Zlin的板子大全

废物Zlin的自用模版合集，代码风格一般，运行效率低下，代码实用性弱，收集完善度弱，路过大佬轻点骂

欢迎提出各种意见本人版权意识薄弱

你看nm呢，回去训练

杂项

对拍

duipai模版

```
#include<iostream>
#include<windows.h>
using namespace std;
int main()
{
    int t=1000;
    while(t)
    {
        t--;
        system("data.exe > data.txt");
        system("a.exe < data.txt > a.txt");
        system("b.exe < data.txt > b.txt");
        if(system("fc a.txt b.txt"))   break;
    }
    if(t==0) cout<<"no error"<<endl;
    else cout<<"error"<<endl;

    return 0;
}

// 注意编译文件的路径
//Mac 记得在终端编译运行，Zlin's MBP默认g++-14
#include<iostream>
#include<stdlib.h>  // 在 Unix 系统下包含 system 函数需要使用 <stdlib.h>

using namespace std;

int main() {
    int t = 1000;
    for (int i = 1; i <= t; i++) {
        system("./data > data.txt");
        system("./a < data.txt > a.txt");
        system("./b < data.txt > b.txt");
        cout << "test " << i << " :";
        if (system("diff a.txt b.txt")) {
            cout << "WA" << '\n';
            break;
        } else cout << "AC" << '\n';
```

```
    }
    return 0;
}
```

常用数据生成方式

```
#include <iostream>
#include <cstdlib> // rand(), srand()
#include <ctime> // time()
#include <set>
#include <vector>
#include <algorithm> // shuffle
#include <utility> // pair

// 随机打乱序列 random_shuffle(sequence.begin(), sequence.end());

int random(int n) {//返回0-n-1之间的随机整数
    cout << rand() % n << '\n';
}

void RandomArray() {//随机生成长度为n的绝对值在1e9之内的整数序列
    int n = random(1e5) + 1;
    int m = 1e9;
    for (int i = 1; i <= n; i++) {
        cout << random(2 * m + 1) - m << '\n';
    }
}

void Intervals() {//随机生成 m个[1,n]的子区间
    int m = 10, n = 100;
    for (int i = 1; i <= m; i++) {
        int l = random(n) + 1;
        int r = random(n) + 1;
        if (l > r) swap(l, r);
        cout << l << " " << r << '\n';
    }
}

void generateTree() {//随机生成一棵n个点的树，用n个点n-1条边的无向图的形式输出
    int n = 10;
    for (int i = 2; i <= n; i++) {//从2 ~ n之间的每个点i向 1 ~ i-1之间的点随机连一条边
        int fa = random(i - 1) + 1;
        int val = random(1e9) + 1;
        cout << fa << " " << i << " " << val << '\n';
    }
}

void generateGraph() {//随机生成一张n个点m条边的无向图，图中不存在重边、自环
    int n = 10, m = 6;
    set<pair<int, int>> edges;//防止重边
    for (int i = 1; i <= n; i++) {//先生成一棵树，保证连通
        int fa = random(i - 1) + 1;
        edges.insert({ fa, i + 1 });
        edges.insert({ i + 1, fa });
    }
```

```
    while (edges.size() < m) {//再生成剩余的 m-n+1 条边
        int x = random(n) + 1;
        int y = random(n) + 1;
        if (x != y) {
            edges.insert({ x, y });
            edges.insert({ y, x });
        }
    }
    // Shuffling and outputting
    vector<pair<int, int>> Edges(edges.begin(), edges.end());
    random_shuffle(Edges.begin(), Edges.end());
    for (auto& edge : Edges) {
        cout << edge.first << " " << edge.second << '\n';
    }
}

// 生成一个随机字符串，包含大小写字母、数字和问号
string String(int length) {
    const string characters =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789?";
    random_device rd; // 随机设备
    mt19937 gen(rd()); // 使用Mersenne Twister算法
    uniform_int_distribution<> dis(0, characters.size() - 1); // 定义一个分布

    string randomString;
    for (int i = 0; i < length; ++i) {
        randomString += characters[dis(gen)]; // 从字符集随机选择字符
    }

    return randomString;
}

vector<int> generate_shuffled_permutation(int n) {
    vector<int> a(n);
    iota(a.begin(), a.end(), 1); // 生成 [1, 2, ..., n]
    mt19937 rng(seed);           // 可选种子，默认使用当前时间
    shuffle(a.begin(), a.end(), rng);
    return a;
}

int main() {
    srand(time(0));
    /*随机生成*/
    return 0;
}
```

离散化

```
inline vi disc(const vi& a)
{
    vi v(a);
    sort(v.begin(), v.end());
    v.erase(unique(v.begin(), v.end()), v.end());
    vi res(a.size());
    for (int i = 0; i < a.size(); i++)
        res[i] = lower_bound(v.begin(), v.end(), a[i]) - v.begin();
    return res;
}
```

莫队

最优大小为$n*m^{-0.5}$

普通莫队

每次先更新右边界，避免出现l大于r的情况

```
const int N = 3e5;
int n, m, len, res = 0;
int w[N], cnt[N], ans[N];

struct Query {
    int qid, l, r;
} q[N];

int get(int i) {
    return i / len;
}

void add(int i) {
    if (!cnt[w[i]]) ++res;
    ++cnt[w[i]];
}

void del(int i) {
    --cnt[w[i]];
    if (!cnt[w[i]]) --res;
}

bool cmp(const Query &a, const Query &b) {
    int la = get(a.l), lb = get(b.l);
    if (la != lb) return la < lb;
    return la & 1 ? a.r < b.r : a.r > b.r;  // 奇偶区块不同方向优化
}

inline void Zlin() {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) cin >> w[i];  // 读入数组

    for (int i = 1; i <= m; i++) {
```

```
        int l, r;
        cin >> l >> r;
        q[i] = {i, l, r};  // 记录查询
    }

    len = sqrt(n) + 1;  // 以 sqrt(n) 为块的大小
    sort(q + 1, q + m + 1, cmp);  // 根据块编号和右端点排序

    res = 0;
    for (int i = 1, l = 1, r = 0; i <= m; i++) {
        // 当前查询范围是 [q[i].l, q[i].r]
        while (r < q[i].r) add(++r);
        while (r > q[i].r) del(r--);
        while (l < q[i].l) del(l++);
        while (l > q[i].l) add(--l);

        ans[q[i].qid] = res;  // 记录答案
    }

    for (int i = 1; i <= m; i++)
        cout << ans[i] << '\n';  // 输出每个查询的结果
}
```

修改莫队

```
struct Query {
    int qid, l, r, cid;
} q[N];

struct Change {
    int p, x;
} c[N];

int cntq = 0, cntc = 0;

int n, m, len, res = 0;
int w[N], cnt[N], ans[N];

int get(int i) {
    return i / len;
}

void add(int i) {
    if (!cnt[i]) ++res;
    ++cnt[i];
}

void del(int i) {
    --cnt[i];
    if (!cnt[i]) --res;
}

bool cmp(const Query &a, const Query &b) {
    int la = get(a.l), ra = get(a.r);
    int lb = get(b.l), rb = get(b.r);
```

```
    if (la != lb) return la < lb;
    if (ra != rb) return ra < rb;
    return a.cid < b.cid;
}

inline void Zlin() {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) cin >> w[i];
    for (int i = 1; i <= m; i++) {
        char op;
        int l, r;
        cin >> op >> l >> r;
        if (op == 'Q') ++cntq, q[cntq] = {cntq, l, r, cntc};
        else c[++cntc] = {l, r};
    }
    len = cbrt((double)n * max(1, cntc)) + 1;
    sort(q + 1, q + cntq + 1, cmp);
    res = 0;
    for (int i = 1, l = 0, r = 0, cid = 0; i <= cntq; i++) {
        while (r < q[i].r) add(w[++r]);
        while (r > q[i].r) del(w[r--]);
        while (l < q[i].l) del(w[l++]);
        while (l > q[i].l) add(w[--l]);
        while (cid < q[i].cid) {
            ++cid;
            if (c[cid].p >= q[i].l && c[cid].p <= q[i].r) {
                del(w[c[cid].p]);
                add(c[cid].x);
            }
            swap(w[c[cid].p], c[cid].x);
        }
        while (cid > q[i].cid) {
            if (c[cid].p >= q[i].l && c[cid].p <= q[i].r) {
                del(w[c[cid].p]);
                add(c[cid].x);
            }
            swap(w[c[cid].p], c[cid].x);
            --cid;
        }
        ans[q[i].qid] = res;
    }
    for (int i = 1; i <= cntq; i++)
        cout << ans[i] << '\n';
}
```

树上莫队

通过欧拉序,将每一个点转换为start和end两个时间戳　 注意:处理链的情况,如果两个没有祖先关系, 记录一点的ed,另一个点的st, 同时要加上他们LCA所产生的贡献,欧拉序是不包含LCA.st

哈希

随机数生成

std::mt19937

这里使用 chrono::steady_clock::now().time_since_epoch().count() 作为随机数种子，这样每次运行代码时种子都会不同，从而保证生成的随机数在不同次运行间不会重复。

```
// 初始化随机数生成器
mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());

// 生成一个64位随机哈希值
ull generateRandomHash() {
    // 返回生成的随机哈希值
    return rng();
}
```

自然溢出

hash[i]=hash[i−1]∗Base+idx(s[i])

数据结构要求ull，不能用ll

单哈希

hash[i]=(hash[i−1]∗Base+idx(s[i]))%MOD

数据结构无要求，可以用ll

双哈希

Base，MOD不同，进行两遍hash

异或哈希

适合无序子集， 类似树同构，求子集种类，找同分异构体

防止被卡操作前先使用mt19937生成随机数异或 然后进行值域离散 之后简单前缀和相加 或者 异或操作

```
const ull mask = mt19937_64(time(nullptr))();

inline ull shift(ull x) {
    x ^= mask;
    x ^= x << 23;
    x ^= x >> 7;
    x ^= x << 17;
    x ^= mask;
    return x;
}
```

洛谷P5043 对于无根树找同构可以先找到重心 *最多可以出现两个所以用pair存两种重心的值* 然后计算重心的hash值进行比较

```
constexpr int N = 55;
```

```cpp
const ull mask = mt19937_64(time(nullptr))();

inline ull shift(ull x) {
    x ^= mask;
    x ^= x << 13;
    x ^= x >> 7;
    x ^= x << 17;
    x ^= mask;
    return x;
}

ull ha[N];
pair<ull, ull> val[N];
vi e[N], rt;
int n, m, siz[N];

inline void findrt(int u, int fa) {
    siz[u] = 1;
    int maxn = 0;
    for (int v: e[u]) {
        if (v == fa) continue;
        findrt(v, u);
        siz[u] += siz[v];
        maxn = max(maxn, siz[v]);
    }
    maxn = max(maxn, n - siz[u]);
    if (maxn <= n / 2) rt.push_back(u);
}

inline void dfs(int u, int fa) {
    siz[u] = 1;
    ha[u] = 1;
    for (int v: e[u]) {
        if (v == fa) continue;
        dfs(v, u);
        ha[u] += shift(ha[v]);
    }
}

inline void Zlin(int id) {
    cin >> n;
    rt.clear();
    for (int i = 0; i <= n; i++) e[i].clear();
    for (int i = 1, x; i <= n; i++) {
        cin >> x;
        if (x) {
            e[x].push_back(i);
            e[i].push_back(x);
        }
    }
    findrt(1, 1);
    vll res;
    dfs(rt[0], rt[0]);
    val[id].first = ha[rt[0]];
    if (rt.size() >= 2) {
        dfs(rt[1], rt[1]);
```

```
        val[id].second = ha[rt[1]];
    } else val[id].second = 0;
    if (val[id].first > val[id].second) swap(val[id].first, val[id].second);
}
```

文件读写操作

std::ios::

数据结构

基础

bitset

进行运算操作时间复杂度为 n/w n为bitset容器长度，w为运行机器编码长度

```
// 定义两个8位的bitset，并通过字符串初始化
bitset<8> b1("1010");   // b1为 00001010
bitset<8> b2("1100");   // b2为 00001100

// 基本操作
cout << "b1: " << b1 << '\n';                          // 输出b1的值
cout << "b2: " << b2 << '\n';                          // 输出b2的值
cout << "b1 size: " << b1.size() << '\n';              // 输出b1的位数（大小）
cout << "b1 count of 1s: " << b1.count() << '\n';// 输出b1中1的个数
cout << "b1 any 1s: " << b1.any() << '\n';             // 检查b1中是否存在至少一个1
cout << "b1 all 1s: " << b1.all() << '\n';             // 检查b1的所有位是否都是1
cout << "b1 none 1s: " << b1.none() << '\n';           // 检查b1的所有位是否都是0

// 位操作
b1.set();                 // 将b1的所有位都设置为1
cout << "b1 after set: " << b1 << '\n';
b1.reset();               // 将b1的所有位都重置为0
cout << "b1 after reset: " << b1 << '\n';
b1.flip();                // 将b1的所有位取反（0变1，1变0）
cout << "b1 after flip: " << b1 << '\n';
b1.set(2);                // 将b1的第2位（从0开始计数）设置为1
cout << "b1 after setting bit 2: " << b1 << '\n';
cout << "b1 test bit 2: " << b1.test(2) << '\n'; // 测试b1的第2位是否为1

// 位运算
cout << "b1 & b2: " << (b1 & b2) << '\n';              // b1 和 b2 的按位与操作
cout << "b1 | b2: " << (b1 | b2) << '\n';              // b1 和 b2 的按位或操作
cout << "b1 ^ b2: " << (b1 ^ b2) << '\n';              // b1 和 b2 的按位异或操作
cout << "~b1: " << (~b1) << '\n';                      // 对b1按位取反
cout << "b1 << 2: " << (b1 << 2) << '\n';              // 将b1左移2位
cout << "b1 >> 2: " << (b1 >> 2) << '\n';              // 将b1右移2位

// 单个位访问与修改
cout << "b1[2]: " << b1[2] << '\n';                    // 访问b1的第2位的值
b1[3] = 1;                                             // 将b1的第3位（从0开始计数）设置为1
cout << "b1 after modifying bit 3: " << b1 << '\n';
```

```
// 转换操作
cout << "b1 to string: " << b1.to_string() << '\n'; // 将b1转换为字符串形式
cout << "b1 to ulong: " << b1.to_ulong() << '\n';   // 将b1转换为无符号长整数


struct cmp {
    bool operator()(const tuple<int, int, int> a, const tuple<int, int, int> b)
const {
        if (get<0>(a) != get<0>(b)) return get<0>(a) < get<0>(b);
        if (get<1>(a) != get<1>(b)) return get<1>(a) < get<1>(b);
        return get<2>(a) < get<2>(b);
    };
};

set<tuple<int, int, int>, cmp> st;
```

