体

```
// 刷新
void init(int num) {
    for (int i = 0; i < num; i++) {
        f[i] = i;
    }
}
// 查找祖先
int find(int v) {
    if (f[v] != v) f[v] = find(f[v]);
    return f[v];
}
// 把 a 集合绑定到 b 集合
void merge(int a, int b) {
    int fa = find(a);
    int fb = find(b);
    f[fa] = fb;
}
```

STL 扩展

```
// STL 并查集容器
map<int, set<int>> toSet;

// 收集到 STL 容器
void collect(int num) {
    for (int i = 0; i < num; i++) {
        toSet[find(i)].insert(i);
    }
}
// 从 STL 容器读取（双现代迭代，快捷操作）
void load() {
    for (auto& aPair : toSet) {
        for (auto& ele : aPair.second) {
            /* 在这里填上 存放的容器 */
        }
    }
```

```
}
// 从 STL 容器读取（单现代迭代，精准操作）
void load() {
    for (auto& aPair : toSet) {
        set<int>& aSet = aPair.second;
        for (auto i = aSet.begin(); i != aSet.end();) {
            /* 在这里填上 存放的容器 */
            if (i++ != aSet.end()) {
                /* 在这里填上 间隔元素 */
            }
        }
        /* 在这里填上 末尾处理方法 */
    }
}

// 从 STL 容器读取，双现代迭代（示范）
void load() {
    ostream_iterator<int> it(cout, " ");
    for (auto& aPair : toSet) {
        copy(aPair.second.begin(), aPair.second.end(), it);
        cout << endl;
    }
}
//从 STL 容器读取，单现代迭代（示范）
void load() {
    for (auto& aPair : toSet) {
        set<int>& aSet = aPair.second;
        for (auto i = aSet.begin(); i != aSet.end();) {
            printf("%d", *i);
            if (i++ != aSet.end()) {
                printf(" ");
            }
        }
        printf("\n");
    }
}
```

平衡二叉树

节点的定义（前置）

```
// 建议指针默认构造为 nullptr（如下）
struct Node {
    int val;
```

```
    int height;
    Node* left;
    Node* right;
    Node(int a, int b) {
        val = a;
        height = b;
        left = nullptr;
        right = nullptr;
    }
};
// 可以用 list 来管理所有动态分配的节点（如下）
list<Node> tree;
```

平衡二叉树要默写的最基本的函数有：

1. getHeight 获取高度
2. getFactor 获取平衡因子
3. update 更新高度
4. L 左旋
5. R 右旋
6. insert 插入

获取高度

```
// 树的高度是倒着计算的
// 树根的高度最高，叶子节点的高度是 1
int getHeight(Node* root) {
    if (root == nullptr) return 0;
    return root->height;
}
```

返回当前节点的成员变量 height。

因为一个节点的高度至少为 1，所以如果节点不存在就返回 0。

获取平衡因子

```
int getFactor(Node* root) {
    return getHeight(root->left) - getHeight(root->right);
}
```

左子树高度-右子树高度

更新高度

```
void update(Node* root) {
    root->height = max(getHeight(root->left), getHeight(root->right)) + 1;
}
```

把当前节点的高度赋值为左右子树的最高高度，然后加 1。（注意：后面这个加 1 很关键）

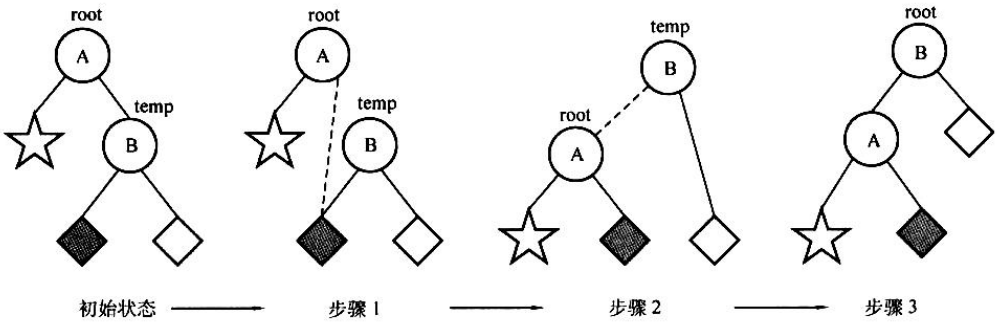
左旋右旋

```
void L(Node*& root) {
    Node* temp = root->right;
    root->right = temp->left;
    temp->left = root;
    update(root);
    update(temp);
    root = temp;
}
void R(Node*& root) {
    Node* temp = root->left;
    root->left = temp->right;
    temp->right = root;
    update(root);
    update(temp);
    root = temp;
}
```

这里采用指针引用，root 实际为树的局部根的“位置”
（若一个指针赋值给了 root，那么这个指针就成为了这部分树的根）

1. 获取子节点
2. 根接管子节点的叶子
3. 子节点接管根
4. 子节点成为根

以下引自《算法笔记》，该书解释得够好。



插入

```
void insert(Node*& root, int val) {
    if (root == nullptr) {
        // 新建节点
        tree.push_back(Node(val, 1));
        root = &tree.back();
    }
    else if (root->val > val) {
        insert(root->left, val);
        update(root);
        if (getFactor(root) == 2) {
            if (getFactor(root->left) == 1) {
                R(root);
            }
            else if (getFactor(root->left) == -1) {
                L(root->left);
                R(root);
            }
        }
    }
    else if (root->val < val) {
        insert(root->right, val);
        update(root);
        if (getFactor(root) == -2) {
            if (getFactor(root->right) == -1) {
                L(root);
            }
            else if (getFactor(root->right) == 1) {
                R(root->right);
                L(root);
            }
        }
```

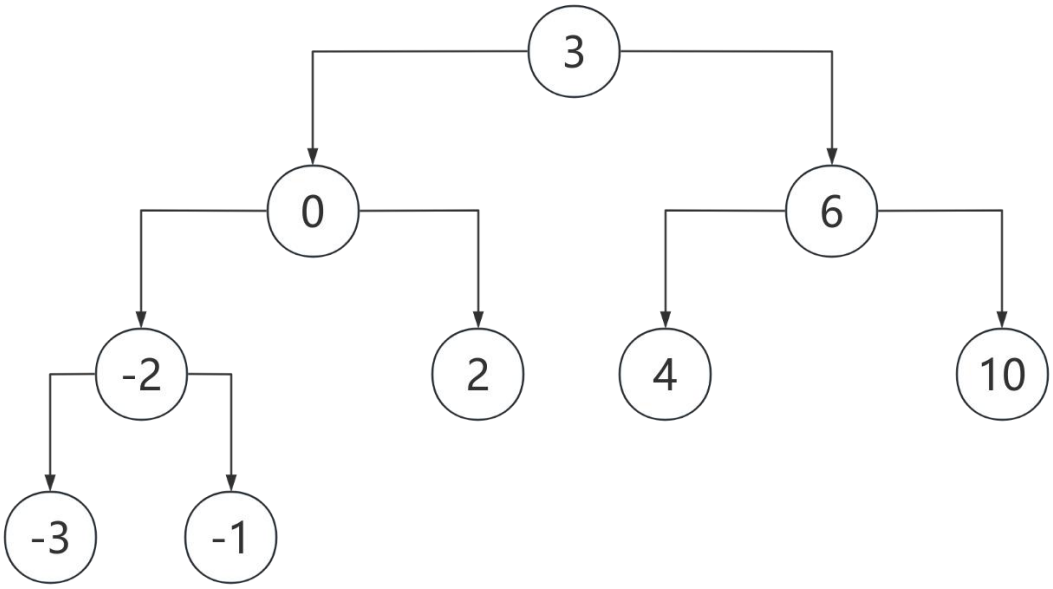
```
    }
}
```

测试

```
// 查询
void search(Node* root, int val) {
    if (root == nullptr) {
        printf("不存在\n");
    }
    else if (val == root->val) {
        // 这里填写查找成功后输出的信息，例如：
        printf("节点 %d 存在，高度为 %d ，", root->val, root->height);
        if (root->left) printf("左孩子为 %d ，", root->left->val);
        else printf("无左孩子，");
        if (root->right) printf("右孩子为 %d \n", root->right->val);
        else printf("无右孩子\n");
    }
    else if (val < root->val) {
        search(root->left, val);
    }
    else {
        search(root->right, val);
    }
}
```

```
// 样例
int main() {
    int data[11] = { 3, 4, 2, 10, 4, 6, 10, 0, -1, -2, -3 };
    int n = 11;
    Node* root = nullptr;
    for (int i = 0; i < n; i++) {
        insert(root, data[i]);
    }
    search(root, 3);
    search(root, 4);
    search(root, 10);
    search(root, -2);
    search(root, 0);
    // 测试一个不存在的值
    search(root, 9);
    return 0;
}
```

测试样例输出的树的模型：



FHQ-Treap

必备模板！全靠同行衬托，其它平衡树要么原理复杂，要么实现困难
不过我猜，通过上节“平衡二叉树”的学习，你多少有些相似的感触了

结构：非严格平衡树（不总是严格平衡），非严格堆（不是完全二叉树）
特性：随机键（key）的堆，值（val）的搜索二叉树（BST）
优点：代码短，不需要旋转，基本操作少

有以下需要注意的点，可以学完以后再回来看：
1. 每次 split 最后一定要有对应的 merge
2. 每次 merge 记得保存函数返回的“根”
3. 结构体的 int size 不建议初始构造为 1（否则计算时会把空节点当作存在的节点）
4. 每次调用常用函数时并不需要给临时变量清零（ins, del, pre, aft, rtv, vtr）

前提内容

结构体＆全局变量

```
// 缠绕器（用 NULL 是为了调试，固定产出）
mt19937 eng(NULL);
// 结构体
struct Node {
    int l, r;
    int key, val;
    int size;
    Node() {
```

```
        key = eng();
    }
} tr[MAXN];
int root, idx;
int x, y, z;
```

更新子树节点数

```
// 更新节点
void update(int p) {
    tr[p].size = tr[tr[p].l].size + tr[tr[p].r].size + 1;
}
```

分裂&合并

用两个引用变量作为函数参数，分别为 x 和 y 代表左右两个“有向路径”，这个路径指向的是“子树”。因为 FHQ-Treap 对值（val）是一种搜索二叉树，所以当前节点的左节点（如果存在）小于等于根节点，当前的右节点（如果存在）大于根节点。

若遇到属于 x 的节点，则它的左子树一定属于 x，右子树可能属于 y，所以要将“该点和它的左子树”加入 x，并递归分裂右子树（加入 x 以后该节点的右节点应视为空，直接用于连接下一个 x 节点）

若遇到属于 y 的节点，则它的右子树一定属于 y，左子树可能属于 x，所以就将“该点和它的右子树”加入 y，并递归分裂左子树（加入 y 以后该节点的左节点应视为空，直接用于连接下一个 y 节点）

