

CP 必备技能及常见算法

张志心 PaperCloud
浙江大学

2025 年 2 月 6 日

写在前面……

讲课内容概览

本节课主要内容包括：

竞技编程 (Competitive Programming) 中必备的技能（非常重要！将伴随你的算法竞赛生涯）

常见算法（非常重要！其中包含着最基础的编程思想，也是赛场上获取部分分的武器）

① CP 必备技能

计算机知识

复杂度估计

高精度计算

文件输入输出

对拍技术

程序的调试

常见 STL 使用

② 数论基础

筛法

③ 常见基础算法

模拟

排序

枚举

贪心

二分

查找

递推

分治

① CP 必备技能

计算机知识

复杂度估计

高精度计算

文件输入输出

对拍技术

程序的调试

常见 STL 使用

② 数论基础

筛法

③ 常见基础算法

模拟

排序

枚举

贪心

二分

查找

递推

分治

在算法竞赛，程序的运行时间和空间都是有严格的限制的。

因此，我们需要对我们所使用的计算机有一定的了解，无需很深入，只要够实用即可。

目前家用电脑（包括 OI/XCPC 的评测机）的运行速度为，每秒

- 2×10^9 次整数的加法、减法、乘法、比较、位运算
- 1.5×10^8 次整数整除取模运算
- 2×10^8 次浮点运算
- 3×10^7 个字符的输入和输出（标准输入输出流）

计算机的精度其实远远没有我们想象的那么高：

- 计算机能快速计算的整数范围是 (`int`) $[-2^{31}, 2^{31} - 1]$ 和 (`long long`) $[-2^{63}, 2^{63} - 1]$ 。超出这个范围就会溢出。更大的范围尽管也有对应的数据类型 (如: `_int128`)，但计算速度明显减慢。
- 计算机的浮点数 `double` 存储范围约为 $\pm[2^{-1022}, 2^{1022}]$ 和 0，超出这个范围均会溢出。
- 计算机的浮点数精度只有 52 个二进制位。任何一个读入计算机的浮点数都自带 2^{-52} 的误差 (机器误差)。

如果不使用高精度算法，其实连精确表示 30 的阶乘都做不到。

计算机的精度

爆精度的常见场景

爆精度是初学者极易遇到的问题。因为有时即使最终结果是在某个数据类型的范围内的，但中间结果可能会溢出或掉精度从而导致最终结果错误。例如：

- 三个 $\leq 10^9$ 的 `int` 相加 ($3 \times 10^9 > 2^{31}$)，解决方法：改用 `long long`
- 两个 $\leq 10^5$ 的 `int` 相乘 ($10^{10} > 2^{31}$)，解决方法：改用 `long long`
- 某个中间结果是小数，被强制转换为整型后因丢失精度而少 1，例如 $8.0/3.0 - 5.0/3.0$ 在计算机中会算成 0.9999999999999998，这时如果强制转换为整型就会得到 0。解决方法：加一个略高于机器误差的 `eps` (如 10^{-12}) 后再转整型。
- 如果要求四舍五入，用 `std::round`。
- 想保留两位小数输出，结果得到了 -0.00，解决方法：特判。

如何优雅地读入浮点数？

虽然大部分良心的出题人都不会在数据读入上卡精度，但是还是有一些题目，需要在读入阶段读入浮点数（如：概率、点的坐标）。

不要直接使用没有 format 的 std::cin 读入浮点数！（也不建议使用 scanf）

建议的读入方式：

```
string sx; cin >> sx; double x = stod(sx); // std::stold
```

以三位小数为例，如果要将其乘 1000 后转整数，建议写成：int y = round(x * 1000);。

现代计算机内存一般为 $4GB$ 到 $32GB$ ，但事实上我们编写的程序能使用的内存只有 $2GB$ 左右（多了会直接报错）。 $1GB = 2^{30}B \approx 10^9B$ 。 $(1B = 8bit)$ 而一个整型 (`int`) 变量的大小为 $4B$ 。也就是说，即使是在本地运行程序，我们也只能定义 4×10^8 个整型变量 ($\approx 1.6GB$)，或者 2×10^8 个长整型 (`long long`)、浮点数 (`double`) 变量。

算法竞赛中，我们能使用的内存则更少。大部分题目的内存限制都在 $64MB$ 和 $1024MB(1GB)$ 之间。

一种压内存的方法：使用 `char` ($1B$) 来存不超过 256 的整型。

另外，一个程序占用的内存严格来说是它在运行期间占用内存的最大值（不是申请过的内存总量）。

① CP 必备技能

计算机知识

复杂度估计

高精度计算

文件输入输出

对拍技术

程序的调试

常见 STL 使用

② 数论基础

筛法

③ 常见基础算法

模拟

排序

枚举

贪心

二分

查找

递推

分治

复杂度估计

衡量算法运行速度的标准

我们一般用**时间复杂度**和**常数**两个标准来衡量一个算法的运行速度。

时间复杂度指当数据规模趋向于 ∞ 时，运行时间的增长速率。更准确地说，设 $T(n)$ 为数据规模为 n 时代码的运行时间，则时间复杂度就是 $T(n)$ 的等价无穷大量。例如 $O(1), O(\log n), O(n), O(n \log n), O(n^2), O(n^3), O(2^n), O(n!)$ 等。

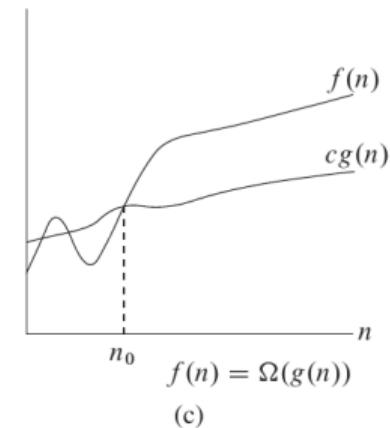
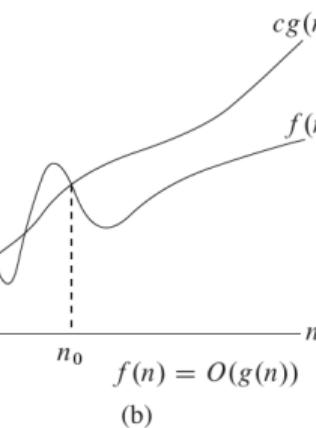
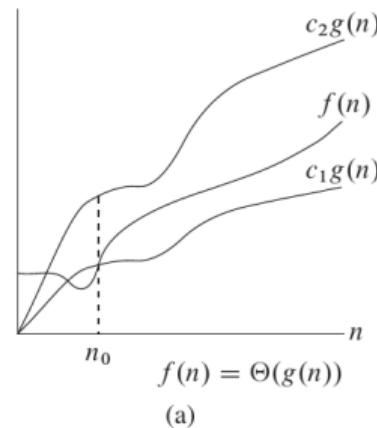
常数则指数据规模和数据强度充分大时，运行时间与时间复杂度比值的极限。它一般不能定量表示，只能定性比较“常数大”或者“常数小”。举个例子，同样的二重循环算法，一个循环体内部只有一次加法和赋值，另一个循环体内部有 20 次加法和赋值。显然后者常数更大，而且基本可以认为后者的运行时间是前者的 20 倍 (n 充分大时)。