64位机

支持区间反转 根据题目自行添加功能

```
struct Bitset {
    ull pool[16000]{};

    Bitset() { memset(pool, 0, sizeof(pool)); }

    ull &operator[](const int i) { return pool[i]; }

    void set(const int i) { pool[i / 64] |= 1ull << i % 64; }

    void flip(const int l, const int r) {
        if (l > r) return;
        if (l / 64 == r / 64) {
            for (int i = l % 64; i <= r % 64; i++) pool[l / 64] ^= 1ull << i;
            return;
        }
        pool[l / 64] ^= ~0ull << (l % 64);
        for (int i = l / 64 + 1; i < r / 64; i++) pool[i] = ~pool[i];
        pool[r / 64] ^= ~0ull >> (63 - r % 64);
    }
} t;

Bitset operator^(Bitset a, Bitset b) {
    Bitset res;
    for (int i = 0; i < 16000; i++) res[i] = a[i] ^ b[i];
    return res;
}

Bitset operator >>(Bitset a, int x) {
    Bitset res;
    int k = x / 64, t = x % 64;
    for (int i = 0; i < 16000 - k; i++) res[i] = a[i + k];
    if (t) {
        for (int i = 0; i < 16000 - 1; i++) {
            res[i] >>= t;
            res[i] ^= res[i + 1] << (64 - t);
```

```
        }
    }
    return res;
}

Bitset operator <<(Bitset a, int x) {
    Bitset res;
    int k = x / 64, t = x % 64;
    for (int i = 0; i < 16000 - k; i++) res[i + k] = a[i];
    if (t) {
        for (int i = 16000 - 1; ~i; i--) {
            res[i] <<= t;
            res[i] ^= res[i - 1] >> (64 - t);
        }
    }
    return res;
}
```

priority_queue

不标注默认大根堆，pair内容先比较第一个元素，然后比较第二个元素

```
//升序队列
priority_queue <int,vector<int>,greater<int> > q;
//降序队列  默认降序
priority_queue <int,vector<int>,less<int> >q;
```

set/multiset

set 自动排序，去重
multiset 自动排序，不去重

```
s.begin(); // 返回set容器的第一个元素的地址（迭代器）
s.end(); // 返回set容器的最后一个元素的地址（迭代器）
s.rbegin(); // 返回逆序迭代器，指向容器元素最后一个位置
s.rend(); // 返回逆序迭代器，指向容器第一个元素前面的位置
s.clear(); // 删除set容器中的所有的元素,返回unsigned int类型O(N)
s.empty(); // 判断set容器是否为空
s.insert(x); // 插入一个元素 O(NlogN)
s.size(); // 返回当前set容器中的元素个数O(1)
s.erase(iterator); //    删除定位器iterator指向的值
s.erase(first, second）; // 删除定位器first和second之间的值
s.erase(key_value); // 删除键值key_value的值O(NlogN) // multiset中是删除这个值的所有元素，要仅删除一个，只能用迭代器
s.find(x); //查找set中的某一元素，有则返回该元素对应的迭代器，无则返回结束迭代器
s.lower_bound(x); //     返回大于等于x的第一个元素的迭代器
s.upper_bound(x); //     返回大于x的第一个元素的迭代器 把这串东西放入代码框，解释加上注释
```

vector

```
vector<int> v;              // 定义空的 int 类型 vector
vector<int> v1(10);         // 初始化大小为 10 的 vector，默认值为 0
vector<int> v2(10, 5);      // 初始化大小为 10 的 vector，每个元素值为 5
vector<int> v3 = {1, 2, 3, 4}; // 使用初始化列表创建 vector
sort(v.begin(), v.end());             // 升序排序
sort(v.begin(), v.end(), greater<int>());  // 降序排序
v.size();         // 返回 vector 的元素个数
v.capacity();     // 返回当前 vector 的容量（可容纳的元素个数）
v.empty();        // 检查 vector 是否为空，返回 true 或 false
v.front();        // 返回第一个元素
v.back();         // 返回最后一个元素
v.push_back(5);       // 在 vector 尾部添加元素 5
v.insert(v.begin(), 10);  // 在第一个位置插入元素 10
v.insert(v.begin() + 2, 7); // 在第 3 个位置插入元素 7
v.pop_back();       // 删除 vector 尾部的元素
v.erase(v.begin() + 1); // 删除第 2 个元素
v.erase(v.begin(), v.begin() + 3); // 删除前 3 个元素
v.clear();          // 清空所有元素
```

stack

```
stk.push(x);  // 将 x 压入栈
stk.pop();  // 移除栈顶元素
stk.empty(); // 判断是否为空
stk.size();  // 获取栈中元素的个数
stk1.swap(stk2);  // 交换 stk1 和 stk2 的内容
```

二进制基础操作

builtin_popcount(x) 统计 x 的二进制表示中 1 的个数（适用于 int 类型） builtin_popcount(5) == 2 (101)

builtin_popcountll(x) 统计 long long 类型的 1 数量 builtin_popcountll(9) == 2 (1001)

builtin_clz(x) 计算 x 的 前导零 个数（适用于 int） builtin_clz(8) == 28 (000...1000)

__builtin_clzll(x) long long 版

builtin_ctz(x) 计算 x 的 后缀零 个数 builtin_ctz(8) == 3 (1000)

__builtin_ctzll(x) long long 版

并查集(DSU)

普通并查集

```
inline int find(int x) { return f[x] == x ? x : f[x] = find(f[x]); }

inline int find(int x) {
    if (f[x] == x)
        return x;
    find(f[x]);
```

```
}
```

种类并查集

根据不同种类开n倍的空间，每个空间存一种关系
每个不同的空间表示一种对立关系，维护一种对立情况

ST表

```
inline void ST_prework() {
    for (int i = 1; i <= n; i++)
        f[i][0] = a[i];
    int t = log(n) / log(2) + 1;
    for (int j = 1; j < t; j++)
        for (int i = 1; i <= n - (1 << j) + 1; i++)
            f[i][j] = max(f[i][j - 1], f[i + (1 << (j - 1))][j - 1]);
}

inline int ST_query(int l, int r) {
    int k = log(r - l + 1) / log(2);
    return max(f[l][k], f[r - (1 << k) + 1][k]);
}
```

树状数组

普通版本

```
#define lowbit(x) (x & (-x))

inline void add(int i, int k) {
    for (; i <= n; i += lowbit(i))
        t[i] += k;
    return;
}

inline int ask(int l, int r) {
    int sum = 0;
    for (; r; r -= lowbit(r))
        sum += t[r];
    --l;
    for (; l; l -= lowbit(l))
        sum -= t[l];
    return sum;
}
```

结构体版本

```
struct Ftree
{
private:
    vi t;
```

```cpp
public:
    void init(int n)
    {
        t.assign(n + 1, 0);
    }

    void upd(int i, int v)
    {
        while (i < t.size())
        {
            t[i] += v;
            i += i & -i;
        }
    }

    int qry1(int i)
    {
        int s = 0;
        while (i > 0)
        {
            s += t[i];
            i -= i & -i;
        }
        return s;
    }

    int qry2(int l, int r)
    {
        return qry1(r) - qry1(l - 1);
    }
} t;
```

树

字典树

```cpp
struct Node {
    unordered_map<char, Node*> nxt;  // 子节点
    int cnt = 0;                     // 以当前节点为结尾的单词个数
    int pre = 0;                     // 以当前节点为前缀的单词个数
};

class Trie {
private:
    Node* root;

public:
    Trie() { root = new Node(); }

    // 插入单词
    void ins(const string& s) {
        Node* cur = root;
        for (char c : s) {
            if (!cur->nxt[c]) cur->nxt[c] = new Node();
```

```
            cur = cur->nxt[c];
            cur->pre++;
        }
        cur->cnt++;
    }

    // 查询单词是否存在
    bool qry(const string& s) {
        Node* cur = root;
        for (char c : s) {
            if (!cur->nxt[c]) return false;
            cur = cur->nxt[c];
        }
        return cur->cnt > 0;
    }

    // 查询前缀是否存在
    bool pre(const string& s) {
        Node* cur = root;
        for (char c : s) {
            if (!cur->nxt[c]) return false;
            cur = cur->nxt[c];
        }
        return cur->pre > 0;
    }

    // 删除单词
    bool del(const string& s) {
        return delHelper(root, s, 0);
    }

    // 查询前缀个数
    int cntPre(const string& s) {
        Node* cur = root;
        for (char c : s) {
            if (!cur->nxt[c]) return 0;
            cur = cur->nxt[c];
        }
        return cur->pre;
    }

private:
    // 删除单词的辅助函数
    bool delHelper(Node* cur, const string& s, int d) {
        if (!cur) return false;
        if (d == s.size()) {
            if (cur->cnt > 0) {
                cur->cnt--;
                cur->pre--;
                return true;
            }
            return false;
        }

        char c = s[d];
        if (delHelper(cur->nxt[c], s, d + 1)) {
```

```
            cur->pre--;
            if (cur->nxt[c]->pre == 0) {
                delete cur->nxt[c];
                cur->nxt.erase(c);
            }
            return true;
        }
        return false;
    }
};
```

线段树

无懒标记懒得写

带懒标记

区间加减

```
struct Tree {
    int l, r, val, tag;
} t[N << 2];

// 建树
void build(int i, int l, int r) {
    if (l == r) {
        t[i].l = l, t[i].r = r;
        t[i].val = w[l];
        return;
    }
    int mid = l + r >> 1;
    build(i << 1, l, mid);
    build(i << 1 | 1, mid + 1, r);
    t[i].l = l, t[i].r = r;
    t[i].val = t[i << 1].val + t[i << 1 | 1].val;
}

//更新懒标记
void pushdown(int i) {
    if (!t[i].tag) return;
    t[i].val += t[i].tag * (t[i].r - t[i].l + 1);
    if (t[i].l != t[i].r) {
        t[i << 1].tag += t[i].tag;
        t[i << 1 | 1].tag += t[i].tag;
    }
    t[i].tag = 0;
}

//区间修改
void modify(int i, int l, int r, int z) {
    if (t[i].l > r || t[i].r < l) return;
    pushdown(i);
    if (t[i].l >= l && t[i].r <= r) {
        t[i].tag += z;
        pushdown(i);
```

```
            return;
        }
        modify(i << 1, l, r, z);
        modify(i << 1 | 1, l, r, z);
        t[i].val = t[i << 1].val + t[i << 1 | 1].val;
}

//区间查询
int query(int i, int l, int r) {
    if (t[i].l > r || t[i].r < l) return 0;
    pushdown(i);
    if (t[i].l >= l && t[i].r <= r) return t[i].val;
    return query(i << 1, l, r) + query(i << 1 | 1, l, r);
}
```

线段树维护区间最大子串价值(单点修区间查)

```
struct STree
{
private:
    struct node
    {
        int l, r;
        ll val, tag;
        ll pre, suf;

        friend node operator +(const node& a, const node& b)
        {
            node res;
            res.l = min(a.l, b.l);
            res.r = max(a.r, b.r);
            res.val = a.val + b.val;
            res.pre = max(a.pre, a.val + b.pre);
            res.suf = max(b.suf, b.val + a.suf);
            res.tag = max({a.tag, b.tag, a.suf + b.pre});
            return res;
        }
    };

    vector<node> t;

    void pushup(int i)
    {
        t[i] = t[i << 1] + t[i << 1 | 1];
    }

    void build(int i, int l, int r)
    {
        t[i].l = l, t[i].r = r;
        if (l == r)
        {
            t[i].val = t[i].tag = t[i].pre = t[i].suf = 0;
            return;
        }
        int mid = l + r >> 1;
```

```
        build(i << 1, l, mid);
        build(i << 1 | 1, mid + 1, r);
        pushup(i);
    }

public:
    void init(int n)
    {
        t.assign(n << 2, {});
        build(1, 1, n);
    }

    void update(int i, int id, int val)
    {
        int lx = t[i].l, rx = t[i].r;
        if (rx < id || lx > id)
            return;
        if (lx == id && rx == id)
        {
            t[i].val += val;
            t[i].tag += val;
            t[i].pre += val;
            t[i].suf += val;
            return;
        }
        update(i << 1, id, val);
        update(i << 1 | 1, id, val);
        pushup(i);
    }

    node query(int i, int l, int r)
    {
        int lx = t[i].l, rx = t[i].r;
        if (lx == l && rx == r)
            return t[i];
        int mid = lx + rx >> 1;
        if (mid >= r)
            return query(i << 1, l, r);
        if (mid + 1 <= l)
            return query(i << 1 | 1, l, r);
        return query(i << 1, l, mid) + query(i << 1 | 1, mid + 1, r);
    }
} t;
```

线段树优化建图

分别创建两颗线段树 第一棵从上往下连val=0的边 第二棵从下往上连val=0的边

对于一个边$s->[l,r]$等价于从第二棵树的$[s,s]$节点连接线第一棵树对应$[l,r]$的节点,价值为这条边的价值,个数最大$log(n)$

对于一个边$[l,r]->[s]$等价于从第二棵树的$[l,r]$对应节点连接线第一棵树$[s,s]$节点,价值为这条边的价值,个数最大$log(n)$

```
struct Dij_Tree {
```

```
struct Node {
    int l, r;
} t[N << 2];

struct edge {
    int to;
    ll val;
};

vector<edge> e[N << 4];

int dif = 5e5;
int idx[N], vis[N << 4];
ll dis[N << 4];


void build(int i, int l, int r) {
    // cout << i << ' ' << l << ' ' << r << endl;
    dis[i] = dis[i + dif] = INF;
    vis[i] = vis[i + dif] = 0;
    t[i].l = l, t[i].r = r;
    if (l == r) {
        // 底边互相连接
        e[i].push_back({i + dif, 0});
        e[i + dif].push_back({i, 0});
        idx[l] = i;
        return;
    }
    // 第一个对应坐标
    e[i].push_back({i << 1, 0});
    e[i].push_back({i << 1 | 1, 0});
    // 第二个对应坐标
    e[(i << 1) + dif].push_back({i + dif, 0});
    e[(i << 1 | 1) + dif].push_back({i + dif, 0});

    int mid = l + r >> 1;
    build(i << 1, l, mid);
    build(i << 1 | 1, mid + 1, r);
}

// op 1 表示从点到线段 点到线段说明是第二棵树的点s 连接 第一棵树的对应node
// op 0 表示从线段到点 线段到点说明是第二棵树的对应node 连接 第一棵树的点s
void update(int i, int l, int r, int s, int val, int op) {
    if (t[i].r < l || t[i].l > r) return;
    if (t[i].l >= l && t[i].r <= r) {
        if (op) e[idx[s] + dif].push_back({i, val});
        else e[i + dif].push_back({idx[s], val});
        return;
    }
    update(i << 1, l, r, s, val, op);
    update(i << 1 | 1, l, r, s, val, op);
}

void dij(int s) {
    priority_queue<pair<ll, int>, vector<pair<ll, int> >, greater<> > pq;
    dis[idx[s]] = 0;
```

```
        pq.emplace(dis[idx[s]], idx[s]);
        while (!pq.empty()) {
            int u = pq.top().second;
            pq.pop();
            if (vis[u]) continue;
            vis[u] = 1;
            // cout << (u >= dif ? u - dif : u) << ' ' << dis[u] << endl;
            for (auto [v, val]: e[u]) {
                if (dis[v] > dis[u] + val) {
                    dis[v] = dis[u] + val;
                    pq.emplace(dis[v], v);
                }
            }
        }
    }

    ll query(int i) { return dis[idx[i]]; }
} t;
```