分裂操作

```
// 分裂操作
void split(int p, int val, int& x, int& y) {
    if (!p) x = y = 0;
    else {
        if (tr[p].val <= val) {
            x = p;
            split(tr[p].r, val, tr[p].r, y);
        }
        else {
            y = p;
            split(tr[p].l, val, x, tr[p].l);
        }
        update(p);
    }
}
```

本次构建大根堆键（key），所以高优先级会放在上边。若有两个节点，将其划分为一个根节点一个插入节点，那么我们要考虑两个因素。一是维持搜索二叉树的性质，二是维持堆的性质。

由分裂操作所得 x 和 y，可知 x 树上的节点要小于 y 树上的节点，那么 x 树应当始终拼接在 y 树的左

侧。**若 x 的 key 大于 y 的 key，那么 x 在 y 的左上方，此时 y 应当拼接在 x 的右子树**，但是我们并不知道 x 的右节点是否已经存在，所以要递归，传入 x 的右节点和 y，然后将返回的结果重新连接到 x 的右节点。**对于 x 的 key 大于 y 的 key 的情况，y 在 x 的右上方，此时 x 应当拼接在 y 的左子树**，但是我们并不知道 y 的左节点是否已经存在，所以要递归，传入 y 的左节点和 x，然后将返回的结果重新连接到 y 的左节点。

合并操作

```
// 合并操作（前提必须保证 x 的 val 比 y 的 val 小）
int merge(int x, int y) {
    if (!x || !y) return x + y;
    if (tr[x].key > tr[y].key) {
        tr[x].r = merge(tr[x].r, y);
        update(x);
        return x;
    }
    else {
        tr[y].l = merge(x, tr[y].l);
        update(y);
        return y;
    }
}
```

常规的操作

插入：将节点按 val 划分，然后向静态数组上申请一个节点，将这个节点当作一颗子树和 xy 拼在一起

插入

```
// 插入节点
void insert(int val) {
    split(root, val, x, y);
    tr[++idx].val = val;
    tr[idx].size = 1;
    root = merge(merge(x, idx), y);
}
```

删除：按 val 的值划分为 x 和 z 两棵树，那么 val 一定在 x。再按 val-1 将 x 划分为 x 和 y，那么 val 一定在 y。此时 y 可能不止一个节点，所以删去根，然后将它的左右节点拼到 y。最后将 xyz 拼接起来即可。

删除

```
// 删除节点
void del(int val) {
    split(root, val, x, z);
    split(x, val - 1, x, y);
```

```
    y = merge(tr[y].l, tr[y].r);
    root = merge(merge(x, y), z);
}
```

由值获取排名：

按（val - 1）划分为 xy 树，那么 x 树上的节点都小于 val，该树（size + 1）就是 rank

由排名获取值：

从起点开始往左走，那么当前节点的 this_rank 为（L_size + 1）或（this_size - R_size）。若 this_rank > rank，说明目标节点在左子树，跳转到左子树；若 this_rank == rank，说明找到了目标节点，返回目标节点的值；若 this_rank < rank，说明目标节点在右子树，先令 **rank - this_rank**，再跳转到右子树，即转换到“局部 rank 匹配模式”

（注：以下函数并未具备异常处理，即默认目标节点存在）

由值获取排名

```
// 获取排名
int valToRank(int val) {
    split(root, val - 1, x, y);
    int rank = tr[x].size + 1;
    root = merge(x, y);
    return rank;
}
```

由排名获取值

```
// 获取值（对于第一层的 p 参数，请传入 root）
int rankToVal(int p, int rank) {
    int temp = tr[tr[p].l].size + 1;
    if (temp > rank) return rankToVal(tr[p].l, rank);
    else if (temp == rank) return tr[p].val;
    else return rankToVal(tr[p].r, rank - temp);
}
```

获取前驱：以 val-1 划分，那么 x 树的最右边的节点就是前驱

获取后继：以 val 划分，那么 y 树的最左边的节点就是后继

获取前驱

```
// 获取前驱
int pre(int val) {
    split(root, val-1, x, y);
    int p = x;
    while (tr[p].r) p = tr[p].r;
```

```
    root = merge(x, y);
    return tr[p].val;
}
```

获取后继

```
// 获取后继
int aft(int val) {
    split(root, val, x, y);
    int p = y;
    while (tr[p].l) p = tr[p].l;
    root = merge(x, y);
    return tr[p].val;
}
```

替罪羊树

实现：若 min{leftNum, rightNum} > alpha * AllNodeNum 则认为不平衡
将该节点为根节点的子树以中序遍历排列，抽取出所有节点，并且通过二分的形式重构

KD 树

前置

```
// 维度
const int k = 2;
// 原数据的结构体及静态数组储存
struct Node {
    int t[k];
    Node(int x = 0, int y = 0) {
        t[0] = x;
        t[1] = y;
    }
} a[MAXN];
// kd 树的节点的存在（可选，储存式算法）
// kd 树的节点的储存（可选，储存式算法）
bool ex[MAXN << k];
Node kd[MAXN << k];
```

创建

```
void build(int i, int l, int r, int d) {
```

```
    // 是否合法
    if (l > r) return;
    // 选择中点和维度
    int mid = l + r >> 1;
    int dim = d % k;
    // 状态描述（储存式）
    ex[i] = 1;
    ex[i * 2] = 0;
    ex[i * 2 + 1] = 0;
    // 接下来把点移到正确的位置
    nth_element(a + l, a + mid, a + r + 1, [&](Node& x, Node& y) {
        return x.t[dim] < y.t[dim];
        });
    // 节点赋值（可选，储存式算法）
    kd[i] = a[mid];
    // 递归
    build(i * 2, l, mid - 1, d + 1);
    build(i * 2 + 1, mid + 1, r, d + 1);
}
```

前置数学函数

```
// 平方
int sq(int a) {
    return a * a;
}
// 距离的平方
int distance(Node& a, Node& b) {
    int res = 0;
    for (int i = 0; i < k; i++) {
        res += sq(a.t[i] - b.t[i]);
    }
    return res?res:INF;
}
```

运行查询

```
class Method {
private:
    Node p;
    int ans = INF;

    void query(int l, int r, int d) {
        if (l > r) return;
```

```
        int mid = l + r >> 1;
        int dim = d % k;
        int dis = distance(a[mid], p);
        if (ans > dis) {
            ans = dis;
        }

        int cir = sq(a[mid].t[dim] - p.t[dim]);
        if (p.t[dim] < a[mid].t[dim]) {
            query(l, mid - 1, d + 1);
            if (cir < ans) {
                query(mid + 1, r, d + 1);
            }
        }
        else {
            query(mid + 1, r, d + 1);
            if (cir < ans) {
                query(l, mid - 1, d + 1);
            }
        }
    }
public:
    void run(int x, int y, int n) {
        p = Node(x, y);
        ans = INF;
        query(1, n, 1);
        cout << ans << endl;
    }
};
```

递归树

Master 公式

公式：T(n) = a*T(n/b) + O(n^c)

若 log(b, a) < c，复杂度为：O(n^c)

若 log(b, a) > c，复杂度为：O(n^log(b,a))

若 log(b, a) == c，复杂度为：O(n^c * logn)

以 b 为底数（Base），a 为真数（Argument）

若对数大于 c，那么时间复杂度以“对数为主导”

若对数小于 c，那么时间复杂度以“c 为主导”

例如：归并排序为 T(n) = 2*T(n/2) + O(n)

T(n) 是排序 n 个元素所需的时间

2*T(n/2) 表示将数组分成“两半”，对“每半”递归

O(n) 表示对单层数据的处理时间复杂度，这里是合并两个已排序的子数组的复杂度

汉诺塔

汉诺塔揭示的几个重要道理（本作者自己想的）：

1) 移动元素等操作（operator）不一定真的需要修改数据，能显示“结论”就行。例如说，汉诺塔中的“移动”就用 printf 来表现的，根本就不涉及真正的元素移动。
2) 参数列表上的元素在进入下一层递归的时候，可以进行一种“交换”操作，以改变元素扮演的角色。显而易见的是，这种交换无需中间变量。
3) 递归应该涉及“返回”，无返回需求不用递归。通常是前面层的答案要从下一层执行结束以后得出。

函数

```
void hanoi(int n, char from, char aux, char to) {
    if (n == 1) printf("%d from %c to %c\n", n, from, to);
    else {
        hanoi(n - 1, from, to, aux);
        printf("%d from %c to %c\n", n, from, to);
        hanoi(n - 1, aux, from, to);
    }
}
```

归并排序

前置

```
// 逆序数（可选）
int cnt = 0;
```

主体

```
// 归并
void merge(int left, int mid, int right, int* arr) {
    // 数组两段指针
    int i = left;
    int j = mid + 1;
    // 临时数组
    int temp[MAXN];
    int t = 0;
    // 选择和并入
```

```
    while (i <= mid && j <= right) {
        if (arr[i] > arr[j]) {
            temp[t++] = arr[j++];
            // 逆序数收集器（可选）
            cnt += mid - i + 1;
        }
        else {
            temp[t++] = arr[i++];
        }
    }
    // 拼接
    while (i <= mid)  temp[t++] = arr[i++];
    while (j <= right)    temp[t++] = arr[j++];
    // 复制临时数组，粘贴回原数组
    for (i = 0; i < t; i++) {
        arr[left + i] = temp[i];
    }
}
// 划分
void mergeSort(int left, int right, int arr[]) {
    if (left < right) {
        int mid = (left + right) / 2;
        mergeSort(left, mid, arr);
        mergeSort(mid + 1, right, arr);
        merge(left, mid, right, arr);
    }
}
```

补充

注意事项：这里的归并排序使用左闭右闭区间

拓展功能：归并排序可以用来求逆序数

线性与指针

滑动窗口

有几种写法：

1. 单指针
2. 双指针
3. 队列

更新答案的时机往往和题目有关。若题目要求“严格的窗口长度”，即仅考虑窗口为设定长度的时候

更新的答案。那么程序在运行到达既定长度之前就不应该更新答案。

注：“不更新答案”不等于“不更新数据”

在到达窗口设定长度之前，可在“主循环内”扩展窗口长度，也可以在“主循环外”。

如果在“内部”进行，那么主循环分两部分，第一部分是到达既定长度之前的操作，到达既定长度以后就解锁第二部分区域的操作。

单指针

要计算“慢指针”的位置，本质还是双指针

```
// 获取最长元音字母子串
class Solution {
private:
    bool isVowel[128] = { 0 };
public:
    int maxVowels(string& s, int k) {
        isVowel['a'] = 1;
        isVowel['e'] = 1;
        isVowel['i'] = 1;
        isVowel['o'] = 1;
        isVowel['u'] = 1;
        int ans = 0;
        int vowel = 0;
        for (int i = 0; i < s.length(); i++) {
            // 第一区域
            if (isVowel[s[i]]) vowel++;
            if (i < k - 1) continue;
            // 第二区域
            ans = max(ans, vowel);
            if (isVowel[s[i - k + 1]]) vowel--;
        }
        return ans;
    }
};
```

