ICPC 模板 20231113

第零部分 引言

11 月 13 日更新说明：

- 新增了树状数组。
- 新增了单调队列。
- 新增了后缀数组。
- 新增了 lucas 定理的公式。
- 新增了 BSGS 的描述。
- 新增了计算几何，代码来自 `https://github.com/F0RE1GNERS/template/blob/master/4-计算几何.md`
- 并查集模板有所更新，新增了带权并查集。
- 线段树模板有所更新，新增了维护 add + cover 的线段树例子，新增了线段树上二分。线段树的代码有所修改，使得可以在 C++ 17 标准下运行。
- 在 ACM 赛场上建议手动取模，故新增了快速幂和逆元。 但 Mint 模板依然保留。
- 从代码长度考虑， ST 表模板的 vector 改为普通数组以避免多次 resize。
- 多项式部分，仅保留 NTT，取消 "多项式" 章节，并入 "数学" 章节。
- 红黑树模板并入第七部分（"杂项"）
- lca 模板，将 dfs 部分和 lca 部分分离。精简了代码。
- 树上背包 dp 优化，精简了代码，并归类为 "杂项"。
- 修改了引言部分的部分表述。
- 修改第一部分为 "常用算法模板"，将常用数据结构划分到第四部分。

这份模板中的代码必须满足：

- 它可以在 C++ 17 标准下运行。
- 优先保证代码简洁易读易抄写，其次是常数。
- 模板中的代码，尽可能封装成函数或类。但是，如果将一道经典例题的 AC 代码放入模板中，则不受这一点限制。
- 模板中的代码，一般而言需要说明 **代码的使用方法（或算法功能，或使用例）** 。

这份模板中的代码**不需要**满足：

- 严格的代码规范。

需要注意的是：

- 默认选手使用了 `#include<bits/stdc++.h>` ，所以一般情况下，不应该引用头文件。
- 默认选手使用了 `using namespace std;` ，所以不应该添加 `std::` 前缀。

- 默认选手使用了 `using ll = long long;` ，所以应该使用 `ll` 代替 long long。
- 允许出现轻微压行。
- 一般而言采用 1-index，除非特别说明。

程序模板

```
#include<bits/stdc++.h>
using namespace std;
using ll = long long;
void solve() {

}
int main() {
    ios::sync_with_stdio(false); cin.tie(nullptr);
    //int t; cin >> t; while(t--)
    solve();
    return 0;
}
```

第一部分 常用算法模板

二分查找

```
using ll = long long;

ll lower(ll l, ll r, ll target, function< ll(ll) > f) {
    if(f(r) < target) return r + 1;
    while (l < r) {
        ll mid = (l + r) / 2;
        if (f(mid) < target)
            l = mid + 1;
        else
            r = mid;
    }
    return l;
}
```

> **功能：在 [l, r] 范围内，求最小的 x 使得 f(x) >= target**
>
> **例子：求 [1, 10^9] 中最小的数 x， 使得递增函数 x^2+5x 的值达到或超过给定的数 K**

```
    ll val = lower(1, 1e9, K, [](ll x){
        return x * (x + 5);
    });
```

Mint 自动取模

```
template<int mod> class mint {
public:
    unsigned int x = 0;
    // int get_modular() { return mod; }
    mint inv() const { return pow(mod-2); }
    mint pow(long long t) const {
        assert(t >= 0 && x >= 0);
        mint res = 1, cur = x;
        for(; t; t>>=1) {
            if(t & 1) res *= cur;
            cur *= cur;
        }
        return res;
    }
    mint() = default;
    mint(unsigned int t): x(t % mod) { }
    mint(int t){ t %= mod; if(t < 0) t += mod; x = t; }
    mint(long long t){ t %= mod; if(t < 0) t += mod; x = t; }

    mint& operator+= (const mint& t){ x += t.x; if(x >= mod) x-=mod; return *this; }
    mint& operator-= (const mint& t){ x += mod - t.x; if(x >= mod) x-=mod; return *this; }
    mint& operator*= (const mint& t){ x = (unsigned long long)x * t.x % mod; return *this; }
    mint& operator/= (const mint& t){ *this *= t.inv(); return *this; }
    mint& operator^= (const mint& t){ *this = this->pow(t.x); return *this; }
    mint operator+ (const mint& t){ return mint(*this) += t; }
    mint operator- (const mint& t){ return mint(*this) -= t; }
    mint operator* (const mint& t){ return mint(*this) *= t; }
    mint operator/ (const mint& t){ return mint(*this) /= t; }
    mint operator^ (const mint& t){ return mint(*this) ^= t; }
    bool operator== (const mint& t){ return x == t.x; }
    bool operator!= (const mint& t){ return x != t.x; }
    bool operator< (const mint& t){ return x < t.x; }
    bool operator<= (const mint& t){ return x <= t.x; }
```

```
    bool operator> (const mint& t){ return x > t.x; }
    bool operator>= (const mint& t){ return x >= t.x; }
    friend istream& operator>>(istream& is, mint& t){ return is >> t.x; }
    friend ostream& operator<<(ostream& os, const mint& t){ return os << t.x; }
    friend mint operator+ (int y, const mint& t){ return mint(y) + t.x; }
    friend mint operator- (int y, const mint& t){ return mint(y) - t.x; }
    friend mint operator* (int y, const mint& t){ return mint(y) * t.x; }
    friend mint operator/ (int y, const mint& t){ return mint(y) / t.x; }
};
const int mod = 998244353;
using Mint = mint<mod>;
```

