ACM 模板

摘要

ACM-模板

刘泽辰

luzechen.coder@qq.com

目录

数学 3
质数 3
试除法判断质数 3
分解质因数 3
线性筛 4
约数 4
试除法求约数 4
最大公约数 5
欧拉函数 5
欧拉函数 5
欧拉筛求欧拉函数 5
逆元 6
快速幂求逆元 6
扩展欧几里得算法求逆元 6
组合数 7
求组合数 1 7
求组合数 2 (用逆元求) 7
卢卡斯定理 8
自动取模类 9
分数类 11
矩阵快速幂 12
欧拉函数 14
FFT 14
欧拉筛求积性函数 15
高斯消元 16
字符串 18
KMP 18
求 next 数组 18
KMP 匹配 19
求最小循环节 19

Trie 树 19
Manachar 算法(求最长回文串长度) 20
字符串哈希 21
AC 自动机 22
后缀数组 24
SAM 26
图论 27
Dijkstra 求最短路 27
spfa 求最短路 28
floyd 求最短路 29
prim 算法求最小生成树 29
Kruskal 求最小生成树 30
二分图 31
染色法判断二分图 31
二分图的最大匹配 32
Lca 33
Dinic 求最大流 36
Tarjan 算法 37
模拟退火 39
数据结构 40
FHQ Treap 40
普通平衡树(值) 40
文艺平衡树(区间) 43
树链剖分 45
CDQ 分治 48
计算几何 51
向量和点 51
自适应辛普森积分 56
动态规划 56
背包问题 56
01 背包问题 56
完全背包问题 57

多重背包问题 I 57
多重背包问题 II 58
分组背包问题 59
区间 DP 59
计数问题 60
其他 61
对拍 61
__int128 61
Stringstream 62
O2/O3 优化 62
BigInterger 62
BigDecimal 63
STL 自定义比较函数 64
编译指令 64
解决爆栈,手动加栈 64
神奇代码 64
赛后反思 65

数学

质数

试除法判断质数

```
bool is_prime(int n)
{
    if (n == 1) return false;
    for (int i = 2; i <= n / i; i++)
    {
        if (n % i == 0) return false;
    }
    return true;
}
```

分解质因数

```
void divide(int n)
{
    for (int i = 2; i <= n / i; i++)
```

```
    {
        if (n % i == 0)
        {
            int s = 0;
            while (n % i == 0) n /= i, s++;
            cout << i << ' ' << s << '\n'; // 输出质数
        }
    }
    if (n > 1) cout << n << ' ' << 1 << '\n'; // 最后一个
}
```

线性筛

时间复杂度:$O(n)$

```
const int N = 1000010;
int primes[N];
bool st[N];
int primes_cnt = 0;

void get_primes(int n)
{
    for (int i = 2; i <= n; i++)
    {
        if (!st[i]) primes[primes_cnt++] = i;
        for (int j = 0; primes[j] <= n / i; j++)
        {
            st[primes[j] * i] = true;
            if (i % primes[j] == 0) break;
        }
    }
}
```

约数

试除法求约数

```
void get_divisors(int n)
{
    vector<int> q;
    for (int i = 1; i <= n / i; i++)
    {
        if (n % i == 0)
        {
            q.push_back(i);
            if (n / i != i) q.push_back(n / i);
        }
    }
    sort(q.begin(), q.end());
    for (auto& item: q) cout << item << ' ';
}
```

最大公约数

一般可以直接使用系统中的 gcd(a,b)

```
int gcd(int a, int b)
{
    return b ? gcd(b, a % b) : a;
}
```

欧拉函数

欧拉函数

欧拉函数的定义:

$1 \sim N$ 中与 N 互质的数的个数被称为欧拉函数，记为 $\phi(N)$ 。

若在算数基本定理中， $N = p_1^{a_1} p_2^{a_2} \cdots p_m^{a_m}$ ，则:

$$\phi(N) = N \times \frac{p_1 - 1}{p_1} \times \frac{p_2 - 1}{p_2} \times \ldots \times \frac{p_m - 1}{p_m}$$

注意: $\phi(1) = 1$

时间复杂度：$O(log)$

```
int get_phi(int n)
{
    int ans = n;
    for (int i = 2; i <= n / i; i++)
    {
        if (n % i == 0)
        {
            ans = ans / i * (i - 1);
            while (n % i == 0) n /= i;
        }
    }
    if (n > 1) ans = ans / n * (n - 1);
    return ans;
}
```

欧拉筛求欧拉函数

```
const int N = 1e6 + 10;
int primes[N];
int phi[N];
bool st[N];
int primes_cnt;

void solve(int n)
{
    phi[1] = 1;
    for (int i = 2; i <= n; i++)
    {
```

```
        if (!st[i])
        {
            primes[primes_cnt++] = i;
            phi[i] = i - 1;
        }
        for (int j = 0; primes[j] <= n / i; j++)
        {
            st[primes[j] * i] = true;
            if (i % primes[j] == 0)
            {
                phi[primes[j] * i] = phi[i] * (primes[j]);
                break;
            }
            phi[primes[j] * i] = phi[i] * (primes[j] - 1);
        }

    }
}
```

逆元

$要求ax \equiv 1(\bmod m)m为质数x \equiv 1(\bmod m)b$存在乘法逆元的充要条件是 b 与模数 m 互质。当模数 m 为质数时，b^{m-2} 即为 b 的乘法逆元

快速幂求逆元

```
int qmi(int a, int b, int mod)
{
    int ans = 1;
    while (b)
    {
        if (b & 1) ans = 1ll * ans * a % mod;
        a = 1ll * a * a % mod;
        b >>= 1;
    }
    return ans;
}
```

扩展欧几里得算法求逆元

```
int exgcd(int a, int b, int& x, int& y)
{
    if (!b)
    {
        x = 1, y = 0;
        return a;
    }
    int d = exgcd(b, a % b, y, x);
    y -= a / b * x;
```

```
    return d;
}

void solve()
{
    int a, p, x, y;
    cin >> a >> p;
    int d = exgcd(a, p, x, y);
    if (d == 1) cout << (x + p) % p << endl; // 保证x是正数
    else puts("impossible");
}
```

组合数

求组合数 1

时间复杂度:$O(n^2)$

1. 适用于数据量小的求法(也可以暴力求)

```
const int N = 1010;
const int mod = 1e9 + 7;
int C[N][N];

void solve()
{
    for (int i = 0; i < N; i++)
        for (int j = 0; j <= i; j++)
            if (!j) C[i][j] = 1;
            else C[i][j] = (C[i - 1][j], C[i - 1][j - 1]) % mod;
}
```

求组合数 2 (用逆元求)

时间复杂度: $log(n)$

```
const int N = 100010;
const int mod = 1e9 + 7;
int fact[N];
int infact[N];

int qmi(int a, int b, int mod)
{
    int ans = 1;
    while (b)
    {
        if (b & 1) ans = 1ll * ans * a % mod;
        a = 1ll * a * a % mod;
        b >>= 1;
    }
    return ans;
}
```

```
void init()
{
    fact[0] = 1;
    infact[0] = 1;
    for (int i = 1; i < N - 1; i++)
    {
        fact[i] = fact[i - 1] * i % mod;
        infact[i] = infact[i - 1] * qmi(i, mod - 2, mod) % mod;
    }
}

void solve()
{
    int a, b;
    cin >> a >> b;
    cout << fact[a] % mod * infact[b] % mod * infact[a - b] % mod << '\
n';
}
```

卢卡斯定理

适用情况:数字较大,但是模数较小

公式:$C_a^b(\text{ lucas }) \equiv C_{\frac{a}{p}}^{\frac{b}{p}}(\text{ lucas }) C_{a \bmod p}^{b \bmod p}(\bmod p)$

```
int qmi(int a, int b, int mod)
{
    int ans = 1;
    while (b)
    {
        if (b & 1) ans = 1ll * ans * a % mod;
        a = 1ll * a * a % mod;
        b >>= 1;
    }
    return ans;
}

int C(int a, int b, int mod)
{
    if (b > a) return 0;
    int ans = 1;
    for (int i = 1, j = a; i <= b; i++, j--)
    {
        ans = ans * j % mod;
        ans = ans % mod * qmi(i, mod - 2, mod) % mod;
    }
    return ans;
}
```

```
int lucas(int a, int b, int mod)
{
    if (a < mod & b < mod) return C(a, b, mod);
    else return C(a % mod, b % mod, mod) * lucas(a / mod, b / mod, mod)
 % mod;
}

void solve()
{
    int a, b, mod;
    cin >> a >> b >> mod;
    cout << lucas(a, b, mod) << '\n';
}
```

自动取模类

```
template<const int T>
struct ModInt
{
    const static int mod = T;
    int x;

    ModInt(int x = 0) : x(x % mod)
    {}

    int val()
    {
        return x;
    }

    ModInt operator+(const ModInt& other) const
    {
        int x0 = x + other.x;
        return ModInt(x0 < mod ? x0 : x0 - mod);
    }

    ModInt operator-(const ModInt& other) const
    {
        int x0 = x - other.x;
        return ModInt(x0 < mod ? x0 + mod : x0);
    }

    ModInt operator*(const ModInt& other) const
    {
        return ModInt(1ll * x * other.x % mod);
    }

    ModInt operator/(const ModInt& other) const
```

```
{
    return *this * other.inv();
}

void operator+=(const ModInt& other)
{
    x += other.x;
    if (x >= mod) x -= mod;
}

void operator-=(const ModInt& other)
{
    x -= other.x;
    if (x < 0) x += mod;
}

void operator*=(const ModInt& other)
{
    x = 1ll * x * other.x % mod;
}

void operator/=(const ModInt& other)
{
    *this = *this / other;
}

bool operator==(const ModInt& other)
{
    return x == other.x;
}

bool operator!=(const ModInt& other)
{
    return x != other.x;
}

friend istream& operator>>(istream& is, ModInt& other)
{
    ll v;
    cin >> v;
    other = ModInt(v);
    return is;
}

friend ostream& operator<<(ostream& os, const ModInt& other)
{
    return os << other.x;
}
```

```
    ModInt qmi(ll b) const
    {
        ModInt ans(1), mul(x);
        while (b)
        {
            if (b & 1) ans = ans * mul;
            mul = mul * mul;
            b >>= 1;
        }
        return ans;
    }

    ModInt inv() const
    {
        int a = x, b = mod, u = 1, v = 0;
        while (b)
        {
            int t = a / b;
            swap(a -= t * b, b);
            swap(u -= t * v, v);
        }
        return (u < 0 ? u + mod : u);
    }
};

typedef ModInt<mod> mint;
```