复杂度估计

渐进符号的定义

定义

- $f(n) = \Theta(g(n)) \Leftrightarrow \exists n_0, c_1, c_2 > 0, \forall n > n_0, 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$
- $f(n) = O(g(n)) \Leftrightarrow \exists n_0, c > 0, \forall n > n_0, 0 \leq f(n) \leq c \cdot g(n)$
- $f(n) = \Omega(g(n)) \Leftrightarrow \exists n_0, c > 0, \forall n > n_0, 0 \leq c \cdot g(n) \leq f(n)$



时间复杂度的分析方法

对简单的顺序/分支/循环结构程序（没有结束条件不明的 while 循环和递归函数），一般可以直接估计总循环次数。（注意不能简单地看循环层数）例如下面三个程序

```
1 // O(n)
2 f[1]=f[2]=f[3]=1;
3 for(int i=4; i<=n; ++i)
4 for(int j=i-3; j<=i; ++j)
5     f[i]+=f[j];
```

```
1 // O(n^2)
2 for(int i=1; i<=n; ++i) {
3     f[i][0]=1;
4     for(int j=1; j<=i; ++j)
5         f[i][j]=f[i-1][j]+f[i-1][j-1];
6 }
```

```
1 // O(n log n)
2 for(int i=1; i<=n; ++i)
3 for(int j=i; j<=n; j+=i)
4     g[j]+=f[i];
```

虽然都是二重循环，但时间复杂度分别为 $O(n)$ 、 $O(n^2)$ 和 $O(n \log n)$ 。

时间复杂度的分析方法

而对于较复杂的程序（比如多维 dp），循环次数难以直接计算。常见的分析方法有：

- 放缩，直接估计最坏情况。这样得到的往往是上界。
- 转化，常见于图论算法的分析，统计每个点或边被访问的次数。（如图上 DFS 的复杂度为 $O(n + m)$ ）。
- **经典复杂度估计问题**：树上联通块优化问题（在树上求一个权值最大的且大小不超过 m 的联通块），通常考虑背包，因为每个点的重量为 1，在树上进行背包的复杂度为 $O(n^2)$ 。
- 在分治问题中，常使用 Master-Theorem（主定理）来求解复杂度。
- 直接用定义（造数据测试）。

主定理 (Master-Theorem)

我们可以使用 Master Theorem 来快速求得关于递归算法的复杂度。Master Theorem 递推关系式如下：假设分治问题中，一个规模为 n 的问题，使用 $f(n)$ 的代价，将其分解为 a 个规模为 n/b 的问题。(如，归并排序使用 $O(n)$ 的代价，将问题分解为 2 个规模为 $n/2$ 的问题)

定理 (Master Thm)

如果

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \quad \forall n > b$$

那么

$$T(n) = \begin{cases} \Theta(n^{\log_b(a)}) & f(n) = O(n^{\log_b(a)-\epsilon}), \epsilon > 0 \\ \Theta(f(n)) & f(n) = \Omega(n^{\log_b(a)+\epsilon}), \epsilon > 0 \\ \Theta(n^{\log_b(a)} \log^{k+1} n) & f(n) = \Theta(n^{\log_b(a)} \log^k n), k \geq 0 \end{cases}$$

归并排序的复杂度为： $O(n^1 \log^1 n)$ 。

时间复杂度分析的“指导性”作用

需要注意的是，时间复杂度分析起到的是“指导性”的作用。很多情况下，我们并不需要精确得到算法的时间复杂度，分析时间复杂度只是为了快速判断这个算法值不值得我们去写或者去提交。

例如，看到 10^5 甚至 10^6 规模数据的题目，就不要提交时间复杂度不低于 $O(n^2)$ 的代码了（虽然在现实生活中我们经常这样做）。而看到规模在 10^4 以下的题目，设计出 $O(n^2)$ 的算法就足够了，即使实际上这个问题有复杂度更低但更难实现的算法。

算法的空间复杂度

空间复杂度的概念和时间复杂度类似。即 n 趋向于 ∞ 时占用内存的增长速率。计算空间复杂度时，一般直接看我们实际申请的内存量即可。其他部分（头文件等）可看作常量，不超过 2MB

如果要对空间复杂度进行优化，一般考虑：

- 滚动数组代替多维数组。
- `bitset` 代替 `bool` 数组。
- 数组代替 STL 容器。
- 时间换空间。
- 重复使用一个数组/容器（不建议）。

① CP 必备技能

计算机知识

复杂度估计

高精度计算

文件输入输出

对拍技术

程序的调试

常见 STL 使用

② 数论基础

筛法

③ 常见基础算法

模拟

排序

枚举

贪心

二分

查找

递推

分治

高精度计算: 即大整数 (**bignum**) 计算, 运用了一些算法结构来支持更大整数间的运算 (数字大小超过语言内建整型)。

设计高精度计算的题目非常建议使用 Python。要注意在 Python 中的整数除法是 `//`, 而如果使用 `/`, 会自动把整型变为浮点数。

高精度计算

由于大整数的位数太多，考虑使用数组来存储高精度整数，数组中的每一位对应大整数 10 进制下一位的值。还需要一个符号位表示是否为负数。

```
int a[1005]; bool sig=0;
```

加法：模拟手算加法的过程，需要注意进位。（正数加法）

```
1 for(int i=0, carry=0; i<LEN; carry=(a[i]+b[i]+carry)/10, ++i)  
2     c[i]=(a[i]+b[i]+carry)%10;
```

减法：模拟手算减法的过程，需要注意借位。（正数减法，大减小）

```
1 for(int i=0, borrow=0; i<LEN; borrow=(a[i]-b[i]-borrow<0), ++i)  
2     c[i]=(a[i]-b[i]-borrow+10)%10;
```

高精度计算

乘法：模拟手算乘法的过程，需要注意进位。（高精乘单精）

```
1 int carry=0;
2 for(int i=0; i<LEN; carry=(a[i]*b+carry)/10, ++i)
3     c[i]=(a[i]*b+carry)%10;
4 c[LEN] = carry;
```

更多高精度计算可参考[LINK](#)

