算法模板

一、数据结构

1 DSU

```
struct DSU {
    std::vector<int> fa, siz;
    std::vector<i64> edge; // 边数

    DSU(int n) {
        init(n);
    }
    void init(int n) {
        fa.resize(n);
        siz.assign(n, 1);
        edge.assign(n, 0ll);
        std::iota(fa.begin(), fa.end(), 0);
    }

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

    bool same(int x, int y) {
        return find(x) == find(y);
    }

    bool merge(int x, int y) {
        int tx = find(x), ty = find(y);
        if (tx == ty) {
            edge[tx]++;
            return false;
        }
        fa[ty] = tx;
        siz[tx] += siz[ty];
        edge[tx] += edge[ty] + 1;
        return true;
    }

    int V(int x) {
        return siz[find(x)];
    }

    i64 E(int x) {
        return edge[find(x)];
    }
};
```

2 带权DSU

```
i64 fa[size], val[size];
i64 find(i64 x) {
    if (x != fa[x]) {
        i64 t = fa[x];
        fa[x] = find(fa[x]);
        val[x] += val[t];
    }
    return fa[x];
}
```

3 ST表

```
template<typename T>
class ST {
public:
    int n;
    std::vector<T> a;
    std::vector<std::vector<T>> fmin, fmax, fgcd;
```

```cpp
    ST(int _n) {
        n = _n;
        a.assign(_n + 1, {});
    };

    void cal_max() {
        fmax.assign(n + 1, std::vector<T>(std::__lg(n) + 1, {}));
        for (int i = 0; i < n; i++) {
            fmax[i][0] = a[i];
        }
        for (int j = 1; j <= std::__lg(n); j++) {
            for (int i = 1; i + (1 << j) - 1 <= n; i++) {
                fmax[i][j] = std::max(fmax[i][j - 1], fmax[i + (1 << (j - 1))][j - 1]);
            }
        }
    }

    void cal_min() {
        fmin.assign(n + 1, std::vector<T>(std::__lg(n) + 1, {}));
        for (int i = 0; i < n; i++) {
            fmin[i][0] = a[i];
        }
        for (int j = 1; j <= std::__lg(n); j++) {
            for (int i = 1; i + (1 << j) - 1 <= n; i++) {
                fmin[i][j] = std::min(fmin[i][j - 1], fmin[i + (1 << (j - 1))][j - 1]);
            }
        }
    }

    void cal_gcd() {
        fgcd.assign(n + 1, std::vector<T>(std::__lg(n) + 1, {}));
        for (int i = 0; i < n; i++) {
            fgcd[i][0] = a[i];
        }
        for (int j = 1; j <= std::__lg(n); j++) {
            for (int i = 1; i + (1 << j) - 1 <= n; i++) {
                fgcd[i][j] = std::gcd(fgcd[i][j - 1], fgcd[i + (1 << (j - 1))][j - 1]);
            }
        }
    }

    T get_max(int l, int r) {
        int len = std::__lg(r - l + 1);
        return std::max(fmax[l][len], fmax[r - (1 << len) + 1][len]);
    }

    T get_min(int l, int r) {
        int len = std::__lg(r - l + 1);
        return std::min(fmin[l][len], fmin[r - (1 << len) + 1][len]);
    }

    T get_gcd(int l, int r) {
        int len = std::__lg(r - l + 1);
        return std::gcd(fgcd[l][len], fgcd[r - (1 << len) + 1][len]);
    }
};
```

4 树状数组

```cpp
// 左闭右开
template <typename T>
struct Fenwick {
    int n;
    std::vector<T> a;

    Fenwick(int n_ = 0) {
        init(n_);
    }

    void init(int n_) {
        n = n_;
        a.assign(n, T{});
    }

    void add(int x, const T &v) {
        for (int i = x + 1; i <= n; i += i & -i) {
            a[i - 1] = a[i - 1] + v;
        }
```

```
    }

    T sum(int x) {
        T ans{};
        for (int i = x; i > 0; i -= i & -i) {
            ans = ans + a[i - 1];
        }
        return ans;
    }

    T rangeSum(int l, int r) {
        return sum(r) - sum(l);
    }

    // 最大 x 使得 sum(x) <= k（注意左闭右开）
    int select(const T &k) {
        int x = 0;
        T cur{};
        for (int i = 1 << std::__lg(n); i; i /= 2) {
            if (x + i <= n && cur + a[x + i - 1] <= k) {
                x += i;
                cur = cur + a[x - 1];
            }
        }
        return x;
    }
};
```

5 (1) 线段树(lazy_tag 区间乘和区间加)

```
i64 mod;
template<class Info, class Tag>
class SegmentTree {
private:
    #define ls(p) (p << 1)
    #define rs(p) (p << 1 | 1)
    int n;
    std::vector<Info> info;
    std::vector<Tag> tag;

    void pull(int p) {
        info[p] = info[ls(p)] + info[rs(p)];
    }

    void settag(int p, Tag v) {
        info[p].val = ((info[p].val * v.mul % mod) + (info[p].sz * v.add % mod)) % mod;
        tag[p] = tag[p] + v;
    }

    void push(int p) {
        if (tag[p].add || tag[p].mul != 1) {
            settag(ls(p), tag[p]);
            settag(rs(p), tag[p]);
            // 标记下传，消除自身标记
            tag[p] = {};
        }
    }

    void build(int p, int pl, int pr) {
        if (pl == pr) {
            info[p].val = a[pl];
            return;
        }
        int mid = pl + pr >> 1;
        build(ls(p), pl, mid);
        build(rs(p), mid + 1, pr);
        pull(p);
    }

    void rangeModify(int l, int r, int p, int pl, int pr, const Tag v) {
        if (l <= pl && pr <= r) {
            settag(p, v);
            return;
        }
        push(p);
        int mid = pl + pr >> 1;
        if (l <= mid) {
            rangeModify(l, r, ls(p), pl, mid, v);
```

```cpp
        }
        if (r > mid) {
            rangeModify(l, r, rs(p), mid + 1, pr, v);
        }
        pull(p);
    }

    Info query(int l, int r, int p, int pl, int pr) {
        if (l <= pl && pr <= r) {
            return info[p];
        }
        push(p);
        int mid = pl + pr >> 1;
        Info res {};
        if (l <= mid) {
            res = res + query(l, r, ls(p), pl, mid);
        }
        if (r > mid) {
            res = res + query(l, r, rs(p), mid + 1, pr);
        }
        return res;
    }
public:
    std::vector<i64> a;
    SegmentTree(int n_) {
        n = n_;
        info.assign(n_ << 2 | 1, {});
        tag.assign(n_ << 2 | 1, {});
        a.assign(n_ + 1, {});
    }
    // 建树
    void build() {
        build(1, 1, n);
    }

    // 区间乘 + 区间加
    void rangeModify(int l, int r, const Tag v) {
        rangeModify(l, r, 1, 1, n, v);
    }

    // 区间查询
    i64 query(int l, int r) {
        Info res = query(l, r, 1, 1, n);
        return res.val;
    }
};
struct Tag {
    i64 add = 0;
    i64 mul = 1;
};
Tag operator + (const Tag& x, const Tag& y) {
    Tag res {};
    res.mul = x.mul * y.mul % mod;
    res.add = ((x.add * y.mul % mod) + y.add) % mod;
    return res;
}

struct Info {
    i64 val {};
    int sz = 1;
};
Info operator + (const Info&x, const Info& y) {
    Info res {};
    res.val = (x.val + y.val) % mod;
    res.sz = (x.sz + y.sz) % mod;
    return res;
}
```