双指针

主循环内的语句并没有什么规律，但是要有以下保证：

1. 移动快慢指针，同时更新窗口收集的信息
2. 更新答案（需在窗口长度为既定长度的时候更新）

若您的窗口在“快指针”移动以后成为既定长度区间，那么答案在“慢指针”移动之前更新；

若您的区间要“快慢指针”都移动了以后才是既定长度区间，那么答案在主循环末尾更新。
若您的区间在“快慢指针”移动之前就是既定长度区间，那么答案在主循环开头更新。

```
// 判断子串的全排序是否出现在主串（代码片段）

    int i = 0;
    for (; i < s1.size(); i++) {
        hashmap[s2[i] - 'a'] += 1;
    }
    if (checkHashMap()) {
        return true;
    }
    for (int j = 0; i < s2.size(); j++, i++) {
        hashmap[s2[i] - 'a'] += 1;
        hashmap[s2[j] - 'a'] -= 1;
        if (checkHashMap()) {
            return true;
        }
    }
    return false;
```

滑动队列

这个是有模板的，先看代码：

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

// 使用队列维护滑动窗口的最大值
vector<int> maxSlidingWindow(vector<int>& nums, int k) {
    deque<int> q;
    vector<int> result;
    for (int i = 0; i < nums.size(); ++i) {
        // 维护队列的单调性，移除队列中所有小于当前元素的元素
        while (!q.empty() && nums[q.back()] < nums[i]) {
            q.pop_back();
        }
        q.push_back(i);

        // 当窗口大小超过 k 时，移除窗口外的元素
        if (i >= k && q.front() + k == i) {
```

```
            q.pop_front();
        }
        // 当窗口大小达到 k 时，开始添加结果
        if (i >= k - 1) {
            result.push_back(nums[q.front()]);
        }
    }
    return result;
}

int main() {
    vector<int> nums = { 1, 3, -1, -3, 5, 3, 6, 7 };
    int k = 3;
    vector<int> max_values = maxSlidingWindow(nums, k);
    for (int val : max_values) {
        cout << val << " ";
    }
    return 0;
}
```

队列通常采用双端队列，存索引。

若是存值：

1) 不能维护边界

2) 若值的类型很大，那么将会浪费空间

若是单端：

1) 不支持维护队列的单调性

总上所述：

双端存索引的“队列滑动窗口”更为常见

尺取法

用来解决重复数区间的问题。比如说，求数组中满足 A+B=target 的组合的个数，A 和 B 都可能重复出现，这时简单的双指针就不够用了。要用到三指针，前两个指针都同向移动，这是同向尺取法，而三指针整体看上去是异向尺取法。

STL 提供的 equal_range 函数可以很好地解决“两指针同向尺取”

它很好地利用了二分，因此计算速度特别快。

以下算法设置了一个额外的二元组（pair）变量 p 来储存两个迭代器。

由于需求不同，传入 equal_range 函数时选择了 p 上的不同迭代器。例如：

第一个函数填 p.first，每次以上一个区间的开头为起点查询

第二个函数填 p.second，每次从下一个区间开始查询

前置

```
#include<algorithm>
#include<utility>
#include<vector>
```

主体

```
// 数对统计（合等于 target 的数对的数量）
int pairCount(vector<int>& nums, int target) {
    sort(nums.begin(), nums.end());
    int res = 0;
    auto p = make_pair(nums.begin(), nums.begin());
    for (auto i = nums.end() - 1; i - p.first > 0; i--) {
        p = equal_range(p.first, i, target - *i);
        res += p.second - p.first;
    }
    return res;
}
// 数对统计（值相等的数对的数量）
// 力扣 1512
int numIdenticalPairs(vector<int>& nums) {
    sort(nums.begin(), nums.end());
    int res = 0;
    auto p = make_pair(nums.begin(), nums.begin());
    while (p.second != nums.end()) {
        p = equal_range(p.second, nums.end(), *(p.second));
        int len = p.second - p.first;
        res += len * (len - 1) / 2;
    }
    return res;
}
```

补充

时间复杂度：O(nlogn)

致命问题通解

本章节要干什么？解决“计算机原理”或“C 风格特性”产生的问题，例如：

1. 《C++不充分的标准函数》之“我辣么大的一个大数处理问题 C++官方居然不管？”（C 风格特性）
2. 细致入微的计算机原理，居然要我面向过程中的过程？冷得面向对象玩家直哆嗦（计算机原理）

高精度运算（C++）

高精度运算”是“模拟”，采用数组的形式，模拟一个很大的数字

以下模板已经尽可能精简了,采用的都是 string 储存。

为什么建议使用 string？

与数组相比，string 的优良性主要在于它的智能，遍历的时候首尾明确，不需要使用一个指针盯着两端；可以直接翻转、拼接、剪切、复制、粘贴，而且 string 足够直观，每个字符已经足够储存数据。它把大量指针操作交给了 C++自带的函数，这样我们编写代码时犯傻的概率将大幅度降低！

总之，无论如何肯定是薄纱数组的。接下来我们的代码将基于 string。

五种基础算法

	对象	长度对齐	剩余运算	遍历顺序	书写方式	负数运算	借位进位
加法	str_str	需要	进位→补位	反向	倒置 str	不支持	carry
减法	str_str	需要	零位→删位	反向	倒置 str	不支持	if
乘法	str_int	不需要	进位→补位	反向	倒置 str	不支持	carry
除法	str_int	不需要	零位→删位	正向	正常 str	不支持	carry
求模	str_int	不需要	无	正向	统计 int	不支持	res

“对象”：指的是函数的参数列表上的两个参数的类型

“长度对齐”：通过扑上前导 0，可以使 string 的长度一致

“剩余运算”：指当前位在计算过程中产生的对其它位的影响

“遍历顺序”：对第一个参数的遍历顺序，正向是 0 位开始

“书写方式”：答案在运算中逐位产生，是“遍历顺序”所导致的不同

“负数运算”：基础算法都不能直接传入带负号的 string，不支持

“借位进位”：加法、乘法、除法的进位和借位都要依赖 for 循环外的 carry 变量来完成；减法依赖 for 循环内的 if 语句来完成；求模依赖“统计的 int”自身，即可完成。

基础算法原理须知：

显而易见的是，只有 str_str 需要在开始的时候长度对齐，因为它们是同位之间的运算。

乘法和加法的结果只能变大（或不变），于是就有进位问题，在运算完以后若有剩余值，需要补位。加法的剩余值很小，最大为 1，所以用 if 即可；而乘法的剩余值可以很大，最大为 89999999，为“最大 8 位数”乘“1 位数”的值除以 10（在“双大数乘法”的绿色字方框内有证明），所以要使用 while 来补位，只要剩余值不为 0，就一直迭代下去。

减法和除法的结果只能变小（或不变），于是可能会出现前导零。而储存结果的 res（string 类型）出现前导零的方式也是不一样的。由于运算方向的不同，减法的前导零最先出现在右侧（string 末位），而除法的前导零总是从左侧（string 首位）出现。

这里我提供两种函数，可以分别从左右两侧开始查询第一个没有 0 的位置：

```
size_t pos = res.find_first_no_of('0');
size_t pos = res.find_last_no_of('0');
```

加法

```
string add(string a, string b)
{
    string res;
    // 对齐
    int len = max(a.size(), b.size());
    a = string(len - a.size(), '0') + a;
    b = string(len - b.size(), '0') + b;
    // 主循环
    int carry = 0;
    for (int i = len - 1; i >= 0; i--) {
        int numa = a[i] - '0';
        int numb = b[i] - '0';
        int temp = numa + numb + carry;
        res += temp % 10 + '0';
        carry = temp / 10;
    }
    // 补位
    if (carry) res += "1";
    // 翻转
    reverse(res.begin(), res.end());
    return res;
}
```

假如两个 string 有一个负数，请转换为减法运算；假如两个都是负数，先将负数转换为正数，运算结束以后再把负号加回去。

减法

```
string sub(string a, string b)
{
    string res;
    // 对齐
    int len = max(a.size(), b.size());
    a = string(len - a.size(), '0') + a;
    b = string(len - b.size(), '0') + b;
    // 排序
    bool flag = 0;
    if (a < b) {
        swap(a, b);
        flag = 1;
```

```
    }
    // 主循环
    for (int i = len - 1; i >= 0; i--)
    {
        // 借位
        if (a[i] < b[i]) {
            a[i - 1] -= 1;
            a[i] += 10;
        }
        res += a[i] - b[i] + '0';
    }
    // 定零
    size_t pos = res.find_last_not_of('0');
    // 特判
    if (pos == string::npos) {
        res = "0";
    }
    else {
        res = res.substr(0, pos + 1);
        if (flag) res += "-";
        reverse(res.begin(), res.end());
    }
    return res;
}
```

单大数乘法（基础）

```
string mul(string a, int b)
{
    string res;
    int len = a.size();
    int carry = 0;
    for (int i = len - 1; i >= 0; i--) {
        int temp = (a[i] - '0') * b + carry;
        res += temp % 10 + '0';
        carry = temp / 10;
    }
    // 补位
    while (carry) {
        res += carry % 10 + '0';
        carry /= 10;
    }
    // 翻转
```

```
    reverse(res.begin(), res.end());
    return res;
}
```

单大数除法（基础）

```
string div(string a, int b)
{
    string res;
    int len = a.size();
    int carry = 0;
    for (int i = 0; i < len; i++) {
        int temp = carry * 10 + a[i] - '0';
        res += temp / b + '0';
        carry = temp % b;
    }
    // 定零
    size_t pos = res.find_first_not_of('0');
    // 特判
    if (pos == string::npos) {
        res = "0";
    }
    else {
        res = res.substr(pos);
    }
    return res;
}
```

求模

```
int mod(string s, int m)
{
    int len = s.length();
    int res = 0;
    for (int i = 0; i < len; i++) {
        res = (res * 10 + s[i] - '0') % m;
    }
    return res;
}
```

求余符号是 %，求模是 mod，遇到负数的运算结果可能不同。

C++的 %是求余运算，这是一种本源上的优势。何为优势？我们知道，C++的 %的运算结果的正负性只取决于第一位（被除数）的正负性，所以 mod 的正负性取决于第二位（除数）。

好，完美区分开了这两个概念。

（注：这条规则可能不适合其它语言，例如：python 的 %是求模）

双大数乘法

双大数乘法

```
// 后序遍历
string DFS(string a, string b) {
    if (b.size() <= 8) {
        return mul(a, stoi(b));
    }
    int cut = b.size() - 8;
    string left = DFS(a + string(8, '0'), b.substr(0, cut));
    string right = DFS(a, b.substr(cut, 8));
    return add(left, right);
}
```

显而易见，双大数乘法的原理是拆分，然后集合。天哪，这简直完美符合递归的特性，此时不用递归，还要等到何时用？

以下是算法解释：

