

蓝桥杯省赛知识点 & 精练（下）

蓝桥杯省赛知识点 & 精练（下）

4月9日蓝桥杯省赛正式开赛!!!

☑ 蓝桥杯近 3 年常考知识点

📁 蓝桥杯近 5 年真题——知识点对应

📢 省赛一等奖学霸的备赛小贴士

1. 大神备考心得
2. 最后一周时间规划

七. 数学

1) 数论

a. 《质数判定》

题目链接

题目描述

输入描述

输出描述

题解代码

b. 《青蛙的约会》

题目链接

题目描述

输入描述

输出描述

题解代码

2) 组合数学

a. 《斐波那契数列》

题目链接

题目描述

输入描述

输出描述

题解代码

3) 其它

a. 《小明的游戏1》（博弈论）

题目链接

题目描述

输入描述

输出描述

题解代码

b. 《小明的游戏2》（博弈论）

题目链接

题目描述

输入描述

输出描述

题解代码

八. 字符串

1. 《扫雷游戏》

题目描述

输入描述

输出描述

输入输出样例

示例 1

示例 2

2. 《ISBN 号码》



运行限制
题目描述
输入描述
输出描述
输入输出样例
 示例 1
 示例 2
运行限制

九. 图论

1. 《蓝桥公园》
 题目链接
 题目描述
 输入描述
 输出描述
 题解代码
2. 《走多远》
 题目链接
 题目描述
 输入描述
 输出描述
 题解代码

十. 计算几何

1. 《三角形的面积》（计算几何基础）
 题目链接
 输入描述
 输出描述
 题解代码
2. 《奇偶覆盖》（扫描线）
 题目链接
 题目描述
 输入描述
 输出描述
 题目分析

什么是算法复杂度

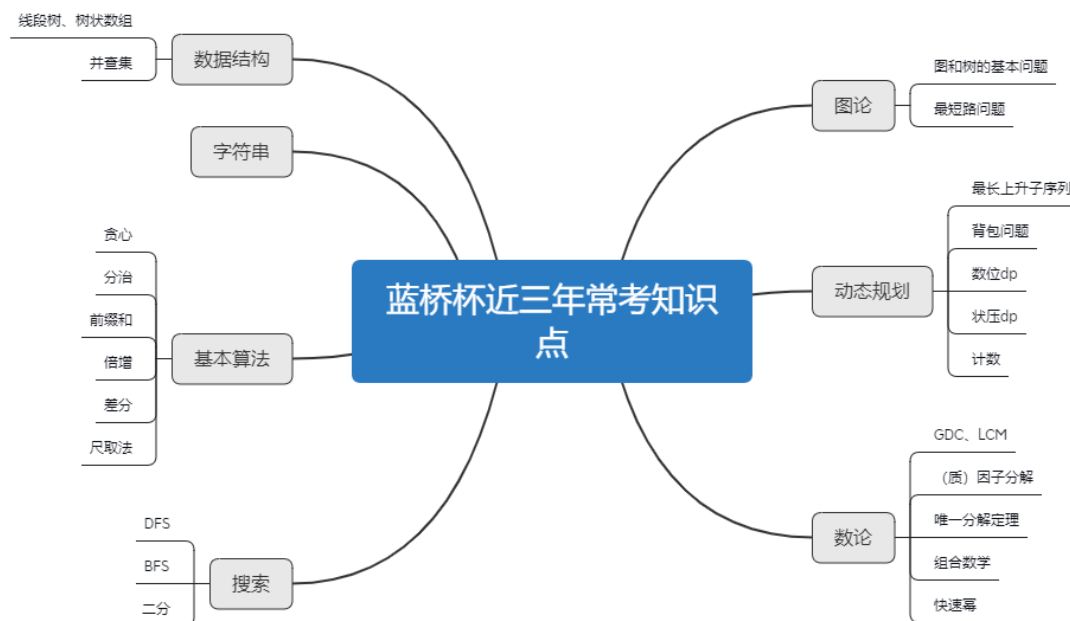
算法复杂度
 冒泡排序
 快速排序
 哈希算法
算法的选择
 什么是算法
 算法复杂度
学习建议



4月9日蓝桥杯省赛正式开赛！！！！



蓝桥杯近 3 年常考知识点



众所周知，算法竞赛以知识点的丰富和编码的高难度闻名。其实也不用太担心，算法竞赛的难度是分级、分阶段的，难题、罕见题并不多见。把学习重点放在基础算法、常见算法上，并用大量的实战练习提高编码能力，足以

在蓝桥杯省赛上获得一等奖。

算法竞赛的题目分为这几个大类：杂题、数据结构、基础算法、搜索、动态规划、数学、字符串、图论、几何等。

其中的「杂题」不需要什么算法和数据结构，只考察思维和编码能力，不过，杂题也可能很难。

蓝桥杯近 5 年真题——知识点对应

下面为大家盘点 2017~2021 年度蓝桥杯省赛的知识点。以 C++ A 组为例，其他组别的题目类似，重合的题目很多。

| 知识点 | 难度 | 题目 |
|--------|-----|---|
| 模拟、思维 | | 2017-10, 2018-10、2019-3、2019-7、2020-3、2020-6、2020-7 |
| 基本数据结构 | * | 二叉树 (2019-6) |
| 基础算法 | * | 枚举 (2018-5)、差分 (2018-7)、倍增 |
| | ** | 二分法 (2017-9) |
| 搜索 | * | DFS (2017-1、2017-4、2017-7)、BFS (2017-2、2018-8、2019-4) |
| 高级数据结构 | * | 并查集 (2019-8、2020-4)、线段树 |
| 动态规划 | * | 线性 DP (2017-5、2017-6、2017-8、2020-10)、记忆化搜索、搜索 (2021) |
| | ** | 状态压缩 DP (2019-9) |
| 简单数学 | | 2018-1, 2018-2, 2018-3、2018-4、2019-1、2019-2、2020-1、2020-5 |
| 数论 | * | 余数 (2018-9)、GCD (2017-8、2020-2) 枚举 (2021) |
| 组合数学 | ** | 卢卡斯定理 (2019-10) |
| | *** | burnside 引理 (2017-3) |
| 其他 | * | 快速幂 (2019-5) |
| 字符串 | * | 简单字符串处理 (2018-6、2019-1、2020-8) |
| 图论 | * | 最短路 BFS (2019-4) |
| 计算几何 | * | 叉积、面积 (2020-9) (2021) |



另外我也统计过前些年蓝桥杯题目，除了上述知识点外，还有一些一星或二星的，例如倍增、线段树等。从上面的统计表格可以看到，省赛涉及的知识点比较基础，考核的是基本的算法思维、基本算法、编码能力。要成功参赛，最重要的是通过大量做基础题目，培养计算思维，提高快速和准确的编码能力。

有一些**几乎是必考的**，因为它们也是整个算法竞赛知识库的基础：

1. 模拟题。不需要算法和数据结构，只需要逻辑、推理的题目，难度可难可易。考察思维能力和编码能力，只能通过大量做题来提高。
2. BFS 搜索和 DFS 搜索，也就是暴搜。这是基本的算法，是基础中的基础。
3. 动态规划，特别是比较简单的线性 DP。
4. 简单数学和简单数论。
5. 简单的字符串处理、输入输出。
6. 基本算法，例如二分法、倍增、差分。
7. 基本数据结构。队列、栈、链表等。

下面列出除「杂题」外几乎所有的算法竞赛知识点，并按知识点本身的难度分成一星、二星、三星。读者应努力

掌握一星、二星知识点。

注意知识点的难度和题目的难度并不一定完全对应，简单的知识点也可能出难题。

并且根据每一知识点配套了1-2道精选习题，下面我们就来根据知识点开启精练（下）吧。

省赛一等奖学霸的备赛小贴士

1. 大神备考心得

通过对蓝桥杯近三年真题的研究，我总结了以下几点极为关键的信息：

1. 蓝桥杯比赛是可以提前入场的。在这段时间内你可以将所需要的模板敲出来并保存。
2. 近年填空题的第一题常考一些不涉及算法的计算机基础问题（例如单位换算），赛前可抽空看看计算机相关基础问题。
3. 填空题不一定需要编程才能得到答案。有时候利用如 Excel、计算器、日历表等工具甚至是更优的选择。
4. 蓝桥杯的命题很有规律，总爱围绕一些知识标签、题型命题（如动态规划）。赛前花点时间整理这
5. 题目的难度不一定是逐级递增的，后面的题目也许比前面的题更好拿分。
6. 蓝桥杯类似 oi 制度，你在提交了程序之后是无法得知答案的正确与否。所以在提交之前，可以先手造几组测试样例对程序进行验证。
7. 压轴题的难度越来越高，如果不是为了冲击省排名，可以减少压轴题花费的时间来检查其它问题。