(2) 线段树单点修改+维护区间最大子段和

```cpp
template<class Info>
class SegmentTree {
private:
    std::vector<Info> info;
    int n;

    void pull(int p) {
        info[p] = info[p * 2 + 1] + info[p * 2 + 2];
```

```cpp
    }

    void modify(int pos, int p, int pl, int pr, const Info v) {
        if (pos == pl && pos == pr) {
            info[p] = v;
            return;
        }
        int mid = pl + pr >> 1;
        if (pos <= mid) {
            modify(pos, p * 2 + 1, pl, mid, v);
        } else {
            modify(pos, p * 2 + 2, mid + 1, pr, v);
        }
        pull(p);
    }

    Info query(int l, int r, int p, int pl, int pr) {
        if (l <= pl && pr <= r){
            return info[p];
        }
        int mid = pl + pr >> 1;
        Info res {};
        if (l <= mid) {
            res = res + query(l, r, p * 2 + 1, pl, mid);
        }
        if (r > mid) {
            res = res + query(l, r, p * 2 + 2, mid + 1, pr);
        }
        return res;
    }
public:
    SegmentTree(int n_) {
        n = n_;
        info.assign(n_ << 2 | 1, {});
    }

    void modify(int pos, const Info v) {
        modify(pos, 0, 0, n, v);
    }

    i64 query_pre(int l, int r) {
        return query(l, r, 0, 0, n).pre;
    }
    i64 query_suf(int l, int r) {
        return query(l, r, 0, 0, n).suf;
    }
};

struct Info {
    i64 ans = 0;
    i64 pre = 0;
    i64 suf = 0;
    i64 sum = 0;
};

Info operator + (const Info& x, const Info& y) {
    Info res {};
    res.pre = std::max(x.pre, x.sum + y.pre);
    res.suf = std::max(y.suf, y.sum + x.suf);
    res.sum = x.sum + y.sum;
    res.ans = std::max({x.ans, y.ans, x.suf + y.pre});
    return res;
}
```

(3) 线段树区间赋值

```cpp
template<class Info, class Tag>
class SegmentTree {
private:
    #define ls(p) (p << 1)
    #define rs(p) (p << 1 | 1)
    int n;
    std::vector<Info> info;
    std::vector<Tag> tag;

    void pull(int p) {
        info[p] = info[ls(p)] + info[rs(p)];
    }
```

```cpp
    void settag(int p, Tag v) {
        if (v.agn != -2e18) {
            info[p].val = v.agn;
        }
        tag[p] = v;
    }

    void push(int p) {
        if (tag[p].agn != -2e18) {
            settag(ls(p), tag[p]);
            settag(rs(p), tag[p]);
            // 标记下传，消除自身标记
            tag[p].agn = -2e18;
        }
    }

    void build(int p, int pl, int pr) {
        if (pl == pr) {
            info[p].val = a[pl];
            return;
        }
        int mid = pl + pr >> 1;
        build(ls(p), pl, mid);
        build(rs(p), mid + 1, pr);
        pull(p);
    }

    void rangeModify(int l, int r, int p, int pl, int pr, const Tag v) {
        if (l <= pl && pr <= r) {
            settag(p, v);
            return;
        }
        push(p);
        int mid = pl + pr >> 1;
        if (l <= mid) {
            rangeModify(l, r, ls(p), pl, mid, v);
        }
        if (r > mid) {
            rangeModify(l, r, rs(p), mid + 1, pr, v);
        }
        pull(p);
    }

    Info query(int l, int r, int p, int pl, int pr) {
        if (l <= pl && pr <= r) {
            return info[p];
        }
        push(p);
        int mid = pl + pr >> 1;
        Info res {};
        if (l <= mid) {
            res = res + query(l, r, ls(p), pl, mid);
        }
        if (r > mid) {
            res = res + query(l, r, rs(p), mid + 1, pr);
        }
        return res;
    }
public:
    std::vector<i64> a;
    SegmentTree(int n_) {
        n = n_;
        info.assign(n_ << 2 | 1, {});
        tag.assign(n_ << 2 | 1, {});
        a.assign(n_ + 1, {});
    }
    // 建树
    void build() {
        build(1, 1, n);
    }

    // 区间修改
    void rangeModify(int l, int r, const Tag v) {
        rangeModify(l, r, 1, 1, n, v);
    }

    // 区间查询
    i64 query(int l, int r) {
        Info res = query(l, r, 1, 1, n);
```

```
        return res.val;
    }
};
struct Tag {
    i64 agn = -2e18;
};
Tag operator + (const Tag& x, const Tag& y) {
    return y;
}

struct Info {
    i64 val {};
    int sz = 1;
};
Info operator + (const Info&x, const Info& y) {
    Info res {};
    res.val = x.val + y.val;
    res.sz = x.sz + y.sz;
    return res;
}
```