```
    // 已知 string * int
    // 如何计算 string * string ?
    //
    // 可拆分，例如：
    // 17,800 * 18,545,454 = (178,000 * 18,545,45) + (17,800 * 4)
    //                       (1,780,000 * 18,545,4) + (178,000 * 5) + (17,800 * 4)
    //                       ......
    // 但是有没有一种可能，计算机的单次乘法运算恒为一个固定值
    // 那么我们这样计算：
    // 17,800 * 18,545,454 = (17,800,000 * 18,545) + (17,800 * 454)
    //                     = (17,800,000,000 * 18) + (17,800,000 * 545) + (17,800 * 454)
    //
    // 岂不是更快？
    //
    // 首先我们只考虑一种思路：string 只能逐项计算
    // 由 string 上的单位数最大为 9，可知 string 每次提供的最大被乘数为 9
    // 设从右侧拆分出来的 int 变量为 num
    // 则进位最大为 9*num/10，当前位乘积最大为 9*num
```

```
    //
    // 考虑到溢出，需满足：9*num/10 + 9*num <= 2,147,483,647
    // 通过简单的计算可得 num <= 21,474,836,470/99 = 216,917,540
    // 所以每次应当提取 8 位数
```

本质上是拆分，过程是二叉树，很简单吧。可是大数除法就难了。

双大数除法

双大数除法

```
bool cmp(const string& a, const string& b) {
    return a.size() < b.size() || a.size() == b.size() && a < b;
}

string doubleDiv(string a, string b) {
    if (cmp(a, b)) {
        return "0";
    }
    size_t pos = a.find_first_not_of('0');
    if (pos == string::npos) {
        return "0";
    }
    string res;
    for (int right = pos + b.size() - 1; right < a.size(); right++) {
        int num = 0;
        string temp;
        // get pos
        while (pos < a.size() && a[pos] == '0') {
            pos++;
        }
        // copy from pos to right
        temp = a.substr(pos, right - pos + 1);

        // if temp bigger than b, sub
        while (!cmp(temp, b)) {
            temp = sub(temp, b);
            num++;
        }
        // write a
        temp = string((right - pos + 1) - temp.size(), '0') + temp;
        for (int i = pos, j = 0; i <= right; i++, j++) {
            a[i] = temp[j];
```

```
            }
            res += num + '0';
        }

        pos = res.find_first_not_of('0');
        if (pos == string::npos) res = "0";
        else res = res.substr(pos);
        return res;
}
```

前置是：大数除法

思路是：化为减法，从高项往下减

负数支持

选择器

```
string selector(string a, string b) {
    // - - → +
    // - + → -
    // + - → -
    // + + → +

    //记录负号信息
    bool flag1 = a[0] == '-';
    bool flag2 = b[0] == '-';
    //去负号运算
    if (flag1) a = a.substr(1);
    if (flag2) b = b.substr(1);
    string res = DFS(a, b);
    //正负判断
    if (flag1 ^ flag2) res = '-' + res;
    return res;
}
```

低精度求模（C/C++）

双单词英文名缩写

```
using LL = long long;
using LD = long double;
using ULL = unsigned long long;
```

“低精度求模”是“对 long long 变量的处理”
这个看起来简单，但是运算可能会涉及一些非常难的公式（如果出题人想这么考）。

以下公式都为求逆元运算，逆元主要运用于**取代除法运算**。例如：
题目给出 mod，求 a/b，那么

1. 求 a % b → rem
2. 求 1 / b → inv
3. 则 a / b == (a - rem) * inv % mod

第一步： 求模（这大家都会）。
第二步： 求 b 的逆元。这个逆元要通过“扩展欧几里得”或“费马小定理”来获取。
特别要注意的是求得的 inv 只能用在整除运算中，这也是第一步求 rem 的原因。
第三步： 将 a 减去 rem 得到“可整除”的被除数，然后将这个数和逆元相乘，再求模。

逆元问题 - 扩展欧几里得

扩展欧几里得の推导

裴蜀定理指出，若干倍个大于 0 的 a 和 b 相加可得 a 和 b 的最大公约数，即：

$$ax + by = gcd(a, b)$$

而“扩展欧几里得”的作用就是**推理出该公式的一组（x，y）特解**
该公式可以推广到 gcd 运算的所有中间层。由 gcd 的原理就可推理出下一层，
x'*b + y'*(a-a/b*b) = gcd(b, a%b)，整理可得：

$$y'*a + (x' - y'*(a/b))*b = gcd(b, a\%b)$$

显然对于原 a 和 b 来说：

$$x = y',\ y = x' - y'*(a/b)$$

该公式指出，若要想得到上层的 x 和 y，只要得到下一层的 x'和 y'即可
而我们又知道当进行到 a+0 = gcd(a, 0)时，算法结束，此时有 x=1，y=0，b=0

于是我们知道**当前层的 x 和 y 可以由下一层得到**，而**末层的 x 和 y 又是固定的**
那么从顶层递归下去，逐层返回 x'和 y'，不就可以得到最终特解（x，y）了么？

代码原型

```
// 扩展欧几里得（方便理解版）
int d, x, y;
void exgcd(int a, int b) {
    if (b == 0) {
        d = a;   // 获取 最大公约数
        x = 1;
        y = 0;
```

```
    }
    else {
        exgcd(b, a % b);
        int px = x; // 抓住下一层传回的 X
        int py = y; // 抓住下一层传回的 Y
        x = py;
        y = px - (a / b) * py;
    }
}
```

补充

时间复杂度（不考虑余除的时间复杂度）：O(log min{a, b})
时间复杂度（考虑余除的时间复杂度）：O(log^3 min{a, b})

接下来我们探讨如何改进这个“原型函数”：

若将最大公约数 d 当作返回值传出，将 x 和 y 写为引用参数，即可去除全局变量

那么原为 x = y', y = x' - a/b*y' 的状态转移方程，现为：x = y，y = x - a/b*y
注意，该公式需要同步进行，先算出了 x 或先算出了 y 都会导致另一个变量运算出错
这也就意味着**需要一个临时变量**来储存 x 或 y 的中间体

将 exgcd 函数的下层 exgcd 的传入参数中的 x 和 y 调换一下，即可节省临时变量

原为 exgcd(b, a%b, x, y)，现为 exgcd(b, a%b, y, x)，可得：

x = y'= x

y = x' - a/b*y' = y - a/b*x

可以看到，x 即是本身，所以不用写；而 y 的运算不会影响 x，所以还省去了临时变量

函数

```
// 扩展欧几里得（教科书版）
LL exgcd(LL a, LL b, LL& x, LL& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    LL temp = exgcd(b, a % b, y, x);
    y -= a / b * x;
    return temp;
}
```

补充

时间复杂度（不考虑余除的时间复杂度）：O(log min{a, b})

时间复杂度（考虑余除的时间复杂度）：O(log^3 min{a, b})

算法的逆元应用

若 x 为 a 的逆元，则有 ax % p = 1 变换可得：ax + py = 1，y∈Z
显然，求出其中一组（x，y）特解，其中的 x 就是我们想要的逆元

主体

```
// 逆元
LL inv(LL a, LL p) {
    LL x = 1, y = 0;
    exgcd(a, p, x, y);
    return (x % p + p) % p;
}
```

补充

时间复杂度：O(logn)
使用条件：只要“整型 x”和“模数 p”互质，即可求得 x 的逆元

逆元问题 - 快速幂&费马小定理

主体

```
// 快速幂
LL qpow(LL a, LL n, LL mod) {
    LL res = 1;
    a %= mod;
    while (n) {
        if (n & 1) {
            res = res * a % mod;
        }
        a = a * a % mod;
        n >>= 1;
    }
    return res;
}
// 逆元
LL inv(LL x, LL p) {
    return qpow(x, p - 2, p);
}
```

补充

快速幂：

时间复杂度：O(logn)

费马小定理：

时间复杂度：O(logn)

使用条件：**对于 int x，只要 x 与质数 mod 互质，那么可求得 1/x（逆元）**

溢出运算求模 - 大乘法

对于 a * b mod p ，若 a*b 连 long long 类型会溢出，而 a%p 和 b%p 也无济于事的时候

那么就要用上“大乘法”了，这个原理是将乘法拆分成多个快速加法，实际是“减半加倍”，

对每部分都进行求模以保证最终的运算不会溢出：

右边 b 不断减半；左边 a 不断加倍，直到右边 b 变成 0 为止

当右边 b 为奇数时，将左边 a 统计入答案。答案初始为 0

写起来非常简单，会快速幂就会写。

函数

```
// 稳定的大乘法
LL qmul(LL a, LL b, LL p) {
    LL res = 0;
    a %= p;
    b %= p;
    while (b) {
        if (b & 1) {
            res = (res + a) % p;
        }
        a = (a + a) % p;
        b >>= 1;
    }
    return res;
}
```

这个算法实际上并不是很快，时间复杂度是 O(logn)，优点就是极其稳定不依赖平台。

有更快的算法，这个接下来讲。

溢出运算求模 - 快速乘

特别注意，这个算法极度依赖平台

对于 16 字节的 long double，可处理 long long

对于 12 字节的 long double，可处理 18 位十进制

对于 10 字节的 long double，可处理 18 位十进制

对于 8 字节的 long double，可处理 15 位十进制

以下讲解基于 10 字节的 long double

基础公式

“a * b mod p”的数学公式可拆解为：**a*b - floor(a*b/p)* p**

以“a*b”为左项，以“floor(a*b/p)”为右项

实际该数学公式可用于负数运算，但编程中不是如此。**处理负数极其复杂，所以本算法只考虑正整数**

右项 floor 向下取整筛去了除法运算产生的小数的小数部分，因此，左项应当必然大于等于右项

但是，事实真是如此吗？不是的，这是程序而不是数学公式，**程序内的左项不一定大于等于右项**！

有两个可能导致左项小于右项的情况，也是我们接下来会讲的，就先放出来吧：

1. 左项比右项多溢出一次
2. 右项的浮点数转为无符号整形时，因为精度不足而在最低位加 1

<—— 如何保证公式在程序中正常进行 ——>

再次强调：该程序只能用于正数运算

<-- 第一部分 -->

首先 **a 和 b 要 mod p**，目的是**防止溢出、限制 a 和 b 的位数小于等于 p**

<-- 第二部分 -->

令右项的“a*b/p”**以 long double 类型运算**。

然后把运算结果**转换回 long long，记作 temp**。

而 long double 只保证 18 位的有效数字正常显示

那么是否需要注意运算结果超过 18 位的情况呢？

答案是不需要。以下是证明：

a 乘 b（long double 类型）的最大位数是 等于 max(digit_a) + max(digit_b)

由于 p 必然大于 a 或 b，所以除后结果是 小于等于 max(digit_a, digit_b)

因此只要 p 在 18 位及以内，就可以完美运行

<-- 第三部分 -->

“左项与右项的差值”就是“求模”的结果，显然我们**只需关注差值**

利用 unsigned 溢出回滚的特性，**左右项同时同次回滚就可以保住差值**

（必须用无符号类型，有符号类型的溢出是未定义行为！）

若回滚次数有差，那么就必定是左项多溢出了一次

而左项会因为这一次回滚，计算而成为负数，然后又回滚一次，这就是二次回滚

例如：

a 为 6，b 为 45，mod 为 110，有符号类型最大 127，无符号最大 255

需要的结果：270 - 220 = 50

现在的结果：(270 -> 14)(一次回滚) - 220 = (-206 + 256)(二次回滚) = 50

<-- 第四部分 -->

当 long double 的精度高于 a，b，p 好几位时，return res；并不会导致错误

如 16 字节 long double 处理 15 位十进制时，就不会有问题

右项“a*b/p”以 long double 类型运算的时候，得到了一个近似结果（浮点型不精确）
由于程序不太确定这个近似结果是否应该进位，**所以它进位了！**这听起来是不是很荒谬？
（另外，long double 转换为 long long 的时候，是截取，不会有问题的。这大可放心）

如何避免发生这样的事情？很遗憾，在这里不行。但是我可以告诉你，只要你离得够远就行了。这里的远是什么？是“我们所需求的数的位数”与“浮点型的最大精度保证的边界”的距离。比如说，double 最大保证有效数字 15 位，那么边界就是这有效数字的右测。

假如“我们所求的数的位数”离之近，就会产生这种莫名奇妙进位问题。一旦靠的近不是说这个浮点型用不了了，而是我们需要开始关注它“是否进了一次不必要的位”。

本算法中我们要算的是 18 位十进制，即使是最大的 16 字节 long double，都没有足够远的距离
所以必须要做处理。**既然是进了一位，也就代表右项多加了一个 p，因此可推测出左项多减了 p**
最终结果 res 要加上 p 再 mod p，这才是我们想要的答案

函数