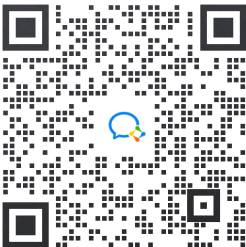
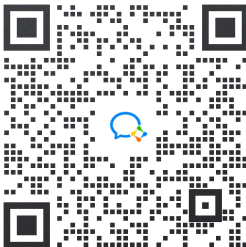
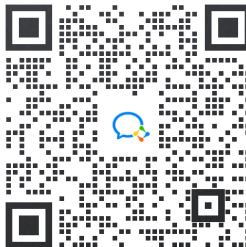
2. 最后一周时间规划



- 最后一周往往是最为关键的一周。如果此前你没有过任何参赛经验，那么我建议你：

1. 花一天时间去整理蓝桥杯的题型、考点、比赛时间、比赛规则等信息。

扫码添加 ▼ 蓝桥杯赛程答疑群

| Java组 | Python组 | C/C++组 |
|---|---|---|
|  |  |  |

2. 花一天时间翻读蓝桥杯真题。

近3年蓝桥杯真题解析（官方出品）

| 组别 | 真题解析链接 |
|-----------|---|
| Java-A组: | https://www.lanqiao.cn/courses/5031/?from=djbd321 |
| Java-B组: | https://www.lanqiao.cn/courses/5032/?from=djbd321 |
| Java-C组: | https://www.lanqiao.cn/courses/5033/?from=djbd321 |
| Python: | https://www.lanqiao.cn/courses/5034/?from=djbd321 |
| C/C++-A组: | https://www.lanqiao.cn/courses/5028/?from=djbd321 |
| C/C++-B组: | https://www.lanqiao.cn/courses/5029/?from=djbd321 |
| C/C++-C组: | https://www.lanqiao.cn/courses/5030/?from=djbd321 |

3. 克隆一场省赛，并花费四个小时模拟参赛。 [蓝桥杯省赛14天夺奖冲刺营](#)

4. 针对第一天整理出的题型、考点进行专题训练，具体如下：

- 对于每个考点，每天挑选两道经典问题刷。
- 在刷题结束后，对会做、不会做的题目都进行一波总结。

5. 调整好心态！

• 如果此前你有过参赛经验，那么我建议你：

1. 翻读历年真题。
2. 进行一些算法思想的训练，如动态规划、构造、贪心等。
3. 调整好心态！



----- 接上册 -----

七. 数学

1) 数论

| 难度 | 知识点 |
|----|---|
| * | GCD, LCM、素数判定、埃氏筛 |
| ** | 整数拆分、ExGCD、欧拉筛（线性筛）、威尔逊定理、原根、费马小定理、欧拉定理、欧拉函数、整除分块、同余，逆元、高斯消元、中国剩余定理、积性函数、莫比乌斯反演 |

a. 《质数判定》

题目链接

<https://www.lanqiao.cn/problems/1152/learning/>

题目描述

给定一个正整数 N ，请你判断它是否为质数。（若为质数输出 1，否则输出 0）

输入描述

第 1 行为一个整数 T ，表示测试数据数量。

接下来的 T 行每行包含一个正整数 N 。

$1 \leq T \leq 10^5$, $1 \leq N \leq 10^{16}$ 。

输出描述

输出共 T 行，每行包含一个整数，表示答案。

题解代码

费马定理：设 p 为素数， a 是整数，且 $\gcd(a, p) = 1$ ，则 $a^{p-1} \equiv 1 \pmod{p}$

二次探测定理：如果 p 是一个素数，且 $0 < x < p$ ，则方程 $x^2 \equiv 1 \pmod{p}$ 的解为 $x = 1, x = p - 1$ 。

Wilson定理：给定的正整数 n ， n 是素数的充要条件为 $(n-1)! \equiv -1 \pmod{n}$

素性测试(即测试给定的数是否为素数)是近代密码学中的一个非常重要的课题。虽然 Wilson 定理给出了一个数是素数的充要条件，但根据它来素性测试所需的计算量太大，无法实现对较大整数的测试。目前，尽管高效的确定性的素性算法尚未找到，但已有一些随机算法可用于素性测试及大整数的因数分解。下面描述的 Miller - Rabin 素性测试算法就是一个这样的算法。

假设 n 是奇素数，则 $n-1$ 为偶数。令 $n-1 = 2^q \cdot m$ 。

随机选择整数 a ($0 < a < n$)，由费马定理， $(a^{2^q \cdot m} = a^{n-1}) \equiv 1 \pmod{n}$ 。由二次探测定理可知： $a^{2^{q-1} \cdot m} \equiv 1 \pmod{n}$ 或者 $a^{2^{q-1} \cdot m} \equiv n-1 \pmod{n}$ ，若 $a^{2^{q-1} \cdot m} \equiv 1 \pmod{n}$ 成立，则再由二次定理可知： $a^{2^{q-2} \cdot m} \equiv 1 \pmod{n}$ 或者 $a^{2^{q-2} \cdot m} \equiv n-1 \pmod{n}$ ，...。如此反复运用二次探测定理，直到某一步 $a^m \equiv 1 \pmod{n}$ ，或者存在 $0 \leq r \leq q-1$ ，使 $a^{2^r \cdot m} \equiv n-1 \pmod{n}$ 。

所以该算法的过程为：

- 给定奇数 n ，为了判断是否为素数，首先测试 $a^{2^q \cdot m} \equiv n-1 \pmod{n}$ 是否成立。若不成立，则 n 一定为合数。否则再做进一步算法测试
- 考察下列 Miller 序列：
 $a^m \pmod{n}, a^{2m} \pmod{n}, a^{4m} \pmod{n}, \dots, a^{2^{q-1} \cdot m} \pmod{n}$

若 $a^m \equiv 1 \pmod{n}$ ，或者存在某个整数 $0 \leq r \leq q-1$ ，使 $a^{2^r \cdot m} \equiv n-1 \pmod{n}$ ，则称 n 通过 Miller 测试。

由上面的分析可知：素数一定通过 Miller 测试。所以，如果 n 不能通过 Miller 测试，则 n 一定是合数，否则 n 很可能是素数。这就是 Miller - Rabin 算法。

可以证明 Miller - Rabin 算法给出错误结果的概率小于 $\frac{1}{4}$ 。若反复测试 k 次，则错误概率降低至 $(\frac{1}{4})^k$ 。保守估计还是挺实用的。

```
#include <bits/stdc++.h>
```

```

#define ll long long
using namespace std;
ll mult(ll a, ll b, ll p)
{
    a %= p, b %= p;
    ll r = 0, v = a;
    while (b)
    {
        if (b & 1)
        {
            r += v;
            if(r > p) r -= p;
        }
        v <<= 1;
        if (v > p) v -= p;
        b >>= 1;
    }
    return r;
}
ll quick_pow(ll a, ll b, ll p)
{
    ll r = 1, v = a % p;
    while (b)
    {
        if (b & 1)
            r = mult(r, v, p);
        v = mult(v, v, p);
        b >>= 1;
    }
    return r;
}
bool CH(ll a, ll n, ll x, ll t)
{
    ll r = quick_pow(a, x, n);
    ll z = r;
    for (ll i = 1; i <= t; i++)
    {
        r = mult(r, r, n);
        if (r == 1 && z != 1 && z != n - 1)
            return true;
        z = r;
    }
    return r != 1;
}
bool Miller_Rabin(ll n)
{
    if (n < 2) return false;
    if (n == 2) return true;
    if (!(n & 1)) return false;
    ll x = n - 1, t = 0;
    while (!(x & 1))
    {
        x >>= 1;
        t++;
    }
    srand(time(NULL));
    ll o = 8;
    for (ll i = 0; i < o; i++)

```




```

    {
        ll a = rand() % (n - 1) + 1;
        if (CH(a, n, x, t)) return false;
    }
    return true;
}

signed main()
{
    int T = 1;
    cin >> T;
    while (T--)
    {
        ll n;
        cin >> n;
        if (Miller_Rabin(n)) cout << 1 << '\n';
        else cout << 0 << '\n';
    }
    return 0;
}