(4) LazySegmentTree

```
// 左闭右开
/*  the way to use it:
    1. create a struct to record the information you want
    2. overload the operator "+" with this struct
    3. declare your SGT with this struct and proper size
    *4. when using optional function, you should declare a lambda function to check if a particular
information is vaild
*/
template<class Info, class Tag>
struct LazySegmentTree {
    int n;
    std::vector<Info> info;
    std::vector<Tag> tag;
    LazySegmentTree() : n(0) {}
    LazySegmentTree(int n_, Info v_ = Info()) {
        init(n_, v_);
    }
    template<class T>
    LazySegmentTree(std::vector<T> init_) {
        init(init_);
    }
    void init(int n_, Info v_ = Info()) {
        init(std::vector(n_, v_));
    }
    template<class T>
    void init(std::vector<T> init_) {
        n = init_.size();
        info.assign(4 << std::__lg(n), Info());
        tag.assign(4 << std::__lg(n), Tag());
        std::function<void(int, int, int)> build = [&](int p, int l, int r) {
            if (r - l == 1) {
                info[p] = init_[l];
                return;
            }
            int m = (l + r) / 2;
            build(2 * p, l, m);
            build(2 * p + 1, m, r);
            pull(p);
        };
        build(1, 0, n);
    }
    void pull(int p) {
        info[p] = info[2 * p] + info[2 * p + 1];
    }
    void apply(int p, const Tag &v) {
        info[p].apply(v);
        tag[p].apply(v);
    }
    void push(int p) {
        apply(2 * p, tag[p]);
        apply(2 * p + 1, tag[p]);
        tag[p] = Tag();
    }
    void modify(int p, int l, int r, int x, const Info &v) {
        if (r - l == 1) {
```

```cpp
            info[p] = v;
            return;
        }
        int m = (l + r) / 2;
        push(p);
        if (x < m) {
            modify(2 * p, l, m, x, v);
        } else {
            modify(2 * p + 1, m, r, x, v);
        }
        pull(p);
    }
    // 单点修改
    void modify(int p, const Info &v) {
        modify(1, 0, n, p, v);
    }
    Info rangeQuery(int p, int l, int r, int x, int y) {
        if (l >= y || r <= x) {
            return Info();
        }
        if (l >= x && r <= y) {
            return info[p];
        }
        int m = (l + r) / 2;
        push(p);
        return rangeQuery(2 * p, l, m, x, y) + rangeQuery(2 * p + 1, m, r, x, y);
    }
    // 区间查询
    Info rangeQuery(int l, int r) {
        return rangeQuery(1, 0, n, l, r);
    }
    void rangeApply(int p, int l, int r, int x, int y, const Tag &v) {
        if (l >= y || r <= x) {
            return;
        }
        if (l >= x && r <= y) {
            apply(p, v);
            return;
        }
        int m = (l + r) / 2;
        push(p);
        rangeApply(2 * p, l, m, x, y, v);
        rangeApply(2 * p + 1, m, r, x, y, v);
        pull(p);
    }
    // 区间修改
    void rangeApply(int l, int r, const Tag &v) {
        return rangeApply(1, 0, n, l, r, v);
    }
    void half(int p, int l, int r) {
        if (info[p].act == 0) {
            return;
        }
        if ((info[p].min + 1) / 2 == (info[p].max + 1) / 2) {
            apply(p, {-(info[p].min + 1) / 2});
            return;
        }
        int m = (l + r) / 2;
        push(p);
        half(2 * p, l, m);
        half(2 * p + 1, m, r);
        pull(p);
    }
    void half() {
        half(1, 0, n);
    }

    // 区间查询满足条件的第一个（下标）找不到返回 -1
    template<class F>
    int findFirst(int p, int l, int r, int x, int y, F &&pred) {
        if (l >= y || r <= x) {
            return -1;
        }
        if (l >= x && r <= y && !pred(info[p])) {
            return -1;
        }
        if (r - l == 1) {
            return l;
        }
        int m = (l + r) / 2;
```

```cpp
        push(p);
        int res = findFirst(2 * p, l, m, x, y, pred);
        if (res == -1) {
            res = findFirst(2 * p + 1, m, r, x, y, pred);
        }
        return res;
    }
    template<class F>
    int findFirst(int l, int r, F &&pred) {
        return findFirst(1, 0, n, l, r, pred);
    }
    // 区间查询满足条件的最后一个
    template<class F>
    int findLast(int p, int l, int r, int x, int y, F &&pred) {
        if (l >= y || r <= x) {
            return -1;
        }
        if (l >= x && r <= y && !pred(info[p])) {
            return -1;
        }
        if (r - l == 1) {
            return l;
        }
        int m = (l + r) / 2;
        push(p);
        int res = findLast(2 * p + 1, m, r, x, y, pred);
        if (res == -1) {
            res = findLast(2 * p, l, m, x, y, pred);
        }
        return res;
    }
    template<class F>
    int findLast(int l, int r, F &&pred) {
        return findLast(1, 0, n, l, r, pred);
    }

    void maintainL(int p, int l, int r, int pre) {
        if (info[p].difl > 0 && info[p].maxlowl < pre) {
            return;
        }
        if (r - l == 1) {
            info[p].max = info[p].maxlowl;
            info[p].maxl = info[p].maxr = l;
            info[p].maxlowl = info[p].maxlowr = -inf;
            return;
        }
        int m = (l + r) / 2;
        push(p);
        maintainL(2 * p, l, m, pre);
        pre = std::max(pre, info[2 * p].max);
        maintainL(2 * p + 1, m, r, pre);
        pull(p);
    }
    void maintainL() {
        maintainL(1, 0, n, -1);
    }
    void maintainR(int p, int l, int r, int suf) {
        if (info[p].difr > 0 && info[p].maxlowr < suf) {
            return;
        }
        if (r - l == 1) {
            info[p].max = info[p].maxlowl;
            info[p].maxl = info[p].maxr = l;
            info[p].maxlowl = info[p].maxlowr = -inf;
            return;
        }
        int m = (l + r) / 2;
        push(p);
        maintainR(2 * p + 1, m, r, suf);
        suf = std::max(suf, info[2 * p + 1].max);
        maintainR(2 * p, l, m, suf);
        pull(p);
    }
    void maintainR() {
        maintainR(1, 0, n, -1);
    }
};

struct Tag {
    int x = 0;
```

```
    void apply(const Tag &t) & {
        x = std::max(x, t.x);
    }
};

struct Info {
    int x = 0;
    void apply(const Tag &t) & {
        x = std::max(x, t.x);
    }
};

Info operator+(const Info &a, const Info &b) {
    return {std::max(a.x, b.x)};
}
```