① CP 必备技能

计算机知识

复杂度估计

高精度计算

文件输入输出

对拍技术

程序的调试

常见 STL 使用

② 数论基础

筛法

③ 常见基础算法

模拟

排序

枚举

贪心

二分

查找

递推

分治

文件输入输出模版

一般情况下，用户使用标准输入流 `stdin` 和标准输出流 `stdout` 与程序进行输入输出的交互。在 OI 系列比赛中，如果一道题使用文件进行输入输出测试 (`sample.in/sample.out`)，我们需要调用文件输出输出流，在 C 中使用 `FILE* fopen()` 来获取文件，并使用 `fread/fwrite/...` 等一系列函数进行文件的读写操作，**上述做法使用起来并不方便，因此不做介绍。**

最方便的做法是在代码的主函数里添加：

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int main()
5 {
6     freopen("sample.in", "r", stdin);
7     freopen("sample.out", "w", stdout);
8     // ...
9 }
```

以上代码的作用其实是标准输入流 `stdin` 和标准输出流 `stdout` 重定向到文件 `sample.in/out`。

C++ 的输入输出流

在 C++ 的 `iostream` 库中，也有文件输入输出流。使用起来也很方便，比如以下是一个 $a + b$ 问题的代码：

```
1 ifstream fin("data.in");
2 int a, b;
3 fin >> a >> b;
4
5 ofstream fout("data.out");
6 fout << a+b << "\n";
```

值得一提的是，`iostream` 中还有字符串流，使得我们可以从一个字符串中读入数据（在一些输入格式很奇怪的题目中很有用）：

```
1 string s = "3 5"; int a, b;
2 stringstream sin(s);
3 sin >> a >> b;
4 cout << a + b << "\n";
```

C++ 的 stringstream

stringstream 的方便之处在于，可以把非字符串的数据转为含有一定格式的字符串（利用 `format` 的函数）。

比如：可以通过以下方式来将一个整数转化为长度为 `len` 的字符串（前导 0 补齐位数）

```
1 int len=10;
2 stringstream ss;
3 ss << setw(len) << setfill('0') << num ;
4 string str;
5 ss >> str;           // 将字符流传给 str
6 cout << str; // 或者 str = ss.str()
```

获取等式字符串：

```
1 int a,b; cin >> a >> b;
2 stringstream ss;
3 ss << format("{}+{}={} ", a, b, a+b);
4 string eq = ss.str();
```

① CP 必备技能

计算机知识

复杂度估计

高精度计算

文件输入输出

对拍技术

程序的调试

常见 STL 使用

② 数论基础

筛法

③ 常见基础算法

模拟

排序

枚举

贪心

二分

查找

递推

分治

对拍是一种进行检验或调试的方法，通过对比两个程序的输出来检验程序的正确性。可以将自己程序的输出与其他程序的输出进行对比，从而判断自己的程序是否正确。

对拍过程要多次进行，因此需要通过批处理的方法来实现对拍的自动化。

具体而言，对拍需要一个数据生成器和两个要进行输出结果比对的程序。

每运行一次数据生成器都将生成的数据写入输入文件，通过重定向的方法使两个程序读入数据，并将输出写入指定文件，最后利用 Windows 下的 `fc` 命令比对文件（Linux 下为 `diff` 命令）来检验程序的正确性。如果发现程序出错，可以直接利用刚刚生成的数据进行调试。

对拍模版

对拍程序的大致框架如下：对拍时 test1.cpp, test2.cpp, gen.cpp 不开文件输入输出！(windows 版本)

```
1 int main() {
2     // 编译部分：可省略
3     system("g++ gen.cpp -o gen");
4     system("g++ test1.cpp -o test1"); system("g++ test2.cpp -o test2");
5     // 对拍部分
6     while (true) {
7         system("gen > test.in");    // 数据生成器将生成数据写入输入文件
8         system("test1.exe < test.in > a.out"); // 获取程序1输出
9         system("test2.exe < test.in > b.out"); // 获取程序2输出
10        if (system("fc a.out b.out")) {
11            // 该行语句比对输入输出，fc返回0时表示输出一致，否则表示有不同处
12            system("pause"); // 方便查看不同处
13            return 0; // 该输入数据已经存放在 test.in 文件中
14        }
15        else cout << "PASS!\n";
16    }
17 }
```

对拍：数据生成器

需要生成与题目输入格式相同的数据，**保证数据的合法性**。

建议使用 `mt19937` 生成随机数（比 `rand()` 质量好）。

```
1 std::mt19937 rnd(time(nullptr)); // ui 类型随机数生成函数
2 // std::mt19937_64 可用于生成 ull 范围数。
3 ofstream fout("data.in");
4 fout<<rnd()%100<<" "<<rnd()%100<<"\n"; // 生成两个 [0, 99] 的整数。
5 fout.close();
6
7 vector<int> a(100);
8 for(int i=0; i<100; ++i) a[i]=i+1;
9 std::shuffle(a.begin(), a.end(), rnd); // 生成随机排列
10 // random_shuffle 已于 C++14 标准中被弃用，于 C++17 标准中被移除。
```

① CP 必备技能

计算机知识

复杂度估计

高精度计算

文件输入输出

对拍技术

程序的调试

常见 STL 使用

② 数论基础

筛法

③ 常见基础算法

模拟

排序

枚举

贪心

二分

查找

递推

分治

程序的调试

对于大部分 OI 中的代码，使用静态调试（眼睛看）和输出调试法即可。

可以结合宏定义，方便调试，并且避免调试较多，提交 oj 时候忘记删完。

```
1 #define DEBUG
2 #ifdef DEBUG
3 // do something when DEBUG is defined
4 #endif
5 // or
6 #ifndef DEBUG
7 // do something when DEBUG isn't defined
8 #endif
```

注：不少 OJ 都开启了 -DONLINE_JUDGE 这一编译选项，善用这一特性可以节约不少时间。所以上面的 `#ifdef DEBUG` 也可以写成 `#ifndef ONLINE_JUDGE`。

① CP 必备技能

计算机知识

复杂度估计

高精度计算

文件输入输出

对拍技术

程序的调试

常见 STL 使用

② 数论基础

筛法

③ 常见基础算法

模拟

排序

枚举

贪心

二分

查找

递推

分治

常见 STL 容器有：

序列式容器：vector, array, deque

关联式容器：set, multiset, map

无序（关联式）容器：unordered_set, unordered_multiset, unordered_map

- `begin()`: 返回指向开头元素的迭代器。
- `end()`: 返回指向末尾的下一个元素的迭代器。
- `size()`: 返回容器内的元素个数。
- `empty()`: 返回容器是否为空。
- `clear()`: 清空容器。

栈、队列、优先队列

不能使用迭代器。有 `size()`/`empty()`/`clear()`。

- 栈 (stack) 后进先出 (LIFO) 的容器。
- 队列 (queue) 先进先出 (FIFO) 的容器。
- 优先队列 (priority_queue) 元素的次序是由某种比较关系决定，默认是从大到小，常用于表示堆，单次操作复杂度为 $O(\log S)$ 。