快速幂 & 逆元

```
const int mod = 1e9 + 7;
ll powmod(ll x, ll t) {
    ll res = 1, cur = x;
    for(; t; t>>=1) {
        if(t & 1) res = res * cur % mod;
        cur = cur * cur % mod;
    }
    return res;
}
ll inv(ll x) {return pow(x, mod - 2);}
```

第二部分 图论

最短路

```
vector<ll> dijkstra(vector<vector<array<ll, 2>>> adj, int start)
{
    int n = adj.size();
    priority_queue<array<ll, 2>, vector<array<ll, 2>>, greater<>> pq;
    vector<ll> dist(n+1, (1ll<<31)-1);
    vector<ll> vis(n+1, 0);
    dist[start] = 0;
    pq.push({0, start});

    while(!pq.empty())
    {
        auto [disx, from] = pq.top(); pq.pop();
        if(vis[from]) continue;
```

```cpp
        vis[from] = 1;

        for(auto [to, dis]: adj[from])
        {
            if(disx + dis < dist[to])
            {
                dist[to] = disx + dis;
                pq.push((array<ll, 2>){dist[to], to});
            }
        }
    }
    return dist;
}
```

> **功能：堆优化 djikstra 求最短路，时间复杂度是 $\Theta((n+m)\log n)$**
>
> **例子：(P4779) 建图，然后求以 s 为源点的最短路**
>
> ```cpp
> int n, m, s; cin >> n >> m >> s;
>
> vector<vector<array<ll, 2>>> adj(n + 1);
>
> for(int i = 1; i <= m; i++) {
>     int x, y, z;
>     cin >> x >> y >> z;
>     adj[x].push_back({y, z});
> }
>
> auto v = dijkstra(adj, s);
> for(int i = 1; i <= n; i++) cout << v[i] << " ";
> ```

最近公共祖先（倍增法）

```cpp
vector<int> d(n + 1);
vector<array<int, 20>> pa(n + 1);
function<void(int, int)> dfs = [&](int now, int fa){
    d[now] = d[fa] + 1;
    pa[now][0] = fa;
    for(int i = 1; i < 20; i++) pa[now][i] = pa[pa[now][i - 1]][i - 1];
    for(int it : v[now]) {
        if(it == fa) continue;
        dfs(it, now);
```

```
    }
};

dfs(1, 1); // root = 1

auto lca = [&](int u, int v) {
    if(d[u] < d[v]) swap(u, v);
    int t = d[u] - d[v];
    for(int i = 19; i >= 0; i--) if(t >> i & 1) u = pa[u][i];

    if(u == v) return u;

    for(int i = 19; i >= 0; i--) {
        if(pa[u][i] != pa[v][i]) {
            u = pa[u][i];
            v = pa[v][i];
        }
    }

    return pa[u][0];
};
```

> **说明：预处理 $\Theta(n\log n)$。用于一棵树上的 LCA 查询，单次查询 $\Theta(\log n)$**
>
> **例子：(P3379) 建树，然后多次查询 LCA**
>
> ```
> int n, m, s;
> cin >> n >> m >> s;
>
> vector<vector<int>> v(n + 1);
> for(int i = 1; i < n; i++) {
>     int x, y;
>     cin >> x >> y;
>     v[x].push_back(y);
>     v[y].push_back(x);
> }
> ...
> while(m--)
> {
>     int u, v; cin >> u >> v;
>     cout << lca(u, v) << '\n';
> }
> ```

第三部分 字符串

KMP 字符串匹配（洛谷 P 3375）

题意：给出 s 和 t，求 s 中 t 的所有出现，然后输出 t 的所有 border 长度。

需要注意，s,t 在这里是 $1-index$ 的，如果采用 std::string，需要使用类似 `s=" "+s;` 的技巧。

```
#include<iostream>
#include<cstring>
#define MAXN 1000010
using namespace std;
int kmp[MAXN];
int la,lb,j;
char a[MAXN],b[MAXN];
int main()
{
    cin>>a+1;
    cin>>b+1;
    la=strlen(a+1);
    lb=strlen(b+1);
    for (int i=2;i<=lb;i++)
    {
       while(j&&b[i]!=b[j+1])
        j=kmp[j];
       if(b[j+1]==b[i])j++;
        kmp[i]=j;
    }
    j=0;
    for(int i=1;i<=la;i++)
    {
          while(j>0&&b[j+1]!=a[i])
            j=kmp[j];
          if (b[j+1]==a[i])
            j++;
          if (j==lb) {cout<<i-lb+1<<endl;j=kmp[j];}
    }

    for (int i=1;i<=lb;i++)
        cout<<kmp[i]<<" ";
    return 0;
}
```

字符串哈希

```cpp
const ll base1 = 93, mod1 = 998244353;
const ll base2 = 97, mod2 = 1e9 + 7;
const int N = 1e6 + 5;
ll p1[N], p2[N];

struct strHash{
    int n;
    vector<ll> h1, h2;
    strHash(string s) : n(s.length() - 1), h1(n + 1), h2(n + 1) {
        p1[0] = p2[0] = 1;
        for(int i = 1; i <= n; i++) {
            p1[i] = p1[i - 1] * base1 % mod1;
            p2[i] = p2[i - 1] * base2 % mod2;
            h1[i] = (h1[i - 1] * base1 + s[i]) % mod1;
            h2[i] = (h2[i - 1] * base2 + s[i]) % mod2;
        }
    }
    array<ll, 2> get_hash(int l,int r) {
        ll res1 = ((h1[r] - h1[l - 1] * p1[r - l + 1]) % mod1 + mod1) % mod1,
           res2 = ((h2[r] - h2[l - 1] * p2[r - l + 1]) % mod2 + mod2) % mod2;
        return {res1, res2};
    }
};
```

