scutsky's template

scutsky

目录

1 数据结构

1.1 ST 表

1.1.1 一维 ST 表

```cpp
int st[N][M], w[N], n; // 使用 ST 表时一定要记得调用 build
void build()
{
    for (int i = 0; i <= __lg(n); i++)
        for (int l = 1; l + (1 << i) - 1 <= n; l++) {
            if (!i) st[l][i] = w[l];
            else st[l][i] = max(st[l][i - 1], st[l + (1 << (i - 1))][i - 1]);
        }
}
int query(int l, int r)
{
    int k = __lg(r - l + 1);
    return max(st[l][k], st[r - (1 << k) + 1][k]);
}
array<int, 2> st[N][M]; // 记录转移位置
void build()
{
    for (int i = 0; i <= __lg(n); i++)
        for (int l = 1; l + (1 << i) - 1 <= n; l++) {
            if (!i) st[l][i] = {w[l], l};
            else st[l][i] = max(st[l][i - 1], st[l + (1 << (i - 1))][i - 1]);
        }
}
array<int, 2> query(int l, int r)
{
    int k = __lg(r - l + 1);
    return max(st[l][k], st[r - (1 << k) + 1][k]);
}
```

1.1.2 二维 ST 表

```cpp
const int N = 1005, M = 11; // M 应该取 max(__lg(N), __lg(N)) + 1
int n, m, w[N][N], st[M][M][N][N];
void build()
{
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            cin >> st[0][0][i][j]; //(st[0][0][i][j] == w[i][j])
    for (int k = 1; k <= __lg(n); k++)
        for (int i = 1; i <= n - (1 << k) + 1; i++)
            for (int j = 1; j <= m; j++)
                st[k][0][i][j] = max(st[k - 1][0][i][j], st[k - 1][0][i + (1 << (k - 1))][j]);
    for (int k = 1; k <= __lg(m); k++)
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= m - (1 << k) + 1; j++)
                st[0][k][i][j] = max(st[0][k - 1][i][j], st[0][k - 1][i][j + (1 << (k - 1))]);
    for (int k1 = 1; k1 <= __lg(n); k1++)
        for (int k2 = 1; k2 <= __lg(m); k2++)
            for (int i = 1; i <= n - (1 << k1) + 1; i++)
                for (int j = 1; j <= m - (1 << k2) + 1; j++)
                    st[k1][k2][i][j] = max(st[k1][k2 - 1][i][j], st[k1][k2 - 1][i][j + (1 <<
                    ↪ (k2 - 1))]);
}
int query(int x1, int y1, int x2, int y2)
{
    int k1 = __lg(x2 - x1 + 1), k2 = __lg(y2 - y1 + 1);
    return max({st[k1][k2][x1][y1], st[k1][k2][x2 - (1 << k1) + 1][y1],
        st[k1][k2][x1][y2 - (1 << k2) + 1], st[k1][k2][x2 - (1 << k1) + 1][y2 - (1 << k2) +
        ↪ 1]});
}
```

1.1.3 状压 RMQ

```cpp
template<typename T>
struct FastRMQ
{
    static constexpr int BLOCK = 64;
    int n, numB, L;
    vector<T> data; // 1...n, 有 data[0] 未用
    vector<u64> mask; // 块内位掩码
    vector<T> pminv, sminv, st; // 扁平化 ST 表, 大小 = L * numB
    // 传入 a[1..n], a[0] 忽略
    void build(const vector<T>& a)
    {
        n = int(a.size()) - 1;
        data = a;
        numB = (n + BLOCK - 1) / BLOCK;
        L = __lg(numB) + 1;
        mask.assign(n + 1, 0ULL);
        pminv.assign(n + 2, numeric_limits<T>::max());
        sminv.assign(n + 2, numeric_limits<T>::max());
        st.assign(L * numB, numeric_limits<T>::max());
        vector<int> stk;
        stk.reserve(BLOCK);
        // 块内预处理: 单调栈掩码 + 前后缀最小
        for (int b0 = 0; b0 < numB; ++b0)
        {
            int l = b0 * BLOCK + 1, r = min(n, l + BLOCK - 1);
            u64 m = 0;
            stk.clear();
            // 单调栈 + 位翻转
            for (int i = l; i <= r; ++i)
            {
                while (!stk.empty() && data[i] < data[stk.back()])
                    m ^= 1ULL << ((stk.back() - 1) & 63), stk.pop_back();
                m |= 1ULL << ((i - 1) & 63);
                stk.push_back(i);
                mask[i] = m;
            }
            // 前缀最小 & 储存整块最小
            T cur = data[l];
            pminv[l] = cur;
            for (int i = l + 1; i <= r; ++i)
            {
                if (data[i] < cur) cur = data[i];
                pminv[i] = cur;
            }
            st[b0] = cur; // 整块最小
            // 后缀最小
            cur = data[r];
            sminv[r] = cur;
            for (int i = r - 1; i >= l; --i)
            {
                if (data[i] < cur) cur = data[i];
                sminv[i] = cur;
            }
        }
        // 构建扁平化稀疏表
        for (int k = 1; k < L; ++k)
        {
            int src = (k - 1) * numB, dest = k * numB, half = 1 << (k - 1);
            for (int b0 = 0; b0 + (1 << k) <= numB; ++b0)
            {
                T x = st[src + b0], y = st[src + b0 + half];
                st[dest + b0] = x < y ? x : y;
            }
        }
    }
    // 查询 1 - indexed 区间 [l..r] 最小值
    inline T query(int l, int r) cons
    {
        if (l < 1 || r > n || l > r) return numeric_limits<T>::max();
        int b0 = (l - 1) / BLOCK, b1 = (r - 1) / BLOCK;
        // 同块: 掩码 + ctz
        if (b0 == b1)
        {
            u64 m = mask[r] & (~0ULL << ((l - 1) & 63));
```

```
            int off = __builtin_ctzll(m);
            return data[b0 * BLOCK + 1 + off];
        }
        // 不同块：两端后缀 / 前缀
        T ans = sminv[l] < pminv[r] ? sminv[l] : pminv[r];
        // 中间整块用稀疏表
        int cnt = b1 - b0 - 1;
        if (cnt > 0)
        {
            int k = __lg(cnt);
            T x = st[k * numB + (b0 + 1)];
            T y = st[k * numB + (b1 - (1 << k))];
            T mid = x < y ? x : y;
            if (mid < ans) ans = mid;
        }
        return ans;
    }
};
```

1.2 单调栈

求数组里每个数左边 / 右边第一个大于 / 小于它的数

例子：求每个数后面第一个大于它的下标

```
vector<int> stk;
for (int i = n; i >= 1; i--)
{
    while (stk.size() && w[stk.back()] <= w[i]) stk.pop_back(); // 栈非空且当前值优于栈顶，那么弹出栈顶
    if (stk.size()) f[i] = stk.back(); // 当前位置的答案就是栈顶
    stk.push_back(i); // 压入栈中
}
```

1.3 单调队列

1.3.1 一维单调队列

求滑动窗口中的最大值/最小值，其中 k 为滑动窗口的大小。

```
int n, k, q[N], w[N], hh = 0, tt = -1;
vector<int> minv, maxv; // 滑动窗口的最小值和最大值
for (int i = 1; i <= n; i++)
{
    if (hh <= tt && i - k >= q[hh]) hh++; // 如果窗口长度超过 k，那么弹出
    while (hh <= tt && w[q[tt]] >= w[i]) tt--;
    q[++tt] = i;
    if (i >= k) minv.push_back(w[q[hh]]);
}
for (int i = 1; i <= n; i++)
{
    if (hh <= tt && i - k >= q[hh]) hh++;
    while (hh <= tt && w[q[tt]] <= w[i]) tt--;
    q[++tt] = i;
    if (i >= k) maxv.push_back(w[q[hh]]);
}
```

1.3.2 二维单调队列

```
int h[N][N], w[N][N], n, m, q[N], hh, tt, ans;
for (int i = 1; i <= n; i++)
{
    hh = 0, tt = -1;
    for (int j = 1; j <= m; j++)
    {
        if (hh <= tt && j - b >= q[hh]) hh++;
        while (hh <= tt && h[i][q[tt]] >= h[i][j]) tt--;
        q[++tt] = j;
        if (j >= b) w[i][j] = h[i][q[hh]];
    }
```

```
}
for (int j = b; j <= m; j++)
{
    hh = 0, tt = -1;
    for (int i = 1; i <= n; i++)
    {
        if (hh <= tt && i - a >= q[hh]) hh++;
        while (hh <= tt && w[q[tt]][j] >= w[i][j]) tt--;
        q[++tt] = i;
        if (i >= a) ans += w[q[hh]][j];
    }
}
```

1.4 可删堆

可删堆使用对顶堆的思想实现删除操作。当不需要查询前驱后继，只需要查询最值并支持删除时，可删堆的效率比 multiset 更高。

```
struct Heap
{
    priority_queue<int> qPush, qErase; // Heap = qPush - qErase
    void push(int x) { qPush.push(x); }
    void extract(int x) { qErase.push(x); }
    int top()
    {
        while (!qErase.empty() && !qPush.empty() && qPush.top() == qErase.top()) qPush.pop(),
        ↪  qErase.pop();
        return qPush.empty() ? -1 : qPush.top();
    }
    void pop()
    {
        while (!qErase.empty() && !qPush.empty() && qPush.top() == qErase.top()) qPush.pop(),
        ↪  qErase.pop();
        if (!qPush.empty()) qPush.pop();
    }
    int size() { return qPush.size() - qErase.size(); }
} heap;
```

1.5 并查集

1.5.1 扩展域并查集

扩展域并查集用于处理敌友关系等二元对立问题。要注意像 2-SAT 一样把该连的边都连上（即要把另一组反向边也连上），同时注意数组需要开大一点（通常是原来的 3 倍）。

```
void solve()
{
    map<string, int> mp;
    for (int i = 1; i <= n; i++)
    {
        string s;
        cin >> s; // 读取字符串
        mp[s] = i, p[i] = i, p[i + n] = i + n;
    }
    while (m--)
    {
        int t, a, b;
        string s1, s2;
        cin >> t >> s1 >> s2; // 读取操作类型和两个字符串
        a = mp[s1], b = mp[s2];
        int pa = find(a), pb = find(b);
        if (pa == pb)
        {
            if (t == 1) cout << "YES\n"; // 相同
            else cout << "NO\n";
        }
        else if (pa == find(b + n))
        {
            if (t == 2) cout << "YES\n"; // 相反
            else cout << "NO\n";
        }
        else
        {
```

```
            cout << "YES\n";
            if (t == 1) p[pa] = pb, p[find(a + n)] = p[find(b + n)];
            else p[pa] = find(b + n), p[find(a + n)] = pb;
        }
    }
    while (q--)
    {
        string s1, s2;
        int a = mp[s1], b = mp[s2];
        int pa = find(a), pb = find(b);
        if (pa == pb) cout << "1\n"; // 相同
        else if (pa == find(b + n)) cout << "2\n"; // 相反
        else cout << "3\n"; // 无关
    }
}
```

例子：食物链（维护的是每个变量属于哪个集合之间的制约关系）

```
int p[N], n, k, res;
void solve()
{
    for (int i = 1; i <= n; i++) p[i] = i, p[i + n] = i + n, p[i + 2 * n] = i + 2 * n;
    while (k--)
    {
        int t, x, y;
        cin >> t >> x >> y;
        if (x > n || y > n || (x == y && t == 2)) res++;
        else if (t == 1) // x 和 y 是同类
        {
            if (find(x) == find(y + n) || find(x + n) == find(y)) res++; // 有吃的关系就是假
            else if (find(x) != find(y))
            {
                p[find(x)] = find(y); // 如果 x 是 A 那么 y 也是 A
                p[find(x + n)] = find(y + n); // 如果 x 是 B 那么 y 也是 B
                p[find(x + 2 * n)] = find(y + 2 * n); // 如果 x 是 C 那么 y 也是 C
            }
        }
        else
        {
            if (find(x) == find(y) || find(x + n) == find(y)) res++; // 有同类或 y 吃 x 就是假
            else if (find(x) != find(y + n))
            {
                p[find(x)] = find(y + n); // 如果 x 是 A 那么 y 是 B
                p[find(x + n)] = find(y + 2 * n); // 如果 x 是 B 那么 y 是 C
                p[find(x + 2 * n)] = find(y); // 如果 x 是 C 那么 y 是 A
            }
        }
    }
}
```

1.5.2 带权并查集

```
int p[N], d[N], sz[N], T, x, y;
for (int i = 1; i < N; i++) p[i] = i, sz[i] = 1;
while (T--)
{
    int px = find(x), py = find(y);
    if (op == 'M') // 将 x 这个集合整体合并到 y(移动到 y 的后面)
    {
        d[px] = sz[py]; // 到根的距离
        sz[py] += sz[px]; // 一定是操作根 不要乱操作 find 之类的
        p[px] = py;
    }
    else if (px != py) cout << "-1\n";
    else cout << max(0, abs(d[x] - d[y]) - 1) << "\n";
}
```

例子：给出一些 $[l,r]$ 之间的异或和或者询问区间 $[l,r]$ 异或和（强制在线）

```cpp
map<int, int> p, d;
int find(int x)
{
    if (p.find(x) == p.end()) return p[x] = x;
    if (p[x] != x)
    {
        int t = find(p[x]);
        d[x] ^= d[p[x]], p[x] = t;
    }
    return p[x];
}
void solve()
{
    int q, ans = 0, t, l, r;
    while (q--)
    {
        if (ans == -1) ans = 1;
        l ^= ans, r ^= ans;
        if (l > r) swap(l, r);
        l--;
        if (t == 1)
        {
            int x;
            x ^= ans;
            if (find(l) == find(r)) continue;
            int pl = find(l), pr = find(r);
            d[pl] = d[r] ^ x ^ d[l], p[pl] = pr;
            // d[find(l)] = d[r] ^ x ^ d[l]; 不能写成这样
            // p[find(l)] = find(r); 一定是要操作根 不然会出错
        }
        else
        {
            if (find(l) != find(r)) ans = -1;
            else ans = d[l] ^ d[r];
            cout << ans << "\n";
        }
    }
}
```

1.6 哈希

为了卡常，可以尝试手写哈希表。

```cpp
template<class T, int Mod>
struct HashTable
{
    using u64 = unsigned long long;
    int hd[Mod], nt[Mod << 1], tot = 0;
    u64 to[Mod << 1];
    T val[Mod << 1];
    void clear() { for (int i = 1; i <= tot; i++) hd[to[i] % Mod] = 0; tot = 0; }
    T operator()(u64 x)
    {
        int u = x % Mod;
        for (int i = hd[u]; i; i = nt[i])
            if (to[i] == x) return val[i];
        return T();
    }
    T& operator[](u64 x)
    {
        int u = x % Mod;
        for (int i = hd[u]; i; i = nt[i])
            if (to[i] == x) return val[i];
        to[++tot] = x, nt[tot] = hd[u], hd[u] = tot;
        return val[tot] = T();
    }
    T find(u64 x)
    {
        int u = x % Mod;
        for (int i = hd[u]; i; i = nt[i])
            if (to[i] == x) return 1;
        return - 1;
    }
```

```
};
HashTable<int, 1090189 > mp; // 大小至少要开和存的东西一个量级
```

1.7 树状数组

1.7.1 基本树状数组 (单点修改 + 区间查询)

```
int tr[N], n;
void add(int x, int val) // 单点修改: a[x] += val
{
    for (int i = x; i <= n; i += i & -i) tr[i] += val;
}
int query(int x) // 查询前缀和: sum(a[1..x])
{
    int res = 0;
    for (int i = x; i > 0; i -= i & -i) res += tr[i];
    return res;
}
int query(int l, int r) // 查询区间和: sum(a[l..r])
{
    return query(r) - query(l - 1);
}
```

1.7.2 树状数组上二分 (倍增)

查询第一个前缀和小于 s 的位置

```
int query_kth(int s) // 查询第一个前缀和小于等于 s 的位置
{
    int pos = 0;
    for (int i = 16; i >= 0; i--) {
        if (pos + (1 << i) <= n && tr[pos + (1 << i)] <= s) {
            pos += (1 << i);
            s -= tr[pos];
        }
    }
    return pos;
}
```

1.7.3 树状数组 (区间修改 + 区间查询)

```
int tr1[N], tr2[N], n;
void add(int *tr, int x, int val) // 辅助函数
{
    for (int i = x; i <= n; i += i & -i) tr[i] += val;
}
int query(int *tr, int x) // 辅助函数
{
    int res = 0;
    for (int i = x; i > 0; i -= i & -i) res += tr[i];
    return res;
}
void add(int l, int r, int val) // 区间修改: [l, r] += val
{
    add(tr1, l, val);
    add(tr2, l, val * l);
    add(tr1, r + 1, -val);
    add(tr2, r + 1, -(r + 1) * val);
}
int sum(int x) // 前缀和
{
    return (x + 1) * query(tr1, x) - query(tr2, x);
}
int query(int l, int r) // 区间查询
{
    return sum(r) - sum(l - 1);
}
```

1.7.4 二维树状数组 (单点修改 + 区间查询)

```cpp
int tr[N][N], n, m;
void add(int x, int y, int val) // 单点修改 (x, y) 权值 += val
{
    for (int i = x; i <= n; i += i & -i)
        for (int j = y; j <= m; j += j & -j) tr[i][j] += val;
}
int query(int x, int y) // 查询以 (x, y) 为右下角的矩形的权值和
{
    int res = 0;
    for (int i = x; i > 0; i -= i & -i)
        for (int j = y; j > 0; j -= j & -j) res += tr[i][j];
    return res;
}
int query(int a, int b, int c, int d) // 查询矩形 (a,b) 到 (c,d) 的权值和
{
    return query(c, d) - query(a - 1, d) - query(c, b - 1) + query(a - 1, b - 1);
}
```