```

b. 《青蛙的约会》

题目链接

<https://www.lanqiao.cn/problems/1317/learning/>



题目描述

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

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

输入描述

输入仅一行包含 5 个整数 x, y, m, n, L 。

$0 \leq x \neq y < 2000000000$, $0 < m, n < 2000000000$, $0 < L < 2100000000$ 。

输出描述

输出碰面所需要的跳跃次数，如果永远不可能碰面则输出一行 `impossible`。

题解代码

exgcd大名扩展欧几里得算法，用来求形如 $\gcd(a, b) = ax + by$ 方程的通解。

前置知识：裴蜀定理

证明：设

$$ax_1 + by_1 = \gcd(a, b)$$

$$bx_2 + (a \bmod b)y_2 = \gcd(b, a \bmod b)$$

由欧几里得定理可知：

$$\gcd(a, b) = \gcd(a, a \bmod b)$$

所以

$$ax_1 + by_1 = bx_2 + (a \bmod b)y_2$$

又因为 $a \bmod b = a - \lfloor \frac{a}{b} \rfloor \times b$

所以

$$ax_1 + by_1 = bx_2 + (a - \lfloor \frac{a}{b} \rfloor \times b)y_2$$

$$ax_1 + by_1 = ay_2 + bx_2 - \lfloor \frac{a}{b} \rfloor \times by_2 = ay_2 + b(x_2 - \lfloor \frac{a}{b} \rfloor y_2)$$

因为 $a = a, b = b$ 所以 $x_1 = y_2, y_1 = x_2 - \lfloor \frac{a}{b} \rfloor y_2$ 。

将 x_2, y_2 不断带入递归求解，直至最大公约数为0，得到 $x = 1, y = 0$ 回溯求解。



```
#include<bits/stdc++.h>
#define ll long long
using namespace std;
ll x, y, m, n, l, ans, md;

ll gcd(ll a, ll b) {
    return b == 0 ? a : gcd(b, a % b);
}

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

ll a, b, c, d, xx, yy;
signed main(){
    cin >> x >> y >> m >> n >> l;
    a = m - n, c = y - x;
    if (a < 0) {
        a = -a; c = -c;
    }
    d = gcd(a, l);
    if (c % d == 0) {
        a /= d; c /= d; l /= d;
```

```

    exgcd(a, 1, xx, yy);
    cout << (((xx + 1) % 1) * c) % 1 + 1) % 1 << '\n';
} else cout << "impossible\n";
return 0;
}

```

2) 组合数学

| 难度 | 知识点 |
|-----|---|
| * | 排列、组合、二项式定理、鸽巢原理、常见恒等式、帕斯卡恒等式、容斥原理、错排问题、斐波那契数列； |
| | 递推方程：线性递推方程、非线性递推方程、求解递推方程(模板) |
| ** | 卢卡斯定理、catalan 数列、stirling 数列、普通母函数、指数母函数、Pólya 定理 |
| *** | Burnside 引理、L 级数、贝尔级数、狄利克雷级数 |

a. 《斐波那契数列》

show: step
version: 1.0
enable_checker: true



题目链接

<https://www.lanqiao.cn/problems/1180/learning>

题目描述

斐波那契数列：
$$\begin{cases} F(0) = F(1) = 1 \\ F(n) = F(n-1) + F(n-2) \quad n \geq 2, n \in \mathbb{N}^* \end{cases}$$
给定一个正整数 N ，求 $F(N)$ 在模 $10^9 + 7$ 下的乘法逆元。

输入描述

第 1 行为一个整数 T ，表示测试数据数量。

接下来的 T 行每行包含一个正整数 N 。

$1 \leq T \leq 10^4, 1 \leq N \leq 10^{18}$ 。

输出描述

输出共 T 行，每行包含一个整数，表示答案。

题解代码

这题直接递推复杂度是肯定会超时的，所以要用矩阵加速递推过程，这就要用到矩阵快速幂，那么什么是矩阵快速幂呢？

由于同阶矩阵是可以进行乘法运算的，那么矩阵快速幂的过程就和普通快速幂一样，只不过需要自己重新定义了乘法。那么矩阵是怎么加速的呢？

我们构造常系数矩阵

$$F_n = 1 \times F_{n-1} + 1 \times F_{n-2}$$

$$F_{n-1} = 1 \times F_{n-1} + 0 \times F_{n-2}$$

也就是下面这个矩阵：

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

我们把递推式子化成矩阵的形式：

$$\begin{bmatrix} F_2 \\ F_1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} F_3 \\ F_2 \end{bmatrix}$$

那么我们可以发现

$$\begin{bmatrix} F_2 \\ F_1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-2} = \begin{bmatrix} F_{n-2} \\ F_{n-1} \end{bmatrix}$$



那么我们只要对

$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-2}$ 这个矩阵用快速幂加速运算，矩阵乘法复杂度为 $O(n^3)$ ， n 为矩阵大小，那么总复杂度就变

成了 $O(2^3 \times \log n)$ ， n 再大也完全不是问题嘛！

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
const int mod = 1e9 + 7;
typedef vector<ll> vec;
typedef vector<vec> mat;
mat mul(mat &A, mat &B){
    mat C(A.size(), vec(B[0].size()));
    for(int i = 0; i < A.size(); i++){
        for(int k = 0; k < B.size(); k++){
            if(A[i][k]){
                for(int j = 0; j < B[0].size(); j++){
                    C[i][j] = (C[i][j] + A[i][k] * B[k][j] % mod + mod) % mod;
                }
            }
        }
    }
}
```

```

        return C;
    }
    mat Pow(mat A, ll n){
        mat B(A.size(), vec(A.size()));
        for(int i = 0; i < A.size(); i++) B[i][i] = 1;
        for(; n > 0; n >= 1, A = mul(A, A))
            if(n & 1) B = mul(B, A);
        return B;
    }

    int main()
    {
        std::ios::sync_with_stdio(false);
        int t;
        cin >> t;
        mat A(2, vec(2));
        while(t--){
            ll n;
            cin >> n;
            A[0][0] = A[0][1] = A[1][0] = 1; A[1][1] = 0;
            A = Pow(A, n - 1);
            cout << A[0][0] << endl;
        }
        return 0;
    }
}

```

3) 其它



| 难度 | 知识点 |
|-----|--|
| * | 高精度、快速幂、矩阵乘法 |
| | 概率与期望、博弈论（公平组合游戏，巴什游戏、P-position、N-position、尼姆游戏、威佐夫游戏）、Simpson 积分 |
| ** | 卢卡斯定理、catalan 数列、stirling 数列、普通母函数、指数母函数、Pólya 定理 |
| *** | 图游戏与 Sprague-grundy 函数、单纯形法解线性规划、快速傅里叶 (FFT) |

a. 《小明的游戏1》（博弈论）

show: step
version: 1.0
enable_checker: true

题目链接

<https://www.lanqiao.cn/problems/1218/learning>

题目描述

蓝桥公司给他们的员工准备了丰厚的奖金，公司主管小明并不希望发太多的奖金，他想把奖金留给智慧的人，于是他决定跟每一个员工玩一个游戏，规则如下：

- 桌面上一共有 n 堆一元钱。
- 双方轮流行动，由小明先行动，每次行动从某一堆钱中拿走若干元（至少一元钱），取走最后一元钱的人获胜。

请问员工们能拿到奖金吗？

输入描述

第一行为一个整数 T ，表示测试数据数量。

每个测试用例包含两行。第一行为一个整数 n ，第二行包括 n 个整数 a_1, a_2, \dots, a_n 表示第 i 堆有 a_i 元。

$$1 \leq T, n \leq 10^5, 1 \leq a_i \leq 10^9.$$

保证所有测试用例的 n 的和不超过 2×10^5 。

输出描述

如果员工能拿到奖金输出 YES，否则输出 NO。



题解代码

这题为 *nim* 博弈原型：

- n 堆石子，俩人轮流从某一堆中取若干个石子，至少取一个，最后一人取完获胜，问先手必胜还是必败？

做法：*nim*和为0 先手必胜。*nim*和为全部石子数量的异或值

证明：定义 P-position 和 N-position，其中 P 代表 Previous，N 代表 Next。直观的说，上一次 move 的人有必胜策略的局面是 P-position，也就是“后手可保证必胜”或者“先手必败”，轮到 move 的人有必胜策略的局面是 N-position，也就是“先手可保证必胜”。更严谨的定义是：

1. 无法进行任何移动的局面（也就是 terminal position）是 P-position；
2. 可以移动到 P-position 的局面是 N-position；
3. 所有移动都导致 N-position 的局面是 P-position。

我们尝试证明上面三个命题即可。

- 显然，terminal position 只有一个，就是全 0，异或仍然是 0。
- 对于某个局面 (a_1, a_2, \dots, a_n) ，若 $a_1 \oplus a_2 \oplus \dots \oplus a_n \neq 0$ ，一定存在某个合法的移动，将 a_i 改变成 a'_i 后满足 $a_1 \oplus a_2 \oplus \dots \oplus a_n = 0$ 。不妨设 $a_1 \oplus a_2 \oplus \dots \oplus a_n = k$ ，则一定存在某个 a_i ，它的二进制表示在 k 的最高位上是 1（否则 k 的最高位那个 1 是怎么得到的）。这时 $a_i \oplus k \leq a_i$ 一定成立。则我们可以将 a_i 改变成 $a'_i = a_i \oplus k$ ，此时 $a_1 \oplus a_2 \oplus \dots \oplus a_n \oplus k = 0$
- 对于某个局面 (a_1, a_2, \dots, a_n) ，若足 $a_1 \oplus a_2 \oplus \dots \oplus a_n = 0$ ，由于异或运算满足消去率，即 $a \oplus b \oplus b = a$ ，故不存在 a'_i 使得将 $a_i (a_i \neq 0)$ 改变成 a'_i 后满足

$$a_1 \oplus a_2 \oplus \dots \oplus a_n = 0$$

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn = 1e5 + 50;
int main()
{
    std::ios::sync_with_stdio(false);
    int t;
    cin >> t;
    while(t--){
        int n;
        cin >> n;
        int sum = 0;
        for(int i = 0; i < n; i++){
            int x; cin >> x;
            sum ^= x;
        }
        if(!sum) cout << "YES\n";
        else cout << "NO\n";
    }
    return 0;
}
```