```
// 真.快速乘
LL qmul(LL a, LL b, LL p) {
    a %= p, b %= p;
    LL temp = (LD)a / p * b;
    LL res = (ULL)a * b - (ULL)temp * p;
    return (res + p) % p;
}
```

测试代码

```
#include<iostream>
#include<algorithm>
#include<random>
#include<iomanip>
using namespace std;
using LL = long long;
using LD = long double;
using ULL = unsigned long long;
// 测试次数
const int TIME = 10000;

LL smul(LL a, LL b, LL p) {
    LL res = 0;
    a %= p;
    b %= p;
```

```
    while (b) {
        if (b & 1) {
            res = (res + a) % p;
        }
        a = (a + a) % p;
        b >>= 1;
    }
    return res;
}
LL qmul(LL a, LL b, LL p) {
    a %= p;
    b %= p;
    LL temp = (LD)a * b / p;
    ULL temp1 = (ULL)a * b;
    ULL temp2 = (ULL)temp * p;
    LL res = temp1 - temp2;
    return (res + p)%p;
    // return res;
    // 如果 res 不加 p，模 p，就无法通过下面这个样例（不止）
    // 882032046897989
    // 612895078083451
    // 939535215211549
}
// 采用对数器测试（将算法与另一稳定算法相比较输出）
int main() {
    random_device rd;
    mt19937 gen(rd());
    // 可填至 999'9999'9999'9999LL，也就是 15 个 9，15 位 10 进制数
    uniform_int_distribution<LL> uid(0, 999'9999'9999'9999LL);

    int errorCount = 0;

    for (int i = 0; i < TIME; i++) {
        // 生成“两个乘数”
        LL a = uid(gen);
        LL b = uid(gen);
        LL bigger = max(a, b);
        // 生成“模数值”
        LL p = uid(gen);
        while (p <= bigger) {
            p = uid(gen);
        }
        // 检查 p 是否大于 a 和 b
        if (p < max(a, b)) {
```

```
            cout << "Bug!" << endl;
        }
        LL res_smul = smul(a, b, p);
        LL res_qmul = qmul(a, b, p);
        if (res_qmul != res_smul) {
            cout << setw(19)<< a<< " "<< setw(19)<< b<< " "<< setw(19)<< p<< " : ";
            cout << "ERROR: " << res_smul << " " << res_qmul << endl;
            errorCount++;
        }
    }
    cout << setw(30) << "long double 字节数：";
    cout << sizeof(long double) << endl;
    cout << setw(30) << "double 字节数：";
    cout << sizeof(double) << endl;
    cout << setw(30) << "float 字节数：";
    cout << sizeof(float) << endl;
    cout << setw(30) << "ERROR 报错比：";
    cout << errorCount << "/" << TIME << endl;
    return 0;
}
```

计算机原理

减半加倍-除法（倍增优化）

原理：被除数每减去除数，商加一。显然这样太慢了，所以可以用倍增法
每次倍增寻找“可减除数与可加商”，时间复杂度优化到了 O(log(a/b))

函数

```
// 创建一个倍增边界
const int MAXNUM = INT_MAX >> 1;
// 减半加倍-除法
int HalfDoubleDiv(int a, int b) {
    int res = 0;
    while (a >= b) {
        int temp = b;
        int count = 1;
        // 倍增优化
        while (temp <= MAXNUM && temp + temp <= a) {
            temp += temp;
            count += count;
        }
```

```
        a -= temp;
        res += count;
    }
    return res;
}
```

究极数论

基础知识-公约公倍

公约倍求解

最大共约数：int gcd(int a, int b){ return b > 0 ? gcd(b, a % b) : a; }
最小公倍数：int lcm = a * b / gcd(a, b);

公因简单定理

最大公约数不变：gcd(a + mb, b) = gcd(a,b)
内外整除无影响：gcd(a, b) = f ⇔ gcd(a/f, b/f) = 1

因数出现论

某因数先前出现的次数：int count = num / factor
出现某次数的最大因数：int factor = num / count
同出现次数的最大因数：int factor_max = num / (num / factor)

基础知识-质数

欧拉函数

欧拉函数是数的“小于等于这个数的互质数”的数量，因此欧拉函数是值，而不是集合。
对于质数 E(p) = p - 1，如：E(7) = {1,2,3,4,5,6} = 6

当一个数 n 等于质 p 的 k 次幂时，任何小于等于 n 且是 p 的倍数的数都与 n 有大于 1 的最大公约数。剩下的是和 n 互质的数。如：7^3 只与 7 的倍数不互质

把上述数列列出来：p, 2p, 3p, ... ,p^(k-1)* p。肉眼可得出以下三条等式：
(1) E(p^k) = p^k - p^(k-1)
(2) E(p^k) = p^(k-1) * (p-1)
(3) E(p^k) = p^k * (1-1/p)

两个质数乘积：n = p^a * q^b

“小于等于 n”且“是 p 的倍数”的质数有：p^(a-1) * q^b，记为 A
“小于等于 n”且“是 q 的倍数”的质数有：p^a * q^(b-1)，记为 B
“小于等于 n”且“是 p * q 的倍数”的质数有：p^(a-1) * q^(b-1)，记为 C
可以推出：n = A + B - C + E(n)（对于正整数，加上非互质的数和互质的数等于它自身）

把这些公式拆开，简单组合可以得到，质数之间的关系：E(pq) = E(p) * E(q)
根据欧拉函数的公式，只要一个数能够分解质因子，就能够求出它的欧拉函数

素数试除法

须知 num 是由数个不同 p^k 相乘组成

以下是伪代码：
对 1 特判，直接返回 1
    遍历所有小于等于“根号 num”的质数 prm
        若 num 可被 prm 整除，则 num 除以 prm，res 乘(prm - 1)
            若该 prm 接下来对 num 仍可除尽，则 num 除以 prm，res 乘 prm
            循环末判断 num 是否为 1，若是则直接返回 res
遍历完后，若 num 还是大于 1，这就表明 num 是一个质数，返回 res * (num - 1)即可

核心代码（参考“英雄哪里出来”）：

```
n /= p;
ans *= (p - 1);
while( 0 == n % p )
    n /= p, ans *= p;
```

为什么第一次 ans *= (p - 1) 以后都是 ans *= p ？
道理就出在上文的（2）式，因为 p-1 仅出现一次

试除法时间复杂度小于 O(n^(1/2))，考虑到欧拉筛只要产生根号 num 以内的数，也是 O(n^(1/2))
所以粗略估计对单时，这个算法只需要 O(n^(1/2))的时间复杂度

试除法预运算

而根据（3）式，可得：E(num) = num * II(k,i=1)((pi-1)/pi)
可见，只要 num 乘上所有的质因子分量((pi-1)/pi)就能得到欧拉值

算法实现

```
const int MAXN = 1024;
bool notprime[MAXN];
int eula[MAXN];

void eulaPre() {
    int i, j;
    notprime[1] = 1;
```

```
    // 先给所有数赋值为自身
    //（对应公式第一部分：num）
    for (i = 1; i < MAXN; ++i)
        eula[i] = i;
    for (i = 2; i < MAXN; ++i) {
        if (!notprime[i]) {
            // 对素数直接按公式赋值
            eula[i] = i - 1;
            // 标记 i 的倍数为非素数
            // 并更新这些倍数的欧拉函数值，累积乘以(pi-1)/pi)
            //（对应公式第二部分：II(k,i=1)((pi-1)/pi)）
            for (j = i + i; j < MAXN; j += i) {
                notprime[j] = 1;
                eula[j] /= i;
                eula[j] *= (i - 1);
            }
        }
    }
}
```

这是一个时间复杂度为 O(nloglogn)的算法，适用于处理大量查询的情况。

有意思的扩展

1. gcd(i, n) = k，则 i 的数量为“n/k 的欧拉函数”

质数检验

欧拉筛

一次产出一片质数。但是只能从 2 开始，所以很尴尬
一片的时间复杂度为 O(n)，每个的时间复杂度为 O(1)

高效运行的要点：质数中只有 2 是偶质数。先对 2 进行特判，**把 2 的倍数都筛掉**
接下来进入主循环，就可以**针对奇数进行筛选**，同时外层循环可以两步**两步地跳**

函数

```
// 欧拉筛，返回一个包含所有小于等于 n 的素数的数组
vector<int> eulerSieve(int n) {
    vector<bool> isPrime(n + 1, true);
    vector<int> res;
    // 0 和 1 不是素数
```

```
    isPrime[0] = isPrime[1] = false;
    isPrime[2] = true;
    // 先将 2 的所有倍数除去
    for (int j = 2 * 2; j <= n; j += 2) {
        isPrime[j] = false;
    }
    for (int i = 3; i * i <= n; i += 2) {
        if (isPrime[i]) {
            // 将 i 的所有倍数标记为非素数
            for (int j = i * i; j <= n; j += i) {
                isPrime[j] = false;
            }
        }
    }
    // 收集所有素数
    for (int i = 2; i <= n; ++i) {
        if (isPrime[i]) {
            res.push_back(i);
        }
    }
    return res;
}
```

六倍速朴素判断法（简称：六素法）

速度是普通的素数判断法的六倍
不过时间复杂度仍然是 O(n^(1/2))，建议在 32 位 int 以内使用

除了 2 和 3，其它所有质数都出现在(6k+1)或(6k+5)。先对 1，2，3 进行特判
然后通过 2 和 3 的余除，筛掉所有在(6k+2)、(6k+3)、(6k+4)、6k 的 num
此时剩(6k+1)、(6k+5)上的 num，令循环从 5 和 7 开始，六倍步进
若 num 能被小于自身的任意形如 6k±1 的数整除，则 num 非质数

函数