1.7.5 二维树状数组 (区间查询 + 区间修改)

```cpp
void add(int x, int y, int k)
{
    for (int i = x; i <= n; i += i & -i)
        for (int j = y; j <= m; j += j & -j)
            tr1[i][j] += k, tr2[i][j] += k * (x - 1),
            tr3[i][j] += k * (y - 1), tr4[i][j] += k * (x - 1) * (y - 1);
}
int query(int x, int y)
{
    int ans1 = 0, ans2 = 0, ans3 = 0, ans4 = 0;
    for (int i = x; i; i -= i & -i)
        for (int j = y; j; j -= j & -j)
            ans1 += tr1[i][j], ans2 += tr2[i][j], ans3 += tr3[i][j], ans4 += tr4[i][j];
    ans1 *= x * y;
    return ans1 - ans2 * y - ans3 * x + ans4;
} // 将 (a, b), (c, d) 为顶点的矩形区域内的所有数字加上 k
void add(int a, int b, int c, int d, int k) { add(a, b, k), add(c + 1, b, -k), add(a, d + 1,
↪  -k), add(c + 1, d + 1, k); }
int query(int a, int b, int c, int d) { return query(c, d) - query(c, b - 1) - query(a - 1, d)
↪  + query(a - 1, b - 1); } // 求 (a, b), (c, d) 为顶点的矩形区域内的所有数字的和

```

1.7.6 离线二维差分 + 二维查询

利用二维差分 + 二维偏序解决

如果是一边差分一边查询，可以利用增加一维时间轴，利用三维偏序解决

首先利用容斥，将待查询矩形转化为四个矩形的查询。对于一个左上角 $(1,1)$，右下角为 (x,y) 的矩形，记作矩形的面积为 $sum_{x,y}$ 二维差分就是对某个点右下角的区域全部加上某个值，所以对于矩形 $\{x_1,y_1,x_2,y_2,val\}$ 的矩形，可以拆分成四个差分 $\{x_1,y_1,val\}$, $\{x_1,y_2+1,-val\}$,$\{x_2+1,y_1,-val\}$,$\{x_2+1,y_2+1,val\}$

考虑一下差分数组和 sum 的关系：

对于一个位置 (i,j)，$d_{i,j}$ 表示差分数组的值，$p_{i,j}$ 表示点权，则 $p_{i,j}=\sum_{s=1}^{i}\sum_{t=1}^{j}d_{s,t}$，$\mathrm{sum}_{x,y}=\sum_{i=1}^{x}\sum_{j=1}^{y}p_{i,j}$，画一个图可知，$d_{i,j}$ 对 $\mathrm{sum}_{x,y}$ 的贡献为 $(x-i+1)\times(y-j+1)\times d_{i,j}$

即 $\mathrm{sum}_{x,y}=\sum_{i=1}^{x}\sum_{j=1}^{y}(x-i+1)\times(y-j+1)\times d_{i,j}$

可以将式子拆为如下四部分，用四个树状数组维护即可：

$(x+1)\times(y+1)\times\sum_{i=1}^{x}\sum_{j=1}^{y}d_{i,j}$

$-(x+1)\times\sum_{i=1}^{x}\sum_{j=1}^{y}j\times d_{i,j}$ $-(y+1)\times\sum_{i=1}^{x}\sum_{j=1}^{y}i\times d_{i,j}$

$\sum_{i=1}^{x}\sum_{j=1}^{y}i\times j\times d_{i,j}$

若 $d_{i,j}$ 对 $sum_{x,y}$ 产生贡献，则必有 $i\leqslant x\wedge j\leqslant y$，所以利用二维偏序处理即可

例题：询问区间 $[l,r]$ 中所有子区间的最小值的和。

考虑每一个值作为最小值的贡献区间 (用单调栈可以轻松求得)，则问题变为一个二维加，二维查询的问题。

```cpp
int n, q, w[N], l[N], r[N], tr[4][N], ans[N];
void addp(int x, int y, int k) { add(tr[0], y, k), add(tr[1], y, y * k),
add(tr[2], y, x * k), add(tr[3], y, x * y * k); }
int query(int x, int y)
{
```

```
    return (x + 1) * (y + 1) * query(tr[0], y) - (x + 1) * query(tr[1], y)-
        (y + 1) * query(tr[2], y) + query(tr[3], y);
}
void solve()
{
    w[0] = w[n + 1] = -INF;
    vector<int> stk;
    stk.push_back(0);
    for (int i = 1; i <= n; i++)
    {
        while (w[stk.back()] > w[i]) stk.pop_back();
        l[i] = stk.back() + 1, stk.push_back(i);
    }
    stk.clear(), stk.push_back(n + 1);
    for (int i = n; i >= 1; i--)
    {
        while (w[stk.back()] >= w[i]) stk.pop_back();
        r[i] = stk.back() - 1, stk.push_back(i);
    }
    vector<array<int, 4>> v;
    for (int i = 1; i <= n; i++)
    {
        v.push_back({i, 0, l[i], w[i]}), v.push_back({i, 0, i + 1, -w[i]});
        v.push_back({r[i] + 1, 0, l[i], -w[i]}), v.push_back({r[i] + 1, 0, i + 1, w[i]});
    }
    for (int i = 1; i <= q; i++)
    {
        int l, r;
        v.push_back({r, i, r, 1}), v.push_back({r, i, l - 1, -1});
        v.push_back({l - 1, i, r, -1}), v.push_back({l - 1, i, l - 1, 1});
    }
    sort(v.begin(), v.end());
    for (auto [x, ty, y, val]:v)
        if (!ty) addp(x, y, val);
        else ans[ty] += val * query(x, y);
}
```

1.8 线段树

1.8.1 线段树上二分

用途：在区间上找满足条件的最右/左端点，如前缀和 $\leq k$ 的最大位置。

原理：利用线段树分治性质，如果整个节点都满足条件就直接跳过，否则递归子节点。

关键：递归时保证从左到右/从右到左的顺序，维护连续性。

由于线段树的分治性质，可以做到线段树上二分

实际上，根据线段树的递归性质，是可以严格从左遍历到右或者严格从右遍历到左

故采取遍历获得满足条件的端点，是可以保持连续的

二分通常都是一部分连续的可以，一部分连续的不可以

假如满足条件的点都会在左边，那么每次都优先递归左区间，就会在查询区间中从左到右查询

如果一个区间中的点都满足的话，那么把端点更新为线段树节点的右端点

一旦出现一个点不满足了，立刻停止

代码说明：sum 是剩余容量，x 是当前找到的最右端点。如果节点和 $\leq$sum 就更新 x 并减去该节点和，否则递归子节点。

```
// 要初始化一下 x 为第一个能满足的点的左边的那个点
void query_sum(int u, int l, int r, int & sum, int & x) // 查询 l~r 之间 l~x 的和小于等于 sum 的最
↪  大的 x
{
    if (l<= tr[u].l && r >= tr[u].r) // 如果线段树节点被查询区间完全包含
    {
        if (x < tr[u].l - 1) return ; // 如果不能和满足的点接上的话 说明已经有断点 退出
        if (tr[u].sum <= sum) x = tr[u].r, sum -= tr[u].sum; // 如果能完全满足的话 减去前缀和 更新端
        ↪  点
        else if (tr[u].l == tr[u].r) return ; // 如果完全不能满足 而且是叶节点 那么退出
        else // 否则继续往下找
        {
            pushdown(u); // 这里也要下传懒标记
            query_sum(u << 1, l, r, sum, x), query_sum(u << 1 | 1, l, r, sum, x);
        }
    }
    else
    {
```

```
        pushdown(u);
        int mid = (tr[u].l + tr[u].r) >> 1;
        if (l <= mid) query_sum(u << 1, l, r, sum, x);
        if (r > mid) query_sum(u << 1 | 1, l, r, sum, x);
    }
}
```

1.8.2 标记永久化 (堆式线段树)

用途：优化线段树的懒标记下传，减少 pushdown 操作。

原理：不下传标记，而是在查询时累加沿途的所有标记。

优势：代码更简洁，常数更小，适合简单的区间加法操作。

将修改的影响直接作用在当前节点, 并打上一个懒标记

在查询的时候, 递归查询的时候累加标记, 就不需要像普通线段树一样下传标记

```
struct Node
{
    int l, r, sum, tag;
} tr[N << 2];
void build(int u, int l, int r)
{
    if (l == r) tr[u] = {l, r, w[r], 0};
    else
    {
        tr[u] = {l, r, 0, 0};
        int mid = (l + r) >> 1;
        build(u << 1, l, mid);
        build(u << 1 | 1, mid + 1, r);
        tr[u].sum = tr[u << 1].sum + tr[u << 1 | 1].sum;
    }
}
void modify(int u, int l, int r, int val)
{
    tr[u].sum += (min(tr[u].r, r) - max(tr[u].l, l) + 1) * val;
    if (l <= tr[u].l && r >= tr[u].r)
    {
        tr[u].tag += val;
        return;
    }
    int mid = (tr[u].l + tr[u].r) >> 1;
    if (l <= mid) modify(u << 1, l, r, val);
    if (r > mid) modify(u << 1 | 1, l, r, val);
}
int query(int u, int l, int r, int s)
{
    if (l <= tr[u].l && r >= tr[u].r) return tr[u].sum + (tr[u].r - tr[u].l + 1) * s;
    int mid = (tr[u].l + tr[u].r) >> 1, res = 0;
    if (l <= mid) res += query(u << 1, l, r, s + tr[u].tag);
    if (r > mid) res += query(u << 1 | 1, l, r, s + tr[u].tag);
    return res;
}
```

1.8.3 动态开点线段树

注意：动态开点线段树的空间复杂度为 $O(m \log n)$，其中 m 为操作次数。通常需要预留 $4 \times m \times \log n$ 的空间，防止 MLE。

例子：默认一开始所有值都是 1。$t = 1$：$[l, r]$ 都变为 0；$t = 2$：$[l, r]$ 都变为 1，求总体的 sum。

动态开点线段树的核心是 create 函数。当需要访问一个节点但该节点尚未创建时，需要提前创建。

注意：传参时基本都要把 l, r 传进去，便于修改操作。

```
const int N = 16000010;
struct Node
{
    int l, r, sum, tag;
} tr[N];
int n, m, root, idx;
void create(int &p, int l, int r)
{
    p = ++idx; // 分配节点
    tr[p].sum = r - l + 1; // 初始化
    tr[p].tag = -1; // 初始化
}
```

```cpp
void pushup(int p, int l, int r)
{
    int mid = (l + r) >> 1;
    if (!tr[p].l) create(tr[p].l, l, mid); // 如果不存在就要先创建出来
    if (!tr[p].r) create(tr[p].r, mid + 1, r); // 如果不存在就要先创建出来
    tr[p].sum = tr[tr[p].l].sum + tr[tr[p].r].sum;
}
void pushdown(int p, int l, int r)
{
    if (tr[p].tag != -1) {
        int tag_val = tr[p].tag, mid = (l + r) >> 1;
        tr[p].tag = -1;
        if (!tr[p].l) create(tr[p].l, l, mid); // 如果不存在就要先创建出来
        if (!tr[p].r) create(tr[p].r, mid + 1, r); // 如果不存在就要先创建出来
        tr[tr[p].l].sum = (mid - l + 1) * tag_val;
        tr[tr[p].r].sum = (r - mid) * tag_val;
        tr[tr[p].l].tag = tr[tr[p].r].tag = tag_val;
    }
}
int modify(int p, int l, int r, int ql, int qr, int val)
{
    if (!p) create(p, l, r); // 如果不存在就要先创建
    if (ql <= l && qr >= r) {
        tr[p].sum = (r - l + 1) * val;
        tr[p].tag = val;
        return p;
    }
    pushdown(p, l, r);
    int mid = (l + r) >> 1;
    if (ql <= mid) tr[p].l = modify(tr[p].l, l, mid, ql, qr, val);
    if (qr > mid) tr[p].r = modify(tr[p].r, mid + 1, r, ql, qr, val);
    pushup(p, l, r);
    return p;
}
void solve()
{
    root = modify(root, 1, n, 1, n, 1);
    while (m--)
    {
        int t, l, r;
        if (t == 1) root = modify(root, 1, n, l, r, 0);
        else root = modify(root, 1, n, l, r, 1);
        cout << tr[root].sum << "\n";
    }
}
```

1.8.4 标记永久化 (可持久化/动态开点)

标记永久化一般搭配主席树使用

```cpp
struct Node
{
    int l, r, sum, tag;
}tr[N << 6];
int w[N], root[N], idx;
int build(int p, int l, int r)
{
    int q = ++idx;
    tr[q] = tr[p];
    if (l == r)
    {
        tr[q].sum = w[l];
        return q;
    }
    int mid = (l + r) >> 1;
    tr[q].l = build(tr[p].l, l, mid), tr[q].r = build(tr[p].r, mid + 1, r);
    tr[q].sum = tr[tr[q].l].sum + tr[tr[q].r].sum;
    return q;
}
int modify(int p, int l, int r, int ql, int qr, int val)
{
    int q = ++idx;
    tr[q].sum = tr[p].sum + (min(r, qr) - max(l, ql) + 1) * val, tr[q].tag = tr[p].tag;
    if (ql<= l && qr >= r)
```

```
    {
        tr[q].tag += val;
        tr[q].l = tr[p].l, tr[q].r = tr[p].r;
        return q;
    }
    int mid = (l + r) >> 1;
    if (ql <= mid) tr[q].l = modify(tr[p].l, l, mid, ql, qr, val);
    else tr[q].l = tr[p].l;
    if (qr > mid) tr[q].r = modify(tr[p].r, mid + 1, r, ql, qr, val);
    else tr[q].r = tr[p].r;
    return q;
}
int query(int p, int l, int r, int ql, int qr, int s)
{
    if (ql<= l && qr >= r) return tr[p].sum + (r - l + 1) * s;
    int mid = (l + r) >> 1, res = 0;
    if (ql <= mid) res += query(tr[p].l, l, mid, ql, qr, s + tr[p].tag);
    if (qr > mid) res += query(tr[p].r, mid + 1, r, ql, qr, s + tr[p].tag);
    return res;
}
```

1.8.5 主席树

用途： 静态区间第 k 小，可持久化数据结构，支持查询历史版本。

原理： 对每个前缀建立一棵权值线段树，利用前缀和思想：$query(l,r) = query(r) - query(l-1)$。

关键： 相邻版本只有 $O(\log n)$ 个节点不同，通过复用节点节省空间。

空间： $O(n \log n)$ 个节点，开 $N \times 20$ 一般够用。

```
struct Node
{
    int l, r, cnt;
}tr[N * 20];
int n, m, w[N], root[N], idx;
int insert(int p, int l, int r, int x)
{
    int q = ++idx;
    tr[q] = tr[p];
    if (l == r)
    {
        // do 对叶节点的操作
        return q;
    }
    int mid = (l + r) >> 1;
    if (x <= mid) tr[q].l = insert(tr[p].l, l, mid, x);
    else tr[q].r = insert(tr[p].r, mid + 1, r, x);
    // tr[q].cnt = tr[tr[q].l].cnt + tr[tr[q].r].cnt; pushup
    return q;
}
int query(int p, int l, int r, int k) // 查询 以查询第 k 小数为例
{
    if (l == r) return r;
    int cnt = tr[tr[p].l].cnt, mid = l + r >>1;
    if (cnt >= k) return query(tr[p].l, l, mid, k);
    else return query(tr[p].r, mid + 1, r, k - cnt);
}
// 如果查询区间 (l, r) 的第 k 小数
for (int i = 1; i <= n; i++) root[i] = insert(root[i - 1], 0, INF, w[i]);
int query(int q, int p, int l, int r, int k)
{
    if (l == r) return r;
    int cnt = tr[tr[q].l].cnt - tr[tr[p].l].cnt;
    int mid = (l + r) >> 1;
    if (k <= cnt) return query(tr[q].l, tr[p].l, l, mid, k);
    else return query(tr[q].r, tr[p].r, mid + 1, r, k - cnt);
}// 查询 query(root[r], root[l - 1], 0, v.size() - 1, k)
```

例: 在线询问区间 mex

```
struct Node
{
    int l, r, minv;
}tr[N << 5];
```

```cpp
int n, m, w[N], root[N], idx, pre[N];
int insert(int p, int l, int r, int x, int pos)
{
    int q = ++idx;
    tr[q] = tr[p];
    if (l == r)
    {
        tr[q].minv = pos;
        return q;
    }
    int mid = (l + r) >> 1;
    if (x <= mid) tr[q].l = insert(tr[p].l, l, mid, x, pos);
    else tr[q].r = insert(tr[p].r, mid + 1, r, x, pos);
    tr[q].minv = min(tr[tr[q].l].minv, tr[tr[q].r].minv);
    return q;
}
int query(int p, int l, int r, int x)
{
    if (l == r) return r;
    int mid = (l + r) >> 1;
    if (tr[tr[p].l].minv < x) return query(tr[p].l, l, mid, x);
    else return query(tr[p].r, mid + 1, r, x);
}
int main()
{ // 如果对一个版本线段树修改多次 先让 root[i] = root[i - 1], 再把下面的 root[i - 1] 换成 root[i - 1]
    for (int i = 1; i <= n; i++) root[i] = insert(root[i - 1], 0, (int)2e5, w[i], i);
    while (m--) cout << query(root[r], 0, (int)2e5, l) << "\n";
}
```

1.8.6 区间修改主席树

一个长度为 N 的数组 $\{A\}$,4 种操作:

C l r d: 区间 $[l,r]$ 中的数都加 d, 同时当前的时间戳加 1。

Q l r: 查询当前时间戳区间 $[l,r]$ 中所有数的和。

H l r t: 查询时间戳 t 区间 $[l,r]$ 的和。

B t: 将当前时间戳置为 t 。

```cpp
struct Node
{
    int l, r;
    int sum, tag; // 和 标记永久化的值
}tr[N << 6];
int w[N], root[N], record[N], idx;
int build(int p, int l, int r) // 初始化线段树
{
    int q = ++idx;
    tr[q] = tr[p];
    if (l == r)
    {
        tr[q].sum = w[l];
        return q;
    }
    int mid = (l + r) >> 1;
    tr[q].l = build(tr[p].l, l, mid), tr[q].r = build(tr[p].r, mid + 1, r);
    tr[q].sum = tr[tr[q].l].sum + tr[tr[q].r].sum;
    return q;
}
int modify(int p, int l, int r, int ql, int qr, int val)
{
    int q = ++idx;
    tr[q].sum = tr[p].sum + (min(r, qr) - max(l, ql) + 1) * val, tr[q].tag = tr[p].tag;
    if (ql<= l && qr >= r)
    {
        tr[q].tag += val;
        tr[q].l = tr[p].l, tr[q].r = tr[p].r;
        return q;
    }
    int mid = (l + r) >> 1;
    if (ql <= mid) tr[q].l = modify(tr[p].l, l, mid, ql, qr, val);
    else tr[q].l = tr[p].l;
    if (qr > mid) tr[q].r = modify(tr[p].r, mid + 1, r, ql, qr, val);
    else tr[q].r = tr[p].r;
    return q;
```

```
}
int query(int p, int l, int r, int ql, int qr, int s)
{
    if (ql<= l && qr >= r) return tr[p].sum + (r - l + 1) * s;
    int mid = (l + r) >> 1, res = 0;
    if (ql <= mid) res += query(tr[p].l, l, mid, ql, qr, s + tr[p].tag);
    if (qr > mid) res += query(tr[p].r, mid + 1, r, ql, qr, s + tr[p].tag);
    return res;
}
void solve()
{
    int n, m;
    for (int i = 1; i<= n; i++) cin >> w[i];
    root[0] = build(root[0], 1, n);
    int cur = 0, t = 0;
    while (m--)
    {
        if (opt=='C')
        {
            int l, r, d;
            record[++cur] = ++t;
            root[t] = modify(root[t - 1], 1, n, l, r, d);
        }
        else if (opt=='Q')
        {
            int l, r;
            cout << query(root[t], 1, n, l, r, 0) << "\n";
        }
        else if (opt=='H')
        {
            int l, r, k;
            cout << query(root[record[k]], 1, n, l, r, 0) << "\n";
        }
        else
        {
            cin >> cur;
            root[++t] = root[record[cur]];
        }
    }
}
```

1.8.7 线段树合并

用途：将两棵动态开点线段树合并成一棵，常用于树上问题的信息合并。

核心思想：递归合并对应位置的节点，叶子节点进行信息合并，空节点直接返回非空节点。

关键技巧：合并后 q 树会被破坏，需要先处理子树询问再合并到父节点。

时间复杂度：合并两棵树的时间复杂度为 O(两树节点数之和)。

```
void pushup(int p) // 用子节点信息更新父节点
{
    tr[p].v = max(tr[tr[p].l].v, tr[tr[p].r].v);
}
int modify(int p, int l, int r, int x, int val)
{
    if (!p) p = ++idx;
    if (l == r) // 这里执行一些在叶子节点的具体合并操作
    {
        tr[p].v.first += val, tr[p].v.second = x;
        return p;
    }
    int mid = (l + r) >> 1;
    if (x <= mid) tr[p].l = modify(tr[p].l, l, mid, x, val);
    else tr[p].r = modify(tr[p].r, mid + 1, r, x, val);
    pushup(p);
    return p;
}
int merge(int p, int q, int l, int r) //将 q 合并到 p 上
{
    if (!p||!q) return p | q; // 如果有一个为空 直接合并
    if (l == r)
    {
        tr[p].v.first += tr[q].v.first;
        return p; // 一定要 return
```

```
    }
    int mid = (l + r) >> 1;
    tr[p].l = merge(tr[p].l, tr[q].l, l, mid);
    tr[p].r = merge(tr[p].r, tr[q].r, mid + 1, r);
    pushup(p);
    return p;
}
void merge() { root[x] = merge(root[x], root[y], 1, N - 1); } // 相当于 x += y
void modify() { root[x] = modify(root[x], 1, N - 1, z, 1); } // 在 x 的这棵线段树上的 z 位置加 1
```

注意：对于合并操作 p,q，q 会直接归为 p 的一部分，会影响 q 节点答案的查询。

解决方案：一般要把询问挂到节点上，使用 dfs 解决询问，先解决儿子的，再解决父亲的，从而不会影响儿子的询问。

替代方案：如果想要解决这个问题，可以在线段树合并的时候新建一个节点作为替代：

```
int merge(int p, int q, int l, int r)
{
    if (!p||!q) return p | q;
    int nw = ++idx;
    if (l == r)
    {
        tr[nw].cnt = tr[nw].maxv = tr[p].cnt + tr[q].cnt;
        return nw;
    }
    int mid = (l + r) >> 1;
    tr[nw].l = merge(tr[p].l, tr[q].l, l, mid);
    tr[nw].r = merge(tr[p].r, tr[q].r, mid + 1, r);
    pushup(nw);
    return nw;
}
```

例子：CF600E

每个结点都有一个颜色，颜色是以编号表示的。i 号结点的颜色编号为 c_i。

如果一种颜色在以 x 为根的子树内出现次数最多，称其在以 x 为根的子树中占主导地位。显然，同一子树中可能有多种颜色占主导地位。

对于每一个 $i\in[1,n]$ 求出以 i 为根的子树中，占主导地位的颜色的编号和。

```
struct Node
{
    int l, r, maxcnt, sum;
}tr[N << 6];
int n, root[N], c[N], ans[N], idx, h[N], ne[N << 1], e[N << 1];
void pushup(int p)
{
    if (tr[tr[p].l].maxcnt == tr[tr[p].r].maxcnt)
    {
        tr[p].maxcnt = tr[tr[p].l].maxcnt;
        tr[p].sum = tr[tr[p].l].sum + tr[tr[p].r].sum;
    }
    else if (tr[tr[p].l].maxcnt > tr[tr[p].r].maxcnt)
    {
        tr[p].sum = tr[tr[p].l].sum;
        tr[p].maxcnt = tr[tr[p].l].maxcnt;
    }
    else tr[p].sum = tr[tr[p].r].sum, tr[p].maxcnt = tr[tr[p].r].maxcnt;
}
int modify(int p, int l, int r, int x)
{
    if (!p) p = ++idx;
    if (l == r)
    {
        tr[p].maxcnt++, tr[p].sum = x;
        return p;
    }
    int mid = (l + r) >> 1;
    if (x <= mid) tr[p].l = modify(tr[p].l, l, mid, x);
    else tr[p].r = modify(tr[p].r, mid + 1, r, x);
    pushup(p);
    return p;
}
int merge(int p, int q, int l, int r)
{
    if (!p||!q) return p | q;
```

```
    if (l == r)
    {
        tr[p].maxcnt += tr[q].maxcnt;
        return p;
    }
    int mid = (l + r) >> 1;
    tr[p].l = merge(tr[p].l, tr[q].l, l, mid), tr[p].r = merge(tr[p].r, tr[q].r, mid + 1, r);
    pushup(p);
    return p;
}
void dfs(int u, int fa)
{
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j == fa) continue;
        dfs(j, u);
        root[u] = merge(root[u], root[j], 1, n);
    }
    ans[u] = tr[root[u]].sum;
}
int main()
{
    for (int i = 1; i <= n; i++) root[i] = modify(root[i], 1, n, c[i]);
    dfs(1, -1);
}
```

1.8.8 线段树分裂

用途：将一棵线段树按指定区间分裂成两棵树，常用于区间移动和分组问题。

核心思想：递归地将指定区间从原树中“切出”，分裂后原树该区间被清空。

注意事项：参数 p 必须用引用，因为分裂后原节点会被清空。通常分裂后要重新合并。

```
// 可以考虑分裂完之后再合并
int split(int & p, int l, int r, int ql, int qr) // 对于 p 分裂出 ql 和 qr 的区间
{
    int q = ++idx; // 要新开节点
    if (ql <= l && qr >= r)
    {
        tr[q] = tr[p];
        p = 0; // p 要用引用 因为这里分裂出来之后要清空
        return q;
    }
    int mid = (l + r) >> 1;
    if (ql <= mid) tr[q].l = split(tr[p].l, l, mid, ql, qr);
    if (qr > mid) tr[q].r = split(tr[p].r, mid + 1, r, ql, qr);
    pushup(p), pushup(q);
    return q;
}

```

例子：CF991G

给出一个数列，有 q 个操作，每种操作是把区间 $[l,r]$ 中等于 x 的数改成 y。输出 q 步操作完的数列。

```
struct Node
{
    int l, r, cnt;
}tr[N * M];
int root[N], idx, n, m, w[N];
void pushup(int p) { tr[p].cnt = tr[tr[p].l].cnt + tr[tr[p].r].cnt; }
int modify(int p, int l, int r, int x)
{
    if (!p) p = ++idx;
    if (l == r)
    {
        tr[p].cnt++;
        return p;
    }
    int mid = (l + r) >> 1;
    if (x <= mid) tr[p].l = modify(tr[p].l, l, mid, x);
    else tr[p].r = modify(tr[p].r, mid + 1, r, x);
    pushup(p);
    return p;
```

```
}
int split(int & p, int l, int r, int ql, int qr)
{
    int q = ++idx;
    if (ql <= l && qr >= r)
    {
        tr[q] = tr[p], p = 0;
        return q;
    }
    int mid = (l + r) >> 1;
    if (ql <= mid) tr[q].l = split(tr[p].l, l, mid, ql, qr);
    if (qr > mid) tr[q].r = split(tr[p].r, mid + 1, r, ql, qr);
    pushup(p), pushup(q);
    return q;
}
int merge(int p, int q, int l, int r)
{
    if (!p||!q) return p | q;
    else if (l == r)
    {
        tr[p].cnt += tr[q].cnt;
        return p;
    }
    int mid = (l + r) >> 1;
    tr[p].l = merge(tr[p].l, tr[q].l, l, mid);
    tr[p].r = merge(tr[p].r, tr[q].r, mid + 1, r);
    pushup(p);
    return p;
}
void query(int p, int l, int r, int v)
{
    if (!tr[p].cnt) return ;
    if (l == r)
    {
        w[l] = v;
        return ;
    }
    int mid = (l + r) >> 1;
    if (tr[tr[p].l].cnt) query(tr[p].l, l, mid, v);
    if (tr[tr[p].r].cnt) query(tr[p].r, mid + 1, r, v);
}
int main()
{
    for (int i = 1; i <= n; i++) root[w[i]] = modify(root[w[i]], 1, n, i);
    while (m--)
    {
        int l, r, x, y;
        if (x == y) continue;
        int temp = split(root[x], 1, n, l, r);
        root[y] = merge(root[y], temp, 1, n);
    }
    for (int i = 1; i <= 100; i++) query(root[i], 1, n, i);
    for (int i = 1; i <= n; i++) cout << w[i] << " ";
}
```

1.8.9 线段树分治

用途：处理有删除操作的动态维护问题，特别是删除操作难以直接处理时。

原理：将时间轴建成线段树，每个修改操作对应一个时间区间，在线段树上进行分治处理。

技巧：删除操作转化为添加 + 撤销，配合可撤销数据结构（如可撤销并查集）。

应用：动态图连通性、动态维护二分图性质等。

线段树分治就是维护一些只存在一段时间的元素的贡献。有时候一些元素的删除并不好做，可以将其转化为只在某一段时间内存在。将这个存在时间利用线段树的结构分治，将删除操作转化为添加操作和撤销操作。

线段树分治能够离线维护动态图的连通性，利用扩展域并查集还可以维护一个图是不是二分图，连边 (x, y) 转化为连边 $(x, y+n)$ 和 $(x+n, y)$

给定一棵 n 个点的树，给第 i 个点染上颜色 c_i，其中，c_i 为 $[1, n]$ 的一个整数。现在，对于每一种颜色 k，你要求出有多少条简单路径满足路径上至少有一个点的颜色为 k。

每种颜色是独立的，单独考虑每一种颜色。考虑容斥，$\text{ans} = \frac{n\times(n+1)}{2} -$ 简单路径没有点颜色为 k 的数量。考虑后者，将颜色为 k 的点删去，结果为 $\sum \frac{(sz_i+1)\times sz_i}{2} - cnt_k$。假设一条边两端颜色分别是 c_1, c_2，则将其在 $x \neq c_1 \wedge x \neq c_2$ 时加入，利用线段树分治和可撤销并查集快速维护答案。

```cpp
int n, cur, c[N], ans[N], p[N], sz[N], cnt[N];
vector<int> stk; // 可撤销并查集
struct Node
{
    int l, r;
    vector<array<int, 2>> v;
}tr[N << 2];
void build(int u, int l, int r)
{
    tr[u] = {l, r};
    if (l == r) return ;
    int mid = (l + r) >> 1;
    build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
}
void modify(int u, int l, int r, int x, int y)
{
    if (l > r) return ; // 一般要加这个 防止不合法
    if (l<= tr[u].l && r >= tr[u].r) tr[u].v.push_back({x, y});
    else
    {
        int mid = (tr[u].l + tr[u].r) >> 1;
        if (l <= mid) modify(u << 1, l, r, x, y);
        if (r > mid) modify(u << 1 | 1, l, r, x, y);
    }
}
void merge(int x, int y) // 按秩合并
{
    if (sz[x] > sz[y]) swap(x, y);
    p[x] = p[y], sz[y] += sz[x];
    stk.push_back(x);
}
int find(int x) // 暴力找祖先
{
    while (x != p[x]) x = p[x];
    return x;
}
int cal(int x) { return x * (x + 1) / 2; }
void solve(int u, int l, int r)
{
    int insz = stk.size();
    for (auto [x, y]:tr[u].v)
    {
        int px = find(x), py = find(y);
        if (px == py) continue;
        else
        {
            cur -= cal(sz[px]) + cal(sz[py]);
            merge(px, py);
            cur += cal(sz[find(px)]);
        }
    }
    int mid = (l + r) >> 1;
    if (l == r) ans[l] = cur - cnt[l];
    else solve(u << 1, l, mid), solve(u << 1 | 1, mid + 1, r);
    while (stk.size() > insz)
    {
        int x = stk.back(), y = p[x];
        stk.pop_back();
        cur -= cal(sz[y]);
        sz[y] -= sz[x], p[x] = x;
        cur += cal(sz[x]) + cal(sz[y]);
    }
}
void solve()
{
    for (int i = 1; i <= n; i++) p[i] = i, sz[i] = 1;
    cur = n;
    build(1, 1, n);
    for (int i = 1; i<= n; i++) cin >> c[i], cnt[c[i]]++;
    for (int i = 0; i < n - 1; i++)
    {
        int a, b;
        int c1 = c[a], c2 = c[b];
        if (c1 > c2) swap(c1, c2);
```

```
        modify(1, 1, c1 - 1, a, b);
        modify(1, c1 + 1, c2 - 1, a, b);
        modify(1, c2 + 1, n, a, b);
    }
    solve(1, 1, n);
    for (int i = 1; i <= n; i++) cout << n * (n + 1) / 2 - ans[i] << "\n";
}
```

1.8.10 扫描线

扫描线求面积并

```
struct Node
{
    int l, r, len, tag; // 注意这里用节点 i 代表 i~i + 1 这一段的长度
}tr[N << 3]; //8 倍空间
void pushup(int u)
{
    if (tr[u].tag) tr[u].len = lsh[tr[u].r + 1] - lsh[tr[u].l]; // 如果被覆盖了 长度就是总长
    else if (tr[u].l == tr[u].r) tr[u].len = 0; // 如果是叶节点 那就没有长度
    else tr[u].len = tr[u << 1].len + tr[u << 1 | 1].len; // 否则就是左右儿子长度相加
}
void build(int u, int l, int r)
{
    tr[u] = {l, r};
    if (l == r) return ;
    int mid = (l + r) >> 1;
    build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
}
void modify(int u, int l, int r, int v)
{
    if (l<= tr[u].l && r >= tr[u].r) tr[u].tag += v, pushup(u); // 注意 pushup
    else
    {
        int mid = (tr[u].l + tr[u].r) >> 1;
        if (l <= mid) modify(u << 1, l, r, v);
        if (r > mid) modify(u << 1 | 1, l, r, v);
        pushup(u);
    }
}
void solve()
{
    vector<array<int, 4>> v;
    for (int i = 1; i <= n; i++)
    {
        int x1, y1, x2, y2;
        lsh.push_back(y1), lsh.push_back(y2);
        v.push_back({x1, y1, y2, 1}), v.push_back({x2, y1, y2, -1});
    }
    sort(v.begin(), v.end());
    build(1, 0, lsh.size() - 2);
    for (int i = 0; i < v.size(); i++)
    {
        if (i) ans += (v[i][0] - v[i - 1][0]) * tr[1].len;
        auto [x, y1, y2, sign] = v[i];
        y1 = find(y1), y2 = find(y2);
        modify(1, y1, y2 - 1, sign);
    }
}
```

例题：ABC346G (求序列有多少个子区间中至少有一个数字只出现过一次)

对于每个 a_i 求出其能合法延伸的最左端点和最右端点 L_i 和 R_i, 那么对于 (l,r) 满足 $L_i \leqslant l \leqslant i, i \leqslant r \leqslant R_i$ 的区间都是合法的。

考虑对于每个 a_i 统计区间, 可能会有重复。注意到合法区间可以看成二维平面上的点, 实际上可以求所有形如 $L_i \leqslant l \leqslant i, i \leqslant r \leqslant R_i$ 的矩形在二维平面上的面积并。

```
int n, w[N], l[N], r[N], pos[N];
vector<array<int, 3>> buc[N];
struct Node
{
    int l, r, len, tag;
}tr[N << 2];
```

```cpp
void pushup(int u)
{
    if (tr[u].tag) tr[u].len = tr[u].r - tr[u].l + 1;
    else if (tr[u].l == tr[u].r) tr[u].len = 0;
    else tr[u].len = tr[u << 1].len + tr[u << 1 | 1].len;
}
void build(int u, int l, int r)
{
    tr[u] = {l, r};
    if (l == r) return ;
    int mid = (l + r) >> 1;
    build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
}
void modify(int u, int l, int r, int v)
{
    if (l<= tr[u].l && r >= tr[u].r) tr[u].tag += v, pushup(u); // 注意 pushup
    else
    {
        int mid = (tr[u].l + tr[u].r) >> 1;
        if (l <= mid) modify(u << 1, l, r, v);
        if (r > mid) modify(u << 1 | 1, l, r, v);
        pushup(u);
    }
}
void solve()
{
    for (int i = 1; i <= n; i++) l[i] = pos[w[i]], pos[w[i]] = i;
    for (int i = 1; i <= n; i++) pos[w[i]] = n + 1;
    for (int i = n; i >= 1; i--) r[i] = pos[w[i]], pos[w[i]] = i;
    for (int i = 1; i <= n; i++)
    {
        buc[l[i] + 1].push_back({i, r[i] - 1, 1});
        buc[i + 1].push_back({i, r[i] - 1, -1});
    }
    build(1, 1, n);
    int ans = 0;
    for (int i = 1; i <= n; i++)
    {
        for (auto [x, y, v]:buc[i]) modify(1, x, y, v);
        ans += tr[1].len;
    }
}
```