李超线段树

```
const ll N = 1e6 + 5;
const ll MOD = 998244353;
const ll inf = 0x7fffffff;
const double eps = 1e-12;

struct line {
    db k, b;//斜率和与y轴
    int l, r;
    int tag;
} t[N << 2];

//计算某条线段在某一个横坐标的纵坐标值
db calc(line a, int pos) { return a.k * pos + a.b; }

//求两条线段交点的横坐标
int cross(line a, line b) { return floor((a.b - b.b) / (b.k - a.k)); }

void build(int root, int l, int r) {
    t[root] = {0, 0, 1, 50000, 0};
    if (l == r) return;
    int mid = (l + r) >> 1;
    build(root << 1, l, mid);
    build(root << 1 | 1, mid + 1, r);
}

void modify(int root, int l, int r, line k) {
    if (k.l > r || k.r < l) return;
    if (k.l <= l && r <= k.r) {
        if (!t[root].tag) {
            // 1.这个区间内没有记录有过优势线段：直接把这个区间的势线段修改为这条线段
            t[root] = k;
            t[root].tag = 1;
        } else if (calc(k, l) - calc(t[root], l) > eps && calc(k, r) -
calc(t[root], r) > eps) {
```

```
            // 2.新线段完全覆盖了之前记录的线段：优势线段为新线段，直接赋值替换
            t[root] = k;
        } else if (calc(k, l) - calc(t[root], l) > eps || calc(k, r) -
calc(t[root], r) > eps) {
            // 3.区间内线段有交点的情况：判断哪根线段为优势线段，把区间记录的值给修改一下，然后把短的那一半递归处理
            int mid = (l + r) >> 1;//取出区间的中点
            // 与中点交点更高的一条直线作为优势线段
            if (calc(k, mid) - calc(t[root], mid) > eps) swap(t[root], k);
            if (mid - cross(k, t[root]) > eps) {
                // 交点在中点的左侧，此时老线可能比被标记的优势线段高需要修改
                modify(root << 1, l, mid, k);
            } else {
                // 交点在中点的右侧，同理需要修改右侧区间的优势线段
                modify(root << 1 | 1, mid + 1, r, k);
            }
        }
        return;
    }
    int mid = (l + r) >> 1;
    modify(root << 1, l, mid, k);
    modify(root << 1 | 1, mid + 1, r, k);
}

db query(int root, int l, int r, int x) {
    //由于是标记永久化，查询就比较类似于标记永久化的线段树
    //那么就要从线段树一层层递归，直到递归到某个点
    //每个区间的最优线段的交点取 max
    if (l == r)return calc(t[root], x);
    else {
        int mid = (l + r) >> 1;
        db ans = calc(t[root], x);
        if (x <= mid)return max(ans, query(root << 1, l, mid, x));
        else return max(ans, query(root << 1 | 1, mid + 1, r, x));
    }
}
```

扫描线

```
struct Line {
    double x1, x2, y;
    int type; // +1 表示矩形底边，-1 表示矩形顶边
    Line(double a, double b, double c, int d) : x1(a), x2(b), y(c), type(d) {}
};

bool cmp(const Line &l1, const Line &l2) {
    return l1.y < l2.y;
}

struct Node {
    int cnt;      // 区间被覆盖的次数
    double len;   // 当前区间的总长度
};

vector<double> xs;  // 保存去重后的 x 坐标
```

```
vector<Node> seg;   // 线段树

void build(int p, int l, int r) {
    seg[p].cnt = seg[p].len = 0;
    if (l == r) return;
    int mid = (l + r) / 2;
    build(p * 2, l, mid);
    build(p * 2 + 1, mid + 1, r);
}

void update(int p, int l, int r, int ql, int qr, int v) {
    if (ql > r || qr < l) return;
    if (ql <= l && r <= qr) {
        seg[p].cnt += v;
    } else {
        int mid = (l + r) / 2;
        update(p * 2, l, mid, ql, qr, v);
        update(p * 2 + 1, mid + 1, r, ql, qr, v);
    }

    if (seg[p].cnt > 0) {
        seg[p].len = xs[r + 1] - xs[l];  // 完全覆盖的区间
    } else {
        if (l == r) {
            seg[p].len = 0;
        } else {
            seg[p].len = seg[p * 2].len + seg[p * 2 + 1].len;  // 合并区间
        }
    }
}
```

主席树

求静态区间K小值

```
struct PStree
{
private:
    struct node
    {
        int l, r;
        int val;
        node* ls = nullptr;
        node* rs = nullptr;
    };

    vector<node*> t;

    void pushup(node& t)
    {
        t.val = t.ls->val + t.rs->val;
    }

    void build(node& t, int l, int r)
    {
```

```cpp
    t.l = l, t.r = r;
    if (l == r)
    {
        t.val = 0;
        return;
    }
    t.ls = new node();
    t.rs = new node();
    int mid = l + r >> 1;
    build(*t.ls, l, mid);
    build(*t.rs, mid + 1, r);
    pushup(t);
}

void update(const node& bef, node& now, int k, int val)
{
    now.l = bef.l, now.r = bef.r;
    if (now.l == k && k == now.r)
    {
        now.val = bef.val + val;
        return;
    }
    int mid = now.l + now.r >> 1;
    if (k <= mid)
    {
        now.ls = new node();
        update(*bef.ls, *now.ls, k, val);
        now.rs = bef.rs;
    }
    else
    {
        now.ls = bef.ls;
        now.rs = new node();
        update(*bef.rs, *now.rs, k, val);
    }
    pushup(now);
}

int query(const node now, int k)
{
    if (now.l == now.r)
        return now.l;
    if (now.ls->val >= k)
        return query(*now.ls, k);
    return query(*now.rs, k - now.ls->val);
}

int query1(const node bef, const node now, int k)
{
    if (now.l == now.r)
        return now.l;
    if (now.ls->val - bef.ls->val >= k)
        return query1(*bef.ls, *now.ls, k);
    return query1(*bef.rs, *now.rs, k - now.ls->val + bef.ls->val);
}
```

```
public:
    void init(int n, int val)
    {
        t.resize(n + 1, nullptr);
        t[0] = new node();
        build(*t[0], 1, val);
    }

    void upd(int bef, int now, int k, int val)
    {
        t[now] = new node();
        update(*t[bef], *t[now], k, val);
    }

    int qry(int now, int k)
    {
        return query(*t[now], k);
    }

    int qry1(int bef, int now, int k)
    {
        return query1(*t[bef - 1], *t[now], k);
    }
} t;
```

求区间不重复个数

```
const int maxn = 30010;
struct node {
    int l, r;
    int v;
} T[maxn * 40];
int cnt = 0;
int a[maxn], past[1000100], root[maxn];

void update(int pos, int l, int r, int &cur, int pre, int val) {
    cur = ++cnt;
    T[cur] = T[pre];
    T[cur].v += val;
    if (l == r)return;
    int mid = l + r >> 1;
    if (pos <= mid)
        update(pos, l, mid, T[cur].l, T[pre].l, val);
    else
        update(pos, mid + 1, r, T[cur].r, T[pre].r, val);
}

int query(int L, int R, int l, int r, int rt) {
    if (L == l && r == R)
        return T[rt].v;
    int mid = l + r >> 1;
    if (R <= mid)
        return query(L, R, l, mid, T[rt].l);
    else if (L >= mid + 1)
        return query(L, R, mid + 1, r, T[rt].r);
```

```
    else
        return (query(L, mid, l, mid, T[rt].l) + query(mid + 1, R, mid + 1, r,
T[rt].r));
}

void slove() {
    ll n;
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }
    for (int i = 1; i <= n; i++) {
        if (past[a[i]]) {
            update(past[a[i]], 1, n, root[i], root[i - 1], -1);
            update(i, 1, n, root[i], root[i], 1);
        } else
            update(i, 1, n, root[i], root[i - 1], 1);
        past[a[i]] = i;
    }
    ll q;
    cin >> q;
    while (q--) {
        int l, r;
        cin >> l >> r;
        cout << query(l, r, 1, n, root[r]) << "\n";
    }
}
```

左偏树/可并堆

更适合处理合并工作，合并最坏复杂度 logn

平衡树(Splay)

适合用来维护有序队列

```
struct Node {
    int s[2], p, val, cnt, siz; // s左右儿子,p父节点

    void init(int p1, int val1) {
        p = p1, val = val1;
        cnt = siz = 1;
    }
} t[N];

int root = 0, tot = 0;

inline void pushup(int x) // 更新点x的大小
{
    t[x].siz = t[t[x].s[0]].siz + t[t[x].s[1]].siz + t[x].cnt;
}

inline void rotate(int x) // 旋转x
{
    int y = t[x].p, z = t[y].p;
```

```
    int k = t[y].s[1] == x;
    t[z].s[t[z].s[1] == y] = x;
    t[x].p = z;
    t[y].s[k] = t[x].s[k ^ 1];
    t[t[x].s[k ^ 1]].p = y;
    t[x].s[k ^ 1] = y;
    t[y].p = x;
    pushup(x), pushup(y);
}

inline void splay(int x, int k) {
    while (t[x].p != k) {
        int y = t[x].p, z = t[y].p;
        if (z != k)
            (t[y].s[0] == x) ^ (t[z].s[0] == y) ? rotate(x) : rotate(y);
        rotate(x);
    }
    if (k == 0)
        root = x;
}

inline void find(int val) // 找到权值等于val的点并把它转为根
{
    int x = root;
    while (t[x].s[val > t[x].val] && t[x].val != val)
        x = t[x].s[val > t[x].val];
    splay(x, 0);
}

inline int get_pre(int val) // 求前驱
{
    find(val);
    int x = root;
    if (t[x].val < val)
        return x;
    x = t[x].s[0];
    while (t[x].s[1])
        x = t[x].s[1];
    return x;
}

inline int get_suc(int val) // 求后继
{
    find(val);
    int x = root;
    if (t[x].val > val)
        return x;
    x = t[x].s[1];
    while (t[x].s[0])
        x = t[x].s[0];
    return x;
}

inline void del(int val) {
    int pre = get_pre(val);
    int suc = get_suc(val);
```

```cpp
    splay(pre, 0);
    splay(suc, pre);
    int del = t[suc].s[0];
    if (t[del].cnt > 1)
        --t[del].cnt, splay(del, 0);
    else
        t[suc].s[0] = 0, splay(suc, 0);
}

// 因为预处理插入了两个无穷大和无穷小的数，所以排名不需要+1
inline int get_rank(int val) // 查询val的排名
{
    find(val);
    if (t[root].val < val) // 如果val没有出现过，要判断根节点和val的大小关系
        return t[t[root].s[0]].siz + t[root].cnt;
    return t[t[root].s[0]].siz;
}

// 因为插入了无穷大和无穷小，所以传入k时要+1,k为实际情况的k+1
inline int get_val(int k) // 查询排名为k的val
{
    int x = root;
    while (1) {
        int y = t[x].s[0];
        if (t[x].cnt + t[y].siz < k) {
            k -= t[x].cnt + t[y].siz;
            x = t[x].s[1];
        } else {
            if (t[y].siz >= k)
                x = t[x].s[0];
            else
                break;
        }
    }
    splay(x, 0);
    return t[x].val;
}

inline void insert(int val) {
    int x = root, p = 0;
    while (x && t[x].val != val)
        p = x, x = t[x].s[val > t[x].val];
    if (x)
        ++t[x].cnt;
    else {
        x = ++tot;
        t[p].s[val > t[p].val] = x;
        t[x].init(p, val);
    }
    splay(x, 0);
}
```

字典树

遇到相似题目可以选择离散化，在套入字典树

```
void insert(string s)//建立字典树
{
    int p = 0;//根结点是0
    for(auto it : s)
    {
        ll j;
        if(it >= '0' && it <= '9') j = it - '0';
        else if(it < 'a') j = it - 'A' + 10;
        else j = it - 'a' + 26 + 10;//A从26开始
        // 上面三行是映射字符，将字符变为数字好处理
        if(!ch[p][j]) ch[p][j] = ++ idx;//没有找到
        p = ch[p][j];
        cnt[p] ++;
    }

}

ll query(string s)//查询函数
{
    int p = 0;
    for(auto it : s)
    {
        ll j;
        if(it >= '0' && it <= '9') j = it - '0';//数字
        else if(it < 'a') j = it - 'A' + 10;//A从10开始
        else j = it - 'a' + 26 + 10;//小写字母
        if(!ch[p][j]) return 0;//字节点的编号不是0，如果是0则没有这条边
        p = ch[p][j];
    }
    return cnt[p];
}
```

树链剖分

重链剖分

快速处理一条链上的查询和修改操作

```
const int N = 2e5 + 5;

struct Node {
    int dep, fa, son, siz, val, top, dfn;
} tn[N];

struct edge {
    int to, nxt;
} e[N];

int tot = 0, head[N];


```

```cpp
void add(int u, int v) {
    e[++tot] = {v, head[u]};
    head[u] = tot;
}


// 预处理，找出树的所有重儿子和重链
void dfs1(int u, int f) {
    tn[u].fa = f;
    tn[u].dep = tn[f].dep + 1;
    tn[u].siz = 1;
    int tmp = -1; // 临时变量，用来存储结点u的重儿子
    for (int i = head[u], v; i; i = e[i].nxt) {
        v = e[i].to;
        if (v == f) continue;
        dfs1(v, u);
        tn[u].siz += tn[v].siz;
        if (tn[v].siz > tmp) { // 如果结点v.siz更大，更新u的重儿子为v
            tn[u].son = v;
            tmp = tn[v].siz;
        }
    }
}

int tim = 0, w[N]; // w用来存储对应dfn序下的树上结点val，tim为dfn计数器

void dfs2(int u, int top) {
    tn[u].top = top;
    tn[u].dfn = ++tim;
    w[tim] = tn[u].val;
    if (!tn[u].son) return; // 如果没有重儿子，说明为叶节点
    dfs2(tn[u].son, top); // 向下传递重链，重链的top一样
    for (int i = head[u], v; i; i = e[i].nxt) {
        v = e[i].to;
        if (v == tn[u].fa || v == tn[u].son) continue;
        dfs2(v, v); // 轻链的top为他自身
    }
}

// 详细见线段树-带懒标记区间修改
void modify(int i, int l, int r, int z) {
    // 线段树区间修改，省略，线段树每一位对应的是dfn
}

int query(int i, int l, int r) {
    // 线段树区间查询
}

// 修改结点u和他的子树，因为dfn连续，所以映射在线段树上是区间修改
void change_1(int u, int z) {
    modify(1, tn[u].dfn, tn[u].dfn + tn[u].siz - 1, z);
}

// 同理
int query_1(int u) {
    return query(1, tn[u].dfn, tn[u].dfn + tn[u].siz - 1);
```

```
}

// 修改一条链上的所有值，重链上的dfn都是连续的
void change_2(int x, int y, int z) {
    while (tn[x].top != tn[y].top) { // 如果他两不在同一条重链上，找出top深度大的，往上翻
        if (tn[tn[x].top].dep < tn[tn[y].top].dep) swap(x, y);
        modify(1, tn[tn[x].top].dfn, tn[x].dfn, z);
        x = tn[tn[x].top].fa;
    }
    if (tn[x].dep > tn[y].dep) swap(x, y);
    modify(1, tn[x].dfn, tn[y].dfn, z);
}

// 查询一条链上的所有值，同理
int query_2(int x, int y) {
    int res = 0;
    while (tn[x].top != tn[y].top) { // 如果他两不在同一条重链上，找出top深度大的，往上翻
        if (tn[tn[x].top].dep < tn[tn[y].top].dep) swap(x, y);
        res += query(1, tn[tn[x].top].dfn, tn[x].dfn);
        x = tn[tn[x].top].fa;
    }
    if (tn[x].dep > tn[y].dep) swap(x, y);
    res += query(1, tn[x].dfn, tn[y].dfn);
    return res;
}
```