```
1 stack<int> st; st.push(1); int x=st.top(); st.pop();
2 queue<int> q; q.push(1); int y=q.front(); q.pop();
3 priority_queue<int> pq; pq.push(1); int z=pq.top(); pq.pop();
4 priority_queue<int, vector<int>, greater<int>> pq2; // 从小到大。
```

向量：vector

内存连续的、可变长度的数组。能够提供线性复杂度的插入和删除，以及常数复杂度的随机访问。

```
1 vector<int> v0, v1(3), v2(3, 2), v3(v2), v4(v2.begin()+1, v2.end());
2 int x=v2[0], y=v2.front(), z=v2.back();
3 vector<int> v5({1,2,3});
4 for(auto x:v5); // range-for
5 for(auto it=v5.begin(); it!=v5.end(); ++it)
6     cout<<*it<<" "; // 迭代器
7 for(auto x: {1,2,3});
```

- `insert()`: 支持在某个迭代器位置插入元素 `v1.insert(v1.begin(), 1)`、可以插入多个 `v1.insert(v1.begin()+1, v2.begin(), v2.end())`。复杂度与 `pos` 距离末尾长度成线性而非常数的。
- `erase()`: 删除某个迭代器 `v2.erase(v2.begin()+1)` 或者区间的元素 `v2.erase(v2.begin()+1, v2.begin()+2)`，返回最后被删除的迭代器。复杂度与 `insert` 一致。
- `push_back()`, `pop_back()`: 插入、删除最后一个元素（注意在 `pop_back()` 前判断容器非空）。

集合： set/multiset

搜索、移除和插入拥有对数复杂度。set 内部通常采用红黑树实现。平衡二叉树的特性使得 set 非常适合处理需要同时兼顾查找、插入与删除的情况。和数学中的集合相似，set 中不会出现值相同的元素。如果有相同元素的集合，需要使用 multiset，使用方法与 set 的使用方法基本相同。

```
1 set<int> s1({1, 2, 3}), s2; s2.insert(1);
2 for(auto x : s1); // range-for
3 for(auto x=s1.begin(); x!=s1.end(); ++x); // 迭代器
4 for(auto x=s1.rbegin(); x!=s1.rend(); ++x); // 反向迭代
```

- insert(x)：当容器中没有等价元素的时候，将元素 x 插入到 set 中。
- erase(x)：删除值为 x 的所有元素，返回删除元素的个数。multiset 如果只要删除一个元素，请使用对迭代器的删除：ms.erase(ms.find(x))。
- earse(pos)：删除迭代器为 pos 的元素，要求迭代器必须合法。
- count(x)：返回 x 的元素数量。
- find(x)：存在键为 x 的元素时会返回该元素的其中一个迭代器，否则返回 end()。
- lower_bound(x), upper_bound(x)：返回首个大于等于/大于 x 的元素的迭代器，如果不存在，返回 end()。

映射：map

有序键值对容器，它的元素的键是唯一的。搜索、移除和插入操作拥有对数复杂度。`map` 通常实现为红黑树。

```
1 map<string, int> mp1; mp1["A"] = 0, mp1["B"] = 1;
2 mp1.insert({"C", 2});
3 if (mp1.count("C")) cout << mp1["C"];
4 if (mp1.find("C") != mp1.end()) cout << mp1.at("C");
5 cout << mp1.lower_bound("B")->second << " " << mp1.upper_bound("B")->second;
6 // 1 2
7 for (auto [x, y] : mp1); // range-for
```

如果访问次数很多，且常常访问到不存在键值对的位置，建议使用 `at(x)`，如果使用 `[]` 访问，会在不存在的位置插入一个键值对，并让其值为该类型的默认构造值（如 `int` 为 0）。

以下函数适用于 STL 容器。使用以下函数的目的是方便编写代码，并不能提升运行速度。

```
1 count(v.begin(), v.end(), 1) // 1 的个数
2 int mx=*max_element(v.begin(), v.end()); // 最大值
3 int mx=*min_element(v.begin(), v.end()); // 最小值
4 reverse(v.begin(), v.end()); // 翻转容器
5 remove(v.begin(), v.end()); // 删除元素 5
6 sort(v.begin(), v.end()); unique(v.begin(), v.end()); // 去重
7 // remove 和 unique 并没有真的删除元素所占用的内存！
8 int sum=accumulate(v.begin(), v.end(), 0); // 元素求和
9 fill(v.begin(), v.end(), 0); // 元素赋初值（序列容器）
```

`next_permutation(v.begin(), v.end())`: 将当前排列更改为全排列中的下一个排列。如果当前排列已经是全排列中的最后一个排列（元素完全从大到小排列），函数返回 `false` 并将排列更改为全排列中的第一个排列（元素完全从小到大排列）；否则，函数返回 `true`。

bitset 优化 `bool` 数组/位数很大的二进制数的位运算（与或非，左移右移），由于其内部使用了压位优化，因此内存和各种操作的时间复杂度为 $O(\frac{n}{w})$ ，其中 $w = 32/64$ 为计算机的位数。

- `bitset<1000> bs;` 开一个大小为 1000 的 bitset，默认全为 0。
- `operator []`: 访问特定一位。
- `operator <</operator >>/operator <<=/operator >>=`: 进行二进制左移/右移。

bitset 的成员函数

- `count()`: 返回 `true` 的数量。
- `size()`: 返回 `bitset` 的大小。
- `test(pos)`: 它和 `vector` 中的 `at()` 的作用是一样的，和 `[]` 运算符的区别就是越界检查。
- `any()`: 若存在某一位是 `true` 则返回 `true`，否则返回 `false`。
- `none()`: 若所有位都是 `false` 则返回 `true`，否则返回 `false`。
- `all()`: 若所有位都是 `true` 则返回 `true`，否则返回 `false`。
- - a. `set()`: 将整个 `bitset` 设置成 `true`。
 - b. `set(pos, val = true)`: 将某一位设置成 `true / false`。
- - a. `reset()`: 将整个 `bitset` 设置成 `false`。
 - b. `reset(pos)`: 将某一位设置成 `false`。相当于 `set(pos, false)`。
- - a. `flip()`: 翻转每一位。 $(0 \leftrightarrow 1$, 相当于异或一个全是 1 的 `bitset`)
 - b. `flip(pos)`: 翻转某一位。
- `to_string()`: 返回转换成的字符串表达。
- `to_ulong()`: 返回转换成的 `unsigned long` 表达 (`long` 在 NT 及 32 位 POSIX 系统下与 `int` 一样，在 64 位 POSIX 下与 `long long` 一样)。
- `to_ullong()`: (`C++11` 起) 返回转换成的 `unsigned long long` 表达。

另外，`libstdc++` 中有一些较为实用的内部成员函数¹:

- `_Find_first()`: 返回 `bitset` 第一个 `true` 的下标，若没有 `true` 则返回 `bitset` 的大小。回
- `_Find_next(pos)`: 返回 `pos` 后面（下标严格大于 `pos` 的位置）第一个 `true` 的下标，若 `pos` 后面没有 `true` 则返回 `bitset` 的大小。

STL 总结

STL 为写代码提供了很大的便利，希望大家在今后的学习中可以慢慢积累，慢慢领悟。

随着 C++ 语言的不断更新，其功能也变得越来越强大。如