6、 Trie

```
const int N = 5e5 + 1;

int tree[N][26];
int cnt[N];
int tot;

int newNode() {
    int x = ++tot;
    for (int i = 0; i < 26; i++) {
        tree[x][i] = 0;
    }
    cnt[x] = 0;
    return x;
}

void init() {
    tot = 0;
    newNode();
}

void insert(std::string s) {
    int p = 1;
    for (auto i : s) {
        int x = i - 'a';
        if (!tree[p][x]) {
            tree[p][x] = newNode();
        }
        p = tree[p][x];
    }
    cnt[p]++;
}

int query(std::string s) {
    int p = 1;
    for (auto i : s) {
        int x = i - 'a';
        if (tree[p][x]) {
            p = tree[p][x];
        } else {
            return 0;
        }
    }
    return p;
}
```

7、李超线段树（最小值）

```
struct Line {
    i64 a, b;

    Line() : a(0), b(1e18) {}
    Line(i64 a_, i64 b_) : a(a_), b(b_) {}

    i64 cal(i64 x) {
        return a * x + b;
    }
};

struct Lichao {
```

```
    int n;
    std::vector<Line> t;

    Lichao() {}
    Lichao(int n_) {
        n = n_;
        t.assign(n_ << 2, {});
    }

    void add(int u, int l, int r, Line p) {
        int mid = (l + r) >> 1;
        if (p.cal(mid) < t[u].cal(mid)) {
            std::swap(p, t[u]);
        }
        if (p.cal(l) < t[u].cal(l)) {
            add(2 * u + 1, l, mid, p);
        }
        if (p.cal(r) < t[u].cal(r)) {
            add(2 * u + 2, mid + 1, r, p);
        }
    }

    i64 query(int u, int l, int r, int x) {
        i64 cur = t[u].cal(x);
        if (l == r) {
            return cur;
        }
        int mid = (l + r) >> 1;
        if (x <= mid) {
            return std::min(cur, query(2 * u + 1, l, mid, x));
        } else {
            return std::min(cur, query(2 * u + 2, mid + 1, r, x));
        }
    }

    void add(Line p) {
        add(0, 0, n - 1, p);
    }

    i64 query(int x) {
        return query(0, 0, n - 1, x);
    }
};
```

二、数论

0、基本数据运算方式

```
const int N = 1e5 + 10;
const int mod = 998244353;
std::vector<i64> fac(N + 1, 1), invfac(N + 1, 1);
i64 fpow(i64 a, i64 b) {
    i64 res = 1;
    while (b) {
        if (b & 1) res = res * a % mod;
        a = a * a % mod;
        b >>= 1;
    }
    return res;
}
void init(int n) {
    fac[0] = 1;
    for (int i = 1; i <= n; i++) {
        fac[i] = fac[i - 1] * i % mod;
    }
    invfac[n] = fpow(fac[n], mod - 2);
    for (int i = n - 1; i >= 0; i--) {
        invfac[i] = invfac[i + 1] * (i + 1) % mod;
    }
}
i64 C(int n, int m) {  // 组合数
    if (m > n || m < 0) {
        return 0;
    }
    return fac[n] * invfac[m] % mod * invfac[n - m] % mod;
}
i64 A(int n, int m) {  // 排列数
    if (m > n || m < 0) {
```

```
        return 0;
    }
    return fac[n] * invfac[n - m] % mod;
}
i64 catalan(int n) {  // 卡特兰数
    if (n < 0) {
        return 0;
    }
    return C(2 * n, n) * fpow(n + 1, mod - 2) % mod;
}
```

1 裴蜀定理

设$a_1, a_2, \ldots, a_n$是不全为零的整数，则存在整数$x_1, x_2, \ldots, x_n$，使得$a_1x_1 + a_2x_2 + \cdots + a_nx_n = \gcd(a_1, a_2, \ldots, a_n)$。

逆定理:

设$a_1, a_2, \ldots, a_n$是不全为零的整数，$d > 0$是$a_1, a_2, \ldots, a_n$的公因数，若存在整数$x_1, x_2, \ldots, x_n$，使得$a_1x_1 + a_2x_2 + \cdots + a_nx_n = d$，则$d = \gcd(a_1, a_2, \ldots,$

2、组合数C(n, k) (杨辉三角递推)

```
    vector<vector<i64>> C(size, vector<i64> (size));
    C[0][0] = 1;
    for (int i = 1; i < size; i++) {
        C[i][0] = 1;
        for (int j = 1; j <= i; j++) {
            C[i][j] = (C[i - 1][j] + C[i - 1][j - 1]) % q;
        }
    }
```

3、快速幂，快速乘

```
i64 fpow(i64 a, i64 b, i64 p) {
    i64 res = 1;
    while (b) {
        if (b & 1) res = res * a % p;
        a = a * a % p;
        b >>= 1;
    }
    return res;
}
i64 mul(i64 a, i64 b, i64 mod) {
    i64 res = a * b - i64(1.0L * a * b / mod) * mod;
    res %= mod;
    if (res < 0) {
        res += mod;
    }
    return res;
}
```

4、线性筛

```
// minp[i] 表示 i 除了 1 以外的最小除数（一定是个素数）
// minp[i] 为 0 表示 i 是素数
// pri 中存储了所有的 素数
constexpr int N = 4e5 + 1;
int minp[N];
std::vector<int> pri;
void sieve(int n) {
    minp[1] = 1;
    for (int i = 2; i <= n; i++) {
        if (!minp[i]) {
            pri.push_back(i);
        }
        for (auto j : pri) {
            if (i * j > n) break;
            minp[i * j] = j;
            if (i % j == 0) break;
        }
    }
}
```