长链剖分

```
```

实链剖分

```
```

分块

普通分块

```
inline void init()
{
   int len = sqrt(n), tot = (n - 1) / len + 1;
   for (int i = 1; i <= tot; i++)
        l[i] = r[i - 1] + 1, r[i] = i * len;
   r[tot] = n;
   for (int i = 1; i <= tot; i++)
        for (int j = l[i]; j <= r[i]; j++)
            belong[j] = i;
}
```

时间分块

数学

基础

cbrt() 返回立方根

Ceil 向上取整

floor 向下取整

2ab = $(a+b)^2-a^2-b^2$

2ab+2ac+2bd = $(a+b+c)^2-a^2-b^2-c^2$

多元同理

大数不能直接用sqrt，要自己用二分查找求值

叉积

AB*AC小于零说明AB能顺时针旋转到AC，大于零说明逆时针
pi = 3.14159265358979323846

cout保留几位小数
cout << fixed << setprecision(12) << ans << '\n';

裴蜀定理

对于任意整数a, m 不全为0

$$a \cdot x + m \cdot y = gcd(a, m)$$

极角排序

o表示原点

```
bool cmp(node a, node b) {
    if (cross(o, a, b) == 0) return a.x < b.x;
    return cross(o, a, b) > 0;
}
```

扩展欧几里得公式

$$\gcd(a,b)=\gcd(b, a \mod b)$$
$$ax+by=\gcd(a,b) \quad \Rightarrow \quad a \mod b = a - k \cdot b \, (k = \lfloor a/b \rfloor)$$

递归到更小的子问题后，可以逐步构造出 x 和 y 。 同时可以求一个值mod另一个值的逆元

```cpp
// 扩展欧几里得算法，返回 gcd(a, b)，并且计算出 x 和 y
// 使得 ax + by = gcd(a, b)
int exgcd(int a, int b, int &x, int &y) {
    if (!b) {
        x = 1, y = 0;
        return a;
    }
    int x1, y1, gcd = exgcd(b, a % b, x1, y1);
    x = y1, y = x1 - a / b * y1;
    return gcd;
}
```

位运算

前缀和异或：从0 ~ x连续异或的结论

$$f(x) = \begin{cases} x, & \text{if } x \mod 4 = 0 \\ 1, & \text{if } x \mod 4 = 1 \\ x+1, & \text{if } x \mod 4 = 2 \\ 0, & \text{if } x \mod 4 = 3 \end{cases}$$

面积计算

三角形计算:
海伦公式:

$$A = \sqrt{s(s-a)(s-b)(s-c)} \ (s = \frac{a+b+c}{2})$$

素数

```cpp
vi primes;
vector<bool> is_p;

inline void init(int n) {
    is_p.assign(n + 1, true);
    is_p[0] = is_p[1] = false;
    for (int i = 2; i <= n; i++) {
        if (is_p[i]) primes.push_back(i);
        for (int j: primes) {
            if (i * j > n) break;
            is_p[i * j] = false;
            if (i % j == 0) break;
        }
    }
}
```

FFT

快速计算多项式乘法/大数乘法

主体

```
const double PI = acos(-1.0);

struct Cp {
    double r, i;

    Cp(double _r = 0.0, double _i = 0.0) : r(_r), i(_i) {}

    Cp operator+(const Cp &o) const {
        return Cp(r + o.r, i + o.i);
    }

    Cp operator-(const Cp &o) const {
        return Cp(r - o.r, i - o.i);
    }

    Cp operator*(const Cp &o) const {
        return Cp(r * o.r - i * o.i, r * o.i + i * o.r);
    }
};

// 进行 FFT 或 IFFT，d == 1 表示 FFT，d == -1 表示 IFFT
void fft(vector<Cp> &a, int n, int d) {
    for (int p = 1, q = 0; p < n - 1; p++) {
        for (int k = n >> 1; (q ^= k) < k; k >>= 1);
        if (p < q) swap(a[p], a[q]);
    }
    for (int m = 2; m <= n; m <<= 1) {
        Cp wm(cos(2 * PI / m), sin(d * 2 * PI / m));
        for (int p = 0; p < n; p += m) {
            Cp w(1, 0);
            for (int j = 0; j < m / 2; j++) {
                Cp u = a[p + j];
                Cp t = w * a[p + j + m / 2];
                a[p + j] = u + t;
                a[p + j + m / 2] = u - t;
                w = w * wm;
            }
        }
    }
    if (d == -1) {
        for (int p = 0; p < n; p++) {
            a[p].r /= n;
            a[p].i /= n;
        }
    }
}
```

大数乘法

```cpp
// 大数乘法主函数
vector<int> multiply(const vector<int>& A, const vector<int>& B) {
    int n = 1;
    while (n < A.size() + B.size()) n <<= 1;  // 找到大于等于 A.size() + B.size() 的最小 2 的幂
    vector<Cp> a(n), b(n);

    for (int p = 0; p < A.size(); p++) a[p] = Cp(A[p], 0);
    for (int p = 0; p < B.size(); p++) b[p] = Cp(B[p], 0);

    fft(a, n, 1);
    fft(b, n, 1);

    for (int p = 0; p < n; p++) a[p] = a[p] * b[p];  // 点乘
    fft(a, n, -1);

    vector<int> res(n);
    for (int p = 0; p < n; p++) res[p] = int(a[p].r + 0.5);  // 四舍五入取整
    for (int p = 0; p < n - 1; p++) {
        res[p + 1] += res[p] / 10;  // 处理进位
        res[p] %= 10;
    }
    while (res.size() > 1 && res.back() == 0) res.pop_back();  // 去掉前导0
    return res;
}
```

多项式乘法

```cpp
// 多项式乘法
vector<int> multiply(const vector<int> &A, const vector<int> &B) {
    int n = 1;
    while (n < A.size() + B.size()) n <<= 1;  // 取大于等于 A.size() + B.size() 的最小2的幂
    vector<Cp> a(n), b(n);

    for (int p = 0; p < A.size(); p++) a[p] = Cp(A[p], 0);
    for (int p = 0; p < B.size(); p++) b[p] = Cp(B[p], 0);

    // 进行 FFT 变换
    fft(a, n, 1);
    fft(b, n, 1);

    // 点乘：每个位置上的系数相乘
    for (int p = 0; p < n; p++) a[p] = a[p] * b[p];

    // 逆 FFT 变换
    fft(a, n, -1);

    // 提取结果并处理进位
    vector<int> res(n);
    for (int p = 0; p < n; p++)
        res[p] = round(a[p].r);
```

```
    return res;
}
```

NTT

主体

受模数的限制，数也比较大，但精度不易缺失

```
const int MOD = 998244353;  // 质数模数 p
const int G = 3;            // 原根 g

// 快速幂计算 a^b % mod
int mod_pow(int a, int b, int mod) {
    int res = 1;
    while (b > 0) {
        if (b % 2 == 1) res = 1LL * res * a % mod;
        a = 1LL * a * a % mod;
        b /= 2;
    }
    return res;
}

// NTT 核心函数
void ntt(vector<int> &a, int n, int inv) {
    // 二进制反转置换
    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        while (j >= bit) {
            j -= bit;
            bit >>= 1;
        }
        j += bit;
        if (i < j) swap(a[i], a[j]);
    }

    // 进行 NTT
    for (int len = 2; len <= n; len <<= 1) {
        int wlen = inv == 1 ? mod_pow(G, (MOD - 1) / len, MOD) :
mod_pow(mod_pow(G, (MOD - 1) / len, MOD), MOD - 2, MOD);
        for (int i = 0; i < n; i += len) {
            int w = 1;
            for (int j = 0; j < len / 2; j++) {
                int u = a[i + j];
                int v = 1LL * a[i + j + len / 2] * w % MOD;
                a[i + j] = (u + v) % MOD;
                a[i + j + len / 2] = (u - v + MOD) % MOD;
                w = 1LL * w * wlen % MOD;
            }
        }
    }

    // 如果是逆变换，需要除以 n (即乘以 n 的逆元)
    if (inv == -1) {
```

```
        int n_inv = mod_pow(n, MOD - 2, MOD);
        for (int &x : a) x = 1LL * x * n_inv % MOD;
    }
}
```

多项式求逆

```
// 多项式乘法
vector<int> poly_mult(const vector<int> &a, const vector<int> &b) {
    int n = 1;
    while (n < a.size() + b.size()) n <<= 1;

    vector<int> A(a.begin(), a.end()), B(b.begin(), b.end());
    A.resize(n);
    B.resize(n);

    ntt(A, false);
    ntt(B, false);

    for (int i = 0; i < n; i++)
        A[i] = (1LL * A[i] * B[i]) % MOD;

    ntt(A, true);

    return A;
}

// 多项式求逆
vector<int> poly_inv(const vector<int> &a) {
    int n = a.size();
    vector<int> res(1, pow_mod(a[0], MOD - 2)); // 初始逆多项式为 a[0] 的逆元

    for (int len = 1; len < n; len *= 2) {
        vector<int> temp(res.begin(), res.end());
        temp.resize(2 * len);
        vector<int> mult = poly_mult(temp, a);
        for (int i = 0; i < len; i++) {
            res.push_back((2LL * res[i] - mult[i] + MOD) % MOD); // 更新逆多项式
        }
    }
    return res;
}
```

如何求原根

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

// 快速幂
long long mpow(long long b, long long e, long long m) {
    long long r = 1;
    while (e) {
        if (e & 1) r = (r * b) % m;
```

```
        b = (b * b) % m;
        e >>= 1;
    }
    return r;
}

// 查找原根
long long g_r(long long p) {
    long long p1 = p - 1;
    vector<long long> f;

    // 找到 p-1 的质因数
    for (long long i = 2; i * i <= p1; i++) {
        if (p1 % i == 0) {
            f.push_back(i);
            while (p1 % i == 0) p1 /= i;
        }
    }
    if (p1 > 1) f.push_back(p1);

    // 寻找原根
    for (long long g = 2; g < p; g++) {
        bool is_r = true;
        for (long long q : f) {
            if (mpow(g, (p - 1) / q, p) == 1) {
                is_r = false;
                break;
            }
        }
        if (is_r) return g;
    }
    return -1; // 如果没有找到
}

int main() {
    long long p = 7; // 可以替换为任意素数
    cout << "Primitive root of " << p << " is: " << g_r(p) << endl;
    return 0;
}
```

计算几何

高斯面积计算公式

$$A = \frac{1}{2}\left|\sum_{i=1}^{n-1}(x_i y_{i+1} - x_{i+1} y_i) + (x_n y_1 - x_1 y_n)\right|$$

计算向量夹角

计算坐标系中两个线段之间的夹角

$$\cos\theta = \frac{\mathbf{v_1}\cdot\mathbf{v_2}}{|\mathbf{v_1}||\mathbf{v_2}|}$$
$$\mathbf{v_1}\cdot\mathbf{v_2} = v_{1x}\cdot v_{2x} + v_{1y}\cdot v_{2y} + v_{1z}\cdot v_{2z}$$
$$|\mathbf{v_1}| = \sqrt{v_{1x}^2 + v_{1y}^2 + v_{1z}^2}$$
$$|\mathbf{v_2}| = \sqrt{v_{2x}^2 + v_{2y}^2 + v_{2z}^2}$$

叉积

$$\text{如果叉积为正 } (\mathbf{A}\times\mathbf{B} > 0)\text{：表示向量 }\mathbf{B}\text{ 在向量 }\mathbf{A}\text{ 的逆时针方向。}$$
$$\text{如果叉积为负 } (\mathbf{A}\times\mathbf{B} < 0)\text{：表示向量 }\mathbf{B}\text{ 在向量 }\mathbf{A}\text{ 的顺时针方向。}$$
$$\text{如果叉积为零 } (\mathbf{A}\times\mathbf{B} = 0)\text{：表示向量 }\mathbf{A}\text{ 和 }\mathbf{B}\text{ 是共线的（即它们在同一直线上）。}$$
$$\mathbf{A}\times\mathbf{B} = \begin{vmatrix}\mathbf{i} & \mathbf{j} & \mathbf{k}\\ A_x & A_y & A_z\\ B_x & B_y & B_z\end{vmatrix} = (A_yB_z - A_zB_y)\mathbf{i} - (A_xB_z - A_zB_x)\mathbf{j} + (A_xB_y - A_yB_x)\mathbf{k}(\text{三维坐标系})$$

$$\mathbf{A}\times\mathbf{B} = A_xB_y - A_yB_x\text{（二维坐标系）}$$

构建凸包

```cpp
struct P {
    int x, y;

    // 比较函数，先按 x 排序，若 x 相同则按 y 排序
    bool operator<(const P &p) const {
        return x < p.x || (x == p.x && y < p.y);
    }
};

// 计算向量 cross product (AB × AC)，用于判断点的相对位置
int cross(const P &a, const P &b, const P &c) {
    return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
}

// 求凸包
vector<P> convexHull(vector<P> &pts) {
    int n = pts.size();
    if (n < 3) return pts;  // 点数小于3无法构成凸包

    // 先对点集进行排序
    sort(pts.begin(), pts.end());

    vector<P> h;

    // 构建下半凸包
    for (int i = 0; i < n; ++i) {
        while (h.size() >= 2 && cross(h[h.size() - 2], h.back(), pts[i]) <= 0) {
            h.pop_back();  // 移除不满足凸包性质的点
        }
        h.push_back(pts[i]);
    }

    // 构建上半凸包
```

```
    int t = h.size() + 1;  // 记录下半部分点的个数
    for (int i = n - 1; i >= 0; --i) {
        while (h.size() >= t && cross(h[h.size() - 2], h.back(), pts[i]) <= 0) {
            h.pop_back();  // 移除不满足凸包性质的点
        }
        h.push_back(pts[i]);
    }

    h.pop_back();  // 移除最后一个点，因为它在上下两部分中都出现了
    return h;
}
```

旋转卡壳

旋转卡壳，求凸包的直径，可以处理三点共线

```
struct P {
    double x, y;
};

// 计算两点之间的欧几里得距离
double dist(const P &p1, const P &p2) {
    return sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));
}