分数类

```
struct Frac
{
    long long x, y;

    Frac(long long a, long long b = 1ll)
    {
        long long _gcd = gcd(a, b);
        x = a / _gcd, y = b / _gcd;
    }

    Frac operator+(const Frac& other) const
    {
        long long son = 1ll * x * other.y + 1ll * other.x * y;
        long long mat = 1ll * y * other.y;
        return Frac(son, mat);
    }

    Frac operator-(const Frac& other) const
    {
        long long son = 1ll * x * other.y - 1ll * other.x * y;
        long long mat = 1ll * y * other.y;
```

```
        return Frac(son, mat);
    }

    Frac operator*(const Frac& other) const
    {
        long long son = 1ll * x * other.x;
        long long mat = y * other.y;
        return Frac(son, mat);
    }

    Frac operator/(const Frac& other) const
    {
        long long son = x * other.y;
        long long mat = y * other.x;
        return Frac(son, mat);
    }

    bool operator<(const Frac& other) const
    {
        return 1ll * x * other.y < 1ll * y * other.x;
    }

    bool operator>(const Frac& other) const
    {
        return 1ll * x * other.y > 1ll * y * other.x;
    }

    bool operator==(const Frac& other) const
    {
        return 1ll * x * other.y == 1ll * y * other.x;
    }

    bool operator<=(const Frac& other) const
    {
        return 1ll * x * other.y <= 1ll * y * other.x;
    }

    bool operator>=(const Frac& other) const
    {
        return 1ll * x * other.y >= 1ll * y * other.x;
    }
};
```

矩阵快速幂

1. mat 矩阵是系数矩阵
2. f1 是初始矩阵

```
const int N = 3;
int n;
```

```
ll m;

void mul(int c[], int a[], int b[][N])
{
    int temp[N] = {0};
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            temp[i] = (1ll * temp[i] + 1ll * a[j] * b[j][i]) % m;
        }
    }
    memcpy(c, temp, sizeof(temp));
}

void mul(int c[][N], int a[][N], int b[][N])
{
    int temp[N][N] = {0};
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            for (int k = 0; k < N; k++)
                temp[i][j] = (1ll * temp[i][j] + 1ll * a[i][k] * b[k]
[j]) % m;

    memcpy(c, temp, sizeof(temp));
}

void solve()
{
    cin >> n >> m;
    int f1[N] = {1, 1, 1};
    int mat[][N] = {
            {0, 1, 0},
            {1, 1, 1},
            {0, 0, 1},
    };
    n--;

    while (n)
    {
        if (n & 1) mul(f1, f1, mat);
        mul(mat, mat, mat);
        n >>= 1;
    }
    cout << f1[2] % m << '\n';
}
```

欧拉函数

$$a^{\varphi(n)} \equiv 1(\bmod m)$$

如果说若m,a为正整数，且m和 a 互质 那么就成立,当m为质数的时候,就是小费马定理

FFT

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include <cmath>
using namespace std;
const int N = 300010;
const double PI = acos(-1);
int n, m;
struct Complex
{
    double x, y;
    Complex operator+ (const Complex& t) const
    {
        return {x + t.x, y + t.y};
    }
    Complex operator- (const Complex& t) const
    {
        return {x - t.x, y - t.y};
    }
    Complex operator* (const Complex& t) const
    {
        return {x * t.x - y * t.y, x * t.y + y * t.x};
    }
}a[N], b[N];
int rev[N], bit, tot;

void fft(Complex a[], int inv)
{
    for (int i = 0; i < tot; i ++ )
        if (i < rev[i])
            swap(a[i], a[rev[i]]);
    for (int mid = 1; mid < tot; mid <<= 1)
    {
        auto w1 = Complex({cos(PI / mid), inv * sin(PI / mid)});
        for (int i = 0; i < tot; i += mid * 2)
        {
            auto wk = Complex({1, 0});
            for (int j = 0; j < mid; j ++, wk = wk * w1)
            {
                auto x = a[i + j], y = wk * a[i + j + mid];
                a[i + j] = x + y, a[i + j + mid] = x - y;
```

```
            }
        }
    }
}

int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 0; i <= n; i ++ ) scanf("%lf", &a[i].x);
    for (int i = 0; i <= m; i ++ ) scanf("%lf", &b[i].x);
    while ((1 << bit) < n + m + 1) bit ++;
    tot = 1 << bit;
    for (int i = 0; i < tot; i ++ )
        rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (bit - 1));
    fft(a, 1), fft(b, 1);
    for (int i = 0; i < tot; i ++ ) a[i] = a[i] * b[i];
    fft(a, -1);
    for (int i = 0; i <= n + m; i ++ )
        printf("%d ", (int)(a[i].x / tot + 0.5));

    return 0;
}
```

欧拉筛求积性函数

$f(a \times b) = f(a) \times (b)$这样的函数叫做积性函数$(gcd(a,b) == 1)$

```
typedef long long ll;
const int N = 1e7 + 10;
int idx = 0;
int cnt[N]; // 一个数字的最小质因子出现的次数
int primes[N];
bool st[N];
ll f[N];

void solve()
{
    int n;
    cin >> n;
    cout << f[n] << '\n';
}

ll calc_f(int n, int cnt) // 用于这个 primes 的计算,对于每一个积性函数是不一样的
{
    return cnt + 1;
}

void get_primes(int n)
```

```
{
    f[1] = 1; // 一般 f[1] 都要进行初始化
    for (int i = 2; i <= n; i++)
    {
        if (!st[i])
        {
            primes[idx++] = i;
            f[i] = 2; // 就两个因子
            cnt[i] = 1; // 注意这里也要加上, i 这个最小质因子出现的次数是 1
        }
        for (int j = 0; primes[j] <= n / i; j++)
        {
            st[primes[j] * i] = true;
            if (i % primes[j] == 0)
            {
                cnt[primes[j] * i] = cnt[i] + 1;
                f[primes[j] * i] = f[i] / calc_f(primes[j], cnt[i]) * c
alc_f(primes[j], cnt[i] + 1); // 这里是除去这个数字然后再*上
                break;
            }
            cnt[primes[j] * i] = 1; // 这里表示就是出现了一次
            f[primes[j] * i] = f[i] * calc_f(primes[j], 1); // 出现了一
次,所以这里也要加上
        }


    }
}
```

高斯消元

输入一个包含n个方程n个未知数的线性方程组

方程组中的系数为实数

求解这个方程组

第一包含整数n

接下来n行,每行包含$n+1$个整数,表示一个方程的n个系数以及等号右侧的常数

无数解:输出 `Infinite group solutions`

无解:输出 `No solution`

测试样例:

输入:

```
3
1.00 2.00 -1.00 -6.00
2.00 1.00 -3.00 -9.00
-1.00 -1.00 2.00 7.00
```

输出:

```
1.00
-2.00
3.00
```

```
const int N = 1010;
int n;
double a[N][N];
double eps = 1e-6;

int gauss() // 所有的答案都在a[r][c] 中,然后最后进行求解
{
    int c, r;
    // col 是列,row 是行
    for (c = 0, r = 0; c < n; c++)
    {
        int t = r;
        for (int i = r; i < n; i++) // 找到绝对值最大的行
            if (abs(a[i][c]) > abs(a[t][c])) t = i;
        if (abs(a[t][c]) < eps) continue;
        for (int i = c; i <= n; i++) swap(a[t][i], a[r][i]);
        for (int i = n; i >= c; i--) a[r][i] /= a[r][c]; // 变成1
        for (int i = r + 1; i < n; i++)
            if (abs(a[i][c]) > eps)
            {
                for (int j = n; j >= c; j--)
                {
                    a[i][j] -= a[r][j] * a[i][c];
                }
            }

        r++;
    }
    if (r < n)
    {
        for (int i = r; i < n; i++)
        {
            if (abs(a[i][n]) > eps) return 2; // 无解
        }
        return 1; // 无穷解
    }
    for (int i = n - 1; i >= 0; i--)
    {
```

```
        for (int j = i + 1; j < n; j++)
        {
            a[i][n] -= a[i][j] * a[j][n];
        }
    }
    return 0;

}

int main()
{
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n + 1; j++)
            cin >> a[i][j];
    int t = gauss();
    if (t == 2) cout << "No solution" << '\n';
    else if (t == 1) cout << "Infinite group solutions" << '\n';
    else
    {
        for (int i = 0; i < n; i++)
        {
            if (abs(a[i][n]) < eps) a[i][n] = 0;
            printf("%.2lf\n", a[i][n]);
        }
    }
    return 0;
}
```

字符串

KMP

1. 建议都使用 string 进行求解

求 next 数组

```
const int N = 1e6 + 10;

int ne[N];

void get_next(string s) // 请从坐标 1 开始
{
    int n = s.size() - 1;
    for (int i = 2; i <= n; i++)
    {
        ne[i] = ne[i - 1];
        while (ne[i] && s[i] != s[ne[i] + 1]) ne[i] = ne[ne[i]];
        ne[i] += s[i] == s[ne[i] + 1];
```