5、矩阵运算

```
using Matrix = std::vector<std::vector<i64>>;

Matrix operator * (const Matrix& a, const Matrix& b) {
    Matrix res(a.size(), std::vector<i64> (b[0].size()));

    for (int i = 0; i < a.size(); i++) { // a 行
        for (int j = 0; j < b[0].size(); j++) { // b 列
            for (int k = 0; k < b.size(); k++) { // a 列, b 行
                res[i][j] = (res[i][j] + (a[i][k] * b[k][j]) % mod) % mod;
            }
        }
    }
    return res;
}

Matrix MatrixPow(Matrix a, i64 b) {
    int n = a.size();
    Matrix res(n, std::vector<i64> (n));

    for (int i = 0; i < n; i++) {
        res[i][i] = 1;
    }
    while (b > 0) {
        if (b & 1) {
            res = res * a;
        }
        a = a * a;
        b >>= 1;
    }
    return res;
}
```

6、CRT

```
template<typename T>
struct CRT {
    int n;
    std::vector<T> a, b, c, m;
    CRT(int n_) {
        n = n_;
        a.assign(n_, {});
        b.assign(n_, {});
        c.assign(n_, {});
        m.assign(n_, {});
    }

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

    T cal_inv(T a, T b) {
        T x, y;
        T g = exgcd(a, b, x, y);
        assert(g == 1);
        return (x % b + b) % b;
    }

    i64 mul(i64 a, i64 b, i64 mod) {
        i64 res = a * b - i64(1.0L * a * b / mod) * mod;
        res %= mod;
        if (res < 0) {
            res += mod;
        }
        return res;
    }

    T cal() {
        T M = 1;
        for (int i = 0; i < n; i++) {
```

```
            M *= m[i];
        }
        for (int i = 0; i < n; i++) {
            b[i] = M / m[i];
        }
        for (int i = 0; i < n; i++) {
            c[i] = mul(b[i], cal_inv(b[i], m[i]), M);
            c[i] %= M;
        }

        T res = 0;
        for (int i = 0; i < n; i++) {
            res += mul(c[i], a[i], M);
            res %= M;
        }
        return res;
    }
};
```

7、欧拉函数

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

// 求全部数的欧拉函数
constexpr int N = 1e6;
bool isprime[N + 1];
int phi[N + 1];
std::vector<int> pri;
void get_phi(int n) {
    std::fill(isprime + 2, isprime + n + 1, true);
    phi[1] = 1;
    for (int i = 2; i <= n; i++) {
        if (isprime[i]) {
            pri.push_back(i);
            phi[i] = i - 1;
        }
        for (auto p : pri) {
            if (i * p > n) {
                break;
            }
            isprime[i * p] = false;
            if (i % p == 0) {
                phi[i * p] = phi[i] * p;
                break;
            }
            phi[i * p] = phi[i] * (p - 1);
        }
    }
}
```

8、扩展欧几里得及计算逆元

```
i64 exgcd(i64 a, i64 b, i64 &x, i64 &y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    i64 g = exgcd(b, a % b, y, x);
    y -= a / b * x;
    return g;
}
```

```
i64 cal_inv(i64 a, i64 b) {
    i64 x, y;
    i64 g = exgcd(a, b, x, y);
    assert(g == 1);
    return (x % b + b) % b;
}
```

9、高斯消元

```
struct Gauss {
    int n;
    std::vector<std::vector<double>> a;
    int rank; // 系数矩阵的秩
    double eps = 1e-7;

    Gauss(int n_) {
        n = n_;
        a.assign(n_, std::vector<double> (n_ + 1, {}));
    }

    int cal() {
        rank = 0;
        int col = 0; // 新增列跟踪变量

        for (int row = 0; row < n && col < n; col++) {
            // 1. 找当前col列的主元
            int max = row;
            for (int i = row; i < n; i++) {
                if (fabs(a[i][col]) > fabs(a[max][col])) {
                    max = i;
                }
            }
            // 2. 跳过全零列
            if (fabs(a[max][col]) < eps) {
                continue; // 修复括号错误
            }
            // 3. 交换行
            std::swap(a[row], a[max]);
            // 4. 归一化当前行（针对col列）
            double pivot = a[row][col];
            for (int j = col; j <= n; j++) { // 从col开始
                a[row][j] /= pivot;
            }
            // 5. 消去其他行（仅处理非当前行）
            for (int j = 0; j < n; j++) {
                if (j != row && fabs(a[j][col]) > eps) {
                    double factor = a[j][col];
                    for (int k = col; k <= n; k++) {
                        a[j][k] -= factor * a[row][k];
                    }
                }
            }
            row++; // 只有找到有效主元才增加行
            rank++; // 正确统计有效主元数
        }
        // 检查无解（后续行的常数项非零）
        for (int i = rank; i < n; i++) {
            if (fabs(a[i][n]) > eps) {
                return 0;
            }
        }
        // 2 无穷多解
        // 1 唯一解
        return (rank == n) ? 1 : 2;
    }
};
```

三、杂项

1、归并排序求逆序对

```
int c[size], res;
inline void ms(int l, int r, int t[]) {
    if (l == r) return;
    int mid = l + r >> 1;
    ms(l, mid, t), ms(mid + 1, r, t);
```

```
    int p1 = l, p2 = mid + 1, idx = 0;
    while (p1 <= mid && p2 <= r) {
        if (t[p1] <= t[p2]) c[++idx] = t[p1++];
        else {
            res += mid - p1 + 1;
            c[++idx] = t[p2++];
        }
    }
    while (p1 <= mid) c[++idx] = t[p1++];
    while (p2 <= r) c[++idx] = t[p2++];
    for (int i = 1; i <= idx; i++) t[l + i - 1] = c[i];
}
```

2、__int128输入输出(注意不要关闭同步流)