```
cout << format("{}+{}={}", a, b, a+b);
```

有关 C++ 的语法可以在 <https://zh.cppreference.com/w/cpp/language> 中查找。

① CP 必备技能

计算机知识

复杂度估计

高精度计算

文件输入输出

对拍技术

程序的调试

常见 STL 使用

② 数论基础

筛法

③ 常见基础算法

模拟

排序

枚举

贪心

二分

查找

递推

分治

① CP 必备技能

计算机知识

复杂度估计

高精度计算

文件输入输出

对拍技术

程序的调试

常见 STL 使用

② 数论基础

筛法

③ 常见基础算法

模拟

排序

枚举

贪心

二分

查找

递推

分治

素数和素数判定

素数： ≥ 2 且因数只有 1 和本身的整数。

单个整数 n 的素数判定需要判断 $2, 3, \dots, \lfloor \sqrt{n} \rfloor$ 是否整除 n ，时间复杂度是 $O(\sqrt{n})$ 的。当我们需要求 1 到 10^6 的所有素数时，这样试除就太慢了。

筛法

埃氏筛法可以以几乎线性的时间复杂度找到 $1 \sim n$ 中的所有素数。筛法的流程非常简单，把所有数列出来，然后

划掉除 2 外所有被 2 整除的数字；

划掉除 3 外所有被 3 整除的数字；

划掉除 5 外所有被 5 整除的数字；

……

划到 $\lfloor \sqrt{n} \rfloor$ 为止，剩下的就都是质数了。

时间复杂度 $O(n \log \log n)$ (证明需要解析数论的知识，此处省略)。

```
1  bool np[N];
2  void sieve(int n){
3      np[0]=np[1]=1;
4      for(int i=2; i*i<=n; ++i) if(!np[i])
5          for(int j=i*i; j<=n; j+=i) np[j]=1;
6  } // 如果从 i 开始筛，复杂度为 O(n log n)
```

筛法可以在 1 秒内找到不超过 10^7 的全部素数。

一个简单优化：除 2 和 3 外，仅判断 $6k+1$ 和 $6k+5$ 型的整数。常数 /=3。

- 
- ① CP 必备技能
 - 计算机知识
 - 复杂度估计
 - 高精度计算
 - 文件输入输出
 - 对拍技术
 - 程序的调试
 - 常见 STL 使用
 - ② 数论基础

筛法

③ 常见基础算法

模拟

排序

枚举

贪心

二分

查找

递推

分治

- 
- ① CP 必备技能
 - 计算机知识
 - 复杂度估计
 - 高精度计算
 - 文件输入输出
 - 对拍技术
 - 程序的调试
 - 常见 STL 使用
 - ② 数论基础

筛法

③ 常见基础算法

模拟

排序

枚举

贪心

二分

查找

递推

分治

模拟 就是按题意一步步实现题目所描述的流程。

纯模拟题一般不涉及其他算法和数据结构（或者即使涉及到了，也只需要调用一些简单的 STL），也无需过多关注时间和空间复杂度的问题，其更多考察的是选手读题和检索有效信息的能力。

建议大家在遇到这类题目时将题面上所有有效信息都用笔圈出来，并且在开始写程序之前，先思考应该用什么结构来存储数据。这里强烈建议用 `struct` 来描述同一个实体的所有数据，非常不建议在变量很多时用 `a,b,c` 等无实义的变量名。还应该合理地用多个函数来封装不同的功能，提高代码的模块化，便于调试。

- 
- ① CP 必备技能
 - 计算机知识
 - 复杂度估计
 - 高精度计算
 - 文件输入输出
 - 对拍技术
 - 程序的调试
 - 常见 STL 使用
 - ② 数论基础

筛法

③ 常见基础算法

模拟

排序

枚举

贪心

二分

查找

递推

分治

排序

排序问题

排序 (Sorting): 给一个序列，要求从小到大排序或从大到小排序。

常见的基于**比较**的排序算法包括：

冒泡排序、选择排序、插入排序、快速排序、归并排序、堆排序等。这些我们都不讲

STL 的 `algorithm` 库里已经封装了排序算法，其底层实现为快速排序，时间复杂度为 $O(n \log n)$ ，实际测试时可以在 1 秒内对 2×10^6 个无序整数进行排序。格式为

```
sort(begin, end, cmp);
```

这里 `begin` 和 `end` 分别为待排序序列的头指针和尾指针（左闭右开），`cmp` 为自定义比较函数（若省略则默认从小到大）。例如对 `a[1...n]` 排序就是 `sort(a+1, a+n+1);`。

排序

自定义比较运算

在很多题目中我们可能要对一些结构体进行排序。例如结构体 node 有三个整型成员 x,y,z，我们要以 x 从小到大为第一关键字，以 y 从大到小为第二关键字排序。那么可以这样实现：

```
1 struct node{  
2     int x, y, z;  
3     bool operator < (const node& p) const  
4     { return x < p.x || (x == p.x && y > p.y); }  
5 } a[N];  
//...  
6 sort(a + 1, a + n + 1);  
7
```

排序

自定义比较运算

或者也可以自定义比较函数（一般在需要以不同方式对同种结构体多次排序时使用）：

```
1 struct node {
2     int x, y, z;
3 } a[N];
4 // ...
5 sort(a + 1, a + n + 1,
6      [&] (node a, node b) -> bool
7 {return a.x < b.x || a.x == b.x && a.y > b.y;});
```

桶排序 (Bucket Sort) 与其他排序算法不同，它不是基于比较的，而是基于值域的。它的复杂度只有 $O(n + W)$, W 为值域。

例如，我们需要对 $n \leq 10^7$ 个数进行排序，但它们的值域 $a_i \leq 10^6$ 。此时再使用 `sort` 就会超时，但使用桶排序则可以在 1 秒内完成排序。

```
1 int n, m=1e6, a[N], buc[N];
2 //...
3 for(int i=1; i<=n; ++i) ++buc[a[i]];
4 int cnt=0;
5 for(int i=1; i<=m; ++i)
6     for(int j=1; j<=buc[i]; ++j)
7         a[++cnt]=i;
```

- 
- ① CP 必备技能
 - 计算机知识
 - 复杂度估计
 - 高精度计算
 - 文件输入输出
 - 对拍技术
 - 程序的调试
 - 常见 STL 使用
 - ② 数论基础

筛法

③ 常见基础算法

模拟

排序

枚举

贪心

二分

查找

递推

分治

枚举

何为枚举

“计算机最擅长的就是重复。”

枚举，就是对大量的元素做同一件事。因此它也常被称为“暴力”“穷举”“brute force”。

这个“元素”，可以是一个变量，也可以是几个变量或结构，甚至是更抽象的一些东西……

总之，就是