```
// 六倍素判断法
bool isPrime(int num) {
    // 循环，2 和 3 需要特判
    if (num == 1) return false;
    if (num == 2 || num == 3) return true;
    if (num % 2 == 0 || num % 3 == 0) return false;
    for (int i = 5; i * i <= num; i += 6) {
        if (num % i == 0 || num % (i + 2) == 0) {
            return false;
```

```
        }
    }
    return true;
}
```

使用技巧之打表

朴素算法适合用来打表，但是有限制，假如某个测试平台限制上传的代码占 100KB，那么估计你只能打 1E4 的表格。假设每个素数的字符由 UTF-16 来储存（问就是不会算 UTF8［滑稽］），每个素数之间要以逗号分隔，那么要 10671B 的空间（包含括号），即 10KB 多。这个可以保证四位数及以内的判断时间复杂度为 O(1)

米勒拉宾素性检验

幻神 Miller Rabin ！一旦要求的素数大于 32 位 int，使用之。
时间复杂度最遭为 O(k(logn)^3)，最好为 O(logn)

费马素性检验

由费马小定理可知，任意素数 p 与一个互质的数 a 满足：**a^(p - 1) ≡ 1(mod p)**

但是这条定理**不能逆用**，因为存在一些合数可以满足这个逆定理，
这些数是“费马伪素数”（如：341）

而多选取一些底数 a，虽然可以降低错误的概率，但**仍不能百分百正确**，
这些仍不能被筛去的数就是“卡迈克尔(Carmichael)数”（据说 1 亿内有 255 个，最小的是 561）

若 n 为卡迈克尔数，则 2^n - 1 也是卡迈克尔数

二次探测定理

对素数 p，若 x^2 ≡ 1(mod p)，则小于 p 的正整数解只有两个，1 和 p - 1
有证明：
变式：(x + 1)(x - 1) ≡ 0(mod p)
因此(x + 1)(x - 1)只能是 0 或 p 的倍数

定理合并（得出：米勒拉宾的证明）

将“费马小定理”拆分为“二次探测定理”的形式可得：
(a^((p-1)/2))^2 ≡ 1(mod p)。显然可得：
若 a^((p-1)/2) mod p 不是 1 或 p - 1，则 p 不是素数

若给(a^((p-1)/2))^2 连续开根，可以一直分解到 a 的指数为奇数 u 为止
则有 (p - 1)/2^t = u ，数列 a^u, a^u*2, a^u*2^2, ... , a^u*2^(t-1)

引用知乎“朝夕”的话：

对这个数列进行检验，它们的解**要么全是 1，要么出现 p-1 后全是 1（之前不能有 1）**，
否则就不是素数。当然，**要注意 p-1 不能出现在最后一个数**，否则就连“费马小定理”都不满足了。

另外要注意，**底数不能是 p 的倍数**，否则也不满足“费马小定理”。
这倒不是说测试失败了，**而是这个底数不能用来测试，跳过即可**。

代码解决

一些底数的组合使用，可以使得米勒拉宾检验在一定范围内百分百正确。
这样一组底数称为 SPRP 基（Strong Probable Prime Test Base）

Jim Sinclair 发现了一组底数，可保证该算法在 2^64 以内不会出错：
[2, 325, 9375, 28178, 450775, 9780504, 1795265022]

另外还有一些其它简单组合：
[2, 7, 61]，保证在 int（32 位）以内不出错
[2, 5, 7, 11]，保证在 2E12 以内不出错

函数（要写三个：qmul、qpow、Miller_Rabin）

```
using LL = long long;
using LD = long double;
using ULL = unsigned long long;
LL qmul(LL a, LL b, LL mod); /* 填写：快速乘 */
LL qpow(LL a, LL n, LL mod); /* 填写：快速幂 */
// 米勒拉宾检验
bool Miller_Rabin(LL n) {
    // 对 1 和 2 特判，并筛去偶数
    if (n < 3) return n == 2;
    if (n % 2 == 0) return false;
    // 计算队列的长度
    LL u = n - 1, t = 0;
    while (u % 2 == 0) {
        t++;
        u /= 2;
    }
    // 检验的主循环
    LL ud[] = { 2,325,9375,28178,450775,9780504,1795265022 };
    for (LL a : ud) {
        // 第一部分
        LL v = qpow(a, u, n);
        if (v == 1 || v == n - 1 || v == 0) continue;
        // 第二部分
        for (int j = 1; j <= t; j++) {
            v = qmul(v, v, n);
```

```
            // 出现 n-1，后面必然为连续的 1，直接跳出
            if (v == n - 1 && j != t) {
                v = 1;
                break;
            }
            // 若未出现 n-1 就出现了 1，那么不通过测试
            if (v == 1) return false;
        }
        // 费马检验
        if (v != 1) return false;
    }
    return true;
}
```

因数分解

质因数分解

函数

```
// 时间复杂度小于 O(n^(1/2))
map<int, int> primeFactor(int num) {
    map<int, int> mp;
    // 处理唯一的偶质因数
    while (num % 2 == 0) {
        mp[2]++;
        num = num / 2;
    }
    // 处理奇质因数
    for (int i = 3; i * i <= num; i += 2) {
        while (num % i == 0) {
            mp[i]++;
            num = num / i;
        }
    }
    // 大于 √num 的质因数至多有一个
    if (num > 2) {
        mp[num]++;
    }
    return mp;
}
```

波拉德罗算法

Pollard-Rho 是一种概率算法（它奶奶嘀，为什么快的算法都是概率算法 [恼怒]）

时间复杂度是 O(n^(1/4))。虽然不是最快的，但是已经很快了。

改进的基础概率算法

```
int find_factor(int n)
{
    if (is_prime(n)) /* 填写判断素数的函数 */
        return n;
    int x, d;
    do
    {
        x = randint(2, n - 1); /* 填写获取随机数的函数 */
        d = gcd(n, x); /* 填写求最大因数的函数 */
    } while (d == 1);
    return d;
}
```

通过引入 gcd，即使 randint 随机抽到的不是因数，也可以通过 gcd 的计算来得到其中一个，因此大大提高了命中率。

在最差的情况下，n = p^2，此时[1, p^2 - 1]内仅有一个 p 是 n 的因数，然而只要 x 取到 p 的正倍数时，都可以得到 p，所以最差期望时间复杂度是 O(n^(1/2)*logn)。其中 logn 来自 gcd 的时间复杂度；n^(1/2)来自抽取的概率，p^2-1 个数中有 p-1 个符合条件，接近 n^(1/2)。

但是显然这连朴素的质因数分解都打不过。

生日悖论

函数（要写三个：isPrime，randint，Pollard_Rho）

```
using LL = long long;
bool isPrime(LL n); /* 填写一个质数判断函数 */
LL randint(LL l, LL r) {
    mt19937_64 eng(time(nullptr));
    uniform_int_distribution<LL> uid(l, r);
    return uid(eng);
}
LL Pollard_Rho(LL n) {
    if (n == 4) return 2;
    if (isPrime(n)) return n;
    while (true) {
        LL c = randint(1, n - 1);
```