```
__int128 read() {
    char arr[30];
    __int128 res = 0;
    scanf("%s", arr);
    for (int i = 1; i <= strlen(arr); i++) {
        res *= 10;
        res += arr[i]-'0';
    }
    return res;
}
void show (__int128 num) {
    if (num > 9) { show(num / 10); }
    putchar(num % 10 + '0');
}
```

3、异或哈希

```
    std::vector<u64> a(n + 1), pre(n + 1);
    for (int i = 1; i <= n; i++) {
        std::cin >> a[i];
    }

    u64 max = *std::max_element(a.begin(), a.end());

    std::mt19937_64 rnd(time(0));
    std::vector<u64> code(max + 1); // max是a[i]的最大值
    for (int i = 1; i <= max; i++) {
        code[i] = rnd();
    }

    for (int i = 1; i <= n; i++) {
        pre[i] = pre[i - 1] ^ code[a[i]];
    }
```

4、对拍

```
#!/bin/bash
t=0
while true; do
    let "t = $t + 1"
    printf $t
    printf ":\n"
    ./random > data.txt
    ./solve < data.txt > solve.out
    ./std < data.txt > std.out

    if diff solve.out std.out; then
        printf "AC\n"
    else
        printf "WA\n"
        cat data.txt
        cat std.out
        cat solve.out
        break
    fi
done
```

```
#include <bits/stdc++.h>
int main() {
    int t = 0;
```

```cpp
    while (1) {
        std::cout << "test: " << t++ << std::endl;
        system("gen.exe > data.in");
        system("std.exe < data.in > std.out");
        system("solve.exe < data.in > solve.out");

        if (system("fc std.out solve.out > diff.log")) {
            std::cout << "WA" << std::endl;
            break;
        }
        std::cout << "AC" << std::endl;
    }
    return 0;
}
```

```cpp
#include <bits/stdc++.h>
std::string rand_str(const int len, int k)  /*参数为字符串的长度*/
{
    /*初始化*/
    std::string str;                 /*声明用来保存随机字符串的str*/
    char c;                     /*声明字符c，用来保存随机生成的字符*/
    int idx;                    /*用来循环的变量*/
    /*循环向字符串中添加随机生成的字符*/
    for(idx = 0; idx < len; idx++)
    {
        /*rand()%26是取余，余数为0~25加上'a',就是字母a~z,详见asc码表*/
        c = 'a' + rand() % k;
        str.push_back(c);       /*push_back()是string类尾插函数。这里插入随机字符c*/
    }
    return str;                 /*返回生成的随机字符串*/
}
int main() {
    std::mt19937 rnd(time(0));
    return 0;
}
```

```cpp
for (int i = dp[u]._Find_first(); i < dp[u].size(); i = dp[u]._Find_next(i)) {
    // _Find_first() 找到 bitset 中第一个为 1 的位置。
    // _Find_next(i) 找到下一个为 1 的位置。
}
```

四、图论

1、倍增求lca

```cpp
    int t = int(log(n) / log(2)) + 1;
    std::vector<std::vector<int>> f(n + 1, std::vector<int> (t + 1));

    std::vector<int> d(n + 1);
    d[s] = 1;
    std::queue<int> q;
    q.push(s);
    while (!q.empty()) {
        int x = q.front();
        q.pop();
        for (auto y : e[x]) {
            if (d[y]) continue;
            d[y] = d[x] + 1;
            f[y][0] = x;
            for (int i = 1; i <= t; i++) {
                f[y][i] = f[f[y][i - 1]][i - 1];
            }
            q.push(y);
        }
    }

    auto lca = [&](int x, int y) {
        if (d[x] > d[y]) std::swap(x, y);
        for (int i = t; i >= 0; i--) {
            if (d[f[y][i]] >= d[x]) {
                y = f[y][i];
            }
        }
        if (x == y) return x;
        for (int i = t; i >= 0; i--) {
            if (f[x][i] != f[y][i]) {
                x = f[x][i];
```

```
            y = f[y][i];
        }
    }
    return f[x][0];
};
```

2、树的重心

```
std::vector<int> p(n), dep(n), siz(n), in(n), ord(n);
int cur = 0;
auto dfs = [&](auto&& self, int u) -> void {
    siz[u] = 1;
    in[u] = cur++;
    ord[in[u]] = u;
    for (auto v : e[u]) {
        if (v == p[u]) {
            continue;
        }
        p[v] = u;
        dep[v] = dep[u] + 1;
        self(self, v);
        siz[u] += siz[v];
    }
};

p[0] = -1;
dfs(dfs, 0);

auto find = [&](auto&& self, int u) -> int {
    for (auto v : e[u]) {
        if (v == p[u] || 2 * siz[v] <= n) {
            continue;
        }
        return self(self, v);
    }
    return u;
};

int rt = find(find, 0);
dep[rt] = 0;
p[rt] = -1;
cur = 0;
dfs(dfs, rt);
```

3、树的直径

```
// bfs
std::vector<int> dis, fa;
auto bfs = [&](int s) -> int {
    dis.assign(n, -1);
    fa.assign(n, -1);

    std::queue<int> q;
    q.push(s);
    dis[s] = 0;

    while (!q.empty()) {
        auto u = q.front();
        q.pop();
        for (auto v : adj[u]) {
            if (dis[v] == -1) {
                dis[v] = dis[u] + 1;
                fa[v] = u;
                q.push(v);
            }
        }
    }

    // 最大 dis 索引
    return std::max_element(dis.begin(), dis.end()) - dis.begin();
};


// dfs 记录路径 （无法处理负权边）
int tar = 0, max = 0;
std::vector<int> pre(n + 1);
auto dfs = [&](auto &&self, int u, int fa, int w, int tag) -> void {
```

```
        if (w > max) {
            max = w;
            tar = u;
        }
        for (auto [y, ww] : e[u]) {
            if (y == fa) continue;
            if (tag == 1) {
                pre[y] = u;
            }
            self(self, y, u, w + ww, tag);
        }
    };
    dfs(dfs, 1, -1, 0, 0);
    int p = tar;
    tar = 0, max = 0;
    dfs(dfs, p, -1, 0, 1);
    int q = tar;