1.8.11 李超线段树

维护线段 (使用前记得 build)

```cpp
struct LCT
{
    using pdi = pair<double, int>;
    const double eps = 1e - 9;
    double k[N], b[N];
    struct Node
    {
        int l, r, id;
    }tr[M << 2];
    int cnt;
    void build(int u, int l, int r)
    {
        tr[u] = {l, r};
        if (l == r) return ;
        int mid = (l + r) >> 1;
        build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
    }
    int cmp(double x, double y)
    {
        if (x - y > eps) return 1;
        if (y - x > eps) return - 1;
        return 0;
    }
    double cal(int id, int x) { return k[id] * x + b[id]; }
    void modify(int u, int p)
    {
```

```
        int & q = tr[u].id, l = tr[u].l, r = tr[u].r, mid = l + r >>1;
        int bmid = cmp(cal(p, mid), cal(q, mid));
        if (bmid == 1 || (!bmid && p < q)) swap(p, q);
        int bl = cmp(cal(p, l), cal(q, l)), br = cmp(cal(p, r), cal(q, r));
        if (bl == 1 || (!bl && p < q)) modify(u << 1, p);
        if (br == 1 || (!br && p < q)) modify(u << 1 | 1, p);
    }
    void modify(int u, int l, int r, int id)
    {
        if (l<= tr[u].l && r >= tr[u].r) modify(u, id);
        else
        {
            int mid = (tr[u].l + tr[u].r) >> 1;
            if (l <= mid) modify(u << 1, l, r, id);
            if (r > mid) modify(u << 1 | 1, l, r, id);
        }
    }
    void add(int x1, int y1, int x2, int y2)
    {
        if (x1 > x2) swap(x1, x2), swap(y1, y2);
        cnt++;
        if (x1 == x2) k[cnt] = 0, b[cnt] = max(y1, y2);
        else k[cnt] = 1.0 * (y2 - y1) / (x2 - x1), b[cnt] = y1 - k[cnt] * x1;
        modify(1, x1, x2, cnt);
    }
    pdi pmax(pdi x, pdi y)
    {
        if (cmp(x.first, y.first) == -1) return y;
        else if (cmp(x.first, y.first) == 1) return x;
        else return x.second < y.second?x:y;
    }
    pdi query(int u, int x)
    {
        int mid = (tr[u].l + tr[u].r) >> 1;
        pdi res = {cal(tr[u].id, x), tr[u].id};
        if (tr[u].l == tr[u].r) return res;
        if (x <= mid) return pmax(res, query(u << 1, x));
        else return pmax(res, query(u << 1 | 1, x));
    }
}lct;
```

动态开点李超树 (插入一条直线, 使用之前要记得 build)
注意: 空间大小等于插入直线条数, 有很好的空间复杂度。

```
struct LCT
{
    using pdi = pair<double, int>;
    const double eps = 1e - 9, INF = 4e18;
    double k[N], b[N];
    struct Node
    {
        int l, r, id;
    }tr[N];
    int tot = 1, cnt, L, R;
    void build(int l, int r) { L = l, R = r, b[0] = -INF; }
    int cmp(double x, double y)
    {
        if (x - y > eps) return 1;
        if (y - x > eps) return - 1;
        return 0;
    }
    double cal(int id, int x) { return k[id] * x + b[id]; }
    int modify(int p, int l, int r, int u)
    {
        if (l > r) return 0;
        if (!p) p = ++tot;
        int & v = tr[p].id, mid = l + r >>1;
        int bmid = cmp(cal(u, mid), cal(v, mid));
        if (bmid == 1 || (!bmid && u < v)) swap(u, v);
        int bl = cmp(cal(u, l), cal(v, l)), br = cmp(cal(u, r), cal(v, r));
        if (bl == 1 || (!bl && u < v)) tr[p].l = modify(tr[p].l, l, mid, u);
        if (br == 1 || (!br && u < v)) tr[p].r = modify(tr[p].r, mid + 1, r, u);
        return p;
```

```
    }
    void add(double kk, double bb)
    {
        k[++cnt] = kk, b[cnt] = bb;
        modify(1, L, R, cnt);
    }
    pdi pmax(pdi x, pdi y)
    {
        if (cmp(x.first, y.first) == -1) return y;
        else if (cmp(x.first, y.first) == 1) return x;
        else return x.second < y.second?x:y;
    }
    pdi query(int p, int l, int r, int x)
    {
        if (!p) return {-INF, 0};
        int mid = (l + r) >> 1;
        pdi res = {cal(tr[p].id, x), tr[p].id};
        if (x <= mid) return pmax(res, query(tr[p].l, l, mid, x));
        else return pmax(res, query(tr[p].r, mid + 1, r, x));
    }
    pdi query(int x) {return query(1, L, R, x); }
}lct;
-------------
lct.build(l, r); //维护的值域
```

如果维护的 x 坐标是实数，可以离散化之后再操作

例题: 有一个二元组集合 S 初始为空, 每次询问给你 (a, b, x, y), 先把二元组 (x, y) 加入集合 S, 再求 $\max_{(x,y)\in S}\{ax + by\}$

设 $res = ax + by$, 则 $\frac{res}{b} = \frac{a}{b}x + y$, 相当于插入斜率为 x, 截距为 y 的直线, 分类讨论 b 的正负即可

```
struct MAXLCT
{
    using pdi = pair<double, int>;
    const double eps = 1e - 12, INF = 4e18;
    double k[N], b[N];
    struct Node
    {
        int l, r, id;
    }tr[N];
    int tot = 1, cnt, L, R;
    void build(int l, int r) { L = l, R = r, b[0] = -2 * INF; }
    int cmp(double x, double y)
    {
        if (x - y > eps) return 1;
        if (y - x > eps) return - 1;
        return 0;
    }
    double cal(int id, int x) { return k[id] * lsh[x - 1] + b[id]; }
    int modify(int p, int l, int r, int u)
    {
        if (l > r) return 0;
        if (!p) p = ++tot;
        int & v = tr[p].id, mid = l + r >>1;
        int bmid = cmp(cal(u, mid), cal(v, mid));
        if (bmid == 1 || (!bmid && u < v)) swap(u, v);
        int bl = cmp(cal(u, l), cal(v, l)), br = cmp(cal(u, r), cal(v, r));
        if (bl == 1 || (!bl && u < v)) tr[p].l = modify(tr[p].l, l, mid, u);
        if (br == 1 || (!br && u < v)) tr[p].r = modify(tr[p].r, mid + 1, r, u);
        return p;
    }
    void add(double kk, double bb)
    {
        k[++cnt] = kk, b[cnt] = bb;
        modify(1, L, R, cnt);
    }
    pdi pmax(pdi x, pdi y)
    {
        if (cmp(x.first, y.first) == -1) return y;
        else if (cmp(x.first, y.first) == 1) return x;
        else return x.second < y.second?x:y;
    }
    pdi query(int p, int l, int r, int x)
    {
        if (!p) return {-INF, 0};
```

```
        int mid = (l + r) >> 1;
        pdi res = {cal(tr[p].id, x), tr[p].id};
        if (x <= mid) return pmax(res, query(tr[p].l, l, mid, x));
        else return pmax(res, query(tr[p].r, mid + 1, r, x));
    }
    pdi query(int x) {return query(1, L, R, x); }
}lct1;
struct MINLCT
{
    using pdi = pair<double, int>;
    const double eps = 1e - 12, INF = 4e18;
    double k[N], b[N];
    struct Node
    {
        int l, r, id;
    }tr[N];
    int tot = 1, cnt, L, R;
    void build(int l, int r) { L = l, R = r, b[0] = 2 * INF; }
    int cmp(double x, double y)
    {
        if (x - y > eps) return 1;
        if (y - x > eps) return - 1;
        return 0;
    }
    double cal(int id, int x) { return k[id] * lsh[x - 1] + b[id]; }
    int modify(int p, int l, int r, int u)
    {
        if (l > r) return 0;
        if (!p) p = ++tot;
        int & v = tr[p].id, mid = l + r >>1;
        int bmid = cmp(cal(u, mid), cal(v, mid));
        if (bmid == -1 || (!bmid && u < v)) swap(u, v);
        int bl = cmp(cal(u, l), cal(v, l)), br = cmp(cal(u, r), cal(v, r));
        if (bl == -1 || (!bl && u < v)) tr[p].l = modify(tr[p].l, l, mid, u);
        if (br == -1 || (!br && u < v)) tr[p].r = modify(tr[p].r, mid + 1, r, u);
        return p;
    }
    void add(double kk, double bb)
    {
        k[++cnt] = kk, b[cnt] = bb;
        modify(1, L, R, cnt);
    }
    pdi pmin(pdi x, pdi y)
    {
        if (cmp(x.first, y.first) == -1) return x;
        else if (cmp(x.first, y.first) == 1) return y;
        else return x.second < y.second?x:y;
    }
    pdi query(int p, int l, int r, int x)
    {
        if (!p) return {INF, 0};
        int mid = (l + r) >> 1;
        pdi res = {cal(tr[p].id, x), tr[p].id};
        if (x <= mid) return pmin(res, query(tr[p].l, l, mid, x));
        else return pmin(res, query(tr[p].r, mid + 1, r, x));
    }
    pdi query(int x) {return query(1, L, R, x); }
}lct2;
void solve()
{
    while (q--)
    {
        int x, y, a, b;
        v.push_back({x, y, a, b});
        if (b) lsh.push_back(1.0L * a / b);
    }
    int maxx = -2E9, minx = 2E9;
    lct1.build(1, lsh.size()), lct2.build(1, lsh.size());
    for (auto [x, y, a, b]:v)
    {
        lct1.add(x, y), lct2.add(x, y);
        maxx = max(maxx, x), minx = min(minx, x);
        if (!b) cout << max(minx * a, maxx * a) << "\n";
        else
```

```
        {
            double XX = 1.0L * a / b;
            int id;
            if (b >0) id = lct1.query(find(XX)).second;
            else id = lct2.query(find(XX)).second;
            auto [X, Y, A, B] = v[id - 1];
            cout << a * X + b * Y << "\n";
        }
    }
}
```

1.8.12 单侧递归线段树更新

单点修改 求有多少个点满足前面所有点和 $(0,0)$ 的连线斜率都小于它和 $(0,0)$ 的连线斜率

```
struct Node
{
    int l, r, h, sum;
    double maxk;
}tr[N << 2];
int cal(int u, double maxk)
{
    if (maxk >= tr[u].maxk) return 0; //如果左侧的斜率完全大于 那么右边全部不可见
    if (tr[u].l == tr[u].r) return tr[u].maxk > maxk; //如果只剩一个点了 那么就看和左侧的斜率大小关系
    if (tr[u << 1].maxk <= maxk) return cal(u << 1 | 1, maxk); //如果左侧斜率完全大于左边斜率 递归右
    ↪ 边计算
    else return tr[u].sum - tr[u << 1].sum + cal(u << 1, maxk); //加上右边的答案 递归计算左边的贡献
}
void pushup(int u)
{
    tr[u].maxk = max(tr[u << 1].maxk, tr[u << 1 | 1].maxk);
    tr[u].sum = tr[u << 1].sum + cal(u << 1 | 1, tr[u << 1].maxk);
}
```

1.8.13 吉司机线段树

区间取 min, 区间求和

吉司机线段树的过程如下:

假设我们要对于区间 $[l,r]$ 中的数对 x 取 min, 我们首先将 $[l,r]$ 拆分成线段树上若干个区间 $[l_i,r_i]$ 对于每个 $[l_i,r_i]$:

如果它的最大值 $\leqslant x$ 那么显然此次操作没有任何效果, 直接 return 即可。

如果 x 小于最大值 $\max\{v\}$, 但大于（注意, 这里必须是严格大于, 否则的话也会出问题）严格次大值 se, 那么显然有且只有 c 个最大值会变为 x, 我们可以简单维护一个标记 tag 表示这段区间内最大值会增加 tag , 然后令 $tag \leftarrow tag+(x-\max\{v\})$ 即可, 由于更新完之后最大值依然严格大于次大值, 因此最大值个数不会发生变化。

如果 x 小于等于严格次大值 se, 这个就没有什么优美的方法了, 直接暴力递归左右子区间即可。

吉司机线段树复杂度是 $O(n\log n)$ 的，证明如下:

我们记一个节点的容为这段区间内不同数的个数, 那么显然在不断取 min 的过程中容只可能越来越小。而如果对于某个区间如果我们对其进行暴力递归, 那么原来是最大值和次大值的位置上的数必然会变为同一个值, 区间的容建一, 而所有区间的长度之和是 $n\log n$ 级别的, 因此所有区间容之和也是 $n\log n$ 级别的, 暴力递归的次数也是 $n\log n$

例题:

有 n 张卡牌, 每张卡牌都有三个属性, 第 i 张卡牌的三个属性记作 a_i,b_i,c_i。卡牌 x 可以打败卡牌 y 当且仅当 x 至少有两个属性值比 y 对应的属性值大。如卡牌 $(1,2,3)$ 可以打败卡牌 $(3,1,2)$, 因为 $2>1,3>2$。

现在请你判断满足 $a\leqslant p,b\leqslant q,c\leqslant r$ 的卡牌中，可以打败给定的所有 n 张卡牌的有多少张。

```
int n, p, q, r, B[N];
struct Node
{
    int l, r;
    int maxv, cnt;
    int se, sum, tag;
}tr[N << 2];
void pushup(int u)
{
    tr[u].maxv = max(tr[u << 1].maxv, tr[u << 1 | 1].maxv);
    tr[u].sum = tr[u << 1].sum + tr[u << 1 | 1].sum;
    if (tr[u << 1].maxv > tr[u << 1 | 1].maxv)
    {
        tr[u].se = max(tr[u << 1].se, tr[u << 1 | 1].maxv);
        tr[u].cnt = tr[u << 1].cnt;
    }
```

```
    else if (tr[u << 1].maxv < tr[u << 1 | 1].maxv)
    {
        tr[u].se = max(tr[u << 1 | 1].se, tr[u << 1].maxv);
        tr[u].cnt = tr[u << 1 | 1].cnt;
    }
    else
    {
        tr[u].se = max(tr[u << 1].se, tr[u << 1 | 1].se);
        tr[u].cnt = tr[u << 1].cnt + tr[u << 1 | 1].cnt;
    }
}
void build(int u, int l, int r)
{
    if (l == r) tr[u] = {l, r, B[l], 1, -INF, B[l]};
    else
    {
        tr[u] = {l, r};
        int mid = (l + r) >> 1;
        build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
        pushup(u);
    }
}
void apply(Node & u, int t) { u.sum += u.cnt * t, u.maxv += t, u.tag += t; }
void pushdown(int u)
{
    if (tr[u].tag)
    {
        int t = tr[u].tag;
        tr[u].tag = 0;
        if (tr[u << 1].maxv > tr[u << 1 | 1].maxv) apply(tr[u << 1], t);
        else if (tr[u << 1 | 1].maxv > tr[u << 1].maxv) apply(tr[u << 1 | 1], t);
        else apply(tr[u << 1], t), apply(tr[u << 1 | 1], t);
    }
}
void modify(int u, int l, int r, int v)
{
    if (v >= tr[u].maxv || l > r) return ;
    if (l<= tr[u].l && r >= tr[u].r)
    {
        if (tr[u].se < v) apply(tr[u], v - tr[u].maxv);
        else
        {
            pushdown(u);
            modify(u << 1, l, r, v), modify(u << 1 | 1, l, r, v);
            pushup(u);
        }
    }
    else
    {
        pushdown(u);
        int mid = (tr[u].l + tr[u].r) >> 1;
        if (l <= mid) modify(u << 1, l, r, v);
        if (r > mid) modify(u << 1 | 1, l, r, v);
        pushup(u);
    }
}
vector<array<int, 2>> v[N];
void solve()
{
    cin >> n >> p >> q >> r;
    for (int i = 1; i <= q; i++) B[i] = r;
    for (int i = 1, a, b, c; i <= n; i++)
    {
        cin >> a >> b >> c;
        v[a].push_back({b, c});
        B[b] = min(B[b], r - c);
    }
    for (int i = q - 1; i >= 1; i--) B[i] = min(B[i], B[i + 1]);
    build(1, 1, q);
    int ans = 0;
    for (int i = p; i >= 1; i--)
    {
        for (auto [b, c]:v[i]) modify(1, 1, b, 0), modify(1, b + 1, q, r - c);
        ans += tr[1].sum;
```

```
    }
    cout << ans << '\n';
}
```