> **说明：双 hash，不采用自然取模**
>
> s 必须是 1-index 的。

SA 后缀数组

```cpp
int m = 75;
int n;
vector<int> rk, tp, sa, t;
void rsort() {
    for(int i=0;i<=m;i++) t[i] = 0;
    for(int i=1;i<=n;i++) t[rk[i]]++;
    for(int i=1;i<=m;i++) t[i]+=t[i-1];
    for(int i=n;i>=1;i--) sa[t[rk[tp[i]]]--] = tp[i];
}
void suffix_sort(const string& s) {
    n = s.length() - 1;
```

```
    rk.resize(n + 1); tp.resize(n + 1); sa.resize(n + 1); t.resize(max(n, m) + 1);
    for(int i=1;i<=n;i++) rk[i] = s[i] - '0' + 1, tp[i] = i;
    rsort();
    int p = 0;
    for (int w = 1, p = 0; p < n; m = p, w <<= 1) {
        p = 0;
        for (int i = 1; i <= w; i++) tp[++p] = n - w + i;
        for (int i = 1; i <= n; i++) if (sa[i] > w) tp[++p] = sa[i] - w;

        rsort();
        swap(tp, rk);

        rk[sa[1]] = p = 1;
        for (int i = 2; i <= n; i++)
            rk[sa[i]] = (tp[sa[i - 1]] == tp[sa[i]] && tp[sa[i - 1] + w] == tp[sa[i] + w]) ? p : ++p;
    }
}
```

得到 Height 数组

```
void get_height() {
    int k=0;
    for(int i=1;i<=n;i++)rk[sa[i]]=i; //初始化rk[i]
    for(int i=1;i<=n;i++)//这里其实是枚举rk[i]
    {
        if(rk[i]==1)continue; //height[1]=0
        if(k)k--; //h[i]>=h[i-1]-1,更新k然后一位位枚举
        int j=sa[rk[i]-1];//前一位字符串
        while(i+k<=n&&j+k<=n&&s[i+k]==s[j+k])k++;//一位位枚举
        height[rk[i]]=k;//h[i]=height[rk[i]]
    }
}
```

> 相关公式:
>
> height[i] >= height[i - 1] - 1
>
> lcp(i, j) = min(height[i..j])

Manacher 算法（洛谷 P3805）

```
#include<bits/stdc++.h>
using namespace std;
#define rep(i, s, t) for(int i = s; i <= t; ++ i)
#define maxn 22000005
int n, m, cnt, p[maxn], mid, mr, Ans;
char c[maxn], s[maxn];
void build() {
    scanf("%s", c + 1), n = strlen(c + 1), s[++ cnt] = '~', s[++ cnt] = '#';
    rep(i, 1, n) s[++ cnt] = c[i], s[++ cnt] = '#';
    s[++ cnt] = '!';
}
void solve() {
    rep(i, 2, cnt - 1) {
        if(i <= mr) p[i] = min(p[mid * 2 - i], mr - i + 1);
        else p[i] = 1;
        while(s[i - p[i]] == s[i + p[i]]) ++ p[i];
        if(i + p[i] > mr) mr = i + p[i] - 1, mid = i;
        Ans = max(Ans, p[i]);
    }
    printf("%d", Ans - 1);
}
int main() { return build(), solve(), 0; }
```

可得到字符串中以每个字符为回文中心的最长回文串长度。

第四部分 数据结构

并查集

```
struct dsu {
    vector<int> p;
    dsu(int n) { p.resize(n + 1); for(int i = 1; i <= n; i++) p[i] = i; }
    int find(int x) { if(x != p[x]) p[x] = find(p[x]); return p[x]; }
    void merge(int x, int y) { p[find(x)] = find(y); }
};
```

> **例子：建立一个并查集，连接 (1，3) 和 (1，2) 两条边，并查询 2 和 3 是否在同一集合**

```
dsu d(0721);
d.merge(1, 3);
d.merge(1, 2);
assert(d.find(2) == d.find(3));
```

拓展：并查集可以实时查询每个集合的信息（例如集合大小），需要对 merge 函数进行一些处理。

使用并查集，连接 (1，3) 和 (1，2) 两条边，然后查询 1 所在集合的大小

```
struct dsu {
    vector<int> p, s;
    dsu(int n) { p.resize(n + 1); s.resize(n + 1); for(int i = 1; i <= n; i++) p[i] = i, s[i] = 1; }
    int find(int x) { if(x != p[x]) p[x] = find(p[x]); return p[x]; }
    void merge(int x, int y) { x = find(x); y = find(y); if(x != y) p[x] = y, s[y] += s[x]; }
};

dsu d(0721);
d.merge(1, 3);
d.merge(1, 2);
assert(d.s[d.find(1)] == 3);
```