// 计算向量叉积
double cross(const P &o, const P &a, const P &b) {
    return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
}

// 使用旋转卡壳算法求凸包的直径（最远点对距离）
double rotCalipers(const vector<P> &h) {
    int n = h.size();
    if (n == 1) return 0.0;
    if (n == 2) return dist(h[0], h[1]);

    int k = 1;
    double maxD = 0.0;
    for (int i = 0; i < n; ++i) {
        while (abs(cross(h[i], h[(i + 1) % n], h[(k + 1) % n])) > abs(cross(h[i],
h[(i + 1) % n], h[k]))) {
            k = (k + 1) % n;
        }
        maxD = max(maxD, dist(h[i], h[k]));
        maxD = max(maxD, dist(h[(i + 1) % n], h[k]));
    }
    return maxD;
}
```

线性基

可以插入也可以删除

异或线形基

最后求出来k个答案，要注意0的情况，如果k!=n+1说明存在0

模版

```
struct LinearBasis {
    ll basis[62];
    bool zero;

    LinearBasis() {
        memset(basis, 0, sizeof(basis));
        zero = false;
    }

    bool insert(ll x) {
        for (int i = 60; ~i; i--) {
            if (x >> i & 1) {
                if (!basis[i]) {
                    basis[i] = x;
                    return true;
                }
                x ^= basis[i];
            }
        }
        zero = true;
        return false;
    }

    // 查询异或最大值
    ll query_max() {
        ll res = 0;
        for (int i = 60; ~i; i--)if ((res ^ basis[i]) > res) res ^= basis[i];
        return res;
    }

    // 查询异或最小非 0 值
    ll query_min() {
        for (int i = 0; i <= 60; i++) if (basis[i]) return basis[i];
        return 0; // 如果都是 0
    }
} linear;
```

高斯消元法

```cpp
inline void gauss() {
    for (int i = 63; ~i && k <= n; i--) {
        for (int j = k; j <= n; j++)
            if (a[j] & (1ll << i)) {
                swap(a[j], a[k]);
                break;
            }
        if (!(a[k] & (1ll << i))) continue;
        for (int j = 1; j <= n; j++)
            if (j != k && (a[j] & (1ll << i))) a[j] ^= a[k];
        ++k;
    }
}
```

区间线性基

更新当前位置永远保证是最右一位

```cpp
inline void insert(int x, int id) {
    int t = id;
    for (int i = 30; ~i; i--) {
        p[id][i] = p[id - 1][i];
        pos[id][i] = pos[id - 1][i];
    }
    for (int i = 30; ~i; i--) {
        if (!(x & (1 << i))) continue;
        if (!p[id][i]) {
            p[id][i] = x;
            pos[id][i] = t;
            return;
        } else if (pos[id][i] < t) {
            swap(p[id][i], x);
            swap(pos[id][i], t);
        }
        x ^= p[id][i];
    }
}
// 最大值高位可能与地位冲突，要比较是否更大
int query_max(int l, int r)
{
    int ans = 0;
    for(int i = 30; ~i; i--)
        if(pos[r][i] >= l && (ans ^ p[r][i]) > ans)
            ans ^= p[r][i];
    return ans;
}

int query_min(int l, int r)
{
    for(int i = 0; i <= 30; i++)
        if(pos[r][i] >= l && p[r][i])
            return p[r][i];
    return 0;
```

```
}
```

高精度

模版

```
struct BigInt
{
    vi now; // 按位存储 低位在前 高位在后
    bool tag = false; // 判断是否是负数

    void init(string s)
    {
        int l = 0, r = s.size() - 1;
        if (s[0] == '-')
            tag = true, l = 1;
        while (r >= l)
            now.push_back(s[r--] - '0');
        trim(now);
    }

    // 清除前导零
    void trim(vi& a)
    {
        while (a.back() == 0)
            a.pop_back();
        if (a.empty())
            a.push_back(0), tag = false;
    }

    // 比较绝对值大小
    bool checkabs(const BigInt& a, const BigInt& b)
    {
        if (a.now.size() != b.now.size())
            return a.now.size() > b.now.size();
        for (int i = a.now.size() - 1; i >= 0; i--)
            if (a.now[i] != b.now[i])
                return a.now[i] > b.now[i];
        return true;
    }

    // 加法
    BigInt add(const BigInt& a, const BigInt& b)
    {
        BigInt res;
        res.tag = a.tag;
        int now = 0;
        for (int i = 0; i < max(a.now.size(), b.now.size()) || now; i++)
        {
            int sum = now;
            if (i < a.now.size())
                sum += a.now[i];
            if (i < b.now.size())
                sum += b.now[i];
```

```
            res.now.push_back(sum % 10);
            now = sum / 10;
        }
        trim(res.now);
        return res;
    }

    // 减法
    BigInt sub(const BigInt& a, const BigInt& b)
    {
        BigInt res;

    }
};
```

加法

> 存储数据
> lenc = max(lena,lenb)，字符串读取输入，翻转存入数组
>
> ```
> int a[N], b[N], c[N];
> int lena, lenb, lenc;
> ```
>
> 相加操作
>
> ```
> vector<int> add(vector<int>& A, vector<int>& B)
> {
>     //如果B更大，因为下面代码都是以第一个形参作为for结束条件，所以要让大的是第一个形参
>     if (A.size() < B.size()) return add(B, A);
>
>     vector<int> C;
>     int t = 0; //用来判断是否进位
>     //注意这里是逆序的数从前往后加的
>     for (int i = 0; i < A.size(); ++i)
>     {//for循环是以大的数来作为循环结束条件的
>         t += A[i];//for循环以A为结束条件，这里不用格外判断
>         if (i < B.size()) t += B[i];//因没以B为结束条件，故这里要格外判断是否可以加
>         C.push_back(t % 10);//加出来的数要的是余数
>         t /= 10; //判断是否有进位
>     }
>     if (t) C.push_back(t);//有可能最后加完还有进位
>
>     return C;
> }
> ```

减法

> a - b
>
> ```
> vector<int> sub(vector<int>& A, vector<int>& B)
> {//利用cmp函数比较，使大的数一定是A，与for循环代码相符
>
>     vector<int> C;
>     int t = 0;//判断借位
> ```

```
    for (int i = 0; i < A.size(); ++i)
    {
        t = A[i] - t;//每次都会减掉借位
        if (i < B.size()) t -= B[i];
        //关于(t+10)%10（t是减出来的数）
        //t若为正数(但<=9)其=t%10+10%10=t
        //t若为负数，正好可以借位+10然后取余数即可
        C.push_back((t + 10) % 10);
        if (t >= 0) t = 0;
        else t = 1; //<0肯定有借位了
    }
    //因为两个数相减会导致有多余的0出现，故去除前导0
//size()>1是因为可能真的相减出现0，这种0不算前导0
    while (C.size() > 1 && C.back() == 0) C.pop_back();

    return C;
}
```

乘法

> 从低位到高位，先累加乘积，然后进位，存余

```
vector<int> mul(vector<int>& A, int b)
{
    vector<int> C;
    for (int i = 0, t = 0; i < A.size() || t; ++i)
    {
        if (i < A.size()) t += A[i] * b;//加上t是因为上一次可能有乘出来的进位
        C.push_back(t % 10);
        t /= 10;//计算进位
    }
    //当b是0时，会出现前导0
    while (C.size() > 1 && C.back() == 0) C.pop_back();

    return C;
}
```

除法

> ***从高位到低位***
> 大数a除以小数b，r保存余数

```
vector<int> div(vector<int>& A, int b, int& r)
{
    vector<int> C;
    for (int i = A.size() - 1; i >= 0; --i)
    {
        r = r * 10 + A[i];
        C.push_back(r / b);
        r %= b; //计算余数
    }
    //逆置:因为我们是正常求，但最后是倒着读的，且便于去除前导0
    reverse(C.begin(), C.end());
    while (C.size() > 1 && C.back() == 0) C.pop_back();
```

```
    return C;
}
```

比较大小

```
//比较哪个数大，注意这里的数是从倒序存的，故后面的才是高位
bool cmp(vector<int>& A, vector<int>& B)
{
    if (A.size() != B.size()) return A.size() > B.size();
    for (int i = A.size() - 1; i >= 0; --i)
        if (A[i] != B[i])
            return A[i] > B[i];//不等从高位开始比

    return true;//相等
}
```

快速幂

快速求a^n的值

```
// 快速幂函数：计算 a^b % mo
ll qpow(ll a, ll b) {
    ll res = 1;
    while (b) {
        if (b & 1) res = res * a % mo;
        a = a * a % mo;
        b >>= 1;
    }
    return res;
}
```

GCD

> 利用更减相损术和builtin内置函数，二进制运算速度更快
>
> ```
> int qGCD(int a, int b)
> {
>     int az = __builtin_ctz(a), bz = __builtin_ctz(b); // 如果数据ll，用函数ctzll
>     int z = min(az, bz), dif;
>     b >>= bz;
>     while (a)
>     {
>         a >>= az;
>         dif = abs(b - a);
>         az = __builtin_ctz(dif);
>         if (a < b)
>             b = a;
>         a = dif;
>     }
>     return b << z;
> }
> ```

$gcd(x,y)=1, x+y=n$ 求x，y对数，欧拉函数

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

EXGCD

求 `ax + by == gcd(a, b)`

```
ll exgcd(ll a, ll b, ll &x, ll &y) {
    if (!b) {
        x = 1, y = 0;
        return a;
    }
    ll x1, y1;
    ll g = exgcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - a / b * y1;
    return g;
}
```

$$A \cdot x \equiv B \pmod{M}$$

$$g = \gcd(A, M)$$

$$\frac{A}{g} \cdot x \equiv \frac{B}{g} \pmod{\frac{M}{g}}$$

$$\text{最小循环节} t = \frac{M}{g}$$

逆元

费马定理

给定两个数a,p，p为质数，a^{p-2}为a模p的乘法逆元

```
// 快速幂函数：计算 a^b % mo
ll qpow(ll a, ll b) {
    ll res = 1;
    while (b) {
        if (b & 1) res = res * a % mo;
        a = a * a % mo;
        b >>= 1;
```

```
    }
    return res;
}

// 费马小定理求逆元：a 的逆元 % mo
ll inv(ll a) {
    return qpow(a, mo - 2);
}
```

递推逆元

递推逆元：如果你需要在区间 [1, n] 内计算逆元，可以使用递推的方式
设 inv[1] = 1 。对于 i大于2小于n的区间

$$inv[i] = (mod - (mod/i) \times inv[mod\%i]) \mod mod$$

预处理法,乘法逆元

适用范围：n，m在1e5以内，且取模的数mod为素数时
利用快速幂求逆元

```
inline void init() // 预处理，fac[]表示阶乘, inf[]表示阶乘的逆元
{
    fac[0] = inf[0] = 1;
    for (int i = 1; i <= N; i++) {
        fac[i] = fac[i - 1] * i % mod;
        inf[i] = inf[i - 1] * quick_pow(i, mod - 2) % mod;
    }
}
```

组合数

基础

$$\sum_{i=0}^{n} \binom{n}{i} \cdot i = n \cdot 2^{n-1}$$

$$\sum_{i=0}^{n} \binom{n}{i} \cdot i^2 = n(n+1) \cdot 2^{n-2}$$

直接定义公式法

组合数公式 C(n, k) 的定义为:

$$C(n,k) = \frac{n!}{k!(n-k)!}$$

其中 (n!) 表示 (n) 的阶乘。这个方法可以用递推或循环计算阶乘，然后利用公式求出组合数。

递推公式法 (Pascal’s Triangle)

$$C(n,k)=C(n-1,k-1)+C(n-1,k)$$

```
const int N = 1000; // 定义最大 N 值 适合N<5e3的情况
long long C[N+1][N+1];

void init_comb() {
    for (int i = 0; i <= N; ++i) {
        C[i][0] = C[i][i] = 1; // 边界条件
        for (int j = 1; j < i; ++j) {
            C[i][j] = C[i-1][j-1] + C[i-1][j]; // 递推公式
        }
    }
}
```

逆元法 (费马小定理)

对于模 (p) (质数) 的组合数计算，利用费马小定理可以高效求组合数。公式为：

$$C(n,k)=\frac{n!}{k!(n-k)!}\mod p$$

利用费马小定理求逆元：

$$a^{-1}\equiv a^{p-2}\mod p$$

这样可以通过预处理阶乘和逆元，快速求出组合数。

```
const int N = 100000; // 定义最大 N 值
const ll mo = 1e9 + 7;
ll fact[N + 1], inv[N + 1];

// 快速幂求 a^b % mo
ll qpw(ll a, ll b) {
    long long res = 1;
    while (b) {
        if (b % 2 == 1) res = res * a % mo;
        a = a * a % mo;
        b /= 2;
    }
    return res;
}

// 预处理阶乘和逆元
void init_fact() {
    fact[0] = inv[0] = 1;
    for (int i = 1; i <= N; ++i) {
        fact[i] = fact[i - 1] * i % mo;
    }
    inv[N] = qpw(fact[N], mo - 2); // 利用费马小定理求逆元
    for (int i = N - 1; i >= 1; --i) {
        inv[i] = inv[i + 1] * (i + 1) % mo;
    }
}j
```

```
// 快速求组合数
ll comb(int n, int k) {
    if (k > n || k < 0) return 0;
    return fact[n] * inv[k] % mo * inv[n - k] % mo;
}
```

逐项计算法

$$[C(n,k) = \frac{n \times (n-1) \times \cdots \times (n-k+1)}{k \times (k-1) \times \cdots \times 1}]$$

```
ll comb(int n, int k) { // 避免溢出
    if (k > n) return 0;
    long long res = 1;
    for (int i = 1; i <= k; ++i) {
        res = res * (n - i + 1) / i;
    }
    return res;
}
```

Lucas 定理

对于较大的 (n) 和 (k)，在模 (p) 的情况下，可以使用 Lucas 定理计算组合数。当 (n) 和 (k) 非常大，但 (p) 是质数时，Lucas
定理是一种有效的求解方法。
将 (n) 和 (k) 分解成模 (p) 的系数来递归计算组合数：

$$[C(n,k) \mod p = C(n \mod p, k \mod p) \times C(n/p, k/p) \mod p]$$

图论

基础

奇数完全图的欧拉路径等于他的所有边，欧拉路径要求图中奇数度的定点不超过二

圆方树

对于每一块个点双,建一个方点连接这个点双的所有圆点

```
void tarjan(int u) {
    dfn[u] = low[u] = ++tot;
    stk[++top] = u;
    for (int v: e[u]) {
        if (!dfn[v]) {
            tarjan(v);
            low[u] = min(low[u], low[v]);
            if (low[v] >= dfn[u]) {
                int nx = ++Tree::n, vx;
                do {
                    vx = stk[top--];
                    Tree::add(nx, vx);
                } while (vx != v);
```

```
                Tree::add(nx, u);
            }
        } else low[u] = min(low[u], dfn[v]);
    }
}
```

哈蜜顿路径

状压DP

mask按位存经过的点
例如：mask=5 换成二进制位 101 说明节点2和0已经经过。 mask=10 换成二进制 1010 说明节点3和1已经经过