带区间加的线段树

额外加一个懒标记 stag, 表示除了最大值之外的其他数都要增加 stag, 区间加 v 时令对应区间的 tag 和 stag 都加 v, 区间取 min 时只会影响 tag, 不会影响 stag 的值。下推标记就和普通吉司机树一样正常推即可。复杂度 $O(n \log^2 n)$

```
struct node
{
    i64 sum;
    int mx, se;
    int cnt, scnt;
    int tag, stag;
} st[N << 2];
#define lc cur << 1
#define rc cur << 1 | 1
void pushup(int cur)
{
    if (!st[lc].cnt)
    {
        st[cur] = st[rc];
        return;
    }
    if (!st[rc].cnt)
    {
        st[cur] = st[lc];
        return;
    }
    st[cur].sum = st[lc].sum + st[rc].sum;
    if (st[lc].mx > st[rc].mx)
    {
        st[cur].mx = st[lc].mx;
        st[cur].se = max(st[lc].se, st[rc].mx);
        st[cur].cnt = st[lc].cnt;
        st[cur].scnt = st[lc].scnt + st[rc].cnt + st[rc].scnt;
    }
    else if (st[lc].mx < st[rc].mx)
    {
        st[cur].mx = st[rc].mx;
        st[cur].se = max(st[rc].se, st[lc].mx);
        st[cur].cnt = st[rc].cnt;
        st[cur].scnt = st[rc].scnt + st[lc].cnt + st[lc].scnt;
    }
    else
    {
        st[cur].mx = st[rc].mx;
        st[cur].se = max(st[lc].se, st[rc].se);
        st[cur].cnt = st[lc].cnt + st[rc].cnt;
        st[cur].scnt = st[lc].scnt + st[rc].scnt;
    }
}
void pushdown(int cur)
{
    if (st[lc].mx == st[rc].mx)
    {
        st[lc].sum += (i64)st[cur].tag * st[lc].cnt;
        st[lc].tag += st[cur].tag;
        st[lc].mx += st[cur].tag;
        st[lc].sum += (i64)st[cur].stag * st[lc].scnt;
        st[lc].stag += st[cur].stag;
        st[lc].se += st[cur].stag;

        st[rc].sum += (i64)st[cur].tag * st[rc].cnt;
        st[rc].sum += (i64)st[cur].stag * st[rc].scnt;
        st[rc].tag += st[cur].tag;
        st[rc].stag += st[cur].stag;
        st[rc].mx += st[cur].tag;
        st[rc].se += st[cur].stag;
    }
    else if (st[lc].mx > st[rc].mx)
    {
        st[lc].sum += (i64)st[cur].tag * st[lc].cnt;
```

```
        st[lc].tag += st[cur].tag;
        st[lc].mx += st[cur].tag;
        st[lc].sum += (i64)st[cur].stag * st[lc].scnt;
        st[lc].stag += st[cur].stag;
        st[lc].se += st[cur].stag;

        st[rc].sum += (i64)st[cur].stag * st[rc].cnt;
        st[rc].sum += (i64)st[cur].stag * st[rc].scnt;
        st[rc].tag += st[cur].stag;
        st[rc].stag += st[cur].stag;
        st[rc].mx += st[cur].stag;
        st[rc].se += st[cur].stag;
    }
    else
    {
        st[lc].sum += (i64)st[cur].stag * st[lc].cnt;
        st[lc].tag += st[cur].stag;
        st[lc].mx += st[cur].stag;
        st[lc].sum += (i64)st[cur].stag * st[lc].scnt;
        st[lc].stag += st[cur].stag;
        st[lc].se += st[cur].stag;

        st[rc].sum += (i64)st[cur].tag * st[rc].cnt;
        st[rc].sum += (i64)st[cur].stag * st[rc].scnt;
        st[rc].tag += st[cur].tag;
        st[rc].stag += st[cur].stag;
        st[rc].mx += st[cur].tag;
        st[rc].se += st[cur].stag;
    }
    st[cur].tag = st[cur].stag = 0;
}
void insert(int cur, int l, int r, int p, int x)
{
    if (l == r)
    {
        st[cur] = {x, x, 0, 1, 0, 0, 0};
        return;
    }
    int mid = (l + r) >> 1;
    pushdown(cur);
    if (p <= mid) insert(lc, l, mid, p, x);
    else insert(rc, mid + 1, r, p, x);
    pushup(cur);
}
void update1(int cur, int l, int r, int a, int b)
{
    if (a > b or !st[cur].cnt) return;
    if (a <= l and r <= b)
    {
        st[cur].tag += 1, st[cur].stag += 1;
        st[cur].sum += st[cur].cnt + st[cur].scnt;
        st[cur].mx += 1, st[cur].se += 1;
        return;
    }
    int mid = (l + r) >> 1;
    pushdown(cur);
    if (a <= mid) update1(lc, l, mid, a, b);
    if (b > mid) update1(rc, mid + 1, r, a, b);
    pushup(cur);
}
void update2(int cur, int l, int r, int a, int b, int x)
{
    if (a > b or st[cur].mx <= x or !st[cur].cnt) return;
    if (a <= l and r<= b and x > st[cur].se)
    {
        st[cur].sum -= (i64)(st[cur].mx - x) * st[cur].cnt;
        st[cur].tag -= st[cur].mx - x;
        st[cur].mx = x;
        return;
    }
    int mid = (l + r) >> 1;
    pushdown(cur);
    if (a <= mid) update2(lc, l, mid, a, b, x);
    if (b > mid) update2(rc, mid + 1, r, a, b, x);
```

```
    pushup(cur);
}
int query(int cur, int l, int r, int a, int b)
{
    if (a > r or b < l or !st[cur].cnt) return 0;
    if (a <= l and r <= b) return st[cur].cnt + st[cur].scnt;
    int mid = (l + r) >> 1;
    pushdown(cur);
    return query(lc, l, mid, a, b) + query(rc, mid + 1, r, a, b);
}
```

同时取 min/max 的吉司机线段树

再维护一个最小值、次小值、最小值标记即可，注意特判最大值等于最小值的情况。复杂度为 $O(n \log n)$。

求一个点被取最小值的次数

额外维护一个标记 ctag 表示被取最小值的次数, 对于上面情况中的第二种 $(smx < x < mx)$, 直接令 ctag 加一即可, 下推 ctag 时就对于左右儿子中最大值等于该区间原来的最大值的区间把 ctag 传给他们即可。

查询时就一路将 ctag 推到叶子节点处, 输出对应叶子节点的 ctag 即可。

线段树维护区间最值操作与区间历史最值

给出一个长度为 n 的数列 A, 同时定义一个辅助数组 B,B 开始与 A 完全相同。接下来进行了 m 次操作, 操作有五种类型, 按以下格式给出:

1 l r k: 对于所有的 $i \in [l, r]$, 将 A_i 加上 k (k 可以为负数)。

2 l r v: 对于所有的 $i \in [l, r]$, 将 A_i 变成 $\min\{A_i, v\}$。

3 l r: 求 $\sum_{i=l}^{r} A_i$。

4 l r: 对于所有的 $i \in [l, r]$, 求 A_i 的最大值。

5 l r: 对于所有的 $i \in [l, r]$, 求 B_i 的最大值。

在每一次操作后, 我们都进行一次更新, 让 $B_i \leftarrow \max\{B_i, A_i\}$。

```
struct Node
{
    int l, r;
    int sum, maxv, se, cnt, maxh;
    int tag1, tag2, tag3, tag4;
}tr[N << 2];
int n, m, w[N];
void change(int k1, int k2, int k3, int k4, int u)
{
    tr[u].sum += k1 * tr[u].cnt + k2 * (tr[u].r - tr[u].l + 1 - tr[u].cnt);
    tr[u].maxh = max(tr[u].maxh, tr[u].maxv + k3);
    tr[u].maxv += k1;
    if (tr[u].se != -INF) tr[u].se += k2;
    tr[u].tag3 = max(tr[u].tag3, tr[u].tag1 + k3);
    tr[u].tag4 = max(tr[u].tag4, tr[u].tag2 + k4);
    tr[u].tag1 += k1, tr[u].tag2 += k2;
}
void pushdown(int u)
{
    int maxn = max(tr[u << 1].maxv, tr[u << 1 | 1].maxv);
    if (tr[u << 1].maxv == maxn) change(tr[u].tag1, tr[u].tag2, tr[u].tag3, tr[u].tag4, u <<
    ↪  1);
    else change(tr[u].tag2, tr[u].tag2, tr[u].tag4, tr[u].tag4, u << 1);
    if (tr[u << 1 | 1].maxv == maxn) change(tr[u].tag1, tr[u].tag2, tr[u].tag3, tr[u].tag4, u
    ↪  << 1 | 1);
    else change(tr[u].tag2, tr[u].tag2, tr[u].tag4, tr[u].tag4, u << 1 | 1);
    tr[u].tag1 = tr[u].tag2 = tr[u].tag3 = tr[u].tag4 = 0;
}
void pushup(int u)
{
    tr[u].sum = tr[u << 1].sum + tr[u << 1 | 1].sum;
    tr[u].maxv = max(tr[u << 1].maxv, tr[u << 1 | 1].maxv);
    tr[u].maxh = max(tr[u << 1].maxh, tr[u << 1 | 1].maxh);
    if (tr[u << 1].maxv == tr[u << 1 | 1].maxv)
    {
        tr[u].se = max(tr[u << 1].se, tr[u << 1 | 1].se);
        tr[u].cnt = tr[u << 1].cnt + tr[u << 1 | 1].cnt;
    }
    else if (tr[u << 1].maxv > tr[u << 1 | 1].maxv)
    {
        tr[u].se = max(tr[u << 1].se, tr[u << 1 | 1].maxv);
        tr[u].cnt = tr[u << 1].cnt;
    }
```

```
    else
    {
        tr[u].se = max(tr[u << 1].maxv, tr[u << 1 | 1].se);
        tr[u].cnt = tr[u << 1 | 1].cnt;
    }
}
int query_sum(int u, int l, int r)
{
    if (l<= tr[u].l && r >= tr[u].r) return tr[u].sum;
    else
    {
        pushdown(u);
        int mid = (tr[u].l + tr[u].r) >> 1, res = 0;
        if (l <= mid) res += query_sum(u << 1, l, r);
        if (r > mid) res += query_sum(u << 1 | 1, l, r);
        return res;
    }
}
int query_maxv(int u, int l, int r)
{
    if (l<= tr[u].l && r >= tr[u].r) return tr[u].maxv;
    else
    {
        pushdown(u);
        int mid = (tr[u].l + tr[u].r) >> 1, res = -INF;
        if (l <= mid) res = max(res, query_maxv(u << 1, l, r));
        if (r > mid) res = max(res, query_maxv(u << 1 | 1, l, r));
        return res;
    }
}
int query_maxh(int u, int l, int r)
{
    if (l<= tr[u].l && r >= tr[u].r) return tr[u].maxh;
    else
    {
        pushdown(u);
        int mid = (tr[u].l + tr[u].r) >> 1, res = -INF;
        if (l <= mid) res = max(res, query_maxh(u << 1, l, r));
        if (r > mid) res = max(res, query_maxh(u << 1 | 1, l, r));
        return res;
    }
}
void build(int u, int l, int r)
{
    if (l == r) tr[u] = {l, r, w[l], w[l], -INF, 1, w[l]};
    else
    {
        tr[u] = {l, r};
        int mid = (l + r) >> 1;
        build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
        pushup(u);
    }
}
void modify_add(int u, int l, int r, int v)
{
    if (l<= tr[u].l && r >= tr[u].r)
    {
        tr[u].sum += (tr[u].r - tr[u].l + 1) * v;
        tr[u].maxv += v;
        tr[u].maxh = max(tr[u].maxv, tr[u].maxh);
        if (tr[u].se != -INF) tr[u].se += v;
        tr[u].tag1 += v, tr[u].tag2 += v;
        tr[u].tag3 = max(tr[u].tag3, tr[u].tag1);
        tr[u].tag4 = max(tr[u].tag4, tr[u].tag2);
    }
    else
    {
        pushdown(u);
        int mid = (tr[u].l + tr[u].r) >> 1;
        if (l <= mid) modify_add(u << 1, l, r, v);
        if (r > mid) modify_add(u << 1 | 1, l, r, v);
        pushup(u);
    }
}
```

```cpp
void modify_min(int u, int l, int r, int v)
{
    if (v >= tr[u].maxv) return ;
    if (l<= tr[u].l && r >= tr[u].r && tr[u].se < v)
    {
        int k = tr[u].maxv - v;
        tr[u].sum -= tr[u].cnt * k;
        tr[u].maxv = v, tr[u].tag1 -= k;
    }
    else
    {
        pushdown(u);
        int mid = (tr[u].l + tr[u].r) >> 1;
        if (l <= mid) modify_min(u << 1, l, r, v);
        if (r > mid) modify_min(u << 1 | 1, l, r, v);
        pushup(u);
    }
}
void solve()
{
    cin >> n >> m;
    for (int i = 1; i<= n; i++) cin >> w[i];
    build(1, 1, n);
    while (m--)
    {
        int opt, l, r, v;
        cin >> opt >> l >> r;
        if (opt == 1)
        {
            cin >> v;
            modify_add(1, l, r, v);
        }
        else if (opt == 2)
        {
            cin >> v;
            modify_min(1, l, r, v);
        }
        else if (opt == 3) cout << query_sum(1, l, r)<<"\n";
        else if (opt == 4) cout << query_maxv(1, l, r)<<"\n";
        else cout << query_maxh(1, l, r)<<"\n";
    }
}
```

1.8.14 历史线段树

用途：维护区间历史最值，支持区间加法、区间赋值操作。

核心思想：在普通线段树基础上，额外维护历史最大值和对应的历史标记。

关键：区间操作时，先用历史标记更新历史最值，再用当前标记更新当前值。

Q X Y: 询问从 X 到 Y 的最大值

A X Y: 询问从 X 到 Y 的历史最大值

P X Y Z:$[x,y]$ 整体 $+=$z

C X Y Z:$[x,y]$ 整体赋值为 z

```cpp
struct Node
{
    int l, r, maxv, add, cover, hismax, hismaxadd, hismaxcover;
}tr[N << 2];
int w[N];
void pushup(int u)
{
    tr[u].maxv = max(tr[u << 1].maxv, tr[u << 1 | 1].maxv);
    tr[u].hismax = max(tr[u << 1].hismax, tr[u << 1 | 1].hismax);
}
void cal_add(Node & u, int add, int hismaxadd)
{
    if (u.cover == -INF)
    {
        u.hismax = max(u.hismax, u.maxv + hismaxadd);
        u.hismaxadd = max(u.hismaxadd, u.add + hismaxadd);
        u.maxv += add, u.add += add;
    }
    else
```

```
    {
        u.hismax = max(u.hismax, u.maxv + hismaxadd);
        u.hismaxcover = max(u.hismaxcover, u.cover + hismaxadd);
        u.maxv += add, u.cover += add;
    }
}
void cal_cover(Node & u, int cover, int hismaxcover)
{
    u.hismaxcover = max(u.hismaxcover, hismaxcover);
    u.hismax = max(u.hismax, hismaxcover);
    u.maxv = cover, u.cover = cover;
}
void pushdown(int u)
{
    cal_add(tr[u << 1], tr[u].add, tr[u].hismaxadd);
    cal_add(tr[u << 1 | 1], tr[u].add, tr[u].hismaxadd);
    if (tr[u].cover != -INF)
    {
        cal_cover(tr[u << 1], tr[u].cover, tr[u].hismaxcover);
        cal_cover(tr[u << 1 | 1], tr[u].cover, tr[u].hismaxcover);
    }
    tr[u].add = 0, tr[u].cover = -INF, tr[u].hismaxadd = 0, tr[u].hismaxcover = -INF;
}
void build(int u, int l, int r)
{
    if (l == r) tr[u] = {l, r, w[l], 0, -INF, w[l], 0, -INF};
    else
    {
        tr[u] = {l, r, 0, 0, -INF, 0, 0, -INF};
        int mid = (l + r) >> 1;
        build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
        pushup(u);
    }
}
void modify_add(int u, int l, int r, int v)
{
    if (l<= tr[u].l && r >= tr[u].r) cal_add(tr[u], v, v);
    else
    {
        pushdown(u);
        int mid = (tr[u].l + tr[u].r) >> 1;
        if (l <= mid) modify_add(u << 1, l, r, v);
        if (r > mid) modify_add(u << 1 | 1, l, r, v);
        pushup(u);
    }
}
void modify_cover(int u, int l, int r, int v)
{
    if (l<= tr[u].l && r >= tr[u].r) cal_cover(tr[u], v, v);
    else
    {
        pushdown(u);
        int mid = (tr[u].l + tr[u].r) >> 1;
        if (l <= mid) modify_cover(u << 1, l, r, v);
        if (r > mid) modify_cover(u << 1 | 1, l, r, v);
        pushup(u);
    }
}
int query_max(int u, int l, int r)
{
    if (l<= tr[u].l && r >= tr[u].r) return tr[u].maxv;
    else
    {
        pushdown(u);
        int mid = (tr[u].l + tr[u].r) >> 1, res = -INF;
        if (l <= mid) res = max(res, query_max(u << 1, l, r));
        if (r > mid) res = max(res, query_max(u << 1 | 1, l, r));
        return res;
    }
}
int query_hismax(int u, int l, int r)
{
    if (l<= tr[u].l && r >= tr[u].r) return tr[u].hismax;
    else
```

```
    {
        pushdown(u);
        int mid = (tr[u].l + tr[u].r) >> 1, res = -INF;
        if (l <= mid) res = max(res, query_hismax(u << 1, l, r));
        if (r > mid) res = max(res, query_hismax(u << 1 | 1, l, r));
        return res;
    }
}
void solve()
{
    build(1, 1, n);
    while (m--)
    {
        char opt;
        int x, y, z;
        cin >> opt >> x >> y;
        if (opt=='Q') cout << query_max(1, x, y)<<"\n";
        else if (opt=='A') cout << query_hismax(1, x, y)<<"\n";
        else if (opt=='P')
        {
            cin >> z;
            modify_add(1, x, y, z);
        }
        else
        {
            cin >> z;
            modify_cover(1, x, y, z);
        }
    }
}
```

历史和问题和历史最值问题的求解思路类似, 需要在标记里多维护一个求历史和轮数。

考虑区间加区间历史和。信息维护区间长度 len, 区间和 s 以及区间历史和 h, 标记维护每个位置加的值 ds。s 通过 ds 和 len 维护(区间加区间和基本操作), 考虑 h。

设每次求历史和的时候加的值分别为 $ds_1, \ldots, ds_t$, 则历史和新增

$$\sum_{i=1}^{t}(s + len \cdot ds_i) = s \cdot t + len \cdot \sum_{i=1}^{t} ds_i$$

于是标记维护每个位置加的值的历史和 $hs = \sum_{i=1}^{t} ds_i$, 以及求历史和轮数 t。一般会让 t 单独加 1 表示求一轮历史和。标记作用在 h 上:$h \leftarrow h + s \cdot t + len \cdot hs$。

根据实际意义合并标记:ds 和 t 简单相加,$hs = hs_l + ds \cdot t + hs_r$。

矩阵乘法与历史和

设区间长度 len, 区间和 s, 区间历史和 h。整体加 v 是

$$s \leftarrow s + v \cdot len,$$

求历史和是

$$h \leftarrow h + s,$$

都是 len, s, h 的线性组合。将一些数的每个数同时变成所有数的线性组合, 想到向量和矩阵乘法。

将信息写成行向量

$$\begin{bmatrix} len & s & h \end{bmatrix},$$

则整体加 v 等价于右乘矩阵

$$\begin{bmatrix} len & s & h \end{bmatrix} \begin{bmatrix} 1 & v & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} len & s + v \cdot len & h \end{bmatrix},$$

求 t 次历史和等价于

$$\begin{bmatrix} len & s & h \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & t \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} len & s & h + t \cdot s \end{bmatrix}.$$