```
    }
}
```

KMP 匹配

```
// p 是子串,s 是模式串
for (int i = 1, j = 0; i <= m; i++)
    {
        while (j && s[i] != p[j + 1]) j = ne[j];
        if (s[i] == p[j + 1]) j++;
        if (j == n)
        {
            printf("%d ", i - n);
            j = ne[j]; // 如果从 0 开始不重复,那么应该设置为 j=0
            // 这里进行相关的操作
        }
    }
```

求最小循环节

1. 在 ne 数组上求解
2. 循环节是指:字符串 s 是由多少个相同的字符串组成

```
// 在 ne 数组上进行求解
void get_min(string s) // 请提前设置好从 坐标 1 开始
{
    int n = s.size() -1 ;
    for (int i = 2; i <= n; i++)
    {
        int t = i - ne[i];
        if (i % t == 0 && ne[i])
        {
            cout << i << ' ' << i / t << '\n';
        }
        int ans = n % (n - ne[n]) ? 1 : n / (n - ne[n]);
        int len = n / ans; // 一个循环节的长度
    }
}
```

Trie 树

1. 增加的 add 表示的是增量,1 表示增加,-1 表示的是减少,相当于删除这整个枝条

```
const int N = 2000010 * 2;
int tr[N][2]; // 得看对应的情况
int has[N];
int idx = 0;

void insert(int x, int add)
{
    int p = 0;
```

```
    for (int i = 31; i >= 0; i--)
    {
        int u = (x >> i) & 1;
        if (!tr[p][u]) tr[p][u] = ++idx;
        p = tr[p][u];
        has[p] += add; // 表示的是有没有这个枝条
    }

}

int query(int x)
{
    int ans = 0;
    int p = 0;
    for (int i = 31; i >= 0; i--)
    {
        int u = (x >> i) & 1;
        if (has[tr[p][!u]])
        {
            ans += (1 << i);
            p = tr[p][!u];
        } else if (has[tr[p][u]]) // 有可能这个枝条不存在,如果没有就直接返回就行了
        {
            p = tr[p][u];
        } else return ans;
    }
    return ans;
}
```

Manachar 算法(求最长回文串长度)

时间复杂度: $O(n)$

1. p 数组不需要 memset
2. 从 0 开始,字符串 abca 会变成$#a$b#c#a#^
3. 这个 p 也包含隐藏数组(就是 r 会变长)

```
const int N = 1000010;
char a[N], b[N * 2]; // a 是原来的串,然后 b 是后来进行扩充的串,
int p[N * 2]; // 包括自身的最长回文串半径
int n; // 回文串长度 最终会变成 b 的回文串长度
void init()
{
    int k = 0;
    b[k++] = '$';
    b[k++] = '#';
    for (int i = 0; i < n; i++) b[k++] = a[i], b[k++] = '#';
    b[k++] = '^';
```

```
    n = k;
}

void manacher()
{
    int mr = 0, mid;
    for (int i = 1; i < n; i++)
    {
        if (i < mr) p[i] = min(p[mid * 2 - i], mr - i);
        else p[i] = 1;
        while (b[i - p[i]] == b[i + p[i]]) p[i]++;
        if (i + p[i] > mr)
        {
            mr = i + p[i];
            mid = i;
        }
    }
}

// abcd 变成$a$b$c$d$
void manacher(const string& _s, vector<int>& r)
{
    string s(_s.size() * 2 + 1, '$');
    for (int i = 0; i < _s.size(); i++)s[2 * i + 1] = _s[i];
    r.resize(_s.size() * 2 + 1);
    for (int i = 0, maxr = 0, mid = 0; i < s.size(); i++)
    {
        if (i < maxr)r[i] = min(r[mid * 2 - i], maxr - i);
        while (i - r[i] - 1 >= 0 && i + r[i] + 1 < s.size() && s[i - r
[i] - 1] == s[i + r[i] + 1]) ++r[i];
        if (i + r[i] > maxr) maxr = i + r[i], mid = i;
    }
}
```

字符串哈希

1. 下标从1开始
2. 将字符串翻转然后哈希两次,就可以直接比较是否是回文串了

```
struct Hash
{
    int n;
    string s;
    static constexpr int base1 = 20023;
    static constexpr int base2 = 20011;
    static constexpr ll mod1 = 2000000011;
    static constexpr ll mod2 = 3000000019;
    vector <array<ll, 2>> hash, pow_base;

    Hash()
```

```
    {}

    Hash(const string& _s)
    {
        n = _s.size();
        s = "?" + _s;
        hash.resize(n + 1);
        pow_base.resize(n + 1);
        pow_base[0][0] = pow_base[0][1] = 1;
        hash[0][0] = hash[0][1] = s[0];
        for (int i = 1; i <= n; i++)
        {
            hash[i][0] = (hash[i - 1][0] * base1 + s[i]) % mod1;
            hash[i][1] = (hash[i - 1][1] * base2 + s[i]) % mod2;
            pow_base[i][0] = pow_base[i - 1][0] * base1 % mod1;
            pow_base[i][1] = pow_base[i - 1][1] * base2 % mod2;
        }
    }

    array<ll, 2> get_hash(const int& l, const int& r)
    {
        auto single_hash = [&](bool flag)
        {
            const ll mod = !flag ? mod1 : mod2; // 注意顺序前面有!
            return (hash[r][flag] % mod - hash[l - 1][flag] % mod * pow
_base[r - l + 1][flag] % mod + mod) % mod;
        };
        return {single_hash(0), single_hash(1)};
    }
}
```

AC 自动机

给定n个模数串s_i 和一个文本串t,求有多少个不同的模式串在文本串中出现过

1. AC 自动机是离线型数据结构，不支持增量添加新的字符串
2. 现将模式串s_i插入到 AC 自动机里面,然后进行查询
3. 步骤：
    1. ac.insert(s);
    2. ac.build();
    3. ac.query(s);

```
struct AC
{
    static constexpr int N = 1e6 + 10;
    int idx = 0;
    int tr[N][26];
    int e[N], fail[N];
```

```
void insert(string s)
{
    int p = 0;
    for (int i = 0; i < s.size(); i++)
    {
        int u = s[i] - 'a';
        if (!tr[p][u]) tr[p][u] = ++idx;
        p = tr[p][u];
    }
    e[p]++; // 节点为p 的串的个数++
}

queue<int> q;

void build()
{
    for (int i = 0; i < 26; i++)
        if (tr[0][i]) q.push(tr[0][i]);

    while (q.size())
    {
        auto u = q.front();
        q.pop();
        for (int i = 0; i < 26; i++)
        {
            if (tr[u][i])
            {
                fail[tr[u][i]] = tr[fail[u]][i];
                q.push(tr[u][i]);
            } else tr[u][i] = tr[fail[u]][i];
        }
    }
}

int query(string s)
{
    int ans = 0, p = 0;
    for (int i = 0; i < s.size(); i++)
    {
        int u = s[i] - 'a';
        p = tr[p][u];
        for (int j = p; j && e[j] != -1; j = fail[j])
        {
            ans += e[j], e[j] = -1;
        }
    }
    return ans;
```

```
    }
};
```

后缀数组

给定一个长度为 n 的字符串，只包含大小写英文字母和数字。

将字符串中的 n 个字符的位置编号按顺序设为 $1\sim n$。

并将该字符串的 n 个**非空后缀**用其起始字符在字符串中的位置编号表示。

现在要对这 n 个非空后缀进行字典序排序，并给定两个数组 SA 和 $Height$。

排序完成后，用 $SA[i]$ 来记录排名为 i 的非空后缀的编号，用 $Height[i]$ 来记录排名为 i 的非空后缀与排名为 $i-1$ 的非空后缀的最长公共前缀的长度（$1 \le i \le n$）。

特别的，规定 $Height[1] = 0$。

请你求出这两个数组。

验证板子正确性:

```
abababab
```

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

```
#pragma GCC optimize(2)

#include <bits/stdc++.h>

struct SA
{
    static constexpr int N = 1e6 + 10;
    string s;
    int n, m;
    vector<int> sa, height, x, y, rk, bucket;

    SA(string _s)
    {
        n = _s.size();
        s = "?" + _s;
        m = 'z';
        sa.resize(n + 1, 0);
        rk.resize(n + 1, 0);
        height.resize(n + 1, 0);
        bucket.resize(N, 0);
        x.resize(n + 1, 0);
        y.resize(n + 1, 0);
    }
```

```
    void get_sa()
    {
        for (int i = 1; i <= n; i++) bucket[x[i] = s[i]]++;
        for (int i = 1; i <= m; i++) bucket[i] += bucket[i - 1];
        for (int i = n; i; i--) sa[bucket[x[i]]--] = i;
        for (int k = 1; k <= n; k <<= 1)
        {
            int num = 0;
            for (int i = n - k + 1; i <= n; i++) y[++num] = i;
            for (int i = 1; i <= n; i++)
            {
                if (sa[i] <= k) continue;
                y[++num] = sa[i] - k;
            }
            for (int i = 0; i <= m; i++) bucket[i] = 0;
            for (int i = 1; i <= n; i++) bucket[x[i]]++;
            for (int i = 1; i <= m; i++) bucket[i] += bucket[i - 1];
            for (int i = n; i; i--) sa[bucket[x[y[i]]]--] = y[i], y[i]
= 0;
            swap(x, y);
            x[sa[1]] = 1, num = 1;
            for (int i = 2; i <= n; i++)
            {
                x[sa[i]] = (y[sa[i]] == y[sa[i - 1]] && y[sa[i] + k] ==
 y[sa[i - 1] + k]) == 1 ? num : ++num;
                if (n == num) return;
                m = num;
            }
        }
    }

    void get_height()
    {
        for (int i = 1; i <= n; i++) rk[sa[i]] = i;
        for (int i = 1, k = 0; i <= n; i++)
        {
            if (rk[i] == 1) continue;
            if (k) k--;
            int j = sa[rk[i] - 1];
            while (i + k <= n && j + k <= n && s[i + k] == s[j + k]) k+
+;
            height[rk[i]] = k;
        }
    }
};

void solve()
{
    string s;
```

```
    cin >> s;
    SA sa(s);
    int n = sa.n;
    sa.get_sa();
    sa.get_height();
    for (int i = 1; i <= n; i++) cout << sa.sa[i] << ' ';
    cout << '\n';
    for (int i = 1; i <= n; i++) cout << sa.height[i] << ' ';
    cout << '\n';
}
```

SAM

给定一个只包含小写字母的字符串 S,

其中:

$ans1$ 表示 S 的所有出现次数不为 1 的子串的出现次数乘上该子串长度的最大值。

$ans2$ 表示S中本质不同的子串个数