b. 《小明的游戏2》（博弈论）



show: step
version: 1.0
enable_checker: true

题目链接

<https://www.lanqiao.cn/problems/1219/learning>

题目描述

蓝桥公司给他们的员工准备了丰厚的奖金，公司主管小明并不希望发太多的奖金，他想把奖金留给智慧的人，于是他决定跟每一个员工玩一个游戏，规则如下：

1. 桌面上一共有 n 堆一元钱
2. 双方轮流行动，由小明先行动，每次行动从某一堆钱中拿走若干元（至少一元钱），取走最后一元钱的人落败。

请问员工们能拿到奖金吗？

输入描述

第一行为一个整数 T ，表示测试数据数量。

每个测试用例包含两行。第一行为一个整数 n ，第二行包括 n 个整数 $a_1, a_2 \dots a_n$ 表示第 i 堆有 a_i 元。

$$(1 \leq T, n \leq 10^5, 1 \leq a_i \leq 10^9)$$

保证所有测试用例的 n 的和不超过 2×10^5 。

输出描述

如果员工能拿到奖金输出 YES，否则输出 NO。

题解代码

该题为经典反 *nim* 游戏 (*anti - nim* 游戏) 博弈

该题原型简介： n 堆石子，两人轮流从某一堆中取若干个石子，至少取一个，最后一人取完获胜。

做法：记 sum 为 *nim* 和，则两个必胜态的条件：

1. 石子个数全部为 1 且 $sum == 0$
2. 至少有一堆石子个数大于 1 且 $sum \neq 0$

证明：

- 1. 当所有堆的石子数均为 1 时：
 - 1.1 石子异或和为 0，即有偶数堆。此时显然先手必胜。
 - 1.2 异或和不为 0，奇数堆。此时显然先手必败。
- 2. 当有一堆的石子数 > 1 时，显然石子异或和不为 0
 - 一定存在操作方法使得通过对大于 1 的石子堆操作将局面转化为 1.2 先手必败，因此当前局面为先手必胜
- 3. 当有两堆及以上的石子数 > 1 时
 - 3.1 石子堆异或和不为 0，根据 *nim* 游戏的证明，可以得到总有一种方法转化 *nim* 和不为 0 状态，也就是转为 3.2 状态。
 - 3.2 石子堆异或和为 0，当石子堆 > 1 的堆数大于 2 时候可以转向 3.1 状态，否则则可以转向 2 状态(必胜态)。

观察 3 我们可以发现，3.2 状态起手可以转化成 3.1，而且 3.1 只能变成 3.2 或者先手必胜态，因此 3.2 为先手必胜状态。

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn = 1e5 + 50;
int main()
{
    std::ios::sync_with_stdio(false);
    int t;
    cin >> t;
    while(t--){
        int n;
        cin >> n;
```




```

int sum = 0;
int ok = 0;
for(int i = 0; i < n; i++){
    int x; cin >> x;
    if(x > 1) ok = 1;
    sum ^= x;
}
if(ok == 0){
    if(sum == 0) cout << "NO\n";
    else cout << "YES\n";
}
else{
    if(sum == 0) cout << "YES\n";
    else cout << "NO\n";
}
}
return 0;
}

```

八. 字符串

| 难度 | 知识点 |
|----|---|
| * | 字符串 hash、字典树 |
| ** | KMP、后缀树、后缀数组、Manacher 回文算法、AC 自动机、后缀自动机、回文自动机 |



1. 《扫雷游戏》

题目描述

扫雷游戏是一款十分经典的单机小游戏。在 nn 行 mm 列的雷区中有一些格子含有地雷（称之为地雷格），其他格子不含地雷（称之为非地雷格）。玩家翻开一个非地雷格时，该格将会出现一个数字-----提示周围格子中有多少个是地雷格。游戏的目标是在不翻出任何地雷格的条件下，找出所有的非地雷格。

现在给出 nn 行 mm 列的雷区中的地雷分布，要求计算出每个非地雷格周围的地雷格数。

注：一个格子的周围格子包括其上、下、左、右、左上、右上、左下、右下八个方向上与之直接相邻的格子。

输入描述

输入的第一行是用一个空格隔开的两个整数 nn 和 mm ，分别表示雷区的行数和列数。

接下来 nn 行，每行 mm 个字符，描述了雷区中的地雷分布情况。字符 '*' 表示相应格子是地雷格，字符 '?' 表示相应格子是非地雷格。相邻字符之间无分隔符。

其中， $1 \leq n \leq 100$ ， $1 \leq m \leq 100$ ， $1001 \leq n \leq 100$ ， $1 \leq m \leq 100$ 。

输出描述

输出包含 nn 行，每行 mm 个字符，描述整个雷区。用 '*' 表示地雷格，用周围的地雷个数表示非地雷格。相邻字符之间无分隔符。

输入输出样例

示例 1

输入