矩阵乘法满足结合律, 可以用线段树维护区间向量乘矩阵、区间向量和。模拟矩阵乘法可知任何标记的主对角线元素为 1, 且下三角(左下角)为 0。标记形如

$$\begin{bmatrix} 1 & ds & hs \\ 0 & 1 & t \\ 0 & 0 & 1 \end{bmatrix},$$

与上面 $h \leftarrow h + t \cdot s + hs \cdot len$ 的线性组合式一致，可知两者本质相同。矩阵法的思维难度较低，但常数大，因为进行了一些无用乘法。变量法给矩阵的每个位置赋予实际含义，抓住了矩阵主对角线及下方恒定的性质。从实际意义理解：区间长度恒定，和贡献至历史和，而历史和不会贡献至和。

例题

用以下方式定义函数 $g(i,j)$：

当 $i \leqslant j$ 时，$g(i,j)$ 是满足

$$\{a_p : i \leqslant p \leqslant j\} \subseteq \{a_q : x \leqslant q \leqslant j\}$$

的最大整数 x；否则 $g(i,j) = 0$。

q 次询问，每次给定 l, r, x, y，请求出

$$\sum_{i=l}^{r} \sum_{j=x}^{y} g(i,j)$$

的值。

扫描线，维护每个颜色最右端的一个出现位置 cur，一个区间的答案就是区间内最左端的 cur，所以每一个以 cur 为答案的都是一段连续的区间。考虑用 set 维护颜色段，每次删除一个元素并且添加一个元素，同时修改一段区间的端点下标即可，可以转化为区间加。

还是维护历史版本和，这很简单了，维护向量

$$\begin{bmatrix} \text{pos} & \text{sum} & \text{len} \end{bmatrix},$$

表示现在的颜色段右端点、历史右端点的版本和以及区间长度。

修改操作乘以矩阵

$$\begin{bmatrix} 1 & 0 & 00 & 1 & 0k & 0 & 1 \end{bmatrix},$$

更新版本和乘以矩阵

$$\begin{bmatrix} 1 & 1 & 00 & 1 & 00 & 0 & 1 \end{bmatrix},$$

还是找出有用的地方，维护对应的位置即可。

打表一下容易发现只有三个有效的位置，我们只用维护三个位置。

矩阵乘法：

$$\begin{bmatrix} 1 & a & 00 & 1 & 0b & c & 1 \end{bmatrix} \times \begin{bmatrix} 1 & A & 00 & 1 & 0B & C & 1 \end{bmatrix} = \begin{bmatrix} 1 & A+a & 00 & 1 & 0b+B & Ab+c+C & 1 \end{bmatrix}$$

向量乘矩阵：

$$\begin{bmatrix} \text{pos} & \text{sum} & \text{len} \end{bmatrix} \times \begin{bmatrix} 1 & A & 00 & 1 & 0B & C & 1 \end{bmatrix} = \begin{bmatrix} \text{pos} + B \cdot \text{len} & A \cdot \text{pos} + \text{sum} + C \cdot \text{len} & \text{len} \end{bmatrix}$$

注意到时间维也是可以差分的，询问可以处理一下变成差分的形式，就很简单了。

```cpp
vector<array<int, 3>> Q[N];
int ans[N], w[N];
struct Node
{
    int l, r;
    Matrix<1, 3> sum;
    Matrix<3, 3> tag;
    bool rt;
}tr[3300000];
void pushup(int u) { tr[u].sum = tr[u << 1].sum + tr[u << 1 | 1].sum; }
void build(int u, int l, int r)
{
    tr[u] = {l, r};
    tr[u].tag = E, tr[u].rt = false;
    if (l == r)
    {
        tr[u].sum(0, 0) = 0, tr[u].sum(0, 1) = 0, tr[u].sum(0, 2) = 1;
        return ;
    }
    int mid = (l + r) >> 1;
    build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
    pushup(u);
}
void apply(Node & u, Matrix<3, 3>& ma)
{
    u.sum = u.sum * ma, u.tag = u.tag * ma, u.rt = true;
}
void pushdown(int u)
{
    if (tr[u].rt)
    {
        tr[u].rt = false;
        apply(tr[u << 1], tr[u].tag), apply(tr[u << 1 | 1], tr[u].tag);
        tr[u].tag = E;
```

```
    }
}
void modify(int u, int l, int r, Matrix<3, 3>& v)
{
    if (l <= tr[u].l && r >= tr[u].r) apply(tr[u], v);
    else
    {
        pushdown(u);
        int mid = (tr[u].l + tr[u].r) >> 1;
        if (l <= mid) modify(u << 1, l, r, v);
        if (r > mid) modify(u << 1 | 1, l, r, v);
        pushup(u);
    }
}
int query(int u, int l, int r)
{
    if (l<= tr[u].l && r >= tr[u].r) return tr[u].sum(0, 1);
    else
    {
        pushdown(u);
        int mid = (tr[u].l + tr[u].r) >> 1, res = 0;
        if (l <= mid) res += query(u << 1, l, r);
        if (r > mid) res += query(u << 1 | 1, l, r);
        return res;
    }
}
int pos[N];
void solve()
{
    E(0, 0) = E(1, 1) = E(2, 2) = 1;
    Matrix<3, 3> P;
    P(0, 0) = 1, P(0, 1) = 1, P(0, 2) = 0;
    P(1, 0) = 0, P(1, 1) = 1, P(1, 2) = 0;
    P(2, 0) = 0, P(2, 1) = 0, P(2, 2) = 1;
    int n, q;
    for (int i = 1; i <= n; i++) read(w[i]);
    for (int i = 1, l, r, x, y; i <= q; i++)
    {
        read(l), read(r), read(x), read(y);
        Q[y].push_back({i, l, r});
        if (x - 1) Q[x - 1].push_back({-i, l, r});
    }
    build(1, 1, n);
    set<int> S;
    S.insert(0);
    Matrix<3, 3> tra;
    tra(0, 0) = 1, tra(0, 1) = 0, tra(0, 2) = 0;
    tra(1, 0) = 0, tra(1, 1) = 1, tra(1, 2) = 0;
    tra(2, 0) = 0, tra(2, 1) = 0, tra(2, 2) = 1;
    for (int i = 1; i <= n; i++)
    {
        int lst = pos[w[i]];
        S.insert(i);
        tra(2, 0) = i;
        modify(1, i, i, tra);
        if (lst)
        {
            S.erase(lst);
            auto it = S.upper_bound(lst);
            int nxt = *it;
            it--;
            int fr = (*it) + 1;
            tra(2, 0) = nxt - lst;
            modify(1, fr, lst, tra);
        }
        modify(1, 1, i, P);
        pos[w[i]] = i;
        for (auto & [sig, l, r]:Q[i])
            if (sig >0) ans[sig] += query(1, l, r);
            else ans[-sig] -= query(1, l, r);
    }
    for (int i = 1; i <= q; i++) write(ans[i]), putchar('\n');
}
```

1.9 平衡树

1.9.1 Splay

以翻转一个正序排列为例

```
struct Node
{
    int s[2], p, v, sz, tag;
    void init(int _v, int _p) { v = _v, p = _p, sz = 1; }
}tr[N];
int root, idx, n, m;
void pushup(int p) { tr[p].sz = tr[tr[p].s[0]].sz + tr[tr[p].s[1]].sz + 1; }
void pushdown(int p)
{
    if (tr[p].tag)
    {
        swap(tr[p].s[0], tr[p].s[1]);
        tr[tr[p].s[0]].tag ^= 1, tr[tr[p].s[1]].tag ^= 1;
        tr[p].tag = 0;
    }
}
void rotate(int x)
{
    int y = tr[x].p, z = tr[y].p;
    int k = tr[y].s[1] == x;
    tr[z].s[tr[z].s[1] == y] = x, tr[x].p = z;
    tr[y].s[k] = tr[x].s[k ^ 1], tr[tr[x].s[k ^ 1]].p = y;
    tr[x].s[k ^ 1] = y, tr[y].p = x;
    pushup(y), pushup(x);
}
void splay(int x, int k)
{
    while (tr[x].p != k)
    {
        int y = tr[x].p, z = tr[y].p;
        if (z != k)
        {
            if ((tr[y].s[0] == x) ^ (tr[z].s[0] == y)) rotate(x);
            else rotate(y);
        }
        rotate(x);
    }
    if (!k) root = x;
}
void output(int p)
{
    pushdown(p);
    if (tr[p].s[0]) output(tr[p].s[0]);
    if (tr[p].v >= 1 && tr[p].v <= n) cout << tr[p].v<<" ";
    if (tr[p].s[1]) output(tr[p].s[1]);
}
void insert(int v)
{
    int u = root, p = 0;
    while (u) p = u, u = tr[u].s[v > tr[u].v];
    u = ++idx;
    if (p) tr[p].s[v > tr[p].v] = u;
    tr[u].init(v, p);
    splay(u, 0);
}
int get_k(int k)
{
    int u = root;
    while (true)
    {
        pushdown(u); //一定要 pushdown
        if (tr[tr[u].s[0]].sz >= k) u = tr[u].s[0];
        else if (tr[tr[u].s[0]].sz + 1 == k) return u;
        else k -= tr[tr[u].s[0]].sz + 1, u = tr[u].s[1];
    }
    return - 1;
}
int main()
```

```
{
    for (int i = 0; i <= n + 1; i++) insert(i); // 0 和 n + 1 为两个哨兵
    while (m--)
    {
        int l, r;
        l = get_k(l), r = get_k(r + 2); //翻转 l, r 要找到 l - 1 和 r + 1 由于哨兵的存在 故要找到 l 和 r + 2
        splay(l, 0), splay(r, l);
        tr[tr[r].s[0]].tag ^= 1;
    }
    output(root); //中序遍历输出
}
```

1.9.2 FHQ_Treap

用途：可持久化平衡树，支持分裂、合并操作，常用于区间操作。

核心操作：split 按值或排名分裂，merge 合并两棵子树。

优势：代码简洁，支持可持久化，分裂合并灵活。

使用模板：插入 x：split (root, x-1, T1, T2); root=merge (T1, merge (newnode (x), T2))

维护集合一般使用按值分裂

```
mt19937 rd(time(0));
struct FHQ_Treap
{
    int l, r, sz;
    int key, val; //rand_key value
}fhq[N];
int root, T1, T2, T3, tot;
int newnode(int v)
{
    fhq[++tot] = {0, 0, 1, rd(), v};
    return tot;
}
void pushup(int p) { fhq[p].sz = fhq[fhq[p].l].sz + 1 + fhq[fhq[p].r].sz; }
void split(int p, int v, int & x, int & y)
{//按值分裂 <= v 的分裂到 x > v 的分裂到 y
    if (!p) x = y = 0;
    else
    {
        if (fhq[p].val > v) y = p, split(fhq[p].l, v, x, fhq[y].l);
        else x = p, split(fhq[p].r, v, fhq[x].r, y);
        pushup(p);
    }
}
int merge(int x, int y)
{ // 合并，一定要保证 x 中的都要小于等于 y
    if (!x||!y) return x | y;
    if (fhq[x].key > fhq[y].key)
    {
        fhq[x].r = merge(fhq[x].r, y);
        pushup(x);
        return x;
    }
    else
    {
        fhq[y].l = merge(x, fhq[y].l);
        pushup(y);
        return y;
    }
}
void insert(int x)//插入一个值 按值排序
{
    split(root, x, T1, T2);
    root = merge(merge(T1, newnode(x)), T2);
}
void del(int x)//删去一个值
{
    split(root, x, T1, T2), split(T1, x - 1, T1, T3);
    T3 = merge(fhq[T3].l, fhq[T3].r);
    root = merge(merge(T1, T3), T2);
}
int find_rk(int x)//查一个值的排名
{
```

```
    split(root, x - 1, T1, T2);
    int res = fhq[T1].sz + 1;
    root = merge(T1, T2);
    return res;
}
int kth(int k)//查第 k 大的值
{
    int p = root;
    while (p)
    {
        int cnt = fhq[fhq[p].l].sz + 1;
        if (cnt == k) break;
        else if (k < cnt) p = fhq[p].l;
        else p = fhq[p].r, k -= cnt;
    }
    return fhq[p].val;
}
int find_pre(int p, int x)//在 root 里找 x 的前驱
{
    if (!p) return - INF;
    if (fhq[p].val < x)
    {
        int res = find_pre(fhq[p].r, x);
        return res == -INF?fhq[p].val:res;
    }
    else return find_pre(fhq[p].l, x);
}
int find_next(int p, int x)//在 root 里找 x 的后继
{
    if (!p) return INF;
    if (fhq[p].val > x)
    {
        int res = find_next(fhq[p].l, x);
        return res == INF?fhq[p].val:res;
    }
    else return find_next(fhq[p].r, x);
}
```

给定 k 个数，再给定 n 次减法，每次减法都给你一个数 c，对于 $\geq c$ 的数都减去 c，求每个数被减了多少次。

考虑按值分裂维护，每次分成三个部分，$[0,c-1],[c,2c],[2c+1,\infty]$，对于第一部分不用管，对于第三部分打一个减法标记和次数标记即可，此时第三部分依然全部大于第一部分，可以合并。对第二部分进行暴力重构后插入平衡树。对于第二部分中的数，每个数至少减少为原来的一半，所以每个数最多重构 $O(\log n)$ 次。

```
struct FHQ_Treap
{
    int l, r;
    int key, id, val, cnt, tagc, tagv;
}fhq[N * 60];
int root, T1, T2, T3, tot, n, k, ans[N];
int newnode(int id, int v, int cnt)
{
    fhq[++tot] = {0, 0, rd(), id, v, cnt, 0, 0};
    return tot;
}
void pushdown(int p)
{
    if (!p) return ;
    if (fhq[p].tagc)
    {
        fhq[fhq[p].l].cnt += fhq[p].tagc;
        fhq[fhq[p].r].cnt += fhq[p].tagc;
        fhq[fhq[p].l].tagc += fhq[p].tagc;
        fhq[fhq[p].r].tagc += fhq[p].tagc;
        fhq[p].tagc = 0;
    }
    if (fhq[p].tagv)
    {
        fhq[fhq[p].l].val += fhq[p].tagv;
        fhq[fhq[p].r].val += fhq[p].tagv;
        fhq[fhq[p].l].tagv += fhq[p].tagv;
        fhq[fhq[p].r].tagv += fhq[p].tagv;
        fhq[p].tagv = 0;
    }
```

```
}
void split(int p, int v, int & x, int & y)
{
    if (!p) x = y = 0;
    else
    {
        pushdown(p);
        if (fhq[p].val > v) y = p, split(fhq[p].l, v, x, fhq[y].l);
        else x = p, split(fhq[p].r, v, fhq[x].r, y);
    }
}
int merge(int x, int y)
{
    if (!x||!y) return x | y;
    pushdown(x), pushdown(y);
    if (fhq[x].key > fhq[y].key)
    {
        fhq[x].r = merge(fhq[x].r, y);
        return x;
    }
    else
    {
        fhq[y].l = merge(x, fhq[y].l);
        return y;
    }
}
void insert(int id, int x, int cnt)
{
    split(root, x, T1, T2);
    root = merge(merge(T1, newnode(id, x, cnt)), T2);
}
void rebuild(int p, int c)
{
    if (!p) return ;
    pushdown(p);
    rebuild(fhq[p].l, c);
    insert(fhq[p].id, fhq[p].val - c, fhq[p].cnt + 1);
    rebuild(fhq[p].r, c);
}
void output(int p)
{
    if (!p) return ;
    pushdown(p);
    output(fhq[p].l);
    ans[fhq[p].id] = fhq[p].cnt;
    output(fhq[p].r);
}
void solve()
{
    for (int i = 1, x; i <= k; i++)
    {
        cin >> x;
        insert(i, x, 0);
    }
    for (int i = 1; i <= n; i++)
    {
        int c = w[i];
        split(root, c - 1, T1, T2);
        split(T2, 2 * c - 1, T2, T3);
        fhq[T3].cnt++, fhq[T3].tagc++;
        fhq[T3].val -= c, fhq[T3].tagv -= c;
        root = merge(T1, T3);
        rebuild(T2, c);
    }
    output(root);
    for (int i = 1; i <= k; i++) cout << ans[i]<<" ";
}
```

维护序列一般使用按排名分裂

```
//只用改变分裂的方式 现在是按排名分裂
void split_rk(int p, int k, int & x, int & y)
{
```

```
    if (!p) x = y = 0;
    else
    {
        int cnt = fhq[fhq[p].l].sz + 1;
        if (cnt == k) x = p, y = fhq[p].r, fhq[p].r = 0;
        else if (k < cnt) y = p, split_rk(fhq[p].l, k, x, fhq[y].l);
        else x = p, split_rk(fhq[p].r, k - cnt, fhq[x].r, y);
        pushup(p);
    }
}
// 如果想知道编号 x 在现在平衡树中的排名，需要维护 fa，只有 split 和 merge 要改变
struct FHQ_Treap
{
    int l, r, sz;
    int key, val, fa;
}fhq[N];
void split_rk(int p, int k, int & x, int & y, int fax = 0, int fay = 0)
{
    if (!p) x = y = 0;
    else
    {
        int cnt = fhq[fhq[p].l].sz + 1;
        if (cnt == k) fhq[p].fa = fax, fhq[fhq[p].r].fa = fay, x = p, y = fhq[p].r, fhq[p].r =
        ↪  0;
        else if (k < cnt) fhq[p].fa = fay, y = p, split_rk(fhq[p].l, k, x, fhq[y].l, fax, y);
        else fhq[p].fa = fax, x = p, split_rk(fhq[p].r, k - cnt, fhq[x].r, y, x, fay);
        pushup(p);
    }
}
int merge(int x, int y)
{
    if (!x||!y) return x | y;
    if (fhq[x].key > fhq[y].key)
    {
        fhq[x].r = merge(fhq[x].r, y);
        fhq[fhq[x].r].fa = x;
        pushup(x);
        return x;
    }
    else
    {
        fhq[y].l = merge(x, fhq[y].l);
        fhq[fhq[y].l].fa = y;
        pushup(y);
        return y;
    }
}
//找到编号 p 现在的排名
int find(int p)
{
    int res = fhq[fhq[p].l].sz + 1;
    while (p)
    {
        if (fhq[fhq[p].fa].r == p) res += fhq[fhq[fhq[p].fa].l].sz + 1;
        p = fhq[p].fa;
    }
    return res;
}
```