```
typedef long long ll;
const int N = 2000010;
int tot = 1, last = 1;
struct Node
{
    int len, fa; // len 表示当前节点的最大长度,fa 表示该节点的父节点
    int ch[26];
} node[N];
string s;
ll f[N]; // f[i]表示当前点出现的次数
ll ans1 = 0;
ll ans2 = 0;
int h[N], e[N], ne[N], idx;

void extend(int c)
{
    int p = last, np = last = ++tot;
    f[tot] = 1;
    node[np].len = node[p].len + 1;
    for (; p && !node[p].ch[c]; p = node[p].fa) node[p].ch[c] = np;
    if (!p) node[np].fa = 1;
    else
    {
        int q = node[p].ch[c];
        if (node[q].len == node[p].len + 1) node[np].fa = q;
        else
        {
            int nq = ++tot;
            node[nq] = node[q], node[nq].len = node[p].len + 1;
```

```
            node[q].fa = node[np].fa = nq;
            for (; p && node[p].ch[c] == q; p = node[p].fa) node[p].ch
[c] = nq;
        }
    }
}

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

void dfs(int u)
{
    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        dfs(j);
        f[u] += f[j];
    }
    if (f[u] > 1) ans1 = max(ans1, f[u] * node[u].len);
    ans2 += node[u].len - node[node[u].fa].len;
}

int main()
{
    cin >> s;
    for (int i = 0; i < s.size(); i++) extend(s[i] - 'a');
    memset(h, -1, sizeof(h));
    for (int i = 2; i <= tot; i++) add(node[i].fa, i);//建立反向边
    dfs(1); // 从 1 开始,然后爆搜

    // cout<<ans1<<'\n';
    cout << ans2 << '\n';
}
```

图论

Dijkstra 求最短路

```
const int N = 200010;
typedef pair<int, int> PII;
int dist[N];
bool st[N];
int e[N], ne[N], w[N], h[N], idx;

void Dijkstra()
{
```

```
    priority_queue<PII, vector<PII>, greater<PII>> q;
    q.push({0, 1});
    dist[1] = 0;
    while (q.size())
    {
        auto v = q.top().second;
        q.pop();
        if (st[v]) continue;
        st[v] = true;
        for (int i = h[v]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > dist[v] + w[i])
            {
                dist[j] = dist[v] + w[i];
                q.push({dist[j], j});
            }
        }
    }
}
```

spfa 求最短路

时间复杂度:$O(nlogn)$

```
const int N=200010;
const int INF=0x3f3f3f3f;
int n,m;
int e[N],ne[N],w[N],h[N],idx;
int dist[N];
bool st[N];
void spfa()
{
    memset(dist,INF,sizeof(dist));
    queue<int> q;
    q.push(1);
    dist[1]=0;
    st[1]=true;
    while(q.size())
    {
        auto t=q.front();
        q.pop();
        st[t]=false;
        for(int i=h[t];i!=-1;i=ne[i])
        {
            int j=e[i];
            if(dist[j]>dist[t]+w[i])
            {
                dist[j]=dist[t]+w[i];
                if(!st[j])
```

```
                {
                    st[j]=true;
                    q.push(j);
                }
            }
        }
    }
}
```

floyd 求最短路

时间复杂度:$O(n^3)$

```
const int N = 1010;
const int INF = 0x3f3f3f3f;
int g[N][N];
int n, m, k;

void init()
{
    memset(g, INF, sizeof(g));
    for (int i = 1; i <= n; i++) g[i][i] = 0;
}

void floyd()
{
    for (int k = 1; k <= n; k++) // 中间转折点
        for (int i = 1; i <= n; i++) // 起点
            for (int j = 1; j <= n; j++) // 终点
                g[i][j] = min(g[i][j], g[i][k] + g[k][j]);

}
```

prim 算法求最小生成树

```
const int N = 510;
bool st[N];
int g[N][N];
int dist[N];
int n, m;

void init()
{
    memset(dist, 0x3f, sizeof dist);
    memset(g, 0x3f, sizeof g);
}
void prim()
{

    int ans = 0;
```

```
    for (int i = 0; i < n; i++)
    {
        int t = -1;
        for (int j = 1; j <= n; j++)
        {
            if (!st[j] && (t == -1 || dist[t] > dist[j])) t = j;
        }

        if (i && dist[t] == 0x3f3f3f3f)
        {
            cout << "impossible" << endl;
            return;
        }
        st[t] = true;
        if (i) ans += dist[t];
        for (int j = 1; j <= n; j++) dist[j] = min(dist[j], g[t][j]);

    }
    cout << ans << endl;
    return;
}
```

Kruskal 求最小生成树

```
const int N = 2e5 + 10;
int n, m;
int p[N];

struct Node
{
    int a, b, w;

    bool operator<(const Node& b) const
    {
        return w < b.w;
    }
} q[N];

int find(int x)
{
    if (x != p[x]) p[x] = find(p[x]);
    return p[x];
}

void Kruskal()
{
    int cnt = 0;
    int ans = 0;
    for (int i = 1; i <= n; i++) p[i] = i;
    sort(q, q + m);
```

```
    for (int i = 0; i < m; i++)
    {
        int a = q[i].a;
        int b = q[i].b;
        int w = q[i].w;
        int pa = find(a);
        int pb = find(b);
        if (pa != pb)
        {
            p[pa] = pb;
            ans += w;
            cnt++;
        }
    }
    if (cnt >= n - 1) cout << ans << '\n';
    else cout << "impossible" << '\n';
}
```

二分图

染色法判断二分图

给定一个n个点m条边,图中可能存在重边和自环,

请判断这个图是否是一个二分图

```
const int N = 1e5 + 10;
int h[N * 2];
int e[N * 2], ne[N * 2], idx;
int color[N];
int n, m;

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

bool dfs(int u, int c)
{
    color[u] = c;
    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!color[j])
        {
            if (!dfs(j, 3 - c)) return false;
        } else
        {
            if (color[u] == color[j]) return false;
        }
```

```
    }
    return true;
}

void solve()
{
    cin >> n >> m;
    for (int i = 1; i <= n; i++) h[i] = -1;
    for (int i = 0; i < m; i++)
    {
        int a, b;
        cin >> a >> b;
        add(a, b), add(b, a);
    }
    bool is = true;
    for (int i = 1; i <= n; i++)
    {
        if (!color[i])
        {
            if (!dfs(i, 1))
            {
                is = false;
                break;
            }
        }
    }
    if (is) cout << "Yes" << '\n';
    else cout << "No" << '\n';
}
```

二分图的最大匹配

给定一个二分图，其中左半部包含 n_+1 个点（编号 $1 \sim n$），右半部包含 n_2 个点（编号 1~n21~n2），二分图共包含 m 条边。

数据保证任意一条边的两个端点都不可能在同一部分中。

请你求出二分图的最大匹配数。

> 二分图的匹配：给定一个二分图 G，在 G 的一个子图 M 中，M 的边集 E 中的任意两条边都不依附于同一个顶点，则称 M 是一个匹配。
>
> 二分图的最大匹配：所有匹配中包含边数最多的一组匹配被称为二分图的最大匹配，其边数即为最大匹配数。
>
> 时间复杂度:$O(\textit{能过})$ (比较玄学)

1. memset 注意每一次找的时候都要清空

```
const int N = 100010;
int e[N], ne[N], h[N], idx;
```

```
bool st[N];
int match[N];
int n1, n2, m;

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

bool find(int u)
{
    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!st[j])
        {
            st[j] = true;
            if (!match[j] || find(match[j]))
            {
                match[j] = u;
                return true;
            }
        }
    }
    return false;
}

void solve()
{
    memset(h, -1, sizeof(h));
    cin >> n1 >> n2 >> m;
    for (int i = 0; i < m; i++)
    {
        int a, b;
        cin >> a >> b;
        add(a, b);
    }
    int ans = 0;
    for (int i = 1; i <= n1; i++)
    {
        memset(st, 0, sizeof(st));
        if (find(i)) ans++; // 对于i 这个点,我们找到了对应的匹配
    }
    cout << ans << '\n';
}
```

Lca

```
const int N = 200010;
int e[N], ne[N], h[N], idx;
```

```
int depth[N];
int fa[N][21];
int n, m;

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

void bfs()
{
    for (int i = 1; i <= n; i++)
    {
        depth[i] = INF;
        dist[i] = INF;
    }
    depth[0] = 0;
    depth[1] = 1;
    queue<int> q;
    q.push(1);
    while (q.size())
    {
        auto v = q.front();
        q.pop();
        for (int i = h[v]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (depth[j] > depth[v] + 1)
            {
                depth[j] = depth[v] + 1;
                dist[j] = dist[v] + 1;
                fa[j][0] = v;
                q.push(j);
                for (int k = 1; k <= 20; k++)
                {
                    fa[j][k] = fa[fa[j][k - 1]][k - 1];
                }