```
for(x s.t. P(x)) do_something(x)
```

枚举

枚举之前的思考

但是，在下手写代码之前，我们最好先停下来问自己几个问题：

- 枚举的时间复杂度是多少？会不会超时？
- 所有情况都枚举到了吗？（充分性）
- 会不会枚举到一定没有用的情况？（必要性）
- 如果有，把它们优化掉会本质上降低算法的复杂度吗？此时应考虑最坏情况。

Nearly Shortest Repeating Substring

给一个字符串 S , 求最小的 k 使得将 S 修改一个字符后可以成为循环节为 k 的串。

$1 \leq n \leq 10^5$ 。字符集为全体小写字母。

Nearly Shortest Repeating Substring

题解

记 $n = |S|$ 。如果枚举修改的字符，需要 $O(n|\Sigma|)$ 轮枚举。而检查循环节需要 $O(nd(n))$ ($d(n)$ 为 n 的约数个数)。总复杂度 $O(n^2|\Sigma|d(n))$ 显然会超时。

但如果我们知道循环节，且假设当前循环节是合法的，那么能不能直接求出修改的字符呢？

Nearly Shortest Repeating Substring

题解

答案是肯定的。

设枚举的循环节为 d , $k = \frac{n}{d}$, 则 $S[1 \dots d], S[d+1 \dots 2d], \dots, S[n-d+1 \dots n]$ 中, 有 $k-1$ 个串相等, 另一个串和它们仅差一个字符。

枚举每一段长为 d 的子串, 将它们和 $S[1 \dots d]$ 比较。

- 若不同的字符多于一个, 则该循环节不合法;
- 否则, 若只有一个串不同, 则该循环节合法;
- 特殊情况: 若所有串都和 $S[1 \dots d]$ 不同, 则可能是 $S[1 \dots d]$ 与其他串不同。此时再检查后面的串是否全相同。

复杂度为 $O(nd(n))$ 。

- 
- ① CP 必备技能
 - 计算机知识
 - 复杂度估计
 - 高精度计算
 - 文件输入输出
 - 对拍技术
 - 程序的调试
 - 常见 STL 使用
 - ② 数论基础

筛法

③ 常见基础算法

模拟

排序

枚举

贪心

二分

查找

递推

分治

贪心法是一种解决最优化问题的思想。某种意义上是枚举法的反面。

枚举是暴力穷举一切情况，而贪心则是观察数据的性质，推导出最优解应该满足的某种性质。即所谓“用数学证明代替程序验证”。

在实际问题中，枚举和贪心一般交替进行。这也正对应着枚举法的“必要性”——贪心法会帮助我们去掉一些隐含的无用情况。

贪心

贪心法的一般过程与适用条件

作为一种思想而非方法，贪心法的过程很难用明确的步骤描述。

需要用贪心法解决的问题具有如下特征：

- 问题往往是最优化一个“选取方案”或者“选取顺序”。
- 直接枚举的复杂度极高甚至不可枚举（无限多种情况）。

贪心法的大体步骤：

- 确定遍历元素的顺序。
- 以该顺序遍历所有元素，每次决策都能达到当前的**局部最优解**。

一个问题能适用贪心法，仅当按顺序考虑每个元素时，**局部最优解等价于整体最优解**，即整体最优解的每一步决策都是最优的，当前的决策仅于下一步可供选择的效益相关，与当前的状态、可能的更长远的效益……都无关。

贪心

不能使用贪心法解决的问题

有一些问题，它们整体最优并不等价于局部最优。例如数字三角形、01 背包问题等等。

这些问题是没有整体最优等价于局部最优的性质的，每一步选择都依赖于当前的状态，或是更长远的效益。它们只能用搜索和动态规划等方法求解。

有 k 组石子，每组有 n 堆。第 i 组的第 j 堆石子为 a_j^i 个，且对所有 i 均有 $\sum_{j=1}^k a_j^i = m$ 。对每组石子，每次操作可以从一堆石子中移动一颗石子到与其相邻的一堆石子中。求最少总移动次数，使每组石子的每堆石子对应相等（即 $\forall i, i', j, a_j^i = a_j^{i'}$ ），并求相等时的石子分布。

$$1 \leq kn \leq 10^5.$$

Game With Stones

题解

设相等时的石子分布为 b_j 。设 $W(a, b)$ 是从分布 a 移动到分布 b 所需移动次数，这是一个类似石子均分问题的过程：

$$W(a, b) = \sum_{j=1}^n |S(a)_j - S(b)_j|,$$

其中 $S(a)$ 是 a 的前缀和数组。因此总操作数为

$$\sum_{i=1}^n W(a^i, b) = \sum_{i=1}^k \sum_{j=1}^n |S(a^i)_j - S(b)_j|.$$

交换求和号，对 i 求和，即 $\sum_{i=1}^k |S(a^i)_j - S(b)_j|$ 。其中 $S(x)$ 为 x 的前缀和数组。

即给定数轴上 k 个点，求一个点与它们的距离之和最小。容易证明选择这些点坐标的中位数即可。

- 
- ① CP 必备技能
 - 计算机知识
 - 复杂度估计
 - 高精度计算
 - 文件输入输出
 - 对拍技术
 - 程序的调试
 - 常见 STL 使用
 - ② 数论基础

筛法

③ 常见基础算法

模拟

排序

枚举

贪心

二分

查找

递推

分治

二分答案

最优化问题和判定性问题

二分算法可以将某些最优化问题转化为判定性问题。

如果一个最优化问题的形式为满足条件 $P(x)$ 的 x 的最大（最小）值，且 $P(x)$ 具有单调性：若 x 满足 $P(x)$ ，则 $y > x (y < x)$ 也满足 $P(y)$ ，则我们可以二分这个最优的 x 。

二分答案

整数二分

当 x 为整数时，我们可以在 $O(\log(M - m))$ 的时间内找到答案。其中 M, m 分别为答案的最大值和最小值（需要提前预估）。

```
1 int l = m, r = M, mid, ans = r+1;
2 while (l <= r) {
3     mid = (l + r) / 2;
4     if (check(mid)) ans = mid, r = mid - 1;
5     else l = mid + 1;
6 }
```

二分答案

实数二分

当 x 为实数时，我们可以在 $O(\log \frac{M-m}{\epsilon})$ 的时间内找到打完。其中 ϵ 为答案所需的绝对精度。