1. 将 $[L,R]$ 这个区间内的所有数加上 V。
2. 将 $[L,R]$ 这个区间翻转, 比如 1 2 3 4 变成 4 3 2 1。
3. 求 $[L,R]$ 这个区间中的最大值。

```
struct FHQ_Treap
{
    int l, r, sz;
    int key, val, maxv, add, rev;
}fhq[N];
int n, m, root, T1, T2, T3, tot;
int newnode(int v)
{
    fhq[++tot] = {0, 0, 1, rd(), v, v};
    return tot;
```

```
}
void pushup(int p)
{
    fhq[p].sz = fhq[fhq[p].l].sz + 1 + fhq[fhq[p].r].sz;
    fhq[p].maxv = max({fhq[fhq[p].l].maxv, fhq[p].val, fhq[fhq[p].r].maxv});
}
void add(int p, int v)
{
    if (!p) return ;
    fhq[p].val += v, fhq[p].maxv += v, fhq[p].add += v;
}
void rev(int p)
{
    if (!p) return ;
    swap(fhq[p].l, fhq[p].r);
    fhq[p].rev ^= 1;
}
void pushdown(int p)
{
    if (fhq[p].add) add(fhq[p].l, fhq[p].add), add(fhq[p].r, fhq[p].add), fhq[p].add = 0;
    if (fhq[p].rev) rev(fhq[p].l), rev(fhq[p].r), fhq[p].rev = 0;
}
void split_rk(int p, int k, int & x, int & y)
{
    if (!p) x = y = 0;
    else
    {
        pushdown(p);
        int cnt = fhq[fhq[p].l].sz + 1;
        if (cnt == k) x = p, y = fhq[p].r, fhq[p].r = 0;
        else if (k < cnt) y = p, split_rk(fhq[p].l, k, x, fhq[y].l);
        else x = p, split_rk(fhq[p].r, k - cnt, fhq[x].r, y);
        pushup(p);
    }
}
int merge(int x, int y)
{
    if (!x||!y) return x | y;
    pushdown(x), pushdown(y);
    if (fhq[x].key > fhq[y].key)
    {
        fhq[x].r = merge(fhq[x].r, y);
        pushup(x);
        return x;
    }
    else
    {
        fhq[y].l = merge(x, fhq[y].l);
        pushup(y);
        return y;
    }
}
void solve()
{
    cin >> n >> m;
    fhq[0].maxv = -INF;
    for (int i = 1; i <= n; i++) root = merge(root, newnode(0));
    while (m--)
    {
        cin >> t >> l >> r;
        split_rk(root, r, T1, T3);
        split_rk(T1, l - 1, T1, T2);
        if (t == 1) add(T2, v);
        else if (t == 2) rev(T2);
        else cout << fhq[T2].maxv<<"\n";
        root = merge(merge(T1, T2), T3);
    }
}
```

1. I pos val 插入一个数字在第 pos 个位置之前，如果 $pos = current_{length}$，那么你需要将这个数字放到序列末尾
2. D pos 删除第 pos 个元素
3. R pos val 将第 pos 个元素变为 val

4. Q l r k 询问 $(\sum_{i=l}^{r} A[i] \times (i-l+1)^k)$, 保证 $0 \leqslant k \leqslant 10$

考虑平衡树上的合并需要合并 $[l, mid-1], [mid, mid], [mid+1, r]$ 三个区间

$$\sum_{i=l}^{r} a[i] \times (i-l+1)^k = \sum_{i=l}^{mid-1} a[i] \times (i-l+1)^k + a[mid] \times (i-mid+1)^k + \sum_{i=mid+1}^{r} a[i] \times (i-l+1)^k$$

$$= \text{ans}_{l,k} + a[mid] \times (i-mid+1)^k + \sum_{i=mid+1}^{r} a[i] \times (i-l+1)^k$$

前两部分是好处理的，考虑如何计算第三部分，可以利用二项式定理展开

$$\sum_{i=mid+1}^{r} a[i] \times (i-l+1)^k = \sum_{i=mid+1}^{r} a[i] \times \sum_{j=0}^{k} \binom{k}{j} (i-mid)^j (mid-l+1)^{k-j}$$

$$= \sum_{j=0}^{k} \sum_{i=mid+1}^{r} (i-mid)^j a[i] \binom{k}{j} (mid-l+1)^{k-j} = \sum_{j=0}^{k} \text{ans}_{r,j} \binom{k}{j} (mid-l+1)^{k-j}$$

```cpp
struct FHQ_Treap
{
    int l, r, sz;
    int key, val;
    array<int, M> v;
}fhq[N];
int root, T1, T2, T3, tot, C[M][M];
int newnode(int v)
{
    fhq[++tot] = {0, 0, 1, rd(), v};
    for (int i = 0; i < M; i++) fhq[tot].v[i] = v;
    return tot;
}
void pushup(int p)
{
    fhq[p].sz = fhq[fhq[p].l].sz + 1 + fhq[fhq[p].r].sz;
    int po[M];
    po[0] = 1;
    for (int i = 1; i < M; i++) po[i] = po[i - 1] * (int)(fhq[fhq[p].l].sz + 1);
    for (int i = 0; i < M; i++)
    {
        fhq[p].v[i] = fhq[fhq[p].l].v[i] + fhq[p].val * po[i];
        for (int j = 0; j <= i; j++) fhq[p].v[i] += C[i][j] * po[i - j] * fhq[fhq[p].r].v[j];
    }
}
void split_rk(int p, int k, int & x, int & y)
{
    if (!p) x = y = 0;
    else
    {
        int cnt = fhq[fhq[p].l].sz + 1;
        if (cnt == k) x = p, y = fhq[p].r, fhq[p].r = 0;
        else if (k < cnt) y = p, split_rk(fhq[p].l, k, x, fhq[y].l);
        else x = p, split_rk(fhq[p].r, k - cnt, fhq[x].r, y);
        pushup(p);
    }
}
int merge(int x, int y)
{
    if (!x || !y) return x | y;
    if (fhq[x].key > fhq[y].key)
    {
        fhq[x].r = merge(fhq[x].r, y);
        pushup(x);
        return x;
    }
    else
    {
        fhq[y].l = merge(x, fhq[y].l);
        pushup(y);
        return y;
    }
}
void solve()
{
    for (int i = 1, x; i <= n; i++)
    {
        cin >> x;
        root = merge(root, newnode(x));
    }
    while (q--)
```

```
    {
        if (opt=='I')
        {
            int pos, val;
            split_rk(root, pos - 1, T1, T2);
            root = merge(merge(T1, newnode(val)), T2);
        }
        else if (opt=='D')
        {
            int pos;
            split_rk(root, pos - 1, T1, T2);
            split_rk(T2, 1, T2, T3);
            root = merge(T1, T3);
        }
        else if (opt=='R')
        {
            int pos, val;
            split_rk(root, pos - 1, T1, T2);
            split_rk(T2, 1, T2, T3);
            root = merge(merge(T1, newnode(val)), T3);
        }
        else
        {
            int l, r, k;
            split_rk(root, r, T1, T2);
            split_rk(T1, l - 1, T1, T3);
            cout << fhq[T3].v[k]<<"\n";
            root = merge(merge(T1, T3), T2);
        }
    }
}
```

1.10 Link-Cut Tree (LCT)

关键操作：

- split (x, y): 提取 x 到 y 路径，y 成为路径根，路径信息在 tr[y] 中
- link (x, y): 连边（自动判断是否已连通）
- cut (x, y): 断边
- connected (x, y): 判断连通性
- path_sum (x, y): 查询路径和

处理边权：每条边 (u, v) 变成点 w，连 (u, w) 和 (w, v)，边权存在 w 上。

LCT 处理点权比较方便，处理边权较为麻烦。而动态森林的变化一般依赖于边的变化，所以可以利用边转点的技巧，将边变为一个新点，原边变成两条边即可。

```
struct LCT
{
    struct Node
    {
        int s[2], fa, rev;
        int v, sum; //维护的信息
    }tr[N];
    int stk[N], top;
    void pushup(int u)
    { // 注意 pushup 的时候会不会取到空，处理方法是提前赋值或者特判
        tr[u].sum = tr[tr[u].s[0]].sum ^ tr[u].v ^ tr[tr[u].s[1]].sum;
    }
    void pushrev(int u) { swap(tr[u].s[0], tr[u].s[1]), tr[u].rev ^= 1; }
    void pushdown(int u)
    {
        if (tr[u].rev) pushrev(tr[u].s[0]), pushrev(tr[u].s[1]), tr[u].rev = 0;
    }
    bool isr(int u) { return tr[tr[u].fa].s[0] != u && tr[tr[u].fa].s[1] != u; }
    void rotate(int x)
    {
        int y = tr[x].fa, z = tr[y].fa, k = tr[y].s[1] == x;
        if (!isr(y)) tr[z].s[tr[z].s[1] == y] = x;
        tr[x].fa = z, tr[y].s[k] = tr[x].s[k ^ 1];
        tr[tr[x].s[k ^ 1]].fa = y, tr[x].s[k ^ 1] = y, tr[y].fa = x;
        pushup(y), pushup(x);
```

```
    }
    void splay(int x)//把 x 旋转到 splay 的根
    {
        int r = x; stk[top = 1] = x;
        while (!isr(r)) stk[++top] = r = tr[r].fa;
        while (top) pushdown(stk[top--]);
        while (!isr(x))
        {
            int y = tr[x].fa, z = tr[y].fa;
            if (!isr(y))
                if ((tr[y].s[1] == x) ^ (tr[z].s[1] == y)) rotate(x);
                else rotate(y);
            rotate(x);
        }
    }
    void access(int x)//在原树上, 建立一条根到 x 的实链
    {
        int z = x;
        for (int y = 0; x; y = x, x = tr[x].fa) splay(x), tr[x].s[1] = y, pushup(x);
        splay(z);
    } // 在原树上, 把 x 变为新根
    void maker(int x) { access(x), pushrev(x); }
    int findr(int x)//在原树上寻找根 可以用来判断两点的连通性
    {
        access(x);
        while (tr[x].s[0]) pushdown(x), x = tr[x].s[0];
        splay(x); return x;
    } // 在原树上创建一条从 x 到 y 的实边路径, 并把 y 变为根节点 所以 split 之后 y 是根 路径的信息都在 y 上
    void split(int x, int y) { maker(x), access(y); }
    void link(int x, int y) { maker(x); if (findr(y) != x) tr[x].fa = y; }
    void cut(int x, int y)
    {
        maker(x);
        if (findr(y) == x && tr[y].fa == x&&!tr[y].s[0])
            tr[x].s[1] = tr[y].fa = 0, pushup(x);
    }

    // 常用应用函数
    bool connected(int x, int y) { return findr(x) == findr(y); } // 判断连通性
    int path_sum(int x, int y) { split(x, y); return tr[y].sum; } // 路径和
    void path_update(int x, int y, int val) { split(x, y); tr[y].v += val; pushup(y); } // 路
    ↪ 径更新
    int get_root(int x) { return findr(x); } // 获取 x 所在树的根
}lct;

```

LCT 维护动态连通图

你有一个无向图 $G(V,E)$, E 中每一个元素用一个二元组 (u,v) 表示。

现在把 E 中的元素排成一个长度为 $|E|$ 序列 A。

然后给你 q 个询问二元组 (l,r),

表示询问图 $G'(V, \bigcup\limits_{i\in[l,r]}\{A_i\})$ 的联通块的个数, 强制在线。

考虑扫描线, 对于每个 r, 维护当前所有 l 的情况, 并尽可能使得当前存在的边尽量靠右

当加入新边时, 若不不成环, 直接加上。若成环, 将环上最靠右的边断掉, 再加入新边

可持久化把树状数组换为主席树即可

```
struct LCT
{
    struct Node
    {
        int s[2], fa, rev, v, minv;
    }tr[N];
    void pushup(int u) { tr[u].minv = min({tr[tr[u].s[0]].minv, tr[u].v,
    ↪ tr[tr[u].s[1]].minv}); }
}lct;
int n, m, q, t;
array<int, 2> E[N];
void solve()
{
    lct.tr[0].v = lct.tr[0].minv = INF;
    cin >> n >> m >> q >> t;
```

```
    for (int i = 1; i <= n; i++) lct.tr[i].v = lct.tr[i].minv = INF;
    for (int i = 1, a, b; i <= m; i++)
    {
        cin >> a >> b;
        E[i] = {a, b}, root[i] = root[i - 1];
        if (a == b) continue;
        if (lct.findr(a) == lct.findr(b))
        {
            lct.split(a, b);
            auto tim = lct.tr[b].minv;
            auto [c, d] = E[tim];
            lct.cut(c, tim + n), lct.cut(d, tim + n);
            root[i] = insert(root[i], 1, m, tim, -1);
        }
        lct.tr[i + n].minv = lct.tr[i + n].v = i;
        root[i] = insert(root[i], 1, m, i, 1);
        lct.link(a, i + n), lct.link(b, i + n);
    }
    int ans = 0;
    while (q--)
    {
        int l, r;
        cin >> l >> r;
        if (t >0) l = (l + ans) % m + 1, r = (r + ans) % m + 1;
        if (l > r) swap(l, r);
        cout << (ans = n - query(root[r], 1, m, l, r))<<"\n";
    }
}
```

最小差值生成树

同理将所有边排序后, 从小到大加入, 维护连通性的时候, 优先断掉边权小的边

```
void solve()
{
    cin >> n >> m;
    for (int i = 1; i <= n; i++) lct.tr[i].v = lct.tr[i].minv = INF;
    vector<array<int, 3>> edge;
    while (m--)
    {
        int a, b, c;
        cin >> a >> b >> c;
        edge.push_back({c, a, b});
    }
    sort(edge.begin(), edge.end());
    int ans = INF;
    multiset<int> s;
    for (int i = 0; i < edge.size(); i++)
    {
        lct.tr[i + 1 + n].minv = lct.tr[i + 1 + n].v = i;
        auto [c, a, b] = edge[i];
        if (lct.findr(a) == lct.findr(b))
        {
            lct.split(a, b);
            int id = lct.tr[b].minv;
            if (id > edge.size()) continue;
            auto [C, A, B] = edge[id];
            lct.cut(A, id + 1 + n), lct.cut(B, id + 1 + n);
            s.extract(C);
        }
        lct.link(a, i + 1 + n), lct.link(b, i + 1 + n);
        s.insert(c);
        if (lct.ee == 2 * n - 2) ans = min(ans, c-*s.begin());
    }
    cout << ans << "\n";
}
```

lct 维护森林路径信息

有 N 个洞, 每个洞有相应的弹力, 能把这个球弹到 $i + power[i]$ 位置。当球的位置 $> N$ 时即视为被弹出

共有两种操作:

0 a b 把 a 位置的弹力改成 b

1 a 在 a 处放一个球, 输出球被弹出前共被弹了多少次

```
struct LCT
{
    struct Node
    {
        int s[2], fa, rev, sz;
    }tr[N];
    int stk[N], top;
    void pushup(int u) { tr[u].sz = tr[tr[u].s[0]].sz + 1 + tr[tr[u].s[1]].sz; }
}lct;
int n, m, ne[N];
void solve()
{
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
    {
        cin >> ne[i];
        lct.link(i, min(i + ne[i], n + 1));
    }
    while (m--)
    {
        int t, a, b;
        cin >> t >> a;
        if (t == 0)
        {
            cin >> b;
            lct.cut(a, min(a + ne[a], n + 1));
            ne[a] = b;
            lct.link(a, min(a + ne[a], n + 1));
        }
        else
        {
            lct.split(a, n + 1);
            cout << lct.tr[n + 1].sz - 1<<"\n";
        }
    }
}
```

LCT 求 LCA

有一个森林最初由 $n(1 \leqslant n \leqslant 100000)$ 个互不相连的点构成

- link A B: 添加从顶点 A 到 B 的边, 使 A 成为 B 的子节点, 其中保证 A 是一个根顶点, A 和 B 在不同的树中。
- cut A: 切断点 A 到其父节点的边, 保证 A 是一个非根节点。
- lca A B: 输出 A 和 B 的最近共同祖先, 保证 A 和 B 在同一棵树中。

```
struct LCT
{
    struct Node
    {
        int s[2], fa, rev;
    }tr[N];
    int stk[N], top;
    void pushrev(int u) { swap(tr[u].s[0], tr[u].s[1]), tr[u].rev ^= 1; }
    void pushdown(int u)
    {
        if (tr[u].rev)
            pushrev(tr[u].s[0]), pushrev(tr[u].s[1]), tr[u].rev = 0;
    }
    bool isr(int u) { return tr[tr[u].fa].s[0] != u && tr[tr[u].fa].s[1] != u; }
    void rotate(int x)
    {
        int y = tr[x].fa, z = tr[y].fa, k = tr[y].s[1] == x;
        if (!isr(y)) tr[z].s[tr[z].s[1] == y] = x;
        tr[x].fa = z, tr[y].s[k] = tr[x].s[k ^ 1];
        tr[tr[x].s[k ^ 1]].fa = y, tr[x].s[k ^ 1] = y, tr[y].fa = x;
    }
    void splay(int x)
    {
        int r = x; stk[top = 1] = x;
        while (!isr(r)) stk[++top] = r = tr[r].fa;
        while (top) pushdown(stk[top--]);
        while (!isr(x))
        {
            int y = tr[x].fa, z = tr[y].fa;
```

```
            if (!isr(y))
                if ((tr[y].s[1] == x) ^ (tr[z].s[1] == y)) rotate(x);
                else rotate(y);
            rotate(x);
        }
    }
    int access(int x)
    {
        int z = x, y;
        for (y = 0; x; y = x, x = tr[x].fa) splay(x), tr[x].s[1] = y;
        splay(z);
        return y;
    }
    void maker(int x) { access(x), pushrev(x); }
    int findr(int x)
    {
        access(x);
        while (tr[x].s[0]) pushdown(x), x = tr[x].s[0];
        splay(x); return x;
    }
    void link(int x, int y) { maker(x); if (findr(y) != x) tr[x].fa = y; }
    void cut(int x)//cut 的时候不能 makeroo
    {
        access(x), splay(x);
        tr[x].s[0] = tr[tr[x].s[0]].fa = 0;
    }
    int lca(int a, int b)
    {
        access(a);
        return access(b);
    }
}lct;
void solve()
{
    while (m--)
    {
        string opt;
        int a, b;
        cin >> opt >> a;
        if (opt=="link")
        {
            cin >> b;
            lct.link(a, b);
        }
        else if (opt=="cut") lct.cut(a);
        else
        {
            cin >> b;
            cout << lct.lca(a, b)<<"\n";
        }
    }
}
```

lct 维护子树信息

A x y 表示在 x 和 y 之间连一条边。保证之前 x 和 y 是不联通的。Q x y 表示询问 (x, y) 这条边两边连通块的大小乘积。保证 x 和 y 之间有一条边。