            }
        }
    }
}

int lca(int a, int b)
{
    if (depth[a] < depth[b]) swap(a, b);
    for (int k = 20; k >= 0; k--)
    {
        if (depth[fa[a][k]] >= depth[b])
```

```
        {
            a = fa[a][k];
        }
    }
    if (a == b) return a;
    for (int k = 20; k >= 0; k--)
    {
        if (fa[a][k] != fa[b][k])
        {
            a = fa[a][k];
            b = fa[b][k];
        }
    }
    return fa[a][0];
}

int get_dist(int a, int b)
{
    int p = lca(a, b);
    return dist[a] + dist[b] - 2 * dist[p];
}

void solve()
{
    cin >> n;
    for (int i = 1; i <= n; i++) h[i] = -1;
    for (int i = 0; i < n; i++)
    {
        int a, b;
        cin >> a >> b;
        if (b == -1) root = a;
        else
        {
            add(a, b);
            add(b, a);
        }
    }
    bfs();
    cin >> m;
    for (int i = 0; i < m; i++)
    {
        int a, b;
        cin >> a >> b;
        cout << lca(a, b) << '\n';
        cout << get_dist(a, b) << '\n';
    }
}
```

Dinic 求最大流

n个点m条边,每条边都都有容量,边的容量非负,有重边和自环,求 S 到 T 的最大流

```
const int N = 2000010;
int e[N], ne[N], h[N], w[N], idx;
int cur[N], d[N];
int n, m, S, T;

void add(int a, int b, int c)
{
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
    e[idx] = a, w[idx] = 0, ne[idx] = h[b];
    h[b] = idx++;
}

bool bfs()
{
    memset(d, -1, sizeof(d));
    queue<int> q;
    q.push(S);
    d[S] = 0, cur[S] = h[S];
    while (q.size())
    {
        auto v = q.front();
        q.pop();
        for (int i = h[v]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (d[j] == -1 && w[i])
            {
                d[j] = d[v] + 1;
                cur[j] = h[j];
                q.push(j);
                if (j == T) return true;
            }
        }
    }
    return false;

}

int find(int u, int limit)
{
    if (u == T) return limit;
    int flow = 0;
    for (int i = cur[u]; i != -1 && flow < limit; i = ne[i])
    {
```

```
        cur[u] = i;
        int j = e[i];
        if (d[j] == d[u] + 1 && w[i])
        {
            int t = find(j, min(w[i], limit - flow));
            if (!t) d[j] = -1;
            w[i] -= t, w[i ^ 1] += t, flow += t;
        }
    }
    return flow;
}

int dinic()
{
    int ans = 0, flow = 0;
    while (bfs()) while (flow = find(S, INF)) ans += flow;
    return ans;
}

void solve()
{
    memset(h, -1, sizeof(h));
    cin >> n >> m >> S >> T;
    for (int i = 0; i < m; i++)
    {
        int a, b, c;
        cin >> a >> b >> c;
        add(a, b, c);
    }
    cout << dinic() << '\n';
}
```

Tarjan 算法

1. dfn:时间戳
2. id:位于哪一个联通分量块上
3. scc_cnt:联通分量的数量
4. sz[i]:第 i 个联通分量的大小

```
int n, m;
int e[N], ne[N], h[N], idx;
int dfn[N], low[N], timestamp;
int stk[N], top;
bool in_stk[N];
int id[N], scc_cnt, sz[N];

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
```

```
}

void tarjan(int u)
{
    dfn[u] = low[u] = ++timestamp;
    stk[++top] = u, in_stk[u] = true;
    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!dfn[j])
        {
            tarjan(j);
            low[u] = min(low[u], low[j]);
        } else if (in_stk[j]) low[u] = min(low[u], dfn[j]);
    }
    if (dfn[u] == low[u])
    {
        int y;
        ++scc_cnt;
        do
        {
            y = stk[top--];
            in_stk[y] = false;
            id[y] = scc_cnt;
            sz[scc_cnt]++;

        } while (y != u);
    }
}

void solve()
{
    cin >> n >> m;
    for (int i = 1; i <= n; i++) h[i] = -1;
    for (int i = 0; i < m; i++)
    {
        int a, b;
        cin >> a >> b;
        add(a, b);
    }
    for (int i = 1; i <= n; i++)
        if (!dfn[i]) tarjan(i);

    for (int v = 1; v <= n; v++)
    {
        for (int i = h[v]; i != -1; i = ne[i])
        {
            int j = e[i];
            int a = id[v], b = id[j];
```

```
            if (a != b) // 位于不同的联通分量上进行连边
            {
                // 进行相关的操作 表示联通分量a 和b 连接一条边
            }
        }
    }
}
```

模拟退火

```
typedef pair<double, double> PDD;
double ans = 1e18;
const int N = 110;
int n;
PDD arr[N];

double rand(double l, double r) // 返回[l,r)之间随机的一点
{
    return (double) rand() / (RAND_MAX) * (r - l) + l;
}

double get_dist(PDD a, PDD b)
{
    double dx = a.first - b.first;
    double dy = a.second - b.second;
    return sqrt(dx * dx + dy * dy);
}

double calc(PDD x)
{
    double temp = 0;
    for (int i = 0; i < n; i++)
    {
        temp += get_dist(x, arr[i]);
    }
    ans = min(ans, temp); // 更新答案
    return temp;
}

void simulate_anneal()
{
    PDD cur(rand(0, 10000), rand(0, 10000));
    for (double t = 1000; t >= 1e-4; t *= 0.97)
    {
        PDD new_point(rand(cur.first - t, cur.first + t), rand(cur.seco
nd - t, cur.second + t));
        double dist = calc(new_point) - calc(cur);
        if (exp(-dist / t) > rand(0, 1)) cur = new_point;//重点
    }
```

```
}

void solve()
{
    cin >> n;
    for (int i = 0; i < n; i++) cin >> arr[i].first >> arr[i].second;
    for (int i = 0; i < 100; i++) simulate_anneal(); // 多次循环求最小值
    cout << fixed << setprecision(0) << ans << '\n';
}
```

数据结构

FHQ Treap

普通平衡树(值)

1. 插入数值 x。
2. 删除数值 x(若有多个相同的数，应只删除一个)。
3. 查询数值 x 的排名(若有多个相同的数，应输出最小的排名)。
4. 查询排名为 x 的数值。
5. 求数值 x 的前驱(前驱定义为小于 x 的最大的数)。
6. 求数值 x 的后继(后继定义为大于 x 的最小的数)。

```
const int N = 2000010;
struct Node
{
    int l, r;
    int val, key; // 值和随机化的值
    int sz; // 子树的大小
} tr[N];
int idx, root;
mt19937 rnd(233);

int new_node(int val) // 创建一个新的节点 返回的是节点编号
{
    tr[++idx].val = val;
    tr[idx].key = rnd();
    tr[idx].sz = 1;
    return idx;
}

void update(int u) // 更新信息
{
    tr[u].sz = tr[tr[u].l].sz + tr[tr[u].r].sz + 1;
}

void spilt(int now, int val, int& x, int& y) // 按照值将树分裂成x,y
```

```
{
    if (!now) x = y = 0;
    else
    {
        if (tr[now].val <= val)
        {
            x = now;
            spilt(tr[now].r, val, tr[now].r, y);
        } else
        {
            y = now;
            spilt(tr[now].l, val, x, tr[now].l);
        }
        update(now);
    }
}

int merge(int x, int y) // 合并x,y 两棵树 合并完之后的编号
{
    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;
    }
}

void insert(int val) // 插入一个新的点
{
    int x, y;
    spilt(root, val, x, y);
    root = merge(merge(x, new_node(val)), y);
}

void del(int val) // 删除val 这个值的点
{
    int x, y, z;
    spilt(root, val, x, z);
    spilt(x, val - 1, x, y);
    y = merge(tr[y].l, tr[y].r);
    root = merge(merge(x, y), z);
}
```

```
void get_rank(int val) // 得到val 这个值的点
{
    int x, y;
    spilt(root, val - 1, x, y);
    cout << tr[x].sz + 1 << '\n';
    root = merge(x, y);
}

void get_num(int rank)
{
    int now = root;
    while (now)
    {
        if (tr[tr[now].l].sz + 1 == rank) break;
        else if (tr[tr[now].l].sz >= rank) now = tr[now].l;
        else
        {
            rank -= tr[tr[now].l].sz + 1;
            now = tr[now].r;
        }
    }
    cout << tr[now].val << '\n';
}

void pre(int val)
{
    int x, y;
    spilt(root, val - 1, x, y);
    int now = x;
    while (tr[now].r) now = tr[now].r;
    cout << tr[now].val << '\n';
    root = merge(x, y);
}

void nxt(int val)
{
    int x, y;
    spilt(root, val, x, y);
    int now = y;
    while (tr[now].l) now = tr[now].l;
    cout << tr[now].val << '\n';
    root = merge(x, y);
}

void solve()
{
    int n;
    cin >> n;
    for (int i = 0; i < n; i++)
```

```
    {
        int op, x;
        cin >> op >> x;
        if (op == 1) insert(x);
        else if (op == 2) del(x);
        else if (op == 3) get_rank(x);
        else if (op == 4) get_num(x);
        else if (op == 5) pre(x);
        else if (op == 6) nxt(x);
    }
}
```

文艺平衡树(区间)