    // 树形dp
    int ans = -inf;
    std::vector<int> dis(n + 1);
    auto dp = [&](auto &&self, int u, int fa) -> void {
        for (auto [y, w] : e[u]) {
            if (y == fa) continue;
            self(self, y, u);
            ans = std::max(ans, dis[y] + dis[u] + w);
            dis[u] = std::max(dis[u], dis[y] + w);
        }
    };
    dp(dp, 1, -1);
```

4、二分图最大匹配（匈牙利算法）$O(nm)$

```
    std::vector<int> vis(n), v(m, -1); // v[y] 表示 y 的匹配，vis[u] 表示 u 是否被访问过
    auto find = [&](auto &&self, int u) -> bool {
        vis[u] = 1;
        for (auto y : e[u]) {
            if (!v[y] || (!vis[v[y]] && self(self, v[y]))) {
                v[y] = u;
                return true;
            }
        }
        return false;
    };
    auto match = [&](int x) {
        int res = 0;
        v.assign(m, -1);
        for (int i = 0; i < x; i++) {
            vis.assign(n, 0);
            if (find(find, i)) {
                res++;
            }
        }
        return res;
    };
```

5、Dijkstra

```
template<typename T>
struct Dijkstra {
    struct Node {
        int u;
        T w;
        bool operator < (const Node& t) const {
            return w > t.w;
        }
    };

    const int inf = 2e9;
    int n;
    Dijkstra() {}
    Dijkstra(int n) {
        init(n); // 从 0 开始存储
    }

    std::vector<std::vector<std::pair<int, T>>> adj; // 邻接表存图
    std::vector<T> dis;                // 距离
```

```
    // 初始化
    void init(int n) {
        this->n = n;
        adj.assign(n, {});
        dis.assign(n, inf);          // 初始化为无穷大
    }

    // 加边 u v是边的顶点，w是边权
    void addEdge(int u, int v, T w) {
        adj[u].push_back({v, w});
        // adj[v].push_back({u, w});
    }

    // 单源非负权最短路 s是源
    void shortest_path(int s) {
        std::vector<bool> vis(this->n);
        // 堆优化
        std::priority_queue<Node> pq;
        pq.push({s, 0});
        dis[s] = 0;

        while (!pq.empty()) {
            int u = pq.top().u;
            pq.pop();
            if (vis[u]) continue;
            vis[u] = true;
            for (auto [y, w] : adj[u]) {
                if (dis[y] > dis[u] + w) {
                    dis[y] = dis[u] + w;
                    pq.push({y, dis[y]});
                }
            }
        }
        // dis 已被更新
    }
};
```

6、SCC(Tarjan)

```
struct SCC {
    int n;
    std::vector<std::vector<int>> adj;
    std::vector<int> stk;
    std::vector<int> dfn, low, bel;     // bel[i] 表示 i 所在的 SCC
    int cur, cnt;   // cur 表示当前时间戳, cnt 表示 SCC 编号

    SCC() {}
    SCC(int n) {
        init(n);
    }

    void init(int n) {
        this->n = n;
        adj.assign(n, {});
        dfn.assign(n, -1);
        low.resize(n);
        bel.assign(n, -1);
        stk.clear();
        cur = cnt = 0;
    }

    void addEdge(int u, int v) {
        adj[u].push_back(v);
    }

    void dfs(int x) {
        dfn[x] = low[x] = cur++;
        stk.push_back(x);

        for (auto y : adj[x]) {
            if (dfn[y] == -1) {
                dfs(y);
                low[x] = std::min(low[x], low[y]);
            } else if (bel[y] == -1) {
                low[x] = std::min(low[x], dfn[y]);
            }
        }
```

```
        if (dfn[x] == low[x]) {
            int y;
            do {
                y = stk.back();
                bel[y] = cnt;
                stk.pop_back();
            } while (y != x);
            cnt++;
        }
    }

    std::vector<int> work() {
        for (int i = 0; i < n; i++) {
            if (dfn[i] == -1) {
                dfs(i);
            }
        }
        return bel;
    }
};
```

7、 EBCC

```
// 用于存储 DFS 过程中经过的边（调试/分析用，不是核心必须的）
std::set<std::pair<int, int>> E;
/*
 * EBCC = Edge-Biconnected Components
 * 作用：对无向图进行 边双连通分量 分解
 * 主要用途：判桥 / 建立桥树 / 缩点分析
*/
struct EBCC {
    int n;
    std::vector<std::vector<int>> adj;
    std::vector<int> stk;
    std::vector<int> dfn, low, bel; // dfn[i] = 时间戳, low[i] = 能回溯的最小时间戳 bel[i] = 节点 i 所属的 EBCC 编号
    int cur, cnt;                   // cur = 当前时间戳, cnt = EBCC 数

    EBCC() {}
    EBCC(int n) {
        init(n);
    }

    void init(int n) {
        this->n = n;
        adj.assign(n, {});
        dfn.assign(n, -1);
        low.resize(n);
        bel.assign(n, -1);
        stk.clear();
        cur = cnt = 0;
    }

    void addEdge(int u, int v) {
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    void dfs(int x, int p) {
        dfn[x] = low[x] = cur++;
        stk.push_back(x);

        for (auto y : adj[x]) {
            if (y == p) {
                continue;
            }
            if (dfn[y] == -1) {
                E.emplace(x, y);
                dfs(y, x);
                low[x] = std::min(low[x], low[y]);
            } else if (bel[y] == -1 && dfn[y] < dfn[x]) {
                E.emplace(x, y);
                low[x] = std::min(low[x], dfn[y]);
            }
        }

        if (dfn[x] == low[x]) {
            int y;
            do {
```

```
                y = stk.back();
                bel[y] = cnt;
                stk.pop_back();
            } while (y != x);
            cnt++;
        }
    }
    // 主接口：运行 EBCC 分解，返回 bel[]（点对应的分量编号）
    std::vector<int> work() {
        dfs(0, -1);
        return bel;
    }

    struct Graph {
        int n;                                      // 压缩后的点数 = 分量数
        std::vector<std::pair<int, int>> edges; // 分量间的边
        std::vector<int> siz;                   // 每个分量的点数
        std::vector<int> cnte;                  // 每个分量内部的边数
    };
    Graph compress() {
        Graph g;
        g.n = cnt;
        g.siz.resize(cnt);
        g.cnte.resize(cnt);
        for (int i = 0; i < n; i++) {
            g.siz[bel[i]]++;
            for (auto j : adj[i]) {
                if (bel[i] < bel[j]) {
                    g.edges.emplace_back(bel[i], bel[j]);
                } else if (i < j) {
                    g.cnte[bel[i]]++;
                }
            }
        }
        return g;
    }
};
```

8、树链剖分