拓展：带权并查集

```
struct dsu {
    vector<ll> p, w;
    dsu(int n) : p(n + 1), w(n + 1) { for(int i = 1; i <= n; i++) p[i] = i, w[i] = 0; }
    int find(int u) { if(p[u] == u) return u; int fa = find(p[u]); w[u] += w[p[u]]; return p[u] = fa; }
    int merge(int u, int v, ll val) {
        int root_u = find(u), root_v = find(v);
        if(root_u == root_v) return w[v] - w[u] == val;
        p[root_v] = root_u;
        w[root_v] = w[u] - w[v] + val;
        return true;
    }
};
```

merge 表示一条 a[u] - a[v] = val 的条件，返回值表示是否插入成功。

单调队列

```
struct Mqueue{
    struct node{
        int index;
        int val;
    };
    int k, now = 0, rev = false;
    deque<node> q;
public:
    Mqueue(int a, bool b = false): k(a), rev(b){}
    void push(int x)
    {
        now++;
        while(!q.empty()&&q.front().index+k<=now) q.pop_front();
        if(!rev) while(!q.empty()&&q.back().val<=x) q.pop_back();
        else while(!q.empty()&&q.back().val>=x) q.pop_back();
        q.push_back((node){now,x});
    }
    int getval(){return q.front().val;}
};
```

说明：构造函数中，第一个参数是窗口大小，第二个参数表示求最大值还是最小值（默认为 false : 最大值）

ST 表

```
template <typename T>
struct ST {
    ST(T a[], int n) {
        int t = __lg(n) + 1;
        for (int i = 1; i <= n; i++) maxv[0][i] = minv[0][i] = a[i];
        for (int j = 1; j < t; j++)
            for (int i = 1; i <= n - (1 << j) + 1; i++) {
                maxv[j][i] = max(maxv[j - 1][i], maxv[j - 1][i + (1 << (j - 1))]);
                minv[j][i] = min(minv[j - 1][i], minv[j - 1][i + (1 << (j - 1))]);
            }
```

```
    }
    T getmax(int l, int r) {
        int k = __lg(r - l + 1);
        return max(maxv[k][l], maxv[k][r - (1 << k) + 1]);
    }
    T getmin(int l, int r) {
        int k = __lg(r - l + 1);
        return min(minv[k][l], minv[k][r - (1 << k) + 1]);
    }
private:
    const int M = 1e6 + 5;
    T maxv[21][M], minv[21][M];
};
```

> **例子：对于一个大小为 n，1-index 的数组 a 建立 ST 表，然后求第 4 个元素至第 8 个元素的最小值**
>
> ```
> ST<int> st(a, n);
> cout << st.getmin(4, 8);
> ```

树状数组

```
const int M = 1e6 + 5;
struct bit{
    int lowbit(int x) {return x & (-x);}
    void add(int x, ll v) { for(int i = x; i < M; i += lowbit(i)) c[i] += v; }
    ll sum(int x) { ll ans = 0; for(int i = x; i; i -= lowbit(i)) ans += c[i];
return ans;}
    ll sum(int x, int y) {return sum(y) - sum(x - 1);}
    ll c[M];
};
```

这里是单点加区间查。对于区间加单点查的情况，维护前缀和数组 s，则一次区间加可转化为两次单点加操作。

线段树

```cpp
template < typename T, T (*op)(T, T), auto e,
           typename F, T (*mapping)(T, F, int, int),
           F (*composition)(F, F), auto e1 >
struct segtree
{
    int n;
    vector< T > v;
    vector< F > lazy;
    void build(T a[], int now, int l, int r) {
        int mid = (l + r) / 2;
        if(l == r) { v[now] = a[l]; return;}
        build(a, now * 2, l, mid); build(a, now * 2 + 1, mid + 1, r);
        pushup(now);
    }

    void pushup(int k){ v[k] = op(v[k * 2], v[k * 2 + 1]); }
    void pushdown(int k, int l, int r) {
        v[k] = mapping(v[k], lazy[k], l, r);
        if(l != r) {
            lazy[k * 2] = composition(lazy[k * 2], lazy[k]);
            lazy[k * 2 + 1] = composition(lazy[k * 2 + 1], lazy[k]);
        }
        lazy[k] = e1;
    }

    void update(int now, int ql, int qr, int l, int r, F x) {
        pushdown(now, l, r);
        if(l > qr || r < ql) return;
        if(l >= ql && r <= qr)
        {
            lazy[now] = x;
            pushdown(now, l, r);
            return;
        }
        update(now * 2, ql, qr, l, (l + r) / 2, x);
        update(now * 2 + 1, ql, qr, (l + r) / 2 + 1, r, x);
        pushup(now);
    }
    T query(int now, int ql, int qr, int l, int r) {
        pushdown(now, l, r);
        if(l > qr || r < ql) return e;
        if(l >= ql && r <= qr) return v[now];
        return op(query(now * 2, ql, qr, l, (l + r) / 2),
```

```
                query(now * 2 + 1, ql, qr, (l + r) / 2 + 1, r));
    }

    segtree(int n) : n(n), v(n << 2, e), lazy(n << 2, e1) {}
    segtree(T a[], int n) : n(n), v(n << 2, e), lazy(n << 2, e1) { build(a, 1,
1, n);}
    void update(int l, int r, F x) { update(1, l, r, 1, n, x); }
    T query(int l, int r) { return query(1, l, r, 1, n); }
};

template < typename T >
T MAX(T x, T y) { if(x > y) return x; else return y; }
template < typename T >
T PLUS(T x, T y) { return x + y; }

/*  Note:
    typename T, T op(T, T), T e(),
    typename F, T mapping(T, F, int, int),
    F composition(F, F), F e1()
*/

ll _e(){return 0;}
ll _map(ll x, ll lazy, int l, int r) { return x + (r - l + 1) * lazy; }
typedef segtree< ll, PLUS< ll >, _e, ll, _map, PLUS< ll >, _e > segtree_add; //
C++17
```