```
const int N = 2e5 + 10;
struct Node
{
    int l, r;
    int val, key;
    int sz;
    bool reverse;
} tr[N];
int idx, root;
mt19937 rnd(233);

int new_node(int val)
{
    tr[++idx].val = val;
    tr[idx].key = rnd();
    tr[idx].sz = 1;
    return idx;
}

void update(int now)
{
    tr[now].sz = tr[tr[now].l].sz + tr[tr[now].r].sz + 1;
}

void pushdown(int now)
{
    swap(tr[now].l, tr[now].r); // 相当于交换了
    tr[tr[now].l].reverse ^= 1;
    tr[tr[now].r].reverse ^= 1;
    tr[now].reverse = false;
}

void spilt(int now, int sz, int& x, int& y)
{
    if (!now) x = y = 0;
    else
```

```
    {
        if (tr[now].reverse) pushdown(now);
        if (tr[tr[now].l].sz < sz)
        {
            x = now;
            spilt(tr[now].r, sz - tr[tr[now].l].sz - 1, tr[now].r, y);
        } else
        {
            y = now;
            spilt(tr[now].l, sz, x, tr[now].l);
        }
        update(now);
    }
}

int merge(int x, int y)
{
    if (!x || !y) return x + y;
    if (tr[x].key < tr[y].key)
    {
        if (tr[x].reverse) pushdown(x);
        tr[x].r = merge(tr[x].r, y);
        update(x);
        return x;
    } else
    {
        if (tr[y].reverse) pushdown(y);
        tr[y].l = merge(x, tr[y].l);
        update(y);
        return y;
    }
}

void reverse(int l, int r)
{
    int x, y, z;
    spilt(root, l - 1, x, y);
    spilt(y, r - l + 1, y, z);
    tr[y].reverse ^= 1;
    root = merge(merge(x, y), z);
}

void ldr(int now)
{
    if (!now) return;
    if (tr[now].reverse) pushdown(now);
    ldr(tr[now].l);
    cout << tr[now].val << ' ';
    ldr(tr[now].r);
```

```
}

int n, m;

void solve()
{
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
    {
        root = merge(root, new_node(i));
    }
    for (int i = 0; i < m; i++)
    {
        int l, r;
        cin >> l >> r;
        reverse(l, r);
    }
    ldr(root);
}
```

树链剖分

给定一棵树,树中包含n个节点(编号$1 \sim n$),其中第i个节点的权值是a_i,树链剖分可以进行一下几种操作:

1. `1 u v k`，修改路径上节点权值，将节点 u 和节点 v 之间路径上的所有节点（包括这两个节点）的权值增加 k。
2. `2 u k`，修改子树上节点权值，将以节点 u 为根的子树上的所有节点的权值增加 k。
3. `3 u v`，询问路径，询问节点 u 和节点 v 之间路径上的所有节点（包括这两个节点）的权值和。
4. `4 u`，询问子树，询问以节点 u 为根的子树上的所有节点的权值和。

对于修改 u 节点和 u 为根的子树上的节点为:`dfn[u]`到 `dfn[u]+sz[u]-1`

```
const int N = 1000010;
// #define int long long
int top[N], w[N], e[N], ne[N], idx;
int h[N];
int sz[N];
int dfn[N];
int fa[N];
int depth[N];
int son[N];
int v[N];
int tim = 0;
struct Node
{
```

```
    int l, r;
    ll sum;
    ll add;
} tr[N << 2];
int n, m;

void pushup(int u)
{
    tr[u].sum = tr[u << 1].sum + tr[u << 1 | 1].sum;
}

void pushdown(int u)
{
    if (tr[u].add)
    {
        tr[u << 1].add += tr[u].add;
        tr[u << 1].sum += (tr[u << 1].r - tr[u << 1].l + 1) * tr[u].add;
        tr[u << 1 | 1].add += tr[u].add;
        tr[u << 1 | 1].sum += (tr[u << 1 | 1].r - tr[u << 1 | 1].l + 1)
 * tr[u].add;
        tr[u].add = 0;
    }
}

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

void dfs1(int u, int father) // 第一次dfs 要遍历的是处理大小和重儿子
{
    fa[u] = father;
    depth[u] = depth[father] + 1;
    sz[u] = 1;
    int max_size = -1;
    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (j == father) continue;
        dfs1(j, u);
        sz[u] += sz[j];
        if (sz[j] > max_size)
        {
            max_size = sz[j];
            son[u] = j;
        }
    }
}
```

```
void build(int u, int l, int r)
{
    tr[u] = {l, r};
    if (l == r)
    {
        tr[u] = {l, r, w[r], 0};
        return;
    }
    int mid = l + r >> 1;
    build(u << 1, l, mid);
    build(u << 1 | 1, mid + 1, r);
    pushup(u);
}

ll query(int u, int l, int r)
{
    if (l <= tr[u].l && tr[u].r <= r) return tr[u].sum;
    pushdown(u);
    int mid = tr[u].l + tr[u].r >> 1;
    ll ans = 0;
    if (l <= mid) ans += query(u << 1, l, r);
    if (r > mid) ans += query(u << 1 | 1, l, r);
    return ans;
}

void dfs2(int u, int t)
{
    dfn[u] = ++tim;
    top[u] = t;
    w[tim] = v[u];
    if (!son[u]) return;
    dfs2(son[u], t);
    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (j == fa[u] || j == son[u]) continue;
        dfs2(j, j);
    }

}

void modify(int u, int l, int r, int k)
{
    if (l <= tr[u].l && tr[u].r <= r)
    {
        tr[u].sum += (tr[u].r - tr[u].l + 1) * k;
        tr[u].add += k;
        return;
    }
```

```
    pushdown(u);
    int mid = tr[u].l + tr[u].r >> 1;
    if (l <= mid) modify(u << 1, l, r, k);
    if (r > mid) modify(u << 1 | 1, l, r, k);
    pushup(u);
}

void modify_path(int x, int y, int k) // 将路径上的和都加上 k
{
    while (top[x] != top[y])
    {
        if (depth[top[x]] < depth[top[y]])swap(x, y);
        modify(1, dfn[top[x]], dfn[x], k);
        x = fa[top[x]];
    }
    if (depth[x] > depth[y]) swap(x, y);
    modify(1, dfn[x], dfn[y], k);
}

ll query_path(int x, int y) // 查询路径 x,y 上的和
{
    ll ans = 0;
    while (top[x] != top[y])
    {
        if (depth[top[x]] < depth[top[y]]) swap(x, y);
        ans += query(1, dfn[top[x]], dfn[x]);
        x = fa[top[x]];
    }
    if (depth[x] > depth[y]) swap(x, y);
    ans += query(1, dfn[x], dfn[y]);
    return ans;
}
```

CDQ 分治

主要解决的是三维偏序的问题

给定n个元素，(编号是1~n),其中第i个元素是a_i,b_i,c_i三种属性，

设$f[i]$ 表示满足以下 4 种条件

1. $a_j \le a_i$
2. $b_j \le b_i$
3. $c_j \le c_i$
4. $j \ne i$

的j的数量。

对于 $d \in [0,n)$ ，求满足 $f(i)=d$ 的 i 的数量。

第一行两个整数n,k,表示元素数量和最大属性值 接下来n行，其中第n行包括三个整数a_i,b_i,c_i,分别表示第i个元素的三个属性值。

输出共n行，每行输出一个整数，其中第$d+1$行表示$f(i)=d$的数量

验证板子

```
10 3
3 3 3
2 3 3
2 3 1
3 1 1
3 1 2
1 3 1
1 1 2
1 2 2
1 3 2
1 2 1

3
1
3
0
1
0
1
0
0
1
```

```cpp
#include <iostream>
#include <cstring>
#include <algorithm>

using namespace std;

const int N = 100010, M = 200010;

int n, m;
struct Data
{
    int a, b, c, s, res;

    bool operator< (const Data& t) const
    {
        if (a != t.a) return a < t.a;
        if (b != t.b) return b < t.b;
```

```
        return c < t.c;
    }
    bool operator== (const Data& t) const
    {
        return a == t.a && b == t.b && c == t.c;
    }
}q[N], w[N];
int tr[M], ans[N];

int lowbit(int x)
{
    return x & -x;
}

void add(int x, int v)
{
    for (int i = x; i < M; i += lowbit(i)) tr[i] += v;
}

int query(int x)
{
    int res = 0;
    for (int i = x; i; i -= lowbit(i)) res += tr[i];
    return res;
}

void merge_sort(int l, int r)
{
    if (l >= r) return;
    int mid = l + r >> 1;
    merge_sort(l, mid), merge_sort(mid + 1, r);
    int i = l, j = mid + 1, k = 0;
    while (i <= mid && j <= r)
        if (q[i].b <= q[j].b) add(q[i].c, q[i].s), w[k ++ ] = q[i ++ ];
        else q[j].res += query(q[j].c), w[k ++ ] = q[j ++ ];
    while (i <= mid) add(q[i].c, q[i].s), w[k ++ ] = q[i ++ ];
    while (j <= r) q[j].res += query(q[j].c), w[k ++ ] = q[j ++ ];
    for (i = l; i <= mid; i ++ ) add(q[i].c, -q[i].s);
    for (i = l, j = 0; j < k; i ++, j ++ ) q[i] = w[j];
}

int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 0; i < n; i ++ )
    {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        q[i] = {a, b, c, 1};
```