```
1 const double eps = 1e-6;
2 double l = m, r = M, mid, ans = r;
3 while (r - l > eps) {
4     mid = (l + r) / 2;
5     if (check(mid)) ans = mid, r = mid;
6     else l = mid;
7 }
```

二分答案

minmax / maxmin

一般来说，对于**最小化最大值、最大化最小值**的问题，我们都可以使用二分答案将它们转化为判定性问题。具体转化步骤如下，以最小化最大值为例：

- 原题意：确定满足某些条件的集合 S ，使其最大值最小。
- 转化：求最小的 x 使 S 的所有元素都不超过 x 。
- 单调性：若 x 满足上述条件，则 $y > x$ 也满足。
- 二分 x ，转化为 S 中元素是否能都不超过 x 。

这个转化可以使一个困难的组合优化问题变为多个简单的判定性问题。

最大化最小值也同理。

跳石头

数轴上有 $n + 2$ 个点 $\{a_i\}$, $a_0 = 0, a_{n+1} = L$ 。在 a_1, \dots, a_n 中移除至多 m 个点, 使剩余点两两之间距离的最小值最大。

$$1 \leq m \leq n \leq 10^5, 1 \leq a_i < L \leq 10^9.$$

跳石头

题解

最大化最小值，考虑二分答案。

问题转化为判断是否存在一种移除方案使任意两个相邻点的距离不小于 x 。

贪心，对每个点，若它和上一个未被移除的点距离小于 x ，则移除这个点。

时间复杂度 $O(n \log n)$

二分答案

0/1 分数规划

我们考虑这样一个问题：

- 有 n 个物品，每个物品的价值为 v_i ，重量为 w_i 。选择恰好 k 个物品使得它们的价值之和与重量之和的比值最大。

二分答案

0/1 分数规划

这种物品可选可不选，在一定限制的前提下最优化比值的问题，就称为 **0/1 分数规划问题**。它也是一类经典的二分答案问题。

转化为判定性问题：判断是否存在选择方案，使价值和与重量和的比值不小于 x 。即

$$\frac{\sum_{i \in S} v_i}{\sum_{i \in S} w_i} \geq x.$$

即

$$\sum_{i \in S} (v_i - w_i x) \geq 0.$$

按 $v_i - w_i x$ 从大到小排序并选择前 k 个即可。

二分答案

最优化第 k 大

最大/最小化第 k 小值同样可以使用二分答案。

具体地，我们可以将它转化成判定性问题：第 k 小值不小于 x 。

它等价于小于 x 的数至多有 $k - 1$ 个。

进而我们可以把所有数字分成两类：大于 x 的和小于 x 的，然后进行 check。

有 n 张卡牌，第 i 张卡牌的正面是 a_i ，反面是 b_i 。初始所有卡牌均正面朝上。你可以至多进行一次操作，选择一个区间 $[l, r]$ 并将第 l 到第 r 张卡牌全部翻面。求操作后所有卡牌朝上的数字的中位数的最大值。

$1 \leq n \leq 3 \times 10^5$ ， n 为奇数。

Flipping Cards

题解

对于这个问题，它可以转化为是否存在一种方案，使得朝上的小于 x 的数字至多有 $\lfloor \frac{n}{2} \rfloor$ 个，或者说大于 x 的数字至少有 $\lfloor \frac{n+1}{2} \rfloor$ 个。

我们将小于 x 的数字看成 0，大于等于 x 的数字看成 1。对一张卡牌，若正面反面相同，则翻转无影响；若正面 0 反面 1，则翻转会让 1（即大于等于 x 的数字）增加一个；若正面 1 反面 0，则翻转会让 1 减少一个。

问题转为判断一个取值在 $\{0, 1, -1\}$ 中的序列 $\{b_i\}$ 的最大子区间和是否不小于 $\frac{n+1}{2} - cnt$ 。

- 
- ① CP 必备技能
 - 计算机知识
 - 复杂度估计
 - 高精度计算
 - 文件输入输出
 - 对拍技术
 - 程序的调试
 - 常见 STL 使用
 - ② 数论基础

筛法

③ 常见基础算法

模拟

排序

枚举

贪心

二分

查找

递推

分治

查找问题

有序序列的查找: 给一个排好序的序列，要查找某个元素 x 在序列中的存在性和排名（即小于等于 x 的元素的数量）

无序集合的查找: 给一个集合，问某个元素是否存在。

有序序列的查找

二分查找

二分查找 可以以 $O(\log n)$ 的时间复杂度查找到 x 的存在性和排名。

假设序列从小到大排序，若当前的查找范围是 $\{a_l, \dots, a_r\}$ ，我们将 x 与 $a_k, k = \lfloor \frac{l+r}{2} \rfloor$ 比较，若 $x \leq a_k$ ，则查找范围可缩小到 $[l, k]$ ；若 $x > a_k$ ，则查找范围可缩小到 $[k+1, r]$ 。则可以将查找范围缩小一半。直到 $l = r$ 为止。

有序序列的查找

二分查找

二分查找的代码 (在 a 数组中查找, 返回下标):

```
1 int search(int x) {
2     int l = 1, r = n;
3     while(l < r) {
4         int mid = (l+r) / 2;
5         if (x <= a[mid]) r = mid;
6         else l = mid+1;
7     }
8     return l;
9 }
```

二分查找同样已在 `algorithm` 库中封装了。库函数的格式为：

```
lower_bound(begin, end, x, cmp);
```

这里`begin/end/cmp`的意义同 `sort`。它返回的是一个指针，指向数组中第一个大于等于 x 的元素的位置，若全部小于 x ，则指向`end`。注意，在使用`lower_bound`前必须保证数组有序，否则返回值是无法预知的。

另外，用法类似的还有`upper_bound`。它与`lower_bound`的唯一不同点在于返回值是第一个大于 x 的元素的位置。

无序集合的查找

哈希查找 可以以 $O(1)$ 的时间复杂度查找到 x 在某个集合中的存在性。但它需要 $O(n)$ 或 $O(W)$ 的额外空间。

哈希查找是通过数据结构**哈希表**实现的。哈希表的实现比较复杂，但 STL 已经对其进行了封装 (`unordered_set`/`unordered_map`)，其用法与 `set`/`map` 相同，复杂度为 $O(1)$ 但没有`lower_bound`/`upper_bound`的接口。

离散化

离散化是一个映射，它把一个集合中的元素映射为一个排列，且保持它们的大小关系不变。当我们只关心数据的大小关系而不关心它们具体的数值时，我们就要对数据进行离散化。

离散化分为三步：排序，去重，建立映射。每一步都有 STL 实现。