```
// 全局变量定义
int n, dp[1 << N][N]; // n: 节点数量，dp: 动态规划表
vi e[N]; // 邻接表

// 初始化函数
void init() {
    // 将 dp 表初始化为 -inf，表示未访问的状态
    for (int i = 0; i < (1 << n); i++)
        memset(dp[i], -inf, sizeof(dp[i])); // 每个状态都初始化为 -inf
    for (int i = 0; i < n; i++)
        e[i].clear(); // 清空邻接表
}

// 哈密顿路径计算函数
int hmd() {
    // 从每个节点作为起点初始化
    for (int i = 0; i < n; i++)
        dp[1 << i][i] = 1; // 每个节点的状态设置为可访问，路径长度为1

    // 遍历所有状态和节点
    for (int mask = 1; mask < (1 << n); mask++) {
        for (int u = 0; u < n; u++) {
            if (dp[mask][u] == -inf) continue; // 如果状态不可达，跳过

            // 遍历与节点 u 相邻的所有节点 v
            for (int v: e[u]) {
                if (mask & (1 << v)) continue; // 如果 v 已访问，跳过
                int newMask = mask | (1 << v); // 更新状态，标记节点 v 为已访问
                dp[newMask][v] = max(dp[newMask][v], dp[mask][u] + 1); // 更新经过节点 v 的最大路径长度
            }
        }
    }

    int res = 1; // 至少会有一个节点
    // 查找经过所有节点的最大路径
    for (int i = 0; i < n; i++)
        res = max(res, dp[(1 << n) - 1][i]); // 更新最终结果
    return res; // 返回最大路径长度
}
```

欧拉路径

无向图

存在欧拉通路的充要条件

非零度顶点是连通的
恰有 2 个奇度顶点

存在欧拉回路的充要条件

非零度顶点是连通的
顶点的度数都是偶数

有向图

存在欧拉通路的充要条件

非零度顶点是弱连通的
至多一个顶点的出度与入度之差为 1
至多一个顶点的入度与出度之差为 1
其他顶点的入度和出度相等

存在欧拉回路的充要条件

非零度顶点是强连通的
每个顶点的入度和出度相等

```
void add(int u, int v) {
    e[u].push_back(v);
    out[u]++, in[v]++;
}

bool check_eul() {
    int l = 0, r = 0;
    for (int i = 1; i <= n; i++) {
        if (out[i] - in[i] == 1) l++;
        else if (in[i] - out[i] == 1) r++;
        else if (in[i] != out[i]) return false;
    }
    return (l == 0 && r == 0) || (l == 1 && r == 1);
}

void findEul(int start) {
    stack<int> stk;
    stk.push(start);
    while (!stk.empty()) {
        int u = stk.top();
        if (!e[u].empty()) {
            int v = e[u].back();
            e[u].pop_back();  // 移除已访问的边
            stk.push(v);
        } else {
            eul.push_back(u);
            stk.pop();
        }
```

```
    }
}
```

Hierholzer 算法

贪心思想，每次走到走不下去为止，那个点就为欧拉路径的顶点，把他放入ans中，走过一条边要把那条边删除ßß

```
void dfs(int u) {
    while (!e[u].empty()) {
        int v = e[u].back();
        e[u].pop_back();
        if (!vis[u][v]) continue;
        vis[u][v] = vis[v][u] = 0;
        dfs(v);
    }
    ans.push_back(u);
}
```

找环思路

无向图找环

DFS

```
bool Dfs(int u) {
    vis[u] = 1;
    for (int i = head[u]; i; i = e[i].nxt) {
        int v = e[i].to;
        if (!vis[v]) {
            f[v] = u;
            if (Dfs(v)) return true;
        } else if (v != f[u]) {
            cc.push_back(v);
            for (int x = u; x != v; x = f[x])
                cc.push_back(x);
            return true;
        }
    }
    return false;
}
```

DSU

判断奇数环和偶数环

二分图染色法

有向如找环

计算每个点的链长,同时换上点长度为环的大小

```
inline void dfs(int x, int y) {
    int xx = x + dx[mp[x][y]], yy = y + dy[mp[x][y]];
    if (xx > n || xx < 1 || yy > m || yy < 1) return;
    stk[++top] = {x, y};
    instk[x][y] = vis[x][y] = 1;
    if (instk[xx][yy]) {
        cc.push_back({xx, yy});
        pii now = {x, y};
        do {
            cc.push_back(now);
            now = fa[now.first][now.second];
        } while (now != make_pair(xx, yy));
        for (auto [xs, ys]: cc) val[xs][ys] = cc.size();
        cc.clear();
    } else {
        fa[xx][yy] = {x, y};
        dfs(xx, yy);
        if (val[x][y] == 1) val[x][y] = val[xx][yy] + 1;
    }
    instk[x][y] = 0;
}
```

最短路算法

Dijkstra

主体，优先队列为小根堆

时间复杂度:$nlog^m$

```
inline void dijkstra() {
    dis[s] = 0;
    q.push({0, s});
    while (!q.empty()) {
        int x = q.top().pos;
        q.pop();
        if (vis[x])
            continue;
        vis[x] = 1;
        for (int i = head[x]; i; i = a[i].next) {
            int y = a[i].to;
            if (dis[y] > dis[x] + a[i].w) {
                dis[y] = dis[x] + a[i].w;
                if (!vis[y])
                    q.push({dis[y], y});
            }
        }
    }
}
```

Bellman-Ford

堆优化版——SPFA
时间复杂度:最好O(m)，最坏O(nm)，菊花图的情况

```
inline void spfa() {
    dis[s] = 0;
    vis[s] = 1;
    queue<int> q;
    q.push(s);
    int x;
    while (!q.empty()) {
        x = q.front();
        q.pop();
        vis[x] = 0;
        for (int i = head[x]; i; i = e[i].next) {
            int y = e[i].to, d = e[i].dis;
            if (dis[y] > dis[x] + d) {
                dis[y] = dis[x] + d;
                if (!vis[y])
                    vis[y] = 1, q.push(y);
            }
        }
    }
}
```

Johnson

> Johnson优化，能有处理负权值和有负环的情况
> 时间复杂度:$n^2 \log^m$
>
> 预处理:先给每条边新添加一条0边
>
> ```
> for (int i = 1; i <= n; i++)
>     add_edge(0, i, 0);
> ```
>
> 然后利用SPFA来判断负环，同时创建h数组（等同于势能，处理负权值）
>
> ```
> inline bool spfa(int s)
> {
>     for (int i = 1; i <= n; i++)
>         h[i] = 63, vis[i] = 0;
>     h[s] = 0;
>     vis[s] = 1;
>     queue<int> q;
>     q.push(s);
>     int x;
>     while (!q.empty())
>     {
>         x = q.front();
>         q.pop();
>         vis[x] = 0;
>         for (int i = head[x]; i; i = e[i].next)
>         {
> ```

```
                int y = e[i].to, d = e[i].w;
                if (h[y] > h[x] + d)
                {
                    h[y] = h[x] + d;
                    if (!vis[y])
                    {
                        ++num[y]; // 说明经过当前点，次数加1
                        if (num[y] > n) // n为自己设定的上限，如果循环次数超过n，说明
存在负环，直接返回
                            return false;
                        vis[y] = 1, q.push(y);
                    }
                }
            }
    }
    return true;
}
```

SPFA预处理完成后利用h数组更新权重

```
for (int j = 1; j <= n; j++)
    for (int i = head[j]; i; i = e[i].next)
        e[i].w = e[i].w + h[j] - h[e[i].to];
```

更新完成后可以保证权值全为正数，随后根据题意运行Dijkstra，最终输出答案要注意减去h数组差值

LCA/最近公共祖先

倍增

```
struct edge {
    int to, nxt;
} e[N];

int tot = 0, head[N];

void add(int u, int v) {
    e[++tot] = {v, head[u]};
    head[u] = tot;
}

int dep[N], fa[N][22];

inline void dfs(int u, int f) {
    dep[u] = dep[f] + 1;
    fa[u][0] = f;
    for (int i = 1; i <= 19; i++)
        fa[u][i] = fa[fa[u][i - 1]][i - 1];
    for (int i = head[u], c; i; i = e[i].nxt)
        if (e[i].to != f)
            dfs(e[i].to, u);
}
```

```
inline int lca(int u, int v) {
    if (dep[u] < dep[v])
        swap(u, v);
    for (int i = 19; i >= 0; i--)
        if (dep[fa[u][i]] >= dep[v])
            u = fa[u][i]; // 让u，v处于同一层
    if (u == v)
        return u;
    for (int i = 19; i >= 0; i--)
        if (fa[u][i] != fa[v][i])
            u = fa[u][i], v = fa[v][i]; // 返回祖先的下一层
    return fa[u][0];
}
```

Tarjan算法

Tarjan缩点/有向图

```
vi e[N], E[N];
int in[N], out[N];

int dfn[N], low[N], cnt;
int stk[N], instk[N], top;
int scc[N], siz[N], num;

ll val[N], f2[N], ans2;
int n, m, a[N], f1[N], ans1;

void tarjan(int x) {
    dfn[x] = low[x] = ++cnt;
    stk[++top] = x, instk[x] = 1;
    for (int y: E[x]) {
        if (!dfn[y]) {
            tarjan(y);
            low[x] = min(low[x], low[y]);
        } else if (instk[y]) {
            low[x] = min(low[x], dfn[y]);
        }
    }
    if (low[x] == dfn[x]) {
        int y;
        ++num;
        do {
            y = stk[top--];
            instk[y] = 0;
            scc[y] = num;
            val[num] += a[y];
            siz[num]++;
        } while (x != y);
    }
}

void build_new() {
    for (int i = 1; i <= n; i++) if (!dfn[i]) tarjan(i);
```

```
    for (int x = 1; x <= n; x++)
        for (int y: E[x])
            if (scc[x] != scc[y])
                e[scc[x]].push_back(scc[y]);
    for (int i = 1; i <= num; i++) {
        sort(e[i].begin(), e[i].end());
        e[i].erase(unique(e[i].begin(), e[i].end()), e[i].end());
    }
    for (int x = 1; x <= num; x++)
        for (int y: e[x])
            ++out[x], ++in[y];
}
```

Tarjan缩点/无向图

普通版

```
constexpr int N = 2e5 + 10;
vi e[N], tree[N];
int cnt = 0, comp_count = 0;
int dfn[N], low[N], fa[N], compID[N], vis[N], siz[N];
set<pii> bridges;

inline void tarjan(int u) {
    vis[u] = true;
    dfn[u] = low[u] = ++cnt;
    for (int v: e[u]) {
        if (!vis[v]) {
            fa[v] = u;
            tarjan(v);
            low[u] = min(low[u], low[v]);
            if (low[v] > dfn[u])
                bridges.insert({min(u, v), max(u, v)});
        } else if (v != fa[u]) low[u] = min(low[u], dfn[v]);
    }
}

inline void dfs(int u, int ID) {
    compID[u] = ID;
    for (int v: e[u])
        if (compID[v] == -1 && bridges.find({min(u, v), max(u, v)}) ==
bridges.end())
            dfs(v, ID);
}

inline void build(int n) {
    fill(compID + 1, compID + n + 1, -1);

    for (int i = 1; i <= n; i++)
        if (compID[i] == -1)
            dfs(i, ++comp_count);
    for (auto it: bridges) {
        int u = compID[it.first], v = compID[it.second];
        tree[u].push_back(v);
        tree[v].push_back(u);
```

```
    }
    for (int i = 1; i <= n; i++) siz[compID[i]]++;
}
```

类标准

```
class Brige
{
private:
    int n;
    vector<bool> vis;
    vector<vi> e, tree;
    set<pii> bridges;
    vi parent, dfn, low, compID;
    int tot = 0, comp_ID = 0;

    inline void dfs(int u, int ID)
    {
        compID[u] = ID;
        for (int v : e[u])
            if (!compID[v] && bridges.find({min(u, v), max(u, v)}) !=
bridges.end())
                dfs(v, ID);
    }

    inline void tarjan(int u)
    {
        vis[u] = true;
        dfn[u] = low[u] = ++tot;
        for (int v : e[u])
        {
            if (!vis[v])
            {
                parent[v] = u;
                tarjan(v);
                low[u] = min(low[u], low[v]);
                if (low[v] > dfn[u])
                    bridges.insert({min(u, v), max(u, v)});
            }
            else if (v != parent[u]) low[u] = min(low[u], dfn[v]);
        }
    }

public:
    Brige(vector<vi> ee, int nn): e(ee), n(nn)
    {
        vis.assign(n + 1, false);
        parent.assign(n + 1, 0);
        dfn.assign(n + 1, 0);
        low.assign(n + 1, 0);
        compID.assign(n + 1, -1);
    }

    inline void build(int n)
    {
```

```
        compID.assign(n + 1, -1);

        for (int i = 1; i <= n; i++)
            if (compID[i] == -1)
                dfs(i, ++comp_ID);
        for (auto it : bridges)
        {
            int u = compID[it.first], v = compID[it.second];
            tree[u].push_back(v);
            tree[v].push_back(u);
        }
    }

    inline void get(vector<vi>& ee, vi& compId)
    {
        ee = tree;
        compId = compID;
    }
};
```

标准tarjan

> 时间戳 dfn[x] 节点x第一次被访问的顺序
>
> 追溯值 low[x] 从x节点出发，能到的最早的时间戳
>
> ```
> vi e[N];
> int dfn[N], low[N], tot;
> int stk[N], instk[N], top;
> int scc[N], cnt;
>
> inline void tarjan(int x) {
> // 入x点，盖时间戳，入栈
>     dfn[x] = low[x] = ++tot;
>     stk[++top] = x, instk[x] = 1;
>     for (int y: e[x]) {
>         // 未访问
>          if (!dfn[y]) {
>              tarjan(y);
>              low[x] = min(low[x], low[y]);
>          } else if (instk[y])
>              low[x] = min(low[x], dfn[y]);
>      }
>  // x为强连通图的根,输出分量图
>      if (dfn[x] == low[x]) {
>         int y;
>         ++cnt;
>         do {
>             y = stk[top--];
>              instk[y] = 0;
>              scc[y] = cnt;
>          } while (y != x);
>      }
>  }
> ```

点双连通分量

> 基础性质：
> 1、 除了一种比较特殊的点双，其他的点双都满足：任意两点间都存在至少两条点不重复路径。
> 2、 图中任意一个割点都在至少两个点双中。
> 3、 任意一个不是割点的点都只存在于一个点双中。
>
> 注意点：要在tarjan基础上加特判起点没有祖先的情况