```
    }
    sort(q, q + n);

    int k = 1;
    for (int i = 1; i < n; i ++ )
        if (q[i] == q[k - 1]) q[k - 1].s ++ ;
        else q[k ++ ] = q[i];

    merge_sort(0, k - 1);
    for (int i = 0; i < k; i ++ )
        ans[q[i].res + q[i].s - 1] += q[i].s;

    for (int i = 0; i < n; i ++ ) printf("%d\n", ans[i]);

    return 0;
}
```

计算几何

向量和点

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

using namespace std;
#define IOS ios::sync_with_stdio(false);cin.tie(0);cout.tie(0)
#define eps 1e-8
#define int128 __int128
#define gcd(a, b) __gcd(a,b)
#define lcm(a, b) a/gcd(a,b)*b
#define lowbit(x) (x&-x)
#define all(x) x.begin(), x.end()
#define debug(x...) do { cout<< #x <<" -> "; re_debug(x); } while (0)

void re_debug()
{ cout << '\n'; }

template<class T, class... Ts>
void re_debug(const T& arg, const Ts& ... args)
{
    cout << arg << " ";
    re_debug(args...);
}

int test = 1;

void cut()
{ cout << "test:" << ' ' << test++ << '\n'; }
```

```cpp
typedef long long ll;
typedef unsigned long long ull;
typedef pair<int, int> PII;
const int INF = 0x3f3f3f3f;
const ll LNF = 0x3f3f3f3f3f3f3f3fll;
const double PI = acos(-1.0);

int sign(double x) // 符号函数
{
    if (abs(x) < eps) return 0; // 算是0
    if (x < 0) return -1;
    return 1;
}

struct Point
{
    double x, y;

    Point operator+(const Point& b) const
    {
        return Point{x + b.x, y + b.y};
    }

    Point operator-(const Point& b) const
    {
        return Point{x - b.x, y - b.y};
    }

    Point operator*(const double& k) const
    {
        return Point{x * k, y * k};
    }

    Point operator/(const double& k) const
    {
        return Point{x / k, y / k};
    }

    bool operator==(const Point& b) const
    {
        return sign(x - b.x) == 0 && sign(y - b.y) == 0;
    }

    bool operator<(const Point& b) const
    {
        return x < b.x || (x == b.x && y < b.y);
    }
```

```
};


int cmp(double x, double y) // 比较函数
{
    if (abs(x - y) < eps) return 0;
    if (x < y) return -1;
    return 1;
}

double dot(Point a, Point b) // 点乘
{
    return a.x * b.x + a.y * b.y;
}

double cross(Point a, Point b) // 外积:表示向量A,B 形成的平行四边形面积
{
    return a.x * b.y - a.y * b.x;
}

double get_lenth(Point a) // 求模长 用的是向量
{
    return sqrt(dot(a, a));
}

double get_lenth(Point a, Point b) // 求点a 到点b 的长度
{
    return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}

double get_angle(Point a, Point b) // 返回的是弧度
{
    return acos(dot(a, b) / get_lenth(a) / get_lenth(b));
}

double get_area(Point a, Point b, Point c) // 返回三点构成的平行四边形的
有向面积
{
    return cross(b - a, c - a);
}

Point rotate(Point a, double angle) // 向量A 顺时针旋转C 的角度
{
    return Point{a.x * cos(angle) + a.y * sin(angle), -a.x * sin(angle)
 + a.y * cos(angle)};
}
```

```
Point get_line_intersection(Point p, Point v, Point q, Point w) // 两个直线相交的点
{
    //两个直线是p+tv 和q+tw
    Point u = p - q;
    double t = cross(w, u) / cross(v, w);
    return p + v * t;
}

double distance_to_line(Point a, Point b, Point p) // 点p 到直线ab 的距离
{
    Point v1 = b - a, v2 = p - a;
    return abs(cross(v1, v2) / get_lenth(a, b));
}

double distance_to_segment(Point a, Point b, Point p) // 点p 到线段ab 的距离
{
    if (a == b) return get_lenth(p - a);
    Point v1 = b - a, v2 = p - a, v3 = p - b;
    if (sign(dot(v1, v2)) < 0) return get_lenth(v2);
    if (sign(dot(v1, v3)) > 0) return get_lenth(v3);
    return distance_to_line(a, b, p);
}

Point get_line_projection(Point a, Point b, Point p) // 点p 在向量ab 的投影的坐标
{
    Point v = b - a;
    return a + v * (dot(v, p - a) / dot(v, v));
}

bool is_on_segment(Point a, Point b, Point p) // 点p 是否在线段ab 上
{
    return sign(cross(p - a, p - b)) == 0 && sign(dot(p - a, p - b)) <= 0;
}

bool is_segment_intersection(Point a1, Point a2, Point b1, Point b2) // 线段a 和b 是否相交
{
    /*
      double c1 = cross(a2 - a1, b1 - a1), c2 = cross(a2 - a1, b2 - a1);
      double c3 = cross(b2 - b1, a2 - b1), c4 = cross(b2 - b1, a1 - b1);
      return sign(c1) * sign(c2) <= 0 && sign(c3) * sign(c4) <= 0;

     */
```

```
    /*
     a1    b2
       \   /
        \ /
         /\
    b1 /  \ a2

    */
    double c1 = cross(a2 - a1, b1 - a1), c2 = cross(a2 - a1, b2 - a1);
    double c3 = cross(b2 - b1, a2 - b1), c4 = cross(b2 - b1, a1 - b1);
    return sign(c1) * sin(c2) <= 0 && sign(c3) * sign(c4) <= 0;
}

double get_triangle_area(Point a, Point b, Point c) // 得到三个点围城的三角形面积
{
    //海伦公式 p=(a+b+c)/2 S=sqrt((p-a)*(p-b)*(p-c)));
    double len_a = get_lenth(a - b);
    double len_b = get_lenth(a - c);
    double len_c = get_lenth(b - c);
    double p = (len_a + len_b + len_c) / 2;
    return sqrt(p * (p - len_a) * (p - len_b) * (p - len_c));
}

double polygon_area(Point p[], int n) // 求多边形面积
{
    double ans = 0;
    for (int i = 1; i + 1 < n; i++)
    {
        ans += cross(p[i] - p[0], p[i + 1] - p[i]);
    }
    return ans / 2;
}

void solve()
{
    cout << get_triangle_area({0, 0}, {1, 1}, {1, 1}) << '\n';
}

int main()
{
    IOS;
    int T = 1;
    //  cin>>T;
    while (T--) solve();
    return 0 ^ 0;
}
```

自适应辛普森积分

```
#define eps 1e-8

double f(double x)
{
    return sin(x) / x; // 写对应的函数表达式
}

double simpson(double l, double r)
{
    double mid = (l + r) / 2;
    return (r - l) * (f(l) + 4 * f(mid) + f(r)) / 6; // 公式,记住就行
}

double asr(double l, double r, double area)
{
    double mid = (l + r) / 2;
    double left = simpson(l, mid), right = simpson(mid, r);
    if (abs(left + right - area) < eps) return left + right;
    else return asr(l, mid, left) + asr(mid, r, right);

}

void solve()
{
    double l, r;
    cin >> l >> r;
    cout << fixed << setprecision(6) << asr(l, r, simpson(l, r)) << '\n
';
}
```

动态规划

背包问题

01 背包问题

```
const int N = 1010;
int w[N];
int v[N];
int f[N];
int n, m;

void solve()
{
    cin >> n >> m;
    for (int i = 1; i <= n; i++) cin >> v[i] >> w[i];
    for (int i = 1; i <= n; i++)
```

```
    {
        for (int j = m; j >= v[i]; j--)
        {
            f[j] = max(f[j], f[j - v[i]] + w[i]);
        }
    }
    cout << f[m] << '\n';
}
```

完全背包问题

```
const int N = 1010;
int f[N];
int w[N], v[N];

void solve()
{
    int n, m;
    cin >> n >> m;
    for (int i = 1; i <= n; i++) cin >> v[i] >> w[i];
    for (int i = 1; i <= n; i++)
    {
        for (int j = v[i]; j <= m; j++)
        {
            f[j] = max(f[j], f[j - v[i]] + w[i]);
        }
    }
    cout << f[m] << '\n';
}
```

多重背包问题 I

```
const int N = 1010;
int f[N];
int v[N], w[N];
int s[N];

void solve()
{
    int n, m;
    cin >> n >> m;
    for (int i = 1; i <= n; i++) cin >> v[i] >> w[i] >> s[i];
    for (int i = 1; i <= n; i++)
    {
        for (int k = 0; k < s[i]; k++)
        {
            for (int j = m; j >= v[i]; j--)
            {
                f[j] = max(f[j], f[j - v[i]] + w[i]);
            }
        }
```

```
    }
    cout << f[m] << '\n';
}
```

多重背包问题 II

1. 按照二进制进行枚举进行拆分

```
const int N = 1000010;
int v[N], w[N];
int f[N];

int main()
{
    int n, m;
    scanf("%d%d", &n, &m);
    int cnt = 0;
    for (int i = 1; i <= n; i++)
    {
        int a, b, s;
        cin >> a >> b >> s;
        int k = 1;
        while (k <= s)
        {
            cnt++;
            v[cnt] = a * k;
            w[cnt] = b * k;
            s -= k;
            k *= 2;
        }
        if (s > 0)
        {
            cnt++;
            v[cnt] = a * s;
            w[cnt] = b * s;

        }

    }
    n = cnt;
    for (int i = 1; i <= n; i++)
    {
        for (int j = m; j >= v[i]; j--)
        {
            f[j] = max(f[j], f[j - v[i]] + w[i]);
        }
    }
    cout << f[m] << endl;
    return 0;
}
```

分组背包问题

```
const int N = 110;
int n, m;
int v[N][N], w[N][N], s[N];
int f[N];

int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i++)
    {
        scanf("%d", &s[i]);
        for (int j = 0; j < s[i]; j++)
        {
            scanf("%d%d", &v[i][j], &w[i][j]);
        }
    }
    for (int i = 1; i <= n; i++)
    {
        for (int j = m; j >= 0; j--)
        {
            for (int k = 0; k < s[i]; k++)
            {
                if (j >= v[i][k])
                {
                    f[j] = max(f[j], f[j - v[i][k]] + w[i][k]);
                }
            }
        }
    }
    cout << f[m] << endl;
    return 0;
}
```