```
3 3
*??
???
?*?
```

输出

```
*10
221
1*1
```

示例 2

输入

```
2 3
?*?
*??
```

输出

```
2*1
*21
```



2. 《ISBN 号码》

运行限制

- 最大运行时间: 1s
- 最大运行内存: 256M

题目描述

每一本正式出版的图书都有一个 ISBN 号码与之对应，ISBN 码包括 9 位数字、1 位识别码和 3 位分隔符，其规定格式如 “x-xxx-xxxxx-x”，其中符号“-”是分隔符（键盘上的减号），最后一位是识别码，例如 0-670-82162-4 就是一个标准的 ISBN 码。ISBN 码的首位数字表示书籍的出版语言，例如 0 代表英语；第一个分隔符“-”之后的三位数字代表出版社，例如 670 代表维京出版社；第二个分隔之后的五位数字代表该书在出版社的编号；最后一位为识别码。

识别码的计算方法如下：

首位数字乘以 1 加上次位数字乘以 2 以此类推，用所得的结果 mod 11，所得的余数即为识别码，如果余数为 10，则识别码为大写字母 X。例如 ISBN 号码 0-670-82162-4 中的识别码 4 是这样得到的：对 067082162 这 9 个数字，从左至右，分别乘以 1, 2, ..., 9，再求和，即 $0 \times 1 + 6 \times 2 + \dots + 2 \times 9 = 158$ ，然后取 $158 \bmod 11$ 的结果 4 作为识别码。你的任务是编写程序判断输入的 ISBN 号码中识别码是否正确，如果正确，则仅输出 `Right`；如果错误，则输出你认为是正确的 ISBN 号码。

输入描述

输入一行，是一个字符序列，表示一本书的 ISBN 号码（保证输入符合 ISBN 号码的格式要求）。

输出描述

输出一行，假如输入的 ISBN 号码的识别码正确，那么输出 `Right`，否则，按照规定的格式，输出正确的 ISBN 号码（包括分隔符“-”）。

输入输出样例

示例 1

输入

0-670-82162-4

输出

Right



示例 2

输入

0-670-82162-0

输出

0-670-82162-4

运行限制

- 最大运行时间：1s
- 最大运行内存：128M

九. 图论

| 难度 | 知识点 |
|-----|---|
| * | 图的存储（矩阵、邻接表、链式前向星）、最短路（BFS） |
| ** | 最短路（dijkstra、bellman-ford、spfa、floyd）、最小生成树（kruskal、prim）、拓扑排序、二分图匹配、差分约束、无向图的连通性、有向图的连通性、强连通分量、割点、割边、缩点、桥、分数规划、2-SAT、树的直径和重心、LCA，树链剖分、树分块、虚树 |
| *** | 网络流（Dinic、Sap、ISAP） |

1. 《蓝桥公园》

show: step
version: 1.0
enable_checker: true

题目链接

<https://www.lanqiao.cn/problems/1121/learning/>

题目描述



小明喜欢观景，于是今天他来到了蓝桥公园。

已知公园有 N 个景点，景点和景点之间一共有 M 条道路。小明有 Q 个观景计划，每个计划包含一个起点 st 和一个终点 ed ，表示他想从 st 去到 ed 。但是小明的体力有限，对于每个计划他想走最少的路完成，你可以帮帮他吗？

输入描述

输入第一行包含三个正整数 N, M, Q

第 2 到 $M + 1$ 行每行包含三个正整数 u, v, w ，表示 $u \leftrightarrow v$ 之间存在一条距离为 w 的路。

第 $M + 2$ 到 $M + Q - 1$ 行每行包含两个正整数 st, ed ，其含义如题所述。

$1 \leq N \leq 400, 1 \leq M \leq \frac{N \times (N - 1)}{2}, Q \leq 10^3, 1 \leq u, v, st, ed \leq n, 1 \leq w \leq 10^9$

输出描述

输出共 Q 行，对应输入数据中的查询。

若无法从 st 到达 ed 则输出 -1 。

题解代码

如何求任意两点之间最短路径呢？通过之前的学习我们知道通过深度或广度优先搜索可以求出两点之间的最短路径。所以进行 n^2 遍深度或广度优先搜索，即对每两个点都进行一次深度或广度优先搜索，便可以求得任意两点之间的最短路径。可是还有没有别的方法呢？

*Floyd*算法（*Floyd – Warshall algorithm*）又称为弗洛伊德算法、插点法，是解决给定的加权图中顶点间的最短路径的一种算法，可以正确处理有向图或负权的最短路径问题，同时也被用于计算有向图的传递闭包。该算法名称以创始人之一、1978年图灵奖获得者、斯坦福大学计算机科学系教授罗伯特·弗洛伊德命名。

算法核心：任意节点 i 到 j 的最短路径两种可能：

1. 直接从 i 到 j ;
2. 从 i 经过若干个节点 k 到 j ;

所以我们可以枚举每一个经过的点,对于任意节点 i 和 节点 j , 从上面两种中选择一条最小的路径进行更新。

核心转移方程: $dis[i][j] = \min(dis[i][j], dis[i][k] + dis[k][j]);$

优点：容易理解，可以算出任意两个节点之间的最短距离，代码编写简单

缺点：时间复杂度比较高，不适合计算大量数据。

时间复杂度: $O(n^3)$;

空间复杂度: $O(n^2)$;

```
#include<bits/stdc++.h>
#define int long long
using namespace std;
const int N = 5e2 + 10;
int dis[N][N];
void Floyd(int n)
{
    for(int k = 1 ; k <= n ; k ++ )
    {
        for(int i = 1 ; i <= n ; i ++ )
        {
            for(int j = 1 ; j <= n ; j ++ )
                dis[i][j] = min(dis[i][j] , dis[i][k] + dis[k][j]);
        }
    }
}
signed main()
{
    int n , m , q , inf = 1e15;
    cin >> n >> m >> q;
    memset(dis , 0x3f , sizeof(dis));
    for(int i = 1 ; i <= m ; i ++ )
    {
        int u , v , w;
        cin >> u >> v >> w;
        dis[u][v] = dis[v][u] = min(dis[u][v] , w);
    }
    for(int i = 1 ; i <= n ; i ++ ) dis[i][i] = 0;
```



```

Floyd(n);
while(q --)
{
    int u , v;
    cin >> u >> v;
    if(dis[u][v] > inf) cout << -1 << '\n';
    else cout << dis[u][v] << '\n';
}
return 0;
}

```

2. 《走多远》

show: step
version: 1.0
enable_checker: true

题目链接

<https://www.lanqiao.cn/problems/1337/learning/>

题目描述

给定一个 n 个点, m 条边的有向无环图, 小明从入度为 0 点出发, 顺着边最远能走多远, 若不存在这样的点, 输出 0。

输入描述

第一行输入一个 n, m 。

接下来 m 行, 每行输入两个整数 u, v 代表有一条有向边从 u 到 v 。

$1 \leq n, m \leq 10^6, 1 \leq u, v \leq n$

输出描述

输出一个整数表示最长距离。

题解代码

拓扑排序要解决的问题是给一个图的所有节点排序。

拓扑排序的过程:

- 找出图中 0 入度的顶点;
- 依次在图中删除这些顶点, 删除后再找出 0 入度的顶点;
- 重复上述过程, 直至删除所有顶点, 即完成拓扑排序, 删除的顺序就是拓扑序。

本题我们可以按拓扑序依次进入队列, 初始距离都为 0, 不断更新每个节点的最大距离, 代码的核心是维持一个入度为 0 的顶点的集合。因为题目保证是有向无环图, 所以一定能进行拓扑排序。

```
#include<bits/stdc++.h>
```

```

using namespace std;
const int maxn = 1e5 + 50;
const int INF = 0x3f3f3f3f;
vector<int> G[maxn];
int in[maxn];
int d[maxn];
int main()
{
    std::ios::sync_with_stdio(false);
    int n, m;
    cin >> n >> m;
    while(m--){
        int u, v;
        cin >> u >> v;
        in[v]++;
        G[u].push_back(v);
    }
    int ans = 0;
    queue<int> q;
    for(int i = 1; i <= n; i++){
        if(!in[i]) q.push(i);
    }
    while(!q.empty()){
        int now = q.front(); q.pop();
        for(auto i : G[now]){
            in[i]--;
            d[i] = max(d[i], d[now] + 1);
            ans = max(ans, d[i]);
            if(!in[i]) q.push(i);
        }
    }
    cout << ans << endl;
    return 0;
}

```



十. 计算几何

| 难度 | 知识点 |
|----|------------------------------------|
| * | 点积、叉积、点、线、面、二维几何、三维几何、面积、体积 |
| ** | 凸包、最近点对、半平面交、旋转卡壳、三角剖分、最小圆覆盖、最小球覆盖 |

1. 《三角形的面积》（计算几何基础）

show: step
version: 1.0
enable_checker: true

题目链接

<https://www.lanqiao.cn/problems/1231/learning>

##题目描述

平面直角坐标系中有一个三角形，请你求出它的面积。

输入描述

第一行输入一个 T ,代表测试数据量

每组测试数据输入有三行，每行一个实数坐标 (x, y) 代表三角形三个顶点。

$$1 \leq T \leq 10^3, -10^5 \leq x, y \leq 10^5$$

输出描述

输出一个实数表示三角形面积。结果保留2位小数，误差不超过 10^{-2}

题解代码

根据三角形面积公式：

$$S_{ABC} = \frac{1}{2} \left| \overrightarrow{AB} \right| \left| \overrightarrow{AC} \right| \sin A = \frac{1}{2} \left| \overrightarrow{AB} \times \overrightarrow{AC} \right|$$

其中乘号代表向量的叉积，那么在给出三角形三个点坐标的时候，我们就可以直接利用叉积来计算三角形的面积，相对利用海伦公式求解，结果更加精确。