```
            auto f = [=](LL x) { return (x * x + c) % n; };
            for (LL t = f(0), r = f(f(0)); t != r; t = f(t), r = f(f(r))) {
                LL d = gcd(abs(t - r), n);
                if (d > 1) return d;
            }
        }
    }
```

余数相关定理

中国剩余定理

问题：一个正整数分别被若干个质数求模，且已知每个的结果，求这个数的最小正数解

可以找到这样的数，与 pi 求模为 ri，且与 pj(j!=i)求模均为 0。公式为：

$$ci = ri * (\prod pn)/pi * inv(ri, (\prod pn)/pi)$$

该公式对 pi 来说就是对 ri 求模；对 pj(j!=i)来说就是对 pj 的倍数求模

然而$\sum ci$，所得到的只是其中的一组通解，而对于所有的通解都可以写成如下形式：

$$k(\prod pn) + Min\_positive\_solution，k \in N$$

我们所求是 Min_positive_solution，所以$\sum ci \mod \prod pn$ 就可以得到最小正数解了

方程组知识

裴蜀定理の扩展

若 a 和 b 不全为 0，则 a 和 b 的“最小正数差值”为 gcd(a, b)
证明：随着 gcd 的进行，左右差值逐渐缩小
若 a 和 b 不全为 0 且存在 x 和 y 满足 ax + by = 1，则 a 和 b 互质
证明：显而易见，瞪眼法
若 a 和 b 不全为 0 且存在 x 和 y 满足 ax + by = c，则 c 为 gcd(a, b)的整数倍
（反过来说，若 c 不是 gcd(a, b)的整数倍，那么 c 不是 ax + by 的一个答案）
证明：若 c 不是整数倍，那么通过简单的数学堆叠计算即可求出新的更小的 gcd
若 a 和 b 不全为 0 且 ax + by = gcd(a, b)有解，则(x, y)有无穷组解
证明：存在 an+bm = 0，那么就有 an+bm + ax+by = 1
裴蜀定理公式不止 a 和 b 两项，可以推广到多项
证明：摊牌了，本作者不会证

二元一次不定方程

既然 ax + by = gcd(a, b) 有无穷多组（x, y）解，那么来探讨一下通解的形式
加上偏移量，可得 a(x+Δx) + b(y+Δy) = gcd(a, b)，化简得：a/d * Δx = -b/d * Δy

有定理：设 gcd(m, a) = 1, m|ab，则 m|b

而 gcd(a/d, b/d) = 1，(a/d)|(b/d * Δy)，那么就有 (a/d)|Δy，同理可得 (b/d)|Δx
那么通解为：X_com = x + b/d*k，Y_com = y - a/d*k，k∈Z

其它知识（选学）

威尔逊定理

若 p 为素数，则 (p - 1)! ≡ -1(mod p)
该定理可逆：若(p - 1)! ≡ -1(mod p)，则 p 为素数
且可以得出：(p - 1)! + 1 一定是 p 的倍数
但鉴于这个定理用的是阶乘，所以实用性并不高

两同增数的最大公约数

（蓝桥杯 2133）
若 a>b>0，k>0 则 d=a-b; k=d-a%d;
// 设 b>a,由更相减损法得 gcd(a+k,b+k) = gcd(b + k,a - b)
// 当 k = b - b % (a - b)时为使得 b + k 为 a - b 的倍数的最小值
// 此时 a+k 和 b+k 的最大公约数即为 a - b,使得 gcd(a+k,b+k)尽可能的达到了最大，
// 同时 k 也为最小的可取值

数组处理

二分

简单而复杂，一看就会，一写就废
首先解决最基础的问题，再深入

基础知识

主循环条件

区间分为：闭区间、开区间、半区间
left 为下界、right 为上界、mid 为“向下取整的中点”

while 循环必须合法（如：必须可以容纳一个元素，否则循环无意义）
若 left 和 right 可以相等（闭区间），那么填 left <= right
若 left 和 right 不能相等（半区间），那么填 left < right
若 left 和 right 间隔一项（开区间），那么填 left + 1 < right

边界收缩方法

若是开边界，每次收缩到 mid 即可
若是闭边界，每次收缩到 mid 还要额外走一格
left = mid + 1，right = mid - 1

很简单的道理。既然是收缩，就代表 mid 点“没用”了
闭区间收缩到 mid，包含 mid，所以要额外走一格
开区间收缩到 mid，就没包含 mid
注：“没用”不等于“不是答案”

可以轻松证明 mid 为何“没用”。
对于只有一项答案的二分查询，若 array[mid]是答案，那么跳出循环而不再有收缩。若有收缩，则是未找到答案，显然此时 mid 已经确认不是答案，那么可以舍弃。
对于有多项答案并要求返回其中最小（或最大）下标的二分查询，若 array[mid]是符合条件的答案，那么其中一个边界会跨过 mid，停止在某位上，直到 while 结束。最终这个边界的旁边即是答案，也用不上 mid

基础类型

查找单项

特点：明显分为三段。若找到值，直接返回结果

对左闭右开区间：
```
if (nums[mid] > target) right = mid;
if (nums[mid] < target) left = mid + 1;
else return mid;
```

红蓝染色

特点：分为两部分。即使找到值，依然进行循环
作用：若存在多个答案，可返回之中最小或最大的那个下标

以下展示其中两种写法：
对最小下标的左闭右开区间（类似 lower_bound）：
```
if (nums[mid] >= target) right = mid;
if (nums[mid] < target) left = mid + 1;
```
对最大下标的左闭右开区间（类似 upper_bound）：
```
if (nums[mid] > target) right = mid;
if (nums[mid] <= target) left = mid + 1;
```

作者评价：没错，求“最大最小下标”的区别就是交换个等号

循环结束以后如何得到答案？很简单，参考**循环不变量**。
将数组分为三区，小于 target 为红区，大于 target 为蓝区，白区为未知
总结规律，可知：
左闭“[”：小于等于【left - 1】 全是红区
左开“(”：小于等于【left】 全是红区
右闭“]”：大于等于【right + 1】全是蓝区
右开“)”：大于等于【right】 全是蓝区

在判断语句 if 中
把等号放在“>”上，得“nums[mid] <= target”，就是将答案并入到蓝区
把等号放在“<”上，得“nums[mid] >= target”，就是将答案并入到红区
“对最小下标的左闭右开区间”，是返回 right 或 left
“对最大下标的左闭右开区间”，是返回 right 或 left

例题：力扣 34

思维纠正

1. 答案不一定在区间内

差分

简单科普

差分和前缀是什么

“前缀”是把前面及当前所有元素相加
“差分”是把当前元素减去前一元素（前面没有元素就不减）

三种类型的数组

“普通数组”前缀合得“前缀数组”
“差分数组”前缀合得“普通数组”

例如：

对于普通数组：[2, 4, 5, 1]
它的前缀数组是：[2, 2+4, 2+4+5, 2+4+5+1] → [2, 6, 11, 12]
它的差分数组是：[2, 4-2, 5-4, 1-5] → [2, 2, 1, -4]

C++的 numeric 提供了两个函数用来处理前缀差分
（具体在“散装文件知识”章节已介绍）
前缀：partial_sum(beg, end, dest)
差分：adjacent_difference(beg, end, dest)

但是很可惜，在复杂情况下，这两个函数就是“依托答辩”，用不上的说。

这俩函数执行的“时间复杂度”和“空间复杂度都”是 O(n)，
如果某道题很苛刻，空间不足以容纳一个前缀数组，时间复杂度要小于 O(n)
那么就必须用到“标记”和“收集”了

标记和收集是什么

“标记”是在“差分数组”中两个位置作修改，从而实现对“普通数组”的区间修改
“收集”是把“差分数组”中的所有位置的修改统计，排序好并依次写入数组

标记和收集的演示

以下示例基于一道题目，链接是：

https://www.matiji.net/exam/brushquestion/18/4498/F16DA07A4D99E21DFFEF46BD18FF68AD?from=1

前置

```
map<int, int> mp;
int tab[MAXN << 1][2];
int cnt = 0;
int ans = 0;
```

map 用来记录对应位置上的差分标记
tab 用来读出所有的 map 记录
cnt 储存读取出的 map 记录数

读出

```
for (auto& p : mp) {
    tab[cnt][0] = p.first;
    tab[cnt][1] = p.second;
    cnt++;
}
```

统计

```
for (int i = 1; i < cnt; i++) {
    tab[i][1] += tab[i - 1][1];
    if (tab[i - 1][1] % 4 == 1) {
```

```
            ans += tab[i][0] - tab[i - 1][0];
        }
    }
```

字符串 KMP

以下 next 算法模板，运用了“动态规划”的思想。由于算法过于复杂，这里只做总结。

首先准备好两个指针。一个指向 string 第一个串的末尾，另一个指向 string 最后推测出 next 值的位置。例如：

```
          |    |
      abbaabbabaabbaa
```

补充知识：next 数组在一些区域上是按公差为 1 单调递增的。一个更大的数字出现时，其前面一定比它小 1 的数。举个例子：[-1,0,0,0,1,1,2,3,4,2,1,1,2,3,4,5,0,0,0,0]。可以很容易发现，诸如[1,2,3,4]这样的递增数列。

既然获取 next 数组的时候，是从左到右的，那么用来遍历的指针的左侧的 next 数组就是已完成的状态。我们可以利用这个已完成的部分推测出下一个节点的 next 值。所以当字符对应得上时，挪动快指针往下一个控填入 next 值+1（p_next[++j] = ++k;）；当字符对应不上时，利用 p_next 数组，对慢指针进行回溯（k = p_next[k]），以此来找到下一个匹配点。

获取 next 数组

前置

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

string s;
int p_next[MAXN];
```

主体

```
// 获取 next 数组
void getNext(string& p) {
    fill(p_next, p_next+MAXN, 0);
    int j = 0;
    int k = -1;
    p_next[0] = -1;
    while (j < p.length()) {
        if (k == -1 || p[j] == p[k]) p_next[++j] = ++k;
        else k = p_next[k];
```

```
    }
}
```

模式匹配

前置

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

主体

```
// 模式匹配
bool KMP(string& s, string& p, int pos) {
    int i = pos;
    int j = 0;
    int slen = s.length();
    int plen = p.length();
    getNext(p);
    while (i < slen && j < plen) {
        if (j == -1 || s[i] == p[j]) {
            i++;
            j++;
        }
        else j = p_next[j];
    }
    if (j >= plen) return true;
    else return false;
}
```

基础数据结构

栈（Stack）

逆波兰式

```
int evalRPN(vector<string>& tokens) {
    stack<int> stk;
    int a = 0;
    int b = 0;

    auto isNumber = [](const string& str)->bool {
```

```
        return !(str == "+" || str == "-" || str == "*" || str == "/");
        };

    for (auto& token : tokens) {
        if (isNumber(token)) {
            stk.emplace(stoi(token));
        }
        else {
            b = stk.top(); stk.pop();
            a = stk.top(); stk.pop();
            switch (token[0]) {
            case '+': stk.emplace(a + b); break;
            case '-': stk.emplace(a - b); break;
            case '*': stk.emplace(a * b); break;
            case '/': stk.emplace(a / b); break;
            }
        }
    }
    return stk.top();
}
```

队（Queue）

目前想不到什么内容，先空着

惨痛中总结的规范

每条规范都有属性，例如说：思维纠正、建议 ...
思维纠正：对某种方法还可以这样做，不要被局限了
建议：你应该遵守的条款，往往是作者亲自踩过的坑

规范-循环体

一. 尽量不要写“自动机”（建议）

错误示范：

```
int state = ?;
for(int i=0; i<time; i++){
    if(state == 0){ ...state = newNum;}
    else if(state == 1){ ... state = newNum;}
    else if(state == 2){ ... state = newNum;}
    else { ... state = newNum;}
```

```
}
```

理由：
非必要情况下，写自动机极度容易出错。而且自动机状态在边界上往往不好处理。例如说，边界导致的漏统计。当 status 为 0 时不统计，status 为 1 时统计这条段路上的答案。结果自动机某个阶段开始 status 为 0，到了边界没有检测到 1 就退出了循环（程序员忘记给边界做特判），于是答案统计少了这一段。

二．涉及一维相邻元素时，指针应从第二位开始，令该位与前位进行运算（建议）

示例：

```
for(int i=1; i<n; i++){
    if( nums[i-1]与 nums[i]的比较){ ...}
}
```

理由：

1. 从第二位开始：规避开头的边界问题，不读取 array[-1]
2. 与前位进行运算：规避了末尾边界问题，不读取 array[MAX]
3. 与前位进行运算：全面顾及，求不同相邻元素数，不会漏
4. 整体上：参考 std::adjacent_difference，官方都是这么写的

规范-链表

一．链表可以用 for 循环（思维纠正）

写法：

```
for(Node* i = &head; i != nullptr; i = i->next){ ...}
```

二．写会改变链表结构的算法，最好加“哨兵/哑元”（建议）

写法：

```
Node dummy;
dummy.next = &root;
```

规范-矩阵

一． 不要用 C 风格的数组，否则传递二维数组够你吃一壶的（建议）

写法：

```
function(int (*p)[MAXD], pr, pd) { ...}
```

二． 如果嫌 vector 矩阵类型名称太长，可以尝试写 map 矩阵（思维纠正）

缺点:

1. 占用为 6 倍
2. 无法获取行数列数

转换：

```
vector<vector<int>> v(r, vector<int>(c, 0));
```

→ map<int, map<int, int>> mp;

三. 尝试阻止算法生成无法容纳的临时矩阵（建议）

左行右列，叉乘的结果取决于：左矩阵有几行，右矩阵有多少列
因此我们可以让“左矮扁右高瘦”的两个矩阵先运算，这样产生的矩阵的所占缓存就会尽可能小

例如：K[100][2] × W[2][100] × Q[100][2]
顺着来，第一步就会得到一个巨大的临时矩阵：cache[100][100]

附录 A（表格样式）

代码示例

正文（代码片段）

小标（6 号标题字体）
正文（函数片段、补充、前置、功能）

对于代码演示窗口，最好上下留空。
而代码通常是大括号独占最后一行，相当于空行，那么正文贴底即可。

其它内容

超链接

成员函数	正文 ...