区间 DP

1. 优先枚举长度

```
const int N = 310;
int s[N];
int a[N];
int f[N][N];

void solve()
{
    /*
    每一次只能合并相邻两堆
    对于任何一个区间'
    */
    int n;
```

```
    cin >> n;
    for (int i = 1; i <= n; i++) cin >> a[i];
    for (int i = 1; i <= n; i++) s[i] = s[i - 1] + a[i];
    for (int len = 2; len <= n; len++)
    {
        for (int i = 1; i + len - 1 <= n; i++)
        {
            int l = i;
            int r = i + len - 1;
            f[l][r] = 0x3f3f3f3f;
            for (int k = l; k < r; k++)
            {
                f[l][r] = min(f[l][r], f[l][k] + f[k + 1][r] + s[r] - s
[l - 1]);
            }
        }
    }
    cout << f[1][n] << '\n';
}
```

计数问题

```
const int mod = 1e9 + 7;
const int N = 100010;
int f[N];

/*
定义状态转移:
f[i][j]表示从从前i 个数字选,其中和为j 的数字
那么选第i 个数字的时候,我们可以选择0 个i 1 一个i 两个i
f[i][j]=f[i-1][j]+f[i-1][j-1*i]+f[i-1][j-2*i]+----f[i-1][j-k*i];
f[i][j-i]=          f[i-1][j-i]+f[i-1][j-2*i] -----f[i-1][j-k*i];
所以可知道 f[i][j]=f[i-1][j]+f[i][j-i];
压缩可知: 对于i,j 只会用到i-1 行和i 行的数据,所以我们可以压缩
f[j]=(f[j]+f[j-i])%mod;
*/
void solve()
{
    int n;
    cin >> n;
    f[0] = 1;
    for (int i = 1; i <= n; i++)
    {
        for (int j = i; j <= n; j++)
        {
            f[j] = (f[j] + f[j - i]) % mod;
        }
    }
    cout << f[n] << '\n';
}
```

其他

对拍

准备一下文件: (放在同一个文件夹里面)

1. `brute.cpp` :暴力程序
2. `std.cpp` :正确程序
3. `makedata.cpp`:生成数据的程序
4. `data.ans` 表示正确答案的输出
5. `data.out` 表示暴力程序的输出
6. `duipai.bat` 表示对拍的 bat

```
@echo off
g++ .\duipai\brute.cpp -o brute -O2 -std=c++17
g++ .\duipai\makedata.cpp -o makedata -O2 -std=c++17
g++ .\duipai\std.cpp -o std -O2 -std=c++17
set cnt=0
:again
    set /a cnt=cnt+1
    echo TEST:%cnt%

    .\makedata >in
    .\std <in >data.ans
    .\brute <in >data.out
    fc data.ans data.out
if not errorlevel 1 goto again
```

__int128

```
using int128 = __int128

int128 read()
{
    int128 x = 0, f = 1;
    char ch = getchar();
    while (ch < '0' || ch > '9')
    {
        if (ch == '-')
            f = -1;
        ch = getchar();
    }
    while (ch >= '0' && ch <= '9')
    {
        x = x * 10 + ch - '0';
```

```
        ch = getchar();
    }
    return x * f;
}

void print(int128 x)
{
    if (x < 0)
    {
        putchar('-');
        x = -x;
    }
    if (x > 9) print(x / 10);
    putchar(x % 10 + '0');
}
```

Stringstream

```
//1 不要开IOS
//2 记得要cin.get()
void solve()
{
    string s;
    cin.get();
    getline(cin, s);
    stringstream ssin(s);
    string t;
    while (ssin >> t)
    {
        cout << t << '\n';
    }
}
```

O2/O3 优化

```
#pragma GCC optimize(2)
#pragma GCC optimize(3)
```

BigInterger

```
BigInteger abs() // 返回大整数的绝对值
BigInteger add(BigInteger val) // 返回两个大整数的和
BigInteger and(BigInteger val) // 返回两个大整数的按位与的结果
BigInteger andNot(BigInteger val) // 返回两个大整数与非的结果
BigInteger divide(BigInteger val) // 返回两个大整数的商
double doubleValue() // 整数的double类型的值
BigInteger gcd(BigInteger val) // 返回大整数的最大公约数
int intValue() // 返回大整数的整型值
long longValue() // 返回大整数的long型值
BigInteger max(BigInteger val) // 返回两个大整数的最大者
BigInteger min(BigInteger val) // 返回两个大整数的最小者
```

```
BigInteger mod(BigInteger val) // 用当前大整数对val 求模
BigInteger multiply(BigInteger val) // 返回两个大整数的积
BigInteger negate() // 返回当前大整数的相反数
BigInteger not() // 返回当前大整数的非
BigInteger or(BigInteger val) // 返回两个大整数的按位或
BigInteger pow(int exponent) // 返回当前大整数的exponent 次方
BigInteger remainder(BigInteger val) // 返回当前大整数除以val 的余数
BigInteger leftShift(int n) // 将当前大整数左移n 位后返回
BigInteger rightShift(int n) // 将当前大整数右移n 位后返回
BigInteger subtract(BigInteger val) // 返回两个大整数相减的结果
byte[] toByteArray(BigInteger val) // 将大整数转换成二进制反码保存在byte
数组中
String toString() // 将当前大整数转换成十进制的字符串形式
BigInteger xor(BigInteger val) // 返回两个大整数的异或
int compareTo(BigInteger val) // 将此BigInteger 与指定的BigInteger 进行比
较。
boolean equals(Object x) // 将此BigInteger 与指定的Object 进行相等性比较
BigInteger  modPow(BigInteger exponent, BigInteger m) // 返回一个值为 (t
hise^xponent mod m)的BigInteger
boolean isProbablePrime(int certainty) // 返回 true 如果此BigInteger 可能
为素数， false ，如果它一定为合
// certainty  取值:1 50%  2 75%  3 87.5% 4 93.75%  5 97.875% 10 99.9%
一般取3 就可以了
```

BigDecimal

```
BigDecimal(String.valueOf(string val)) 使用此方法进行构造,精度不会丢失
BigDecimal add(Big Decimal) // 返回两个BigDecimal 的和
BigDecimal subtract(Big Decimal) // 返回两个BigDecimal 的差
BigDecimal multiply(Big Decimal) // 返回两个BigDecimal 的积
BigDecimal divide(BigDecimal divisor, int scale, int roundingMode) // d
ivide(除数,小数位数,取舍规则)
double  doubleValue() // 将BigDecimal 转换为 double
boolean equals(Object x) // 将此 BigDecimal 与指定的 Object 进行相等性比较
BigDecimal  movePointLeft(int n) // 返回一个 BigDecimal ，相当于这个小数
点向左移动 n 位置
BigDecimal  movePointRight(int n) // 返回一个 BigDecimal ，相当于这个小数
点向右移动 n 位置

/*
取舍规则: 看取舍小数的后面一位来进行判断 是RoundingMode.或者BigDecimal.来
进行调用
ROUND_CEILING 向正无穷方向舍入 向上取整
ROUND_DOWN 向零方向舍入 向0 取整
ROUND_FLOOR 向负无穷方向舍入 向下区中
ROUND_HALF_DOWN 向（距离）最近的一边舍入，除非两边（的距离）是相等,如果是这
```

```
样，向下舍入，例如1.55 保留一位小数结果为1.5
ROUND_HALF_EVEN 向（距离）最近的一边舍入，除非两边（的距离）是相等,如果是这样，如果保留位数是奇数，使用ROUND_HALF_UP，如果是偶数，使用ROUND_HALF_DOWN
ROUND_HALF_UP 四舍五入,看保留小数的后面的一位
ROUND_UNNECESSARY 计算结果是精确的，不需要舍入模式
ROUND_UP 向远离0的方向舍入
*/
```

STL 自定义比较函数

```
struct cmp
{
    long long operator()(pair<int, int> x) const
    {
        return 1ll * x.first * 5221417 + x.second;
    }
};

unordered_map<pair<int, int>, int, cmp> umap;

struct comp
{
    bool operator()(int x, int y)
    {
        return x > y;
    }
};

priprity_queue<int, vector<int>, cmp> q;
```

编译指令

```
g++ -O2 -Wall -std=c++17 -DDEBUG -Wl --stack=268435456
```

解决爆栈,手动加栈

1. 防止爆栈最好的写法是写成 bfs,或者模拟栈,加栈是旁门左道,局谨慎!
2. 不要忘了 exit(0)

```
int main()
{
    int size(512 << 20); // 512M
    __asm__ ( "movq %0, %%rsp\n"::"r"((char*) malloc(size) + size));
    // YOUR CODE
    ...
    exit(0);
}
```

神奇代码

1. 能够输出自身代码的函数

```
#include<cstdio>

char *s={"#include<cstdio>%cchar *s={%c%s%c};%cint main(){printf(s,10,34,s,34,10);return 0;}"};

int main(){printf(s,10,34,s,34,10);return 0;}
```

赛后反思

1. 写题的时候可以选择三一个人一起写一道题,或者两人写题,一人开其他题,两个人一人码代码,一人监工.
2. 上机之前要把思路写到自己的纸上,然后捋清思路,然后才上机,不要耽误时间
3. 交题前一定要看看题面,防止没有注意细节导致 WA 题
4. 多实例题目要注意清空数据等
5. 有点题目可能没有思路,但是过题量和正确率都很高,可以想一想结论
6. 增加少数服从多数的形式,遇到恶心的事情,要举手表决,然后做出建议
7. 遇到一些数据很恶心的题,不要直接用 double 来算,应该转化成乘法来算,谁直接用 double 比谁**
8. 请不要在一道题上浪费过多的时间