```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn = 1e5 + 50;
struct Point{
    double x, y;
    void input(){
        scanf("%lf%lf", &x, &y);
    }
    Point(){}
    Point(double _x, double _y){x = _x, y = _y;}
    Point operator - (const Point &b) const{
        return Point(x - b.x, y - b.y);
    }
    //叉积
    double operator ^(const Point &b){
        return x * b.y - y * b.x;
    }
};
double get_area(Point a, Point b, Point c){
    Point A = a - b;
    Point B = a - c;
    return fabs(A ^ B) * 0.5;
}
int main()
```



```
{
    int t;
    cin >> t;
    while(t--){
        Point a, b, c;
        a.input(); b.input(); c.input();
        double ans = get_area(a, b, c);
        printf("%.2f\n", ans);
    }
    return 0;
}
```

2. 《奇偶覆盖》（扫描线）

show: step
version: 1.0
enable_checker: true

题目链接

<https://www.lanqiao.cn/problems/1040/learning/>

题目描述



在平面内有一些矩形，它们的两条边都平行于坐标轴。我们称一个点被某个矩形覆盖，是指这个点在矩形的内部或者边界上。

请问，被奇数个矩形覆盖和被偶数 (≥ 2) 个矩形覆盖的点的面积分别是多少？

输入描述

输入的第一行包含一个整数 n ，表示矩形的个数。

接下来 n 行描述这些矩形，其中第 i 行包含四个整数 l_i, b_i, r_i, t_i ，表示矩形的两个对角坐标分别为 $(l_i, b_i), (r_i, t_i)$ 。

其中， $1 \leq n \leq 10^5, 0 \leq l_i < r_i \leq 10^9, 0 \leq b_i < t_i \leq 10^9$ 。

输出描述

输出两行。

第一行包含一个整数，表示被奇数个矩形覆盖的点的面积。

第二行包含一个整数，表示被偶数 (≥ 2) 个矩形覆盖的点的面积。

题目分析

一般要求平面内矩形的面积交、面积并，采用扫描线的思想和线段树进行优化。

1. 从扫描线入手：假设令扫描线与 y 轴平行并且从左往右遍历与 y 轴平行的线段，那么我们应该如何计算答案？

在扫描线计算矩形面积时，根据已经遍历过的线段覆盖区间，然后与俩线段 x 坐标的差值相乘计算答案。不难看出每次遍历计算面积时，覆盖次数的奇偶性决定这一段面积是被奇数个矩形覆盖还是偶数个。

设 $len1$ 为遍历过的线段覆盖次数为奇数的区间总长度， $len2$ 为遍历过的线段覆盖次数为偶数的区间总长度。那么从第二次遍历开始对奇偶覆盖面积进行累加
($L[i]$ 从左到右第 i 条是平行 y 轴的线段)

```
odd += 1LL * (L[i].x - L[i - 1].x) * len1[1];
even += 1LL * (L[i].x - L[i - 1].x) * len2[1];
```

2. 考虑每次遍历 $len1$ 、 $len2$ 怎么维护，那当然是线段树啦。还是老套路，将一个区间左闭右开，就可以当做点进行维护。

考虑两个区间怎么进行合并：

第一种：覆盖次数为0的，如果是线段树的根节点这将其置0，代表这段区间没有覆盖，否则就左右子区间相加即可。

第二种：覆盖次数为奇数，这一段区间将对 $len1$ 进行贡献 所以加上 $v[r] - v[l - 1]$ (v 是 y 坐标离散化后的数组) 减去 $len2$ 占去的区间，奇数加奇数等于偶数所以 $len2$ 的区间长度可以由 $len1$ 的左右子树合并而来，覆盖次数为偶数的同理。



```
#include<bits/stdc++.h>
#define ls rt << 1
#define rs rt << 1 | 1
#define lson l, mid, rt << 1
#define rson mid + 1, r, rt << 1 | 1
#define lr2 (l + r) >> 1
using namespace std;
typedef long long ll;
const int maxn = 1e6 + 50;
vector<ll> v;
int getid(int x){
    return lower_bound(v.begin(), v.end(), x) - v.begin() + 1;
}
ll len1[maxn << 2], len2[maxn << 2], lazy[maxn << 2];
void push_up(int rt, int l, int r){
    if(lazy[rt]){
        //cout << l << " " << r << " " << v[l] << " " << v[r] << endl;
        if(lazy[rt] & 1){
            len2[rt] = len1[ls] + len1[rs];
            len1[rt] = v[r] - v[l - 1] - len2[rt];
        }
        else{
            len1[rt] = len1[ls] + len1[rs];
            len2[rt] = v[r] - v[l - 1] - len1[rt];
        }
    }
    else if(l == r){
        len1[rt] = len2[rt] = 0;
    }
}
```

```

    }
    else{
        len1[rt] = len1[ls] + len1[rs];
        len2[rt] = len2[ls] + len2[rs];
    }
}

void update(int a, int b, int k, int l, int r, int rt){
    if(a <= l && b >= r){
        lazy[rt] += k;
        push_up(rt, l, r);
        return ;
    }
    int mid = lr2;
    if(a <= mid) update(a, b, k, lson);
    if(b > mid) update(a, b, k, rson);
    push_up(rt, l, r);
}

struct node{
    int x, l, r, val;
    node(){ }
    node(int _x, int _l, int _r, int _val){l = _l, x = _x, r = _r, val = _val;}
    bool operator <(const node & p)const{
        if(x != p.x) return x < p.x;
        else return val > p.val;
    }
}L[maxn << 2];

int main()
{
    std::ios::sync_with_stdio(false);
    int n;
    cin >> n;
    ll odd = 0, even = 0;
    int cnt = 0;
    for(int i = 1; i <= n; i++){
        ll a, b, c, d;
        cin >> a >> b >> c >> d;
        L[cnt++] = node(a, b, d, 1);
        L[cnt++] = node(c, b, d, -1);
        v.push_back(b), v.push_back(d);
    }
    //离散化y坐标
    sort(v.begin(), v.end());
    v.erase(unique(v.begin(), v.end()), v.end());
    int len = v.size();
    sort(L, L + cnt);
    for(int i = 0; i < cnt; i++){
        if(i){
            odd += 1LL * (L[i].x - L[i - 1].x) * len1[1];
            even += 1LL * (L[i].x - L[i - 1].x) * len2[1];
        }
        int l = getid(L[i].l), r = getid(L[i].r);
        update(l, r - 1, L[i].val, 1, len - 1, 1);
        //cout << len1[1] << " " << len2[1] << "\n";
    }
    cout << odd << "\n" << even << "\n";
    return 0;
}

```

什么是算法复杂度

算法复杂度

本课程是讲算法竞赛的知识点，在后面展开算法知识点的讲解之前，需要先了解：

- 1. 什么是算法？
- 2. 算法复杂度是什么，为什么很重要？

知识点

- 时间复杂度
- 空间复杂度

计算资源

计算机软件运行需要的资源有两种：计算时间和存储空间。资源是有限的，一个算法对这两个资源的使用程度，可以用来衡量该算法的优劣。

■时间复杂度：程序运行需要的时间。

■空间复杂度：程序运行需要的存储空间。

与这两个复杂度对应，算法题目一般会有对运行时间和空间的说明，例如：

- Time Limit: 2000/1000 MS (Java/Others)
- Memory Limit: 65536/65536 K (Java/Others)



其中：

- Time Limit 是对程序运行时间的限制，这个题目要求在 2s(Java)/1s (C, C++)内结束。
- Memory Limit 是对程序使用内存的限制，这里是 65536 K，即 64M。

这 2 个限制条件非常重要，是检验程序性能的参数。

所以，队员拿到题目后，第一步要分析的是：程序运行需要的计算时间和存储空间。

编程竞赛的题目，在逻辑、数学、算法上有不同的难度：简单的，可以一眼看懂；复杂的，往往需要很多步骤才能找到解决方案。它们对程序性能考核的要求是：程序必须在限定的时间和空间内运行结束。

这是因为，问题的「有效」解决，不仅在于能否得到正确答案，更重要的是能在合理的时间和空间内给出答案。

李开复在「算法的力量」一文中写到：

1988 年，贝尔实验室副总裁亲自来访问我的学校，目的就是为了解为什么他们的语音识别系统比我开发的慢几十倍，而且，在扩大至大词汇系统后，速度差异更有几百倍之多。在与他们探讨的过程中，我惊讶地

发现一个 $O(nm)$ 的动态规划居然被他们做成了 $O(n^2m)$ 。

贝尔实验室的研究员当然绝顶聪明，但他们全都是学数学、物理或电机出身，从未学过计算机科学或算法，才犯了这么基本的错误。

上文提到的 $O(nm)$ 和 $O(n^2m)$ 就是时间复杂度。符号 ' O ' 表示复杂度， $O(nm)$ 可以粗略地理解为运行次数是 $n \times m$ 。 $O(n^2m)$ 比 $O(nm)$ 运行时间差不多大 n 倍。

在上面这个语音识别的例子中，设 $n = 100$ ，如果李开复的识别系统运行时间是 1s，那么贝尔实验室的系统需要 100s。显然，一个长达 100s 才能得到结果的语音识别系统，肯定是不实用的。李开复的这个例子，生动地说明了

「好」算法的属性：有合理的时间效率。

那么如何衡量程序运行的时间效率？测量程序在计算机上运行的时间，可以得到一个直观的认识。

下面的代码只有一个 for 语句，它对 k 进行累加，循环次数是 n 。用 `clock()` 函数统计 for 循环的运行时间。

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int i, k, n = 1e8;
    clock_t start, end;
    start = clock();
    for(i = 0; i < n; i++)    k++;    //循环次数
    end = clock();
    cout << (double)(end - start) / CLOCKS_PER_SEC << endl;
}
```