总之，要求不是很严格，自由发挥即可

附录 B（算法常见词）

本书对元素的书写遵循“正则表达式”的语法
如果出现“正则表达式”，就是说这个词通常配合“其它符号”出现

例： +begin 表示 begin 这个词不可单独写（实际会和 C++的 begin 函数冲突）
加号代表至少加一个符号在前面（举例：sbegin 代表 str 的起始迭代器）

常量（算法常量、数据结构常量 ...）

图论常量	写法
最大权重	const int INF = 0x3f3f3f3f;
最大节点数	const int MAXV;
最大边数	const int MAXE;

INF 赋值为 0x3f3f3f3f 的好处是：

1. 易写，重复 4 遍 3f 即可（不像 0x7fffffff 要仔细数 f 的数量）
2. 足够大，大于 10e9 小于 10e10
3. 不会轻易溢出。两倍 INF 依然小于 MAX_INT
4. 但是作者强烈建议先判断再运算，如 Dijkstra 有这么一段：
5. if (... && g[u][v] != INF && dis[v] > dis[u] + g[u][v]) {...}

变量（通用名称、数学名称 ...）

通用名称	书写
输出 ...	
结果	result / res
答案	answer / ans
总和	summary / sum
For 循环中间体 ...	
临时变量	temporary / temp
临时元素	element / ele
行	row
列	column / col
缓冲区 ...你的缓冲区又溢出啦(*/ω＼*)	
高速缓冲存储器	cacho
缓冲区	buffer / buf
偏移量	offset

静态数组の全局成员	书写
元素数	cnt (count)
末尾元素的位	idx (index)

cnt 和 idx 的区别是，假如元素是从 0 开始记的，那么指向最后一个元素的位的数是就是元素总数即 end_element_index = count；假如元素是从 1 开始记的，跳过了 0，那么指向最后一个元素的位的数是就不是元素总数，即 end_element_index != count，那么我建议用 idx 来标记，好作区分。

数学名称	书写
数字	number / num

位数	digit
次数	time
系数	coefficient / coe
指数	exponent / exp
基数	bottom_number / bot, base
逆元	inverse / inv
模数	modulus / mod （一些习惯写法：p）

类型专用名（容器、流类 ...）

容器	专用变量名
map / unordered_map / multimap	mp
set / unordered_set / multiset	se
vector	v
list	ls
string	s / str
deque / queue / priority_queue	dq / q / pq
stack	sk / stk

流类	专用变量名
stringstream	ss
ostringstream	oss
istringstream	iss

参数常见名（函数列表）

术语：arguments

简写：arg

函数列表	写法
	递归参数 ...
下标（整数类型）	index / idx
深度（整数类型）	deep
维度（整数类型）	dimention / dim
层级（整数类型）	level, rank, layer
	二分参数、双指针参数 ...
左参数（整数类型）	left_head_side / lhs
右参数（整数类型）	righr_head_side / rhs
左边界（整数类型）	left / l
右边界（整数类型）	right / r
中点（整数类型）	middle / mid
	树论参数、图论参数 ...
值	value / val

键	key
代价	cost
权重	weight / w
	结构体、平衡二叉树 ...
当前节点下标（整数类型）	p（习惯性写法，无英文原型）
当前节点下标（整数类型）	i（需要计算节点下标的时候常用，例如静态数组完全二叉树）
当前树节点	root

容器	书写
目标容器（迭代器）	destination / dest
目标容器（迭代器）	location / loc
源容器（迭代器）	source / src
比较器（可调用对象）	compare / cmp
判断器（可调用对象）	predicate / pd
操作器（可调用对象）	operation / op
数组（迭代器）	array / arr
容器（迭代器）	container / cot
类型（尖括号内容）	type / T
指针	pointer / ptr / p
引用	reference / ref / r
迭代器	iterator it / i
起始迭代器	+begin / first
结束迭代器	+end / last

注：end 在 C++已经被占用，写的时候建议在前面加上类型缩写，如 send 代表 string 的 end

数据结构

链表	书写
下一个节点	back
上一个节点	previous / pre
头节点	head
尾节点	tail
节点	node
哑元/哨兵	dummy

树	书写
插入（操作）	insert / ins
删除（操作）	delete / del
建树（操作）	create
查询（操作）	search

图	书写
边的起点终点	u → v（符号无实际意义）
图的源点终点	s → t（符号无实际意义）
节点数	n（符号无实际意义）
图的储存	grids / g
到源点的距离	distance / dis
节点是否已访问	isVisited / visit / vis
边	edge / e
节点	vertex / v

矩阵	书写
矩阵	matrix

算法篇

动态规划	书写
二维数组/滚动数组/变量	dynamic_programming / dp

并查集	书写
祖先	father / f

附录 C（如何使用 VScode）

最基础的操作

第一步：创建一个文件夹，在里面创建 main.cpp

写一点简单代码（能跑即可）

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

int main(){
	return 0;
}
```

第二步：上方搜索栏输入“>”（或 Ctrl+Shift+p）

此时进入了命令面板

点击：C/C++ Edit Configurations (UI)
此时会自动进入一个窗口：IntelliSense Configurations

按照以下内容**选择**
Configuration name：Win32
Compiler path：C:/MinGW/bin/g++.exe（这里的路径必须有 bin 和 g++，其它不一样无所谓）
IntelliSense mode：gcc-x64 (legacy)

成功创建了 c_cpp_properties.json

第三步：左上角 Terminal→Configuration Task

选择：C/C++: g++.exe build active file
此时创建并进入了 task.json

第四步：左上角 Terminal→Run Build Task（或 Ctrl+Shift+b）

此时可能弹出命令面板，继续**选择**：C/C++: g++.exe build active file
下方 TERMINAL 会开始执行，执行结束以后，成功创建 main.exe

接下来下列操作可选，对流程无影响
按 Ctrl+Shift+~进入 TERMINAL
输入命令 .\main.exe 即可运行程序

第五步：Run→Add Configuration

此时开始执行一个构建任务。这个功能用于 Debug 调试

选择：C++(GDB/LLDB)
此时创建并进入了 launch.json

右下角点击：按钮 Add Configuration
此时文件会自动补充内容

按照以下内容**修改**

```
    "program": "enter program name, for example ${workspaceFolder}/a.exe"
→   "program": "${workspaceFolder}/main.exe"

    "miDebuggerPath": "/path/to/gdb",
→   "miDebuggerPath": "C:/MinGW/bin/gdb.exe",
```

（拷贝第二步的"compilerPath"内的路径，改 g++为 gdb）

（拷贝 task. json 的"command"内的路径，改 g++为 gdb）

接下来下列操作可选，可改善调试流程
接着在"configurations"的花括号内，写逗号然后**接一条命令**：

"preLaunchTask": "C/C++: g++.exe build active file"

（这条命令的内容就是 task.json 内的"label"的内容）
这条命令的作用是，每次调试程序之前生成新的 main.exe

额外插件

离线安装：进入 cmd，写 code -h。在弹出的表格内找到安装的方法。

Chinese Language Pack

Usage ：
You can override the default UI language by explicitly setting the VS Code display language using the **Configure Display Language** command.

Press **Ctrl + Shift + P** to bring up the **Command Palette** then start typing **display** to filter and display the **Configure Display Language** command.

Press **Enter** and a list of installed languages by locale is displayed,
with the current **locale** highlighted. Select another locale to switch UI language.

code runner

Usage ：
To run code:
use shortcut **Ctrl+Alt+N**
or press **F1** and then select/type **Run Code**,
or right click the Text Editor and then click **Run Code** in editor context menu
or click **Run Code** button in editor title menu
or click **Run Code** button in context menu of file explorer

To stop the running code:
use shortcut **Ctrl+Alt+M**
or press **F1** and then select/type **Stop Code Run**
or click **Stop Code Run** button in editor title menu
or right click the Output Channel and then click **Stop Code Run** in context menu

To select language to run:
use shortcut **Ctrl+Alt+J**
or press **F1** and then select/type **Run By Language**, then type or select the language to run

To run custom command:

use shortcut Ctrl+Alt+K
or press F1 and then select/type Run Custom Command

Debug Visualizer

可视化工具，链表专向破解

Usage ：
After installing this extension, use the command to open a new visualizer view. In this view you can enter an expression that is evaluated and visualized while stepping through your application. This view works the same as the watch view of VS Code, except that the resulting value is presented visually rather than textually and you can only watch one expression (but you can still open multiple windows).Debug Visualizer: New View

Use the command (Shift+F1) to use the currently selected text as expression in the most recently opened debug visualizer.Debug Visualizer: Use Selection as Expression

Prettier - Code formatter

Usage ：
Using Command Palette (CMD/CTRL + Shift + P)
1. CMD + Shift + P -> Format Document
OR
1. Select the text you want to Prettify
2. CMD + Shift + P -> Format Selection

Keyboard Shortcuts
Visual Studio Code provides default keyboard shortcuts for code formatting. You can learn about these for each platform in the VS Code documentation.

If you don't like the defaults, you can rebind and in the keyboard shortcuts menu of vscode.editor.action.formatDocumenteditor.action.formatSelection

CodeLf

Usage ：
Select text, right-click and select "Codelf".

常见问题

Debug 时数组和容器显示地址，不显示值

有一种可能是你的 MinGW 不行

方法 1：凑合着用*(type(*)[size])array_name

这是竞赛环境下，几乎唯一能做的。较为麻烦，犹如弃车徒步

如：*(int(*)[10]) v

注：不用担心数组越界，数组越界以后不会导致调试崩溃

方法 2：换一个 MinGW

非竞赛环境下可行

附录 D（快捷键）

以下基于 VS2022 和 VScode

存在差异的快捷键

	VS2022	VScode
Ctrl + [		退缩进
Ctrl +]	括号匹配	缩进
Alt + 鼠标	方格选择（左扫）	多选择（左点）
Alt + Shift + 鼠标	方格选择（左扫）	方格选择（左扫）
Alt + Shift + ↑	方格选择	抄写整行到上行
Alt + Shift + ↓	方格选择	抄写整行到下行
Ctrl + D	复制粘贴整行	选定单词（多按多选）
Ctrl + J	强制智能感知	呼出状态栏
Ctrl + Shift + L	行删除	变量全选
Ctrl + L	行删除	行选择
Ctrl + Enter	上开行	下开行
Ctrl + Shift + Enter	下开行	上开行

通用快捷键

F	F8	跳到下一个错误
	F12 + (Ctrl)	跳转至定义(声明)
Alt	Alt + 上下方向	移动整行
Ctrl	Ctrl + /	注释和去注释

全局整理

Ctrl + A

Ctrl + K + F

注释和去注释（有先后顺序）

Ctrl + K + C

Ctrl + K + U

VScode 独占

Ctrl + R	搜索面板
Ctrl + Shift + P	命令面板
Ctrl + Shift + N	新开窗口
Ctrl + Shift + W	关闭窗口
Ctrl + Shift + B	编译
Ctrl + F5	运行

VS2022 独占

Ctrl + R + R	重命名变量