> 说明：T，op，e，F，mapping，composition，e1 分别表示线段树中维护的值的类型、加法运算、单位元、线段树中懒标记类型、当前值及懒标记向真实值的映射函数、懒标记的叠加函数，懒标记的单位元。
>
> 例子：使用 `segtree_add` ，完成区间加法和区间查询
>
> ```
> segtree_add s(n);
> s.update(1, 2, 3); // 将 [1, 2] 加上 3
> cout << s.query(1, 2);
> ```
>
> 例子：使用线段树，维护区间加、区间覆盖，查询和、最大值、最小值。
>
> 以下代码是 add + cover + 查询最大值的例子
>
> ```
> struct F
> {
>     ll add;
>     ll cover;
> ```

```
};

const ll inf = 1e18;
ll mapping(ll x, F t, int l, int r) {
    if (t.cover != inf)
        return t.cover;
    else
        return x + t.add;
};
F composition (F t1, F t2){
    if (t2.cover != inf) return { 0, t2.cover };
    else
    {
        if (t1.cover != inf) return { 0, t1.cover + t2.add };
        else return { t1.add + t2.add, inf };
    }
};

ll e() { return -inf; }
F e1() { return {0, inf}; }

segtree< ll, MAX<ll>, e, F, mapping, composition, e1 > s(a, n);

int op;
cin >> op;
if (op == 1) {
    int x, y, val;
    cin >> x >> y >> val;
    s.update(x, y, { 0, val }); // cover
}
if (op == 2) {
    int x, y, val;
    cin >> x >> y >> val;
    s.update(x, y, { val, inf }); // add
}
if (op == 3) {
    int x, y;
    cin >> x >> y;
    cout << s.query(x, y) << '\n';
}
```

查询和的时候，e() 是 0，然后将 op 改为 x + y，然后将 mapping 改为：cover = inf ? (r - l + 1) * cover :

查询最小值的时候，e() 是无穷大，然后需要改变 op 为 min(x, y)。

拓展：线段树二分

如果固定左端点，有单调增函数 $f(r)$ 表示区间 $[l, r]$ 之前的某些信息，那么要找第一个数值满足 $f(u) >= target$，可以采用线段树二分。

例如，CF1878E 给定数组 a，多次询问：给定 l 求最大的 r 使得 $a_l \& a_{l+1} \& \ldots a_r \geq K$。

虽然倍增很好做，但是这里可以尝试线段树二分。线段树维护区间 &，由于单调性可做。

请使用以下模板，其中 check 函数必须是单调的，一开始返回 0，到某个值开始返回 1。那么，程序返回第一个 1 的位置。如果总是返回 0，则答案是 n + 1。

```
int max_right(int now, int ql, T& t, int l, int r, function<bool(T)>
check) {
        pushdown(now, l, r);
        if(r < ql) return n + 1;
        if(ql <= l) {// 可以操作
            if(!check(op(t, v[now]))) {
                t = op(t, v[now]);
                return n + 1;
            }
            if(l == r) return l;
        }
        int pos = max_right(now * 2, ql, t, l, (l + r) / 2, check);
        if(pos != n + 1) return pos;
        return max_right(now * 2 + 1, ql, t, (l + r) / 2 + 1, r, check);
}

int max_right(int ql, function<bool(T)> check) { T t = e(); return
max_right(1, ql, t, 1, n, check); }
```

同理可写出 min_left，check 必须单调，从右向左先是不满足（0），最终满足（1）。返回首个满足的，若都不满足，返回 0。

```
int min_left(int now, int qr, T& t, int l, int r, function<bool(T)>
check) {
    pushdown(now, l, r);
    if(l > qr) return 0;
    if(r <= qr) {// 可以操作
        if(!check(op(t, v[now]))) {
            t = op(t, v[now]);
            return 0;
        }
        if(l == r) return l;
    }
    int pos = min_left(now * 2 + 1, qr, t, (l + r) / 2 + 1, r, check);
    if(pos != 0) return pos;
    return min_left(now * 2, qr, t, l, (l + r) / 2, check);
```

```
}
```

第五部分 数学

拓展欧几里得

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

例子：求 $ax + by = \gcd(a, b)$ 的一组整数解。

```
exgcd(a, b, x, y);
```

则 x, y 是一组整数解。

如果要求 $ax + by = c$ 的一组整数解，则必须满足 $\gcd(a, b)|c$，然后 x 和 y 需要乘以 $c/\gcd(a, b)$ 才是答案。

如果要求 x 是正整数，则 x 增加若干个 $b/\gcd(a, b)$，y 减少若干个 $a/\gcd(a, b)$ 依然是答案。

即：

$$x = x_0 + k \times b/\gcd(a, b)$$
$$y = y_0 - k \times a/\gcd(a, b)$$

组合数

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

```
const int M = 5005;
ll C[M][M];
void init_C(int n) {
    for(int i = 0; i <= n; i++) {
        C[i][0] = 1;
        for(int j = 1; j <= i; j++) {
            C[i][j] = (C[i - 1][j] + C[i - 1][j - 1]) % mod;
        }
    }
}
```