在一台普通配置的电脑上运行，例如 CPU 为 i5-8250U、内存 8G、64 位操作系统，结果如下：

- 当 $n = 1e8 = 10^8$ 时，输出的运行时间是 **0.164s**。
- 当 $n = 1e9$ 时，输出的运行时间是 **1.645s**。



评测用的 OJ 服务器，性能可能比这个好一些，也可能差不多。

$n \leq$ 所以，如果题目要求「Time Limit: 2000/1000 MS (Java/Others)」，那么内部的循环次数应该满足 10^8 ，即 1 亿次以内。

由于程序的运行时间依赖于计算机的性能，不同的机器结果不同。所以直接把运行时间作为判断标准，并不准确。

用程序执行的「次数」来衡量更加合理，例如上述的程序，循环了 n 次，把它的运行效率记为 $O(n)$ 。

竞赛所给的题目，一般都会有多种解法，它考核的是在限定时间和空间内解决问题。如果条件很宽松，那么可以在多种解法中选一个容易编程的算法；如果给定的条件很苛刻，那么，能选用的合适算法就不多了。

下面用一个例子说明，对同样的问题，如何选用不同的解法。

题目地址：<http://acm.hdu.edu.cn/showproblem.php?pid=1425>

Time Limit: 6000/1000MS (Java/Others) Memory Limit: 64M/32M (Java/Others)

给你 n 个整数，请按从大到小的顺序输出其中前 m 大的数。

输入：每组测试数据有两行，第一行有两个数 n, m ($0 < n, m < 1000000$)，第二行包含 n 个各不相同，且都处于区间 $[-500000, 500000]$ 的整数。

输出：对每组测试数据按从大到小的顺序输出前 m 大的数。

Sample Input

5 3

3 -35 92 213 -644

Sample Output

213 92 3

这一题的思路是：先对 100 万个数排序，然后输出前 m 大的数。题目给出的代码运行时间，非 Java 语言的时间是

1s，内存是 32M。

下面分别用冒泡、快速排序、哈希三种算法编程。

冒泡排序

首先，用最简单的冒泡排序算法求解上面的问题。

```
#include<bits/stdc++.h>
using namespace std;
int a[1000001]; //记录数字
#define swap(a, b) {int temp = a; a = b; b = temp;} //交换
int n, m;
void bubble_sort() { //冒泡排序，结果仍放在a[]中
    for(int i = 1; i <= n-1; i++)
        for(int j = 1; j <= n-i; j++)
            if(a[j] > a[j+1])
                swap(a[j], a[j+1]);
}
int main() {
    while(~scanf("%d%d", &n, &m)) {
        for(int i=1; i<=n; i++) scanf("%d", &a[i]);
        bubble_sort();
        for(int i = n; i >= n-m+1; i--) { //打印前m大的数，反序打印
            if(i == n-m+1) printf("%d\n", a[i]);
            else printf("%d ", a[i]);
        }
        return 0;
    }
}
```



`bubble_sort()` 运行后，得到从小到大的排序结果，然后从后往前打印前 m 大的数。冒泡排序算法的步骤是：

1. 第一轮，从第 1 个数到第 n 个数，逐个对比每 2 个相邻的数 a, b ，如果 $a > b$ ，则交换。这一轮的结果，把最大的数「冒泡」到了第 n 个位置，后面不用再管它。

2. 第二轮，从第 1 个数到第 $n-1$ 个数，对比每 2 个相邻的数。这一轮，把第 2 大的数「冒泡」到了第 $n-1$ 个位置。
3. 继续以上过程，直到结束。

下面分析程序的时间和空间效率。

1. 时间复杂度。也就是程序执行了多少步骤？花了多少时间？

在 `bubble_sort()` 中有 **2** 层循环，循环次数是： $n - 1 + n - 2 + \dots + 1 \approx n^2/2$ ；在 `swap(a, b)` 中做了 **3** 次操作；总的计算次数是 $3n^2/2$ ，复杂度记为 $O(n^2)$ 。 $n=100$ 万时，计算超过 **1** 万亿次。如果提交到 **OJ**，由于 **OJ** 每秒只能运行 **1** 亿次，必然返回 **TLE** 超时。可以推论出，只有 $n < 1$ 万时，才勉强能用冒泡算法。

2. 空间复杂度，也就是程序占用的内存空间。程序使用 `int a[1000001]` 存储数据，`bubble_sort()` 也没有使用额外的空间。`int` 是 **32** 位整数，占用 **4** 个字节，所以 `int a[1000001]` 共使用了 **4M** 空间。这是冒泡算法的优点，它不额外占用空间。

```
- name: 检测文件是否存在
script: |
    ls /home/project/bubble_sort.cpp
error: /home/project 路径下不存在 bubble_sort.cpp
timeout: 2
```

快速排序

快速排序是一种基于分治法的优秀排序算法。这里先直接用 **STL** 的 `sort()` 函数，它是改良版的快速排序，称为「内省式排序」。

在上面的程序中，把 `bubble_sort()` ；改为：

```
sort(a + 1, a + n + 1);
```

就完成了 `a[1]` 到 `a[n]` 的排序，结果仍然存在 `a[]` 中。

算法的时间复杂度是 $O(n \log n)$ ，当 $n = 100$ 万时， $100 \text{ 万} \times \log 2 100 \text{ 万} \approx 2000 \text{ 万}$ 。

在 **hdu** 上提交，返回的运行时间是 **600MS**，正好通过 **OJ** 的测试。



哈希算法

哈希算法是一种以空间换取时间的算法。本题的哈希思路是：在输入一个数字 t 的时候，对应 `a[500000 + t]` 这个位置，记录 `a[500000 + t] = 1`；输出的时候，逐个检查 `a[i]`，如果 `a[i]` 等于 1，表示这个数存在，打印出前 m 个。程序如下：

```

#include<bits/stdc++.h>
using namespace std;
const int MAXN = 1000001;
int a[MAXN];

int main(){
    int n,m;
    while(~scanf("%d%d", &n, &m)){
        memset(a, 0, sizeof(a));
        for(int i=0; i<n; i++){
            int t;
            scanf("%d", &t); //此题数据多，如果用很慢的cin输入，肯定TLE
            a[500000+t]=1;    //数字t，登记在500000+t这个位置
        }
        for(int i=MAXN; m>0; i--){
            if(a[i]){
                if(m>1) printf("%d ", i-500000);
                else    printf("%d\n", i-500000);
                m--;
            }
        }
    }
}

```

程序并没有做显式的排序，只是每次把输入的数按哈希插入到对应位置，只有 1 次操作；n 个数输入完毕，就相当于排好了序。总的时间复杂度是 $O(n)$ 。在 hdu 上提交，返回的运行时间是 500MS。