```
inline void tarjan(int u, int fa) {
    int child = 0;
    dfn[u] = low[u] = ++tot;
    for (int v: e[u]) {
        if (!dfn[v]) {
            ++child;
            tarjan(v, u);
            low[u] = min(low[u], low[v]);
            if (fa != -1 && low[v] >= dfn[u]) cut[u] = 1;
        } else if (v != fa) low[u] = min(low[u], dfn[v]);
    }
    if (fa == -1 && child >= 2) cut[u] = 1;
}
```

边双连通分量(DCC)

> 基础性质:
> 1、 割边不属于任意边双，而其它非割边的边都属于且仅属于一个边双。
> 2、 对于一个边双中的任意两个点，它们之间都有至少两条**边不重复**的路径。
>
> ```
> int dfn[N], low[N], cnt, tot;
> struct edge {
> int u, v;
> };
> vector<edge> e;
> vi h[N];
> struct bridge {
> int x, y;
> } bri[N];
>
> inline void add(int x, int y) {
> e.push_back({x, y});
> h[x].push_back(e.size() - 1);
> }
>
> inline void tarjan(int x, int in_edg) {
> dfn[x] = low[x] = ++tot;
> for (int i = 0; i < h[x].size(); i++) {
>     int j = h[x][i], y = e[j].v;
>     if (!dfn[y]) {
>         tarjan(y, j);
>         low[x] = min(low[x], low[y]);
>         if (low[y] > dfn[x]) //如果low值大于dfn值，说明只能从x到y为割边
> ```

```
            bri[++cnt] = {x, y};
        } else if (j != (in_edg ^ 1)) //判断是否为反边
            low[x] = min(low[x], dfn[y]);
    }
}
```

DCC - SCC

```
struct Tree {
    struct edge {
        int v, id;
    };

    vector<edge> e[N], g[N];
    int eid = -1;
    bool bridges[N];
    int dfn[N], low[N], tick;

    void adde(int u, int v) {
        ++eid;
        e[u].push_back({v, eid});
        e[v].push_back({u, eid});
    }

    void tarjan(int u, int pid) {
        dfn[u] = low[u] = ++tick;
        for (auto [v, id]: e[u]) {
            if (!dfn[v]) {
                tarjan(v, id);
                low[u] = min(low[u], low[v]);
                if (low[v] > dfn[u]) bridges[id] = true;
            } else if (id != pid) low[u] = min(low[u], dfn[v]);
        }
    }

    int siz[N], compID[N], cnum;

    void dfs(int u, int cid) {
        ++siz[cid];
        compID[u] = cid;
        for (auto [v, eid]: e[u]) {
            if (compID[v] == -1 && !bridges[eid]) {
                dfs(v, cid);
            }
        }
    }

    vi tree[N];

    void build(int n) {
        fill(dfn + 1, dfn + 1 + n, 0);
        fill(low + 1, low + 1 + n, 0);
        tick = 0;
        for (int i = 1; i <= n; i++) if (!dfn[i]) tarjan(i, -1);
        fill(siz + 1, siz + 1 + n, 0);
```

```
        fill(compID + 1, compID + 1 + n, -1);
        cnum = 0;
        for (int i = 1; i <= n; i++) if (compID[i] == -1) dfs(i, ++cnum);
        for (int u = 1; u <= n; u++) {
            for (auto [v, id]: e[u]) {
                if (compID[u] != compID[v]) tree[compID[u]].push_back(compID[v]);
                else g[u].push_back({v, id});
            }
        }
    }

    bool vis_c[N];
    ll vis_g[N * 2];

    void dfs1(int u) {
        for (auto [v, id]: g[u]) {
            if (vis_g[id]) continue;
            vis_g[id] = 1ll * u * N + v;
            dfs1(v);
        }
    }

    void dcc_scc(int n) {
        for (int i = 1; i <= n; i++) if (!vis_c[compID[i]]) dfs1(i),
vis_c[compID[i]] = true;
    }

    int fa[N];

    void dfs2(int u, int f) {
        fa[u] = f;
        for (int v: tree[u]) {
            if (v == f)continue;
            dfs2(v, u);
        }
    }

    int find_s() {
        int id = 0;
        for (int i = 1; i <= cnum; i++) if (siz[id] < siz[i]) id = i;
        dfs2(id, 0);
        return siz[id];
    }

    bool check(int u, int v, int tid) {
        if (fa[compID[u]] == compID[v]) return true;
        int ux = vis_g[tid] / N, vx = vis_g[tid] % N;
        if (ux == u && vx == v) return true;
        return false;
    }
} t;
```

拓扑排序

有向无环图（DAG），可以判断有向图中是否有环

DFS算法

> 通过c数组来存放颜色，表示不同的状态

```
vector<int> e[N], tp; // e[N]类似邻接表，存放有向边，tp存拓扑排序
int c[N];             // 存放颜色，0表示为经过，1表示已经被经过，-1表示正在被经过
bool dfs(int u)
{
c[u] = -1;
for (auto v : e[u])
{
    if (~c[v]) // 说明有环存在
        return 0;
    if (!c[v] && !dfs(v)) // 递归，说明v下面有环
        return 0;
}
c[u] = 1;
tp.push_back(u);
return 1;
}
bool toposort()
{
memset(c, 0, sizeof c);
for (int i = 1; i <= n; i++) // 遍历每个点，如果颜色没被标记，进行搜索
    if (!c[i] && !dfs(i))
        return 0;
reverse(tp.begin(), tp.end());
return 1;
}
```

卡恩算法（Kahn)

通过队列来维护入度为0的集合

```
vector<int> e[N], tp; // e[N]类似邻接表，存放有向边，tp存拓扑排序
int din[N];           // 存放每个点的入度
bool toposort()
{
queue<int> q;
for (int i = 1; i <= n; i++)
   if (!din[i])
       q.push(i);
 while (!q.empty())
 {
   int u = q.front();
   q.pop();
    tp.push_back(u);
    for (auto v : e[u])
        if (--din[v] == 0)
```

```
            q.push(v);
    }
    return tp.size() == n;
}
```

最小生成树

稀疏图一般选择 prim
稠密图一般选择 Kruskal

Prim

```
struct edge
{
    int v, w;
};
vector<edge> e[N];
int d[N], vis[N];
priority_queue<pair<int, int>> q;

bool prim(int s)
{
    for (int i = 0; i <= n; i++)
        d[i] = inf;
    d[s] = 0;
    q.push({0, s});
    while (!q.empty())
    {
        int u = q.top().second;
        q.pop();
        if (vis[u])
            continue;
        vis[u] = 1;
        ans += d[u];
        ++cnt;
        for (auto ed : e[u])
        {
            int v = ed.v, w = ed.w;
            if (d[v] > w)
            {
                d[v] = w;
                q.push({-d[v], v});
            }
        }
    }
    return cnt == n;
}
```

Kruskal

```
struct Edge {
    int u, v; // 边的两个端点
    int w;    // 边的权重
```

```cpp
    // 重载小于运算符以便于排序
    bool operator<(const Edge &other) const {
        return w < other.w;
    }
};

int fa[N], siz[N];

// 并查集查找，路径压缩
int find(int u) { return fa[u] == u ? u : fa[u] = find(fa[u]); }

// 并查集合并
void merge(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (fx != fy) {
        // 按秩合并
        if (siz[fx] < siz[fy])
            fa[fx] = fy;
        else if (siz[fx] > siz[fy])
            fa[fy] = fx;
        else {
            fa[fy] = fx;
            siz[fx]++;
        }
    }
}

// Kruskal 算法
int kruskal(int n, vector<Edge> &edges) {
    // 初始化并查集
    for (int i = 1; i <= n; i++) {
        fa[i] = i;
        siz[i] = 0;
    }
    sort(edges.begin(), edges.end()); // 按权重升序排序
    int tot = 0; // 最小生成树的权重
    for (const auto &e: edges) {
        int u = e.u;
        int v = e.v;
        if (find(u) != find(v)) {
            tot += e.w; // 加入边的权重
            merge(u, v); // 合并两个集合
        }
    }
    return tot; // 返回最小生成树的总权重
}
```

二分图

最大匹配问题

匈牙利算法

时间复杂度Omn

```
vector<int> g[N];  // 邻接表
int mt[N];         // 存储匹配
bool vis[N];       // 访问标记

// 深度优先搜索（DFS）查找增广路径
bool dfs(int u) {
    for (int v : adj[u]) {
        if (!vis[v]) {
            vis[v] = true;
            // 如果 v 没有匹配或 v 的匹配点可以找到其他匹配
            if (mt[v] == -1 || dfs(mt[v])) {
                mt[v] = u;
                return true;
            }
        }
    }
    return false;
}

// 匈牙利算法求二分图最大匹配
int hungarian(int n) {
    memset(mt, -1, sizeof mt);  // 初始化匹配数组，-1 表示没有匹配
    int res = 0;  // 匹配的数量
    for (int u = 0; u < n; ++u) {
        memset(vis, false, sizeof(vis));  // 每次查找增广路径时重置访问标记
        if (dfs(u)) {  // 如果找到增广路径，匹配数加1
            ++res;
        }
    }
    return res;
}
```

Hopcroft-Karp算法

时间复杂度O$n^{0.5}$m

```
int n, m;// n: 左侧顶点数, m: 右侧顶点数
vi mtl(N), mtr(N), dis(N);// mtl,mtr:左侧和右侧的匹配情况 dis:记录距离（用于 BFS）
vector<vi> g(N);// 存储二分图的邻接表

bool bfs() {
    queue<int> q;
    for (int u = 1; u <= n; u++) {
        if (mtl[u] == -1) {//初始化起点，如果没有被匹配过，距离为零，放入队列
            dis[u] = 0;
            q.push(u);
```

```
        } else {//如果有，则赋值为inf
            dis[u] = inf;
        }
    }
    bool check = false;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int v: g[u]) {
            int vv = mtr[v];//表示右边能到达的点的匹配点
            if (vv == -1) {//如果为-1，说明这个右边的点没有被匹配，能直接使用
                check = true;
            } else if (dis[vv] == inf) {//如果不为-1，说明他和vv匹配，把vv放到队列中，同时更新dis[vv]，说明vv和u是间隔相邻
                dis[vv] = dis[u] + 1;
                q.push(vv);
            }
        }
    }
    return check;
}

bool dfs(int u) {
    for (int v: g[u]) {
        int vv = mtr[v];
        if (vv == -1 || (dis[vv] == dis[u] + 1 && dfs(vv))) {
            mtl[u] = v;
            mtr[v] = u;
            return true;
        }
    }
    dis[u] = inf;//重置距离
    return false;
}

int HK() {
    for (int i = 1; i <= n; i++) mtl[i] = -1;
    for (int i = 1; i <= m; i++) mtr[i] = -1;
    int mt = 0;
    while (bfs()) // 分阶段寻找增广路径
        for (int u = 1; u <= n; u++)
            if (mtl[u] == -1 && dfs(u))// 如果没被匹配过同时找到增广路径，匹配数加1
                ++mt;
    return mt;
}
```

网络流

最大流

V是节点数 E是边数

EK算法

时间复杂度$O(VE^2)$

```
struct EK {
    struct Edge {
        int from, to, cap, flow;

        Edge(int u, int v, int c, int f) : from(u), to(v), cap(c), flow(f) {
        }
    };

    int n, m; // n: 点数，m: 边数
    vector<Edge> edges; // edges: 所有边的集合
    vector<int> G[MAXN]; // G: 点 x -> x 的所有边在 edges 中的下标
    int a[MAXN], p[MAXN];
    // a: 点 x -> BFS 过程中最近接近点 x 的边给它的最大流
    // p: 点 x -> BFS 过程中最近接近点 x 的边

    void init(int n) {
        for (int i = 0; i < n; i++) G[i].clear();
        edges.clear();
    }

    void AddEdge(int from, int to, int cap) {
        edges.push_back(Edge(from, to, cap, 0));
        edges.push_back(Edge(to, from, 0, 0));
        m = edges.size();
        G[from].push_back(m - 2);
        G[to].push_back(m - 1);
    }

    int Maxflow(int s, int t) {
        int flow = 0;
        for (;;) {
            memset(a, 0, sizeof(a));
            queue<int> Q;
            Q.push(s);
            a[s] = INF;
            while (!Q.empty()) {
                int x = Q.front();
                Q.pop();
                for (int i = 0; i < G[x].size(); i++) {
                    // 遍历以 x 作为起点的边
                    Edge &e = edges[G[x][i]];
                    if (!a[e.to] && e.cap > e.flow) {
                        p[e.to] = G[x][i]; // G[x][i] 是最近接近点 e.to 的边
                        a[e.to] = min(a[x], e.cap - e.flow); // 最近接近点 e.to 的
边赋给它的流
                        Q.push(e.to);
                    }
                }
                if (a[t]) break; // 如果汇点接受到了流，就退出 BFS
            }
            if (!a[t]) break; // 如果汇点没有接受到流，说明源点和汇点不在同一个连通分量上
```

```
            for (int u = t; u != s; u = edges[p[u]].from) {
                // 通过 u 追寻 BFS 过程中 s -> t 的路径
                edges[p[u]].flow += a[t]; // 增加路径上边的 flow 值
                edges[p[u] ^ 1].flow -= a[t]; // 减小反向路径的 flow 值
            }
            flow += a[t];
        }
        return flow;
    }
} ek;
```

Dinic算法

在普通情况下， DINIC算法时间复杂度为$O(V^2E)$

在二分图中， DINIC算法时间复杂度为$O(V^{0.5}E)$

```
struct MF {
    struct edge {
        int v, nxt, cap, flow;
    } e[N];

    int fir[N], cnt = 0;

    int n, S, T;
    ll maxflow = 0;
    int dep[N], cur[N];

    void init() {
        memset(fir, -1, sizeof fir);
        cnt = 0;
    }

    void addedge(int u, int v, int w) {
        e[cnt] = {v, fir[u], w, 0};
        fir[u] = cnt++;
        e[cnt] = {u, fir[v], 0, 0};
        fir[v] = cnt++;
    }

    bool bfs() {
        queue<int> q;
        memset(dep, 0, sizeof(int) * (n + 1));

        dep[S] = 1;
        q.push(S);
        while (q.size()) {
            int u = q.front();
            q.pop();
            for (int i = fir[u]; ~i; i = e[i].nxt) {
                int v = e[i].v;
                if (!dep[v] && e[i].cap > e[i].flow) {
                    dep[v] = dep[u] + 1;
                    q.push(v);
                }
            }
```

```
        }
        return dep[T];
    }

    int dfs(int u, int flow) {
        if (u == T || !flow) return flow;

        int ret = 0;
        for (int &i = cur[u]; ~i; i = e[i].nxt) {
            int v = e[i].v, d;
            if (dep[v] == dep[u] + 1 && (d = dfs(v, min(flow - ret, e[i].cap -
e[i].flow)))) {
                ret += d;
                e[i].flow += d;
                e[i ^ 1].flow -= d;
                if (ret == flow) return ret;
            }
        }
        return ret;
    }

    void dinic() {
        while (bfs()) {
            memcpy(cur, fir, sizeof(int) * (n + 1));
            maxflow += dfs(S, INF);
        }
    }
} mf;
```

费用流

EK算法改编

```
struct EK {
    struct Edge {
        int from, to, cap, flow, cost;