```
1 for (int i=1; i<=n; ++i) buc[i] = a[i];
2 sort(buc+1, buc+n+1), m = unique(buc+1, buc+n+1) - buc - 1;
3 for (int i=1; i<=n; ++i) a[i] = lower_bound(buc+1, buc+n+1, a[i]) - buc;
```

这里unique函数可以给一个有序数组进行去重，返回的是去重后的数组的尾指针。

① CP 必备技能

计算机知识

复杂度估计

高精度计算

文件输入输出

对拍技术

程序的调试

常见 STL 使用

② 数论基础

筛法

③ 常见基础算法

模拟

排序

枚举

贪心

二分

查找

递推

分治

递推

递推公式

递推公式就是类似 $a_i = f(a_{i-1}, a_{i-2}, \dots)$ 或者 $a_i = f(a_{i+1}, a_{i+2}, \dots)$ 的公式。它的左端是未知项，右端是已知或在之前的递推中求出来的项。例如下面这些都是递推公式：

- **前缀和** $s_0 = 0, s_i = s_{i-1} + a_i$
- **后缀和** $s_{n+1} = 0, s_i = s_{i+1} + a_i$
- **二维前缀和** $s_{0,0} = 0, s_{i,j} = s_{i-1,j} + s_{i,j-1} - s_{i-1,j-1} + a_{i,j}$
- **斐波那契数列** $f_0 = f_1 = 1, f_i = f_{i-1} + f_{i-2}$ (前八项为 1,1,2,3,5,8,13,21)
- **组合数** $c_{i,0} = 1, c_{i,j} = c_{i-1,j} + c_{i-1,j-1}$
- **卡特兰数** $C_1 = 1, C_n = \frac{4n-2}{n+1} C_{n-1}$ (前七项为 1,2,5,14,42,132,429)

而这个就不是递推公式而是线性方程组：

- $a_{-1} = a_{n+1} = 0, a_i = a_{i-1} + a_{i+1} + 1$

递推

递推公式的推导和实现

递推问题一般都是数学或计数的问题。在推导递推式时，我们可以观察每一项和它的前几项或前缀和之间的关系，然后尝试推导。当然在不要求证明的算法竞赛中，更方便的做法是先写一个更暴力的做法（枚举，搜索等）打出前 10 到 20 项的表，再尝试寻找递推式甚至通项公式。

在推导出递推公式后，我们处理边界条件后直接按顺序用 `for` 循环递推即可。如果顺序难以确定，也可以用记忆化搜索。

递推式和 `dp` 式的定义和形式都非常类似，只不过解决的问题不同（`dp` 解决的是最优化问题）。所以某些题解会把递推和 `dp` 混淆。

将 1 到 n 任意排列，然后在排列的每两个数之间根据他们的大小关系插入 $>$ 和 $<$ 。

问在所有排列中，有多少个排列恰好有 k 个 $<$ 。

例如排列 $(3, 4, 1, 5, 2) \rightarrow 3 < 4 > 1 < 5 > 2$ ，共有 2 个 $<$

答案对 998244353 取模。

$1 \leq k \leq n \leq 5000$

设 $f(n, k)$ 为 n 个数 k 个 $<$ 的方案数，考虑 n 插入一个长度为 $n - 1$ 的排列时的位置，若 n 的左侧是 $<$ 或 n 在最左侧，则 $<$ 的数量不变；若 n 的左侧是 $>$ ，则 $<$ 的数量加 1。因此有

$$f(n, k) = (n - k + 1)f(n - 1, k - 1) + kf(n - 1, k)$$

$O(n^2)$ 递推即可。

① CP 必备技能

计算机知识

复杂度估计

高精度计算

文件输入输出

对拍技术

程序的调试

常见 STL 使用

② 数论基础

筛法

③ 常见基础算法

模拟

排序

枚举

贪心

二分

查找

递推

分治

分治

“分而治之”

分治 (Divide and Conquer)，字面上的解释是“分而治之”，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。

因此分治的流程也可以分为三步：分解 → 解决 → 合并：

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

分治

分治法解决的问题

分治法能解决的问题一般有如下特征：

- 该问题的规模缩小到一定的程度就可以容易地解决。
- 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质，利用该问题分解出的子问题的解可以合并为该问题的解。
- 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。

尽管很少会有题目专门考察分治思想，但分治思想广泛地应用于几乎一切算法中：归并排序，二叉树遍历，二叉搜索树，线段树（等一切二叉树形结构），树分治等等。

归并排序

我们以归并排序为例来理解分治法的流程：

假设我们目前要对 $a[l \dots r]$ 排序。我们先对 $a[l \dots m]$ 和 $a[m + 1 \dots r]$ 排序（“分解”），如果区间内只有一个数就不需要排了（“解决”），然后再用 $O(r - l + 1)$ 的时间复杂度将这两个有序数组合并为一个有序数组（“合并”）。总时间复杂度满足 $T(n) = 2T(\frac{n}{2}) + O(n)$ 故 $T(n) = O(n \log n)$ 。

给一个长为 n 的序列，每个数都是 $[0, k)$ 中的整数。

求这个序列中满足下面两个条件的非空子序列有多少个。

- 子序列中第一个数和最后一个数的位置之差至少为 L ;
- 子序列中所有数之和是 k 的倍数。

由于答案过大，你只需输出它对 998244353 取模的值。

$$1 \leq n \leq 10^5, 1 \leq k \leq 50$$

首先有一个暴力递推：先枚举序列的开头 s ，然后设 $f(i, j)$ 为前 i 个数，和模 k 余 j 的子序列数量：

$$f(s, a_s) = 1, f(i, j) = \sum_{l=s}^{i-1} f(l, (j - a_i) \bmod k), i > s$$

前缀和优化后时间复杂度为 $O(n^2 k)$ 。

考虑分治。 $[l, r]$ 中所有的合法子序列包括： $[l, m]$ 中的合法子序列， $[m + 1, r]$ 中的合法子序列，跨过中点的合法子序列。

前两个继续分治计算即可，分到区间长度小于 L 就可以直接退出了。我们考虑如何在 $O(nk)$ 的时间内计算出跨过终点 m 的合法子序列。这些子序列一定由左右两段拼接而成。

KL-subsequence

题解

设 $f(i, j)$ 表示 $a[l \dots m]$ 中开头为 i , 结尾任意, 和模 k 余 j 的子序列数量;
设 $g(i, j)$ 表示 $a[m + 1 \dots r]$ 结尾为 i , 开头任意, 和模 k 余 j 的子序列数量;
这两个递推都可以 $O(nk)$ 计算 (比起直接暴力少了一步枚举)。

对于每个左半边开头为 i 且和模 k 余 j 的序列, 它都可以与右半边结尾在 $i + L$ 之后且和模 k 与 $k - j$ 的序列配对而形成一个合法的子序列。故这部分总方案数为

$$\sum_{i=l}^m \sum_{j=0}^{k-1} \sum_{t=i+L}^r f(i, j)g(t, k - j)$$

最后一个 \sum 可以用后缀和优化掉。故这部分复杂度也为 $O(nk)$ 。

因此总时间复杂度为 $O(nk \log n)$