算法的选择



从上述 3 种程序可知，对同一个问题，常常存在不同的解决方案，有高效的，也有低效的。算法编程竞赛，主要的考核点，就是在限定的时间和空间内解决问题。虽然在大部分情况下，只有高效的算法才能通过满足判题系统的要求，但是请注意，并不是只有高效算法才是合理的，低效的算法有时也是有用的。对于程序设计竞赛来说，由于竞赛时间极为紧张，解题速度极为关键，只有尽快完成更多的题目，才能获得胜利。在满足限定条件的前提下，用最短的时间完成编码任务，才是最重要的。而低效算法的编码时间往往大大低于高效算法。例如，题目限定时间是 1s，现在有 2 个方案：（1）高效算法 0.01s 运行结束，但是代码有 50 行，编程 40 分钟；（2）低效算法 1s 运行结束，但是代码只有 20 行，编程 10 分钟。显然，此时应该选择低效算法。不过在竞赛时，这种情况通常只发生在数据规模小的简单题中，即所谓的「签到题」，而大部分题目是没有这种好事的。所以，这只是一个小小的技巧，并没有太大用处。

什么是算法

我们常常听说：「程序 = 算法 + 数据结构」，算法是解决问题的逻辑、方法、过程，数据结构是数据在计算机中的存储和访问的方式，两者是紧密结合的。

算法 (Algorithm)：对特定问题求解步骤的一种描述，是指令的有限序列。它有 5 个特征：

1. 输入：一个算法有零个或多个输入。可以没有输入，例如一个定时闹钟程序，它不需要输入，但是能够每隔一段时间就输出一个报警。
2. 输出：一个算法有一个或多个输出。程序可以没有输入，但是一定要有输出。
3. 有穷性：一个算法必须在执行有穷步之后结束，且每一步都在有穷时间内完成。
4. 确定性：算法中的每一条指令必须有确切的含义，对于相同的输入只能得到相同的输出。
5. 可行性：算法描述的操作可以通过已经实现的基本操作执行有限次来实现。

以冒泡排序算法为例子，上一节已经描述过它的执行步骤，它满足上述 5 个特征：

1. 输入：由 n 个数构成的序列 $\{a_1, a_2, a_3, \dots, a_n\}$;
2. 输出：对输入的一个排序 $\{a'_1, a'_2, a'_3, \dots, a'_n\}$, 且 $a'_1 \leq a'_2 \leq a'_3 \leq \dots \leq a'_n$;
3. $O(n^2)$ 有穷性：算法在执行 次后结束，这也是对算法性能的评估，即算法复杂度；
4. 确定性：算法的每个步骤都是确定的；
5. 可行性：算法的步骤能编程实现。

需要指出的是，上述第(5)条可行性，也是很重要的。有些算法并不能编程实现。例如一个有趣的排序算法，珠排序

$O(1)O(\sqrt{n})$ (Bead sort)，如果它用重力法，能够在 或 时间内得到排序结果，效率高到令人惊叹，但是无法编程实现。

算法复杂度

衡量算法性能的主要标准是时间复杂度，本节再针对算法竞赛展开说明。

为什么一般不讨论空间复杂度呢？一般情况下，一个程序的空间复杂度是容易分析的，而时间复杂度往往关系到算法的根本逻辑，更能说明一个程序的优劣。因此，如果不特别说明，提到「复杂度」时，一般指时间复杂度。

请注意，时间复杂度只是一个估计，并不需要精确计算。例如，在一个有 n 个数的无序数列中，查找某个数 x ，可能第一个数就是 x ，也可能最后一个数才是 x ，平均查找时间是 $n/2$ 次；但是把查找的时间复杂度记为 $O(n)$ ，而不是 $O(n/2)$ 。再例如，冒泡排序算法的计算次数约等于 $n^2/2$ 次，但是仍记为 $O(n^2)$ ，而不是 $O(n^2/2)$ 。在算法分析中，规模 n 前面的常数系数被认为是不重要的。还有，OJ 系统所判定的运行时间，是整个程序运行所花的时间，而不是理论上算法所需要的时间。同一个算法，不同的人写出的程序，复杂度和运行时间可能差别很大，跟编程语言、逻辑结构、库函数等等都有关系。

一个程序或算法的复杂度有以下可能：



$O(1)$

计算时间是一个常数，和问题的规模 n 无关。例如：用公式计算时，一次计算的复杂度就是 $O(1)$ ；哈希算法，用 hash 函数在常数时间内计算出存储位置；在矩阵 $A[M][n]$ 中查找 i 行 j 列的元素，只需要一次访问 $A[i][j]$ 就够了。

$O(\log n)$

计算时间是对数，通常是以 2 为底的对数，每一步计算后，问题的规模减小一倍。例如在一个长度为 n 的有序数列中查找某个数，用折半查找的方法，只需要 $\log n$ 次就能找到。再例如分治法，一般情况下，在每个步骤把规模减少一倍，所以一共有 $O(\log n)$ 个步骤。

$O(\log n)$ 和 $O(1)$ 没有太大差别。

$O(n)$

计算时间随规模 n 线性增长。在很多情况下，这是算法可能达到的最优复杂度，因为对输入的 n 个数，程序一般需要处理所有的数，即计算 n 次。例如查找一个无序数列中的某个数，可能需要检查所有的数。再例如图问题，有 V 个点和 E 个边，大多数图的问题，都需要搜索到所有的点和边，复杂度的上限就是 $O(V + E)$ 。

$O(n \log n)$

这常常是算法能达到的最优复杂度。例如分治法，一共 $O(\log n)$ 个步骤，每个步骤对每个数操作一次，所以总复杂度是 $O(n \log n)$ 。用分治法思想实现的快速排序算法和归并排序算法，复杂度就是 $O(n \log n)$ 。

$O(n^2)$

一个两重循环的算法，复杂度是 $O(n^2)$ 。例如冒泡排序，是典型的两重循环。类似的复杂度有 $O(n^3)$ 、 $O(n^4)$ 等等。

$O(2^n)$

一般对应集合问题，例如一个集合中有 n 个数，要求输出它的所有子集，子集有 2^n 个。

$O(n!)$

在集合问题中，如果要求按顺序输出所有的子集，那么复杂度就是 $O(n!)$

把上面的复杂度分成两类：（1）多项式复杂度，包括 $O(1)$ 、 $O(n)$ 、 $O(n\log n)$ 、 $O(nk)$ 等，其中 k 是一个常数；（2）指数复杂度，包括 $O(2^n)$ 、 $O(n!)$ 等。

如果一个算法是多项式复杂度，称它为「高效」算法。如果是指数复杂度，则是一个「低效」算法。可以这样通俗地解释「高效」和「低效」算法的区别：多项式复杂度的算法，随着规模 n 的增加，可以通过堆叠硬件来实现，「砸钱」是行得通的；而指数复杂度的算法，增加硬件也无济于事，其增长的速度超出了想象力。

竞赛题目一般的限制时间是 **1s**，对应普通计算机计算速度是每秒千万次级。那么，上述的时间复杂度，可以换算出能解决问题的数据规模。例如，如果一个算法的复杂度是 $O(n!)$ ，当 $n = 11$ 时， **$11! = 39916800$** ，这个算法只能解决 $n \leq 11$ 以内的问题。

下面这个表需要牢记：

| 问题规模 n | $O(\log n)$ | $O(n)$ | $O(n\log n)$ | $O(n^2)$ | $O(2^n)$ | $O(n!)$ |
|---------------|-------------|--------|--------------|----------|----------|---------|
| $n \leq 11$ | √ | √ | √ | √ | √ | √ |
| $n \leq 25$ | √ | √ | √ | √ | √ | × |
| $n \leq 5000$ | √ | √ | √ | √ | × | × |
| $n \leq 10^6$ | √ | √ | √ | × | × | × |
| $n \leq 10^7$ | √ | √ | × | × | × | × |
| $n > 10^8$ | √ | × | × | × | × | × |

学习建议

本讲啰嗦了这么多，读者如果能赏脸读到这里，估计也已经疲惫不堪、耳朵起茧，不想再听任何话。好吧不说了。还是说一句：学算法竞赛，最重要的是「刷题」！

再说一句：有初学者是零基础，刚开始学语言，例如 c 或者 java，要等学完之后再开始学算法竞赛。不要等！因为算法竞赛的编码用不着什么复杂语法，而且也能把竞赛题当成编程语言的练习题。

再再说一句：找同学一起学，不要自己一个人搞。有难度的学习，需要互相鼓励一起进步。找个异性同学更好！

再再再说一句：自主学习！最大的动力是自己！