        Edge(int u, int v, int c, int f, int co) : from(u), to(v), cap(c),
flow(f), cost(co) {
        }
    };

    int n, m; // n: 点数，m: 边数
    vector<Edge> edges; // edges: 所有边的集合
    vector<int> G[MAXN]; // G: 点 x -> x 的所有边在 edges 中的下标
    int dis[MAXN], a[MAXN], p[MAXN]; // dis: 计算最短路径的数组
    bool inQueue[MAXN]; // 记录是否在队列中

    void init(int n) {
        for (int i = 0; i < n; i++) G[i].clear();
        edges.clear();
    }

    // 添加边 (from -> to, capacity, cost)
    void AddEdge(int from, int to, int cap, int cost) {
```

```
        edges.push_back(Edge(from, to, cap, 0, cost));
        edges.push_back(Edge(to, from, 0, 0, -cost)); // 反向边，费用取反
        m = edges.size();
        G[from].push_back(m - 2);
        G[to].push_back(m - 1);
    }

    // 使用 SPFA (Shortest Path Faster Algorithm) 找增广路径
    bool SPFA(int s, int t) {
        fill(dis, dis + n, INF);
        memset(inQueue, 0, sizeof(inQueue));
        queue<int> Q;
        Q.push(s);
        dis[s] = 0;
        a[s] = INF;
        inQueue[s] = true;
        while (!Q.empty()) {
            int u = Q.front();
            Q.pop();
            inQueue[u] = false;
            for (int i: G[u]) {
                Edge &e = edges[i];
                if (e.cap > e.flow && dis[e.to] > dis[u] + e.cost) {
                    dis[e.to] = dis[u] + e.cost;
                    a[e.to] = min(a[u], e.cap - e.flow);
                    p[e.to] = i;
                    if (!inQueue[e.to]) {
                        Q.push(e.to);
                        inQueue[e.to] = true;
                    }
                }
            }
        }
        return dis[t] != INF;
    }

    // 最大费用流计算
    int MaxFlow(int s, int t) {
        int flow = 0, cost = 0;
        while (SPFA(s, t)) {
            int f = a[t]; // 增广流量
            flow += f;
            cost += f * dis[t]; // 计算费用
            // 更新流量和反向流量
            for (int u = t; u != s; u = edges[p[u]].from) {
                edges[p[u]].flow += f;
                edges[p[u] ^ 1].flow -= f; // 反向边流量减去
            }
        }
        return cost; // 返回最大费用
    }
};
```

Dinic算法改编

```
struct MCMF {
    static const int N = 5e3 + 5, M = 1e5 + 5;
    int tot = 1, lnk[N], cur[N], ter[M], nxt[M], cap[M], cost[M];
    ll dis[N], ret;
    bool vis[N];

    void init(int n) {
        tot = 1;
        memset(lnk, 0, sizeof(int) * (n + 2));
        ret = 0;
    }

    void add(int u, int v, int w, int c) {
        ter[++tot] = v, nxt[tot] = lnk[u], lnk[u] = tot, cap[tot] = w, cost[tot]
= c;
    }

    void addedge(int u, int v, int w, int c) { add(u, v, w, c), add(v, u, 0, -c);
}

    bool spfa(int s, int t) {
        memset(dis, 0x3f, sizeof(dis));
        memcpy(cur, lnk, sizeof(lnk));
        queue<int> q;
        q.push(s), dis[s] = 0, vis[s] = true;
        while (!q.empty()) {
            int u = q.front();
            q.pop(), vis[u] = false;
            for (int i = lnk[u]; i; i = nxt[i]) {
                int v = ter[i];
                if (cap[i] && dis[v] > dis[u] + cost[i]) {
                    dis[v] = dis[u] + cost[i];
                    if (!vis[v]) q.push(v), vis[v] = true;
                }
            }
        }
        return dis[t] != dis[N - 1];
    }

    int dfs(int u, int t, int flow) {
        if (u == t) return flow;
        vis[u] = true;
        int ans = 0;
        for (int &i = cur[u]; i && ans < flow; i = nxt[i]) {
            int v = ter[i];
            if (!vis[v] && cap[i] && dis[v] == dis[u] + cost[i]) {
                int x = dfs(v, t, min(cap[i], flow - ans));
                if (x) ret += x * cost[i], cap[i] -= x, cap[i ^ 1] += x, ans +=
x;
            }
        }
        vis[u] = false;
        return ans;
    }

    int dinic(int s, int t) {
```

```
        int ans = 0;
        while (spfa(s, t)) {
            int x;
            while ((x = dfs(s, t, INF))) ans += x;
        }
        return ans;
    }
} mcmf;
```

上下界费用流

无源汇上下界

```
struct MCMF {
    static const int MAXN = 10010;
    static const int MAXE = 50010;
    static const int INF = 0x3f3f3f3f;

    struct Edge {
        int v, nxt, cap, flow, cost;
    } e[MAXE];

    int fir[MAXN], cur[MAXN], cnt = 0;
    int S, T, superS, superT;
    int dis[MAXN], pre[MAXN];
    bool vis[MAXN];
    int demand[MAXN]; // 点的需求
    long long maxflow = 0, mincost = 0;

    void init(int n) {
        memset(fir, -1, sizeof(int) * (n + 5));
        memset(demand, 0, sizeof(int) * (n + 5));
        cnt = 0;
        maxflow = mincost = 0;
    }

    void addRawEdge(int u, int v, int cap, int cost) {
        e[cnt] = {v, fir[u], cap, 0, cost};
        fir[u] = cnt++;
        e[cnt] = {u, fir[v], 0, 0, -cost};
        fir[v] = cnt++;
    }

    // 添加带上下界的边：下界 l，上界 r，花费 cost
    void addEdge(int u, int v, int l, int r, int cost) {
        // 维护真实花费
        mincost += 1LL * l * cost;

        // 加边容量 r - l
        addRawEdge(u, v, r - l, cost);

        // 调整点需求
        demand[u] -= l;
        demand[v] += l;
    }
```

```cpp
    bool spfa() {
        memset(dis, INF, sizeof dis);
        memset(vis, 0, sizeof vis);
        memcpy(cur, fir, sizeof fir);
        std::queue<int> q;
        dis[S] = 0;
        vis[S] = true;
        q.push(S);

        while (!q.empty()) {
            int u = q.front(); q.pop();
            vis[u] = false;
            for (int i = fir[u]; ~i; i = e[i].nxt) {
                int v = e[i].v;
                if (e[i].cap > e[i].flow && dis[v] > dis[u] + e[i].cost) {
                    dis[v] = dis[u] + e[i].cost;
                    pre[v] = i;
                    if (!vis[v]) {
                        vis[v] = true;
                        q.push(v);
                    }
                }
            }
        }
        return dis[T] != INF;
    }

    int dfs(int u, int flow) {
        if (u == T || flow == 0) return flow;
        vis[u] = true;
        int ret = 0;
        for (int &i = cur[u]; ~i && ret < flow; i = e[i].nxt) {
            int v = e[i].v;
            if (!vis[v] && dis[v] == dis[u] + e[i].cost && e[i].cap > e[i].flow)
 {
                int d = dfs(v, std::min(flow - ret, e[i].cap - e[i].flow));
                if (d) {
                    e[i].flow += d;
                    e[i ^ 1].flow -= d;
                    ret += d;
                    mincost += 1LL * d * e[i].cost;
                }
            }
        }
        vis[u] = false;
        return ret;
    }

    bool dinic() {
        bool has_flow = false;
        while (spfa()) {
            int flow;
            memset(vis, 0, sizeof vis);
            while ((flow = dfs(S, INF))) {
                maxflow += flow;
```

```
                has_flow = true;
            }
        }
        return has_flow;
    }

    // 建立超级源汇
    bool buildSuperGraph(int n) {
        superS = n + 1, superT = n + 2;
        S = superS, T = superT;
        for (int i = 1; i <= n; ++i) {
            if (demand[i] > 0) {
                addRawEdge(superS, i, demand[i], 0);
            } else if (demand[i] < 0) {
                addRawEdge(i, superT, -demand[i], 0);
            }
        }

        // 跑最大流判断是否所有需求都满足
        int total_demand = 0;
        for (int i = 1; i <= n; ++i) {
            if (demand[i] > 0) total_demand += demand[i];
        }
        dinic();

        return maxflow == total_demand;
    }
};
```

字符串

AC自动机

```
// 回跳边:父节点回跳所指节点的儿子
// 转移边:当前节点回跳边所指节点的儿子
struct AC_auto
{
private:
    struct node
    {
        int val = 0;
        node* nex = nullptr;
        node* next[26] = {nullptr};
    };

    static const int N = 1e6 + 5;
    node pool[N];
    int tot = 0;

    node* alloc()
    {
        pool[tot] = node();
        return &pool[tot++];
    }
```

```
    node* root;

public:
    void init()
    {
        tot = 0;
        root = alloc();
    }

    void ins(const string& s, int val)
    {
        node* now = root;
        for (char it : s)
        {
            if (now->next[it - 'a'] == nullptr)
                now->next[it - 'a'] = alloc();
            now = now->next[it - 'a'];
        }
        now->val += val;
    }

    void build()
    {
        queue<node*> q;
        root->nex = root;
        for (int i = 0; i < 26; i++)
        {
            if (root->next[i] != nullptr)
                q.push(root->next[i]);
            else
                root->next[i] = root;
            root->next[i]->nex = root;
        }
        while (!q.empty())
        {
            node* now = q.front();
            q.pop();
            for (int i = 0; i < 26; i++)
            {
                if (now->next[i] != nullptr)
                {
                    now->next[i]->nex = now->nex->next[i];
                    q.push(now->next[i]);
                }
                else now->next[i] = now->nex->next[i];
            }
        }
    }

    int qry(const string& s)
    {
        int res = 0;
        node* now = root;
        for (char it : s)
        {
```

```
            now = now->next[it - 'a'];
            for (node* cal = now; cal != root && ~cal->val; cal = cal->nex)
                res += cal->val, cal->val = -1;
        }
        return res;
    }
} t;
```

后缀自动机

```
/*
 * 基础信息:
 * 合法性:子节点的最短串的最长后缀=父节点的最长串
 * 节点的子串长度:最长子串=len[i] 最短子串=len[i]-len[fa[i]]
 * 节点的子串数量=len[i]-len[fa[i]]
 * 子串的出现次数=cnt[i]
 */
struct SAM
{
private:
    int tot = 1, np = 1;
    vi fa, len, cnt;
    vector<unordered_map<char, int>> ch;

    void extend(char c)
    {
        int p = np;
        np = ++tot;
        len[np] = len[p] + 1;
        cnt[np] = 1;
        // 从链接边向前遍历,更新旧点的链接边
        while (p && !ch[p][c])
        {
            ch[p][c] = np;
            p = fa[p];
        }
        // 更新链接边
        if (!p)
            fa[np] = 1; // 指向根节点,是新字符,直接创建
        else
        {
            int q = ch[p][c];
            if (len[q] == len[p] + 1)
                fa[np] = q; // 相邻,合法,直接加边
            else
            {
                // 不相邻,不合法,要求新建一个链接点,然后把p点之前的所有点的路径更新
                int nq = ++tot;
                len[nq] = len[p] + 1;
                fa[nq] = fa[q];
                fa[q] = nq;
                fa[np] = nq;
                while (p && ch[p][c])
                {
                    ch[p][c] = nq;
```

```
                p = fa[p];
            }
            // 将原先的转移边复制到nq上
            ch[nq] = ch[q];
        }
    }
}

public:
    // n不该是题目给的n,要足够大保证链接点够,建议为 n*2 大小
    void init(string s)
    {
        int n = s.length() * 2;
        tot = np = 1;
        fa.assign(n + 1, 0);
        len.assign(n + 1, 0);
        cnt.assign(n + 1, 0);
        // ch 存放链接边信息
        ch.assign(n + 1, unordered_map<char, int>());

        for (auto it : s)
            extend(it);
    }

    ll get_count()
    {
        ll res = 0;
        for (int i = 1; i <= tot; i++)
            res += len[i] - len[fa[i]];
        return res;
    }
} sam;
```

Manacher 判断回文串

```
inline string get_new(const string& s)
{
    string res = "#";
    for (auto it : s)
    {
        res += it;
        res += '#';
    }
    return res;
}

// res代表以i为中心的回文串的长度
inline vi work(const string& x)
{
    string s = get_new(x);
    int n = s.length();
    vi res(n);
    // 当前最长回文中间点所在位置
    int c = 0;
    for (int i = 0; i < n; i++)
```

```
    {
        int l = 2 * c - i;
        if (i < c + res[c])res[i] = min(res[l], c + res[c] - i);
        while (i - res[i] - 1 >= 0 && i + res[i] + 1 < n && s[i - res[i] - 1] ==
s[i + res[i] + 1]) ++res[i];
        if (i + res[i] > c + res[c]) c = i;
    }
    return res;
}

// 判断是否为回文串,l r为未修改前字符串的坐标
bool ch(int l, int r, vi& p)
{
    int ll = l * 2 + 1, rr = r * 2 + 1;
    int mid = ll + rr >> 1;
    return mid + p[mid] - 1 >= rr;
}
```

矩阵乘法求字符串匹配

```
vector<vll> mul(vector<vll> A, vector<vll> B) {
    int n = A.size();
    vector<vll> C(n, vll(n, 0));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            for (int k = 0; k < n; ++k)
                C[i][j] = (C[i][j] + A[i][k] * B[k][j] % mo) % mo;
    return C;
}

vector<vll> build(string S, char c) {
    int n = S.size();
    vector<vll> F(n + 1, vll(n + 1, 0));
    for (int i = 0; i <= n; ++i) {
        if (i < n && S[i] == c)
            F[i][i + 1] = 1;
        F[i][i] = 1;
    }
    return F;
}

ll cal(string S, string T) {
    int n = S.size();
    vector<vll> res(n + 1, vll(n + 1, 0));
    for (int i = 0; i <= n; ++i) res[i][i] = 1;
    for (char c: T) {
        vector<vll> Fc = build(S, c);
        res = mul(res, Fc);
    }
    return res[0][n];
}
```

KMP

蓝书P82

```
void getNext(string s, int len) {
    next[0] = 0;
    int k = 0; //k = next[0]
    for (int i = 1; i < len; i++) {
        while (k > 0 && s[i] != s[k])k = next[k - 1]; //k = next[k-1]
        if (s[i] == s[k])k++;
        next[i] = k; //next[j+1] = k+1 | next[j+1] = 0
    }
}

//返回模式串T中字串S第一次出现的位置下标，找不到则返回-1
int kmp(string T, string S) {
    int len_T = T.length();
    int len_S = S.length();
    for (int i = 0, j = 0; i < len_T; i++) {
        while (j > 0 && T[i] != S[j])j = next[j - 1];
        if (T[i] == S[j])j++;
        if (j == len_S)return i - len_S + 1;
    }
    return -1;
}

//返回模式串T中字串S出现的次数，找不到则返回0
int kmp(string T, string S) {
    int sum = 0;
    int len_T = T.length();
    int len_S = S.length();
    for (int i = 0, j = 0; i < len_T; i++) {
        while (j > 0 && T[i] != S[j])j = next[j - 1];
        if (T[i] == S[j])j++;
        if (j == len_S) {
            sum++;
            j = next[j - 1];
        }
    }
    return sum;
}
```