```
struct HLD {
    int n;
    std::vector<int> siz, top, dep, parent, in, out, seq;
    std::vector<std::vector<int>> adj;
    int cur;

    HLD() {}
    HLD(int n) {
        init(n);
    }
    void init(int n) {
        this->n = n;
        siz.resize(n);
        top.resize(n);
        dep.resize(n);
        parent.resize(n);
        in.resize(n);
        out.resize(n);
        seq.resize(n);
        cur = 0;
        adj.assign(n, {});
    }
    void addEdge(int u, int v) {
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    void work(int root = 0) {
        top[root] = root;
        dep[root] = 0;
        parent[root] = -1;
        dfs1(root);
        dfs2(root);
    }
    void dfs1(int u) {
        if (parent[u] != -1) {
            adj[u].erase(std::find(adj[u].begin(), adj[u].end(), parent[u]));
        }
```

```
        siz[u] = 1;
        for (auto &v : adj[u]) {
            parent[v] = u;
            dep[v] = dep[u] + 1;
            dfs1(v);
            siz[u] += siz[v];
            if (siz[v] > siz[adj[u][0]]) {
                std::swap(v, adj[u][0]);
            }
        }
    }
    void dfs2(int u) {
        in[u] = cur++;
        seq[in[u]] = u;
        for (auto v : adj[u]) {
            top[v] = v == adj[u][0] ? top[u] : v;
            dfs2(v);
        }
        out[u] = cur;
    }
    int lca(int u, int v) {
        while (top[u] != top[v]) {
            if (dep[top[u]] > dep[top[v]]) {
                u = parent[top[u]];
            } else {
                v = parent[top[v]];
            }
        }
        return dep[u] < dep[v] ? u : v;
    }

    int dist(int u, int v) {
        return dep[u] + dep[v] - 2 * dep[lca(u, v)];
    }

    int jump(int u, int k) { // 返回 u 的深度为 k 的祖先节点
        if (dep[u] < k) {
            return -1;
        }

        int d = dep[u] - k;

        while (dep[top[u]] > d) {
            u = parent[top[u]];
        }

        return seq[in[u] - dep[u] + d];
    }

    bool isAncester(int u, int v) { // 判断 u 是否是 v 的祖先
        return in[u] <= in[v] && in[v] < out[u];
    }

    int rootedParent(int u, int v) { // 当把树“以 u 为根”时，求 v 在什么位置？更常见的用途是
        std::swap(u, v);
        if (u == v) {
            return u;
        }
        if (!isAncester(u, v)) {
            return parent[u];
        }
        auto it = std::upper_bound(adj[u].begin(), adj[u].end(), v, [&](int x, int y) {
            return in[x] < in[y];
        }) - 1;
        return *it;
    }

    int rootedSize(int u, int v) { // 把树“以 u 为根”时计算 v 的子树大小（也可以理解为在把树根切换到 u 后，v 那个连通块/子树的节点数）
        if (u == v) {
            return n;
        }
        if (!isAncester(v, u)) {
            return siz[v];
        }
        return n - siz[rootedParent(u, v)];
    }

    int rootedLca(int a, int b, int c) { // 求三点 a,b,c 的“三点公共祖先” O(logn)
        return lca(a, b) ^ lca(b, c) ^ lca(c, a);
```

```
    }
};
```

9、 SPFA O(nm)

```
    for (int i = 0; i < n; i++) {
        adj[n].push_back({i, 0ll});
    }

    std::vector<i64> dis(n + 1, inf);
    auto spfa = [&]() -> bool {
        dis[n] = 0;
        std::queue<int> q;
        q.push(n);
        // cnt【u]: 到 u 的最短路经过的边数，vis[u]：u 是否在队列中
        std::vector<int> cnt(n + 1), vis(n + 1);
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            vis[u] = 0;
            for (auto [v, w] : adj[u]) {
                if (dis[v] > dis[u] + w) {
                    dis[v] = dis[u] + w;
                    cnt[v] = cnt[u] + 1;
                    if (cnt[v] > n) {
                        return false;
                    }
                    if (!vis[v]) {
                        vis[v] = 1;
                        q.push(v);
                    }
                }
            }
        }
        return true;
    };
```

五、DP

1、数位 DP

```
i64 dp[14], ten[14]; // dp[i] 表示为 i 位数时每种数字有多少个，ten[i] 表示 10^i
void cal(i64 x, std::vector<i64>& cnt) {
    std::vector<int> num(1);
    while (x) {
        num.push_back(x % 10);
        x /= 10;
    }
    // 299
    for (int i = int(num.size()) - 1; i >= 1; i--) {
        // [00, 99]
        for (int j = 0; j <= 9; j++) {
            cnt[j] += dp[i - 1] * num[i];
        }
        // [000, 200) 中的 0 和 1
        for (int j = 0; j < num[i]; j++) {
            cnt[j] += ten[i - 1];
        }
        i64 num2 = 0;
        for (int j = i - 1; j >= 1; j--) {
            num2 = num2 * 10 + num[j];
        }
        // num2: 99 计算 2 在百位出现的次数
        cnt[num[i]] += num2 + 1; // cnt[2] += 99 + 1
        // 去除前导零 [00, 99]
        cnt[0] -= ten[i - 1]; // cnt[0] -= ten[3 - 1] = cnt[0] - 100
    }
}
void init() {
    ten[0] = 1;
    for (int i = 1; i <= 14; i++) {
        dp[i] = i * ten[i - 1];
        ten[i] = 10 * ten[i - 1];
    }
}
```