带上下界插板法【公式】

下界插板法：n 个元素分成 m 堆，每堆至少有 x 个。

> 相当于 $n - xm$ 个物品分成 m 堆。答案是
>
> $$\binom{n+m-mx-1}{m-1}$$

上界插板法：n 个元素分成 m 堆，每堆至多有 x 个。

> 答案是m
>
> $$\sum_{i=0}^{m}(-1)^i\binom{m}{i}\binom{n+m-i(x+1)-1}{m-1}$$

Lucas 定理【公式】

> $$\binom{n}{m} \equiv \binom{n \bmod p}{m \bmod p} \times \binom{n/p}{m/p} \pmod p$$

BSGS

在算法竞赛中，BSGS（baby-step giant-step，大步小步算法）常用于求解离散对数问题。可以在 $\Theta(\sqrt{m})$ 时间解决以下问题：

$$a^x \equiv b \pmod m$$

其中 $\gcd(a, m) = 1$。方程的解满足$0 \le x < m$

这样，令 $x = A\lceil\sqrt{m}\rceil - B$，其中 $0 \le A, B \le \lceil\sqrt{m}\rceil$。那么方程可写成：

$$a^{A\lceil\sqrt{m}\rceil - B} \equiv b \pmod m$$

即

$$a^{A\lceil\sqrt{m}\rceil} \equiv b \times a^B \pmod m$$

采用 meet in the middle 策略可以在 $\Theta(\sqrt{m})$ 解决，采用 map 则多一个 log。

多项式乘法

这里采用的是 `vector<long long>` 版本。

```
namespace convolution {
// ** --------- 重要常数 ------------ **

using ll = long long;
using poly = vector<ll>;
const int mod = 998'244'353, g = 3;
vector< int > r;

// ** --------- 重要常数 ------------ **

// ** --------- 辅助函数 ------------ **

void init(int siz)
{
    const int l = 32 - __builtin_clz(siz - 1);
    // 一般情况，取 l = 20 就行了
    r.resize(1 << l);
    r[0] = 0;
    for (int i = 1; i < (1 << l); i++)
        r[i] = (r[i >> 1] >> 1 | (i & 1) << (l - 1));
}

ll pow(ll a, ll b) {
    ll res = 1, cur = a;
```

```cpp
    for(; b; b >>= 1)
    {
        if(b & 1) res = (res * cur) % mod;
        cur = (cur * cur) % mod;
    }
    return res;
}

ll inv(ll a) {return pow(a, mod - 2);}

// ** --------- 辅助函数 ------------ **

// ** ------- 快速数论变换 ------------ **

void ntt(poly& p, int type)
{
    int limit = p.size();
    init(limit);
    for (int i = 0; i < limit; i++)
        if (i < r[i]) swap(p[i], p[r[i]]);

    ll Wn, w;
    for (int mid = 1; mid < limit; mid <<= 1) {
        Wn = pow(g, (mod - 1) / (mid << 1));
        if (type == -1) Wn = inv(Wn);
        int size = mid << 1;
        for (int i = 0; i < limit; i += size) {
            w = 1;
            int j = i + mid;
            for (int k = i; k < j; k++, w = w * Wn % mod) {
                ll x = p[k], y = w * p[k + mid] % mod;
                p[k] = (x + y) % mod;
                p[k + mid] = (x - y + mod) % mod;
            }
        }
    }
}

// ** ------- 快速数论变换 ------------ **

// ** --------- 乘法 ------------- **

poly operator*(poly lhs, poly rhs)
{
    int siz = lhs.size() + rhs.size();
    int limit = 1 << (32 - __builtin_clz(siz - 1));
```

```cpp
    lhs.resize(limit);
    rhs.resize(limit);

    ntt(lhs, 1);
    ntt(rhs, 1);

    for (int i = 0; i < limit; i++)
        lhs[i] = lhs[i] * rhs[i] % mod;

    ntt(lhs, -1);
    ll t = inv(limit);

    for (int i = 0; i < limit; i++)
        lhs[i] = lhs[i] * t % mod;
    lhs.resize(siz - 1);
    return lhs;
}

// ** --------- 乘法 ------------- **

int main() {

    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    using namespace convolution;

    /* P3803 【模板】多项式乘法（FFT） 589ms (n, m = 1e6)
        int n, m; cin >> n >> m;
        poly a(n + 1), b(m + 1);
        for(int i = 0; i <= n; i++) cin >> a[i];
        for(int i = 0; i <= m; i++) cin >> b[i];
        a = a * b;
        for(int i = 0; i < a.size(); i++) cout << a[i] << " ";
     */

    return 0;

}
```

第六部分 计算几何

来自 github

二维几何：点与向量

```
#define y1 yy1
#define nxt(i) ((i + 1) % s.size())
typedef double LD;
const LD PI = acos(-1);
const LD eps = 1E-10;
int sgn(LD x) { return fabs(x) < eps ? 0 : (x > 0 ? 1 : -1); }
struct L;
struct P;
typedef P V;
struct P {
    LD x, y;
    explicit P(LD x = 0, LD y = 0): x(x), y(y) {}
    explicit P(const L& l);
};
struct L {
    P s, t;
    L() {}
    L(P s, P t): s(s), t(t) {}
};

P operator + (const P& a, const P& b) { return P(a.x + b.x, a.y + b.y); }
P operator - (const P& a, const P& b) { return P(a.x - b.x, a.y - b.y); }
P operator * (const P& a, LD k) { return P(a.x * k, a.y * k); }
P operator / (const P& a, LD k) { return P(a.x / k, a.y / k); }
inline bool operator < (const P& a, const P& b) {
    return sgn(a.x - b.x) < 0 || (sgn(a.x - b.x) == 0 && sgn(a.y - b.y) < 0);
}
bool operator == (const P& a, const P& b) { return !sgn(a.x - b.x) && !sgn(a.y - b.y); }
P::P(const L& l) { *this = l.t - l.s; }
ostream &operator << (ostream &os, const P &p) {
    return (os << "(" << p.x << "," << p.y << ")");
}
istream &operator >> (istream &is, P &p) {
    return (is >> p.x >> p.y);
}

LD dist(const P& p) { return sqrt(p.x * p.x + p.y * p.y); }
LD dot(const V& a, const V& b) { return a.x * b.x + a.y * b.y; }
```

```
LD det(const V& a, const V& b) { return a.x * b.y - a.y * b.x; }
LD cross(const P& s, const P& t, const P& o = P()) { return det(s - o, t - o);
}
// --------------------------------------------
```

象限

```
// 象限
int quad(P p) {
    int x = sgn(p.x), y = sgn(p.y);
    if (x > 0 && y >= 0) return 1;
    if (x <= 0 && y > 0) return 2;
    if (x < 0 && y <= 0) return 3;
    if (x >= 0 && y < 0) return 4;
    assert(0);
}

// 仅适用于参照点在所有点一侧的情况
struct cmp_angle {
    P p;
    bool operator () (const P& a, const P& b) {
//        int qa = quad(a - p), qb = quad(b - p);
//        if (qa != qb) return qa < qb;
        int d = sgn(cross(a, b, p));
        if (d) return d > 0;
        return dist(a - p) < dist(b - p);
    }
};
```

线

```
// 是否平行
bool parallel(const L& a, const L& b) {
    return !sgn(det(P(a), P(b)));
}
// 直线是否相等
bool l_eq(const L& a, const L& b) {
    return parallel(a, b) && parallel(L(a.s, b.t), L(b.s, a.t));
}
// 逆时针旋转 r 弧度
P rotation(const P& p, const LD& r) { return P(p.x * cos(r) - p.y * sin(r), p.x
* sin(r) + p.y * cos(r)); }
P RotateCCW90(const P& p) { return P(-p.y, p.x); }
```

```
P RotateCW90(const P& p) { return P(p.y, -p.x); }
// 单位法向量
V normal(const V& v) { return V(-v.y, v.x) / dist(v); }
```

点与线

```
// 点在线段上  <= 0包含端点 < 0 则不包含
bool p_on_seg(const P& p, const L& seg) {
    P a = seg.s, b = seg.t;
    return !sgn(det(p - a, b - a)) && sgn(dot(p - a, p - b)) <= 0;
}
// 点到直线距离
LD dist_to_line(const P& p, const L& l) {
    return fabs(cross(l.s, l.t, p)) / dist(l);
}
// 点到线段距离
LD dist_to_seg(const P& p, const L& l) {
    if (l.s == l.t) return dist(p - l);
    V vs = p - l.s, vt = p - l.t;
    if (sgn(dot(l, vs)) < 0) return dist(vs);
    else if (sgn(dot(l, vt)) > 0) return dist(vt);
    else return dist_to_line(p, l);
}
```

线与线

```
// 求直线交 需要事先保证有界
P l_intersection(const L& a, const L& b) {
    LD s1 = det(P(a), b.s - a.s), s2 = det(P(a), b.t - a.s);
    return (b.s * s2 - b.t * s1) / (s2 - s1);
}
// 向量夹角的弧度
LD angle(const V& a, const V& b) {
    LD r = asin(fabs(det(a, b)) / dist(a) / dist(b));
    if (sgn(dot(a, b)) < 0) r = PI - r;
    return r;
}
// 线段和直线是否有交   1 = 规范，2 = 不规范
int s_l_cross(const L& seg, const L& line) {
    int d1 = sgn(cross(line.s, line.t, seg.s));
    int d2 = sgn(cross(line.s, line.t, seg.t));
    if ((d1 ^ d2) == -2) return 1; // proper
    if (d1 == 0 || d2 == 0) return 2;
    return 0;
```

```
}
// 线段的交  1 = 规范，2 = 不规范
int s_cross(const L& a, const L& b, P& p) {
    int d1 = sgn(cross(a.t, b.s, a.s)), d2 = sgn(cross(a.t, b.t, a.s));
    int d3 = sgn(cross(b.t, a.s, b.s)), d4 = sgn(cross(b.t, a.t, b.s));
    if ((d1 ^ d2) == -2 && (d3 ^ d4) == -2) { p = l_intersection(a, b); return 1; }
    if (!d1 && p_on_seg(b.s, a)) { p = b.s; return 2; }
    if (!d2 && p_on_seg(b.t, a)) { p = b.t; return 2; }
    if (!d3 && p_on_seg(a.s, b)) { p = a.s; return 2; }
    if (!d4 && p_on_seg(a.t, b)) { p = a.t; return 2; }
    return 0;
}
```

多边形

面积、凸包

```
typedef vector<P> S;

// 点是否在多边形中 0 = 在外部 1 = 在内部 -1 = 在边界上
int inside(const S& s, const P& p) {
    int cnt = 0;
    FOR (i, 0, s.size()) {
        P a = s[i], b = s[nxt(i)];
        if (p_on_seg(p, L(a, b))) return -1;
        if (sgn(a.y - b.y) <= 0) swap(a, b);
        if (sgn(p.y - a.y) > 0) continue;
        if (sgn(p.y - b.y) <= 0) continue;
        cnt += sgn(cross(b, a, p)) > 0;
    }
    return bool(cnt & 1);
}
// 多边形面积，有向面积可能为负
LD polygon_area(const S& s) {
    LD ret = 0;
    FOR (i, 1, (LL)s.size() - 1)
        ret += cross(s[i], s[i + 1], s[0]);
    return ret / 2;
}
// 构建凸包 点不可以重复 < 0 边上可以有点， <= 0 则不能
// 会改变输入点的顺序
const int MAX_N = 1000;
S convex_hull(S& s) {
//    assert(s.size() >= 3);
```

```
    sort(s.begin(), s.end());
    S ret(MAX_N * 2);
    int sz = 0;
    FOR (i, 0, s.size()) {
        while (sz > 1 && sgn(cross(ret[sz - 1], s[i], ret[sz - 2])) < 0) --sz;
        ret[sz++] = s[i];
    }
    int k = sz;
    FORD (i, (LL)s.size() - 2, -1) {
        while (sz > k && sgn(cross(ret[sz - 1], s[i], ret[sz - 2])) < 0) --sz;
        ret[sz++] = s[i];
    }
    ret.resize(sz - (s.size() > 1));
    return ret;
}

P ComputeCentroid(const vector<P> &p) {
    P c(0, 0);
    LD scale = 6.0 * polygon_area(p);
    for (unsigned i = 0; i < p.size(); i++) {
        unsigned j = (i + 1) % p.size();
        c = c + (p[i] + p[j]) * (p[i].x * p[j].y - p[j].x * p[i].y);
    }
    return c / scale;
}
```

圆

```
struct C {
    P p; LD r;
    C(LD x = 0, LD y = 0, LD r = 0): p(x, y), r(r) {}
    C(P p, LD r): p(p), r(r) {}
};
```

三点求圆心

```
P compute_circle_center(P a, P b, P c) {
    b = (a + b) / 2;
    c = (a + c) / 2;
    return l_intersection({b, b + RotateCW90(a - b)}, {c , c + RotateCW90(a -
c)});
}
```

圆线交点、圆圆交点

- 圆和线的交点关于圆心是顺时针的

```
vector<P> c_l_intersection(const L& l, const C& c) {
    vector<P> ret;
    P b(l), a = l.s - c.p;
    LD x = dot(b, b), y = dot(a, b), z = dot(a, a) - c.r * c.r;
    LD D = y * y - x * z;
    if (sgn(D) < 0) return ret;
    ret.push_back(c.p + a + b * (-y + sqrt(D + eps)) / x);
    if (sgn(D) > 0) ret.push_back(c.p + a + b * (-y - sqrt(D)) / x);
    return ret;
}

vector<P> c_c_intersection(C a, C b) {
    vector<P> ret;
    LD d = dist(a.p - b.p);
    if (sgn(d) == 0 || sgn(d - (a.r + b.r)) > 0 || sgn(d + min(a.r, b.r) - max(a.r, b.r)) < 0)
        return ret;
    LD x = (d * d - b.r * b.r + a.r * a.r) / (2 * d);
    LD y = sqrt(a.r * a.r - x * x);
    P v = (b.p - a.p) / d;
    ret.push_back(a.p + v * x + RotateCCW90(v) * y);
    if (sgn(y) > 0) ret.push_back(a.p + v * x - RotateCCW90(v) * y);
    return ret;
}
```

圆圆位置关系

```
// 1:内含 2:内切 3:相交 4:外切 5:相离
int c_c_relation(const C& a, const C& v) {
    LD d = dist(a.p - v.p);
    if (sgn(d - a.r - v.r) > 0) return 5;
    if (sgn(d - a.r - v.r) == 0) return 4;
    LD l = fabs(a.r - v.r);
    if (sgn(d - l) > 0) return 3;
    if (sgn(d - l) == 0) return 2;
    if (sgn(d - l) < 0) return 1;
}
```

第七部分 杂项

红黑树

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
typedef
tree<int,null_type,less<int>,rb_tree_tag,tree_order_statistics_node_update>
rbtree;
```

> 比起 std::set，它更支持排名的查询。
>
> ### 用法
>
> ```
> T.insert(x) 插入
> T.erase(x) 删除
> T.order_of_key(x) 求排名(比它小的元素个数)
> T.find_by_order(k) 找排名为 k 的元素的迭代器
> T.lower_bound / upper_bound(x) 找大于(等于) x 的迭代器
> ```
>
> ### 例子：红黑树的插入和查询
>
> ```
> rbtree rbt;
> rbt.insert(1);
> rbt.insert(2);
> rbt.insert(3);
> cout << rbt.order_of_key(2) << endl; // 查询比 2 小的元素个数，即 1 个
> cout << *rbt.find_by_order(2) << endl; // 查询排名为 2 的元素，即 3。
> ```

树上背包上下界优化

```
for(auto nxt : v[x]) {
    for(int j = sz[x]; j >= 0; j--){
      for(int h = sz[nxt]; h >= 0; h--){
        dp[x][j + h] = min(dp[x][j + h], dp[x][j] ? ? dp[nxt][h]);
        // 由 dp[x][j], dp[nxt][h] -> dp[x][j + h]
        // sz 表示子树大小，初始请设置 sz[x] = 1
      }
    }
    sz[x] += sz[nxt];
}
```