要求维护的信息可差分

```
struct LCT
{
    struct Node
    {
        int s[2], fa, rev;
        int sz, vir_sz; //实链的大小 虚子树的大小
    }tr[N];
    int stk[N], top;
    void pushup(int u)
    {//实儿子的信息加上虚儿子的信息
        tr[u].sz = tr[tr[u].s[0]].sz + 1 + tr[tr[u].s[1]].sz + tr[u].vir_sz;
    } // 查子树 sz 之前要 makeroo
    void pushrev(int u) { swap(tr[u].s[0], tr[u].s[1]), tr[u].rev ^= 1; }
    void pushdown(int u)
```

```
    {
        if (tr[u].rev)
            pushrev(tr[u].s[0]), pushrev(tr[u].s[1]), tr[u].rev = 0;
    }
    bool isr(int u) { return tr[tr[u].fa].s[0] != u && tr[tr[u].fa].s[1] != u; }
    void rotate(int x)
    {
        int y = tr[x].fa, z = tr[y].fa, k = tr[y].s[1] == x;
        if (!isr(y)) tr[z].s[tr[z].s[1] == y] = x;
        tr[x].fa = z, tr[y].s[k] = tr[x].s[k ^ 1];
        tr[tr[x].s[k ^ 1]].fa = y, tr[x].s[k ^ 1] = y, tr[y].fa = x;
        pushup(y), pushup(x);
    }
    void splay(int x)//把 x 旋转到 splay 的根
    {
        int r = x; stk[top = 1] = x;
        while (!isr(r)) stk[++top] = r = tr[r].fa;
        while (top) pushdown(stk[top--]);
        while (!isr(x))
        {
            int y = tr[x].fa, z = tr[y].fa;
            if (!isr(y))
                if ((tr[y].s[1] == x) ^ (tr[z].s[1] == y)) rotate(x);
                else rotate(y);
            rotate(x);
        }
    }
    void access(int x)//在原树上, 建立一条根到 x 的实链
    {
        int z = x;
        for (int y = 0; x; y = x, x = tr[x].fa)
        {
            splay(x); //加上新变成虚边所连的子树的贡献, 减去刚刚变成实边所连的子树的贡献
            tr[x].vir_sz += tr[tr[x].s[1]].sz - tr[y].sz;
            tr[x].s[1] = y, pushup(x);
        }
        splay(z);
    }
    void maker(int x) { access(x), pushrev(x); }
    int findr(int x)
    {
        access(x);
        while (tr[x].s[0]) pushdown(x), x = tr[x].s[0];
        splay(x); return x;
    }//在原树上创建一条从 x 到 y 的实边路径, 并把 y 变为根节点
    void split(int x, int y) { maker(x), access(y); }
    void link(int x, int y)
    {//在父亲结点的虚子树值中加上新子结点的子树大小贡献
        maker(x), maker(y);
        tr[x].fa = y, tr[y].vir_sz += tr[x].sz;
    }
    void cut(int x, int y)
    {
        maker(x);
        if (findr(y) == x && tr[y].fa == x&&!tr[y].s[0])
            tr[x].s[1] = tr[y].fa = 0, pushup(x);
        maker(y); //顺便让 y 变为根
    }
}lct;
void solve()
{
    for (int i = 1; i <= n; i++) lct.tr[i].sz = 1;
    while (q--)
    {
        char opt; int x, y;
        cin >> opt >> x >> y;
        if (opt=='A') lct.link(x, y);
        else
        {
            lct.cut(x, y);
            cout << lct.tr[x].sz * lct.tr[y].sz<<"\n";
            lct.link(x, y);
        }
    }
```

```
}
```

lct 做链修改

例: 对链上的点做加法和乘法

```
struct LCT
{
    struct Node
    {
        int s[2], fa, rev, sz;
        int v, sum, add, mul;
    }tr[N];
    int stk[N], top;
    void pushrev(int u) { swap(tr[u].s[0], tr[u].s[1]), tr[u].rev ^= 1; }
    void pushup(int u)
    {
        tr[u].sum = (tr[tr[u].s[0]].sum + tr[u].v + tr[tr[u].s[1]].sum) % mod;
        tr[u].sz = tr[tr[u].s[0]].sz + 1 + tr[tr[u].s[1]].sz;
    }
    void mul(int u, int v)
    {
        tr[u].v = tr[u].v * v % mod, tr[u].sum = tr[u].sum * v % mod;
        tr[u].add = tr[u].add * v % mod, tr[u].mul = tr[u].mul * v % mod;
    }
    void add(int u, int v)
    {
        tr[u].v = (tr[u].v + v) % mod, tr[u].sum = (tr[u].sum + tr[u].sz * v) % mod;
        tr[u].add = (tr[u].add + v) % mod;
    }
    void pushdown(int u)
    {
        if (tr[u].rev) pushrev(tr[u].s[0]), pushrev(tr[u].s[1]), tr[u].rev = 0;
        if (tr[u].mul != 1)
        {
            mul(tr[u].s[0], tr[u].mul), mul(tr[u].s[1], tr[u].mul);
            tr[u].mul = 1;
        }
        if (tr[u].add)
        {
            add(tr[u].s[0], tr[u].add), add(tr[u].s[1], tr[u].add);
            tr[u].add = 0;
        }
    }
    bool isr(int u) { return tr[tr[u].fa].s[0] != u && tr[tr[u].fa].s[1] != u; }
    void rotate(int x)
    {
        int y = tr[x].fa, z = tr[y].fa, k = tr[y].s[1] == x;
        if (!isr(y)) tr[z].s[tr[z].s[1] == y] = x;
        tr[x].fa = z, tr[y].s[k] = tr[x].s[k ^ 1], tr[tr[x].s[k ^ 1]].fa = y, tr[x].s[k ^ 1] =
        ↪  y, tr[y].fa = x;
        pushup(y), pushup(x);
    }
    void splay(int x)
    {
        int r = x; stk[top = 1] = x;
        while (!isr(r)) stk[++top] = r = tr[r].fa;
        while (top) pushdown(stk[top--]);
        while (!isr(x))
        {
            int y = tr[x].fa, z = tr[y].fa;
            if (!isr(y))
                if ((tr[y].s[1] == x) ^ (tr[z].s[1] == y)) rotate(x); else rotate(y);
            rotate(x);
        }
    }
    void access(int x)
    {
        int z = x;
        for (int y = 0; x; y = x, x = tr[x].fa) splay(x), tr[x].s[1] = y, pushup(x);
        splay(z);
    }
    void maker(int x) { access(x), pushrev(x); }
```

```
    int findr(int x)
    {
        access(x);
        while (tr[x].s[0]) pushdown(x), x = tr[x].s[0];
        splay(x); return x;
    }
    void split(int x, int y) { maker(x), access(y); }
    void link(int x, int y) { maker(x); if (findr(y) ≠ x) tr[x].fa = y; }
    void cut(int x, int y)
    {
        maker(x);
        if (findr(y) == x && tr[y].fa == x&&!tr[y].s[0]) tr[x].s[1] = tr[y].fa = 0, pushup(x);
    }
}lct;
```

1.11 分块

1.11.1 支持区间加法区间内小于某个数的元素个数 (区间重构)

todo: 利用归并可以做到更优的复杂度

```
int n, len, w[N], id[N], add[M];
vector<int> block[M];
void update(int u)
{
    block[u].clear();
    for (int i = (u - 1) * len + 1; i <= u * len && i <= n; i++) block[u].push_back(w[i]);
    sort(block[u].begin(), block[u].end());
}
void modify(int l, int r, int c)
{
    if (id[l] == id[r])
    {
        for (int i = l; i <= r; i++) w[i] += c;
        update(id[l]);
    }
    else
    {
        for (int i = l; i <= id[l] * len; i++) w[i] += c;
        update(id[l]);
        for (int i = (id[r] - 1) * len + 1; i <= r; i++) w[i] += c;
        update(id[r]);
        for (int i = id[l] + 1; i < id[r]; i++) add[i] += c;
    }
}
int query(int l, int r, int v)
{
    int res = 0;
    if (id[l] == id[r])
    {
        for (int i = l; i <= r; i++)
            if (w[i] + add[id[i]] < v) res++;
    }
    else
    {
        for (int i = l; i <= id[l] * len; i++)
            if (w[i] + add[id[l]] < v) res++;
        for (int i = (id[r] - 1) * len + 1; i <= r; i++)
            if (w[i] + add[id[r]] < v) res++;
        for (int i = id[l] + 1; i < id[r]; i++)
        {
            int l = 0, r = block[i].size() - 1;
            if (block[i][0] + add[i] >= v) continue;
            while (l < r)
            {
                int mid = l + r + 1 >>1;
                if (block[i][mid] + add[i] < v) l = mid;
                else r = mid - 1;
            }
            res += l + 1;
        }
    }
    return res;
```

```
}
int main()
{
    len = sqrt(n);
    for (int i = 1; i <= n; i++)
    {
        id[i] = (i - 1) / len + 1;
        block[id[i]].push_back(w[i]);
    }
    for (int i = 1; i <= id[n]; i++) sort(block[i].begin(), block[i].end());
    for (int i = 1; i <= n; i++)
    {
        int k, l, r, c;
        if (k == 0) modify(l, r, c);
        else cout << query(l, r, c) <<"\n";

}
```

1.11.2 分块在线解决区间众数

```
int w[N], id[N], n, m, x, len, cnt[N];
int f[M][M]; // 第 i 块到第 j 块中的众数
int s[M][N]; // 前 m 块中 x 的出现次数
void init()
{
    for (int i = 1; i <= id[n]; i++)
    {
        int l = (i - 1) * len + 1, r = min(n, i * len);
        for (int k = l; k <= r; k++) s[i][w[k]]++;
        for (int j = 0; j < lsh.size(); j++) s[i][j] += s[i - 1][j];
    }
    for (int i = 1; i <= id[n]; i++)
        for (int j = 1; j <= id[n]; j++)
        {
            int mode = f[i][j - 1], l = (j - 1) * len + 1, r = min(n, j * len);
            for (int k = l; k <= r; k++)
            {
                int v1 = s[j][w[k]] - s[i - 1][w[k]], v2 = s[j][mode] - s[i - 1][mode];
                if (v1 > v2) mode = w[k];
                else if (v1 == v2) mode = min(mode, w[k]);
            }
            f[i][j] = mode;
        }
}
int query(int l, int r)
{
    if (id[r] - id[l] <= 1)
    {
        int mode = w[l];
        cnt[w[l]]++;
        for (int i = l + 1; i <= r; i++)
        {
            cnt[w[i]]++;
            if (cnt[w[i]] > cnt[mode]) mode = w[i];
            else if (cnt[w[i]] == cnt[mode]) mode = min(mode, w[i]);
        }
        for (int i = l; i <= r; i++) cnt[w[i]] = 0;
        return lsh[mode];
    }
    else
    {
        int mode = f[id[l] + 1][id[r] - 1], t = mode;
        cnt[mode] = s[id[r] - 1][mode] - s[id[l]][mode];
        for (int i = l; i <= id[l] * len; i++) cnt[w[i]] = s[id[r] - 1][w[i]] -
        ↪  s[id[l]][w[i]];
        for (int i = (id[r] - 1) * len + 1; i <= r; i++) cnt[w[i]] = s[id[r] - 1][w[i]] -
        ↪  s[id[l]][w[i]];
        for (int i = l; i <= id[l] * len; i++)
        {
            cnt[w[i]]++;
            if (cnt[w[i]] > cnt[mode]) mode = w[i];
            else if (cnt[w[i]] == cnt[mode]) mode = min(mode, w[i]);
```

```
        }
        for (int i = (id[r] - 1) * len + 1; i <= r; i++)
        {
            cnt[w[i]]++;
            if (cnt[w[i]] > cnt[mode]) mode = w[i];
            else if (cnt[w[i]] == cnt[mode]) mode = min(mode, w[i]);
        }
        for (int i = l; i <= id[l] * len; i++) cnt[w[i]] = 0;
        for (int i = (id[r] - 1) * len + 1; i <= r; i++) cnt[w[i]] = 0;
        cnt[t] = 0;
        return lsh[mode];
    }
}
int main()
{
    len = sqrt(n);
    for (int i = 1; i<= n; i++) cin >> w[i], lsh.push_back(w[i]);
    for (int i = 1; i <= n; i++) id[i] = (i - 1) / len + 1;
    init();
    while (m--) cout << query(l, r) <<"\n";
}
```

1.12 莫队

```
struct Query
{
    int id, l, r;
}query[N];
bool cmp(Query a, Query b)
{
    if (id[a.l] != id[b.l]) return id[a.l] < id[b.l];
    else if (id[a.l] & 1) return a.r < b.r; //奇偶排序
    else return a.r > b.r;
}
void add(int x, int & res) // 加入一个数的贡献对答案的影响
void del(int x, int & res) //减少一个数的贡献对答案的影响
int main()
{
    sort(query, query + m, cmp); //莫队排序
    for (int i = 0, j = 1, k = 0, res = 0; k < m; k++)
    {
        int id = query[k].id, l = query[k].l, r = query[k].r;
        while (i < r) add(w[++i], res); //先扩展区间
        while (j > l) add(w[--j], res); //先扩展区间
        while (i > r) del(w[i--], res); //后缩小区间
        while (j < l) del(w[j++], res); //后缩小区间
        ans[id] = res;
    }
}
```

例子： 莫队解决区间众数的出现次数

```
struct Query
{
    int id, l, r;
}query[N];
int n, m, len;
int w[N], cnt[N], id[N], ans[N], cntnum[N];
vector<int> lsh;
bool cmp(Query a, Query b)
{
    if (id[a.l] != id[b.l]) return id[a.l] < id[b.l];
    else if (id[a.l] & 1) return a.r > b.r;
    else return a.r < b.r;
}
void add(int x, int & res)
{
    cnt[cntnum[x]]--;
    cntnum[x]++;
    cnt[cntnum[x]]++;
    res = max(res, cntnum[x]);
}
```

```cpp
void del(int x, int & res)
{
    cnt[cntnum[x]]--;
    cntnum[x]--;
    cnt[cntnum[x]]++;
    if (cnt[res] == 0) res--;
}
int main()
{
    for (int i = 1; i<= n; i++) cin >> w[i], lsh.push_back(w[i]);
    for (int i = 1; i <= n; i++) w[i] = find(w[i]);
    len = sqrt(n);
    for (int i = 1; i <= n; i++) id[i] = (i - 1) / len + 1;
    for (int i = 0; i < m; i++) query[i] = {i, l, r};
    sort(query, query + m, cmp);
    for (int i = 0, j = 1, k = 0, res = 0; k < m; k++)
    {
        int l = query[k].l, r = query[k].r, id = query[k].id;
        while (i < r) add(w[++i], res);
        while (j > l) add(w[--j], res);
        while (i > r) del(w[i--], res);
        while (j < l) del(w[j++], res);
        ans[id] = res;
    }
}
```

1.12.1 带修莫队

```cpp
struct Query
{
    int l, r, t, id;
}query[N]; // 记录下每个询问
struct Modify
{
    int x, v;
}modify[N]; // 记录下每个操作
int n, m, len, w[N], cnt[M], cntm, cntq, id[N], ans[N];
bool cmp(Query a, Query b)
{
    if (id[a.l] != id[b.l]) return id[a.l] < id[b.l]; //先按左端点进行块排序
    if (id[a.r] != id[b.r]) return id[a.r] < id[b.r]; //再按右端点进行块排序
    return a.t < b.t; //最后按时间进行排序
}
int main()
{
    for (int i = 0; i < m; i++)
    {
        char op[2];
        int a, b;
        if (*op=='R') modify[++cntm] = {a, b}; // 记录下每个询问及询问对应的时间
        else query[++cntq] = {a, b, cntm, cntq}; // 记录下每个操作并更新时间
    }
    len = cbrt((double)n * max(1, cntm)) + 1;
    sort(query + 1, query + 1 + cntq, cmp);
    for (int i = 0, j = 1, t = 0, k = 1, res = 0; k <= cntq; k++)
    {
        int l = query[k].l, r = query[k].r, tt = query[k].t, id = queey[k].id;
        while (i < r) add(w[++i], res); //先扩展区间
        while (j > l) add(w[--j], res); //先扩展区间
        while (i > r) del(w[i--], res); //后缩小区间
        while (j < l) del(w[j++], res); //后缩小区间
        while (t < tt) //如果说当前时间指针小于询问时间
        {
            t++; //时间指针移动
            if (modify[t].x >= j && modify[t].x <= i) //如果当前时间指针对应的操作的位置在 j 和 i 之
            ↪ 间
            {
                del(w[modify[t].x], res); //删去原来位置的数对答案的影响
                add(modify[t].v, res); //添加上新的数对答案的影响
            }
            swap(w[modify[t].x], modify[t].v); //交换一下
        }
```

```
        while (t > tt) //如果说当前时间指针大于询问时间
        {
            if (modify[t].x >= j && modify[t].x <= i) //如果当前时间指针对应的操作的位置在 j 和 i 之间
            {
                del(w[modify[t].x], res); //删去原来位置的数对答案的影响
                add(modify[t].v, res); //添加上新的数对答案的影响
            }
            swap(w[modify[t].x], modify[t].v); //交换一下
            t--; //时间指针移动
        }
        ans[id] = res; //记录答案
    }
}
```

例子：数颜色

```
struct Query
{
    int l, r, t, id;
}query[N];
struct Modify
{
    int x, v;
}modify[N];
int n, m, len, w[N], cnt[M], cntm, cntq, id[N], ans[N];
bool cmp(Query & a, Query & b)
{
    if (id[a.l] != id[b.l]) return id[a.l] < id[b.l];
    if (id[a.r] != id[b.r]) return id[a.r] < id[b.r];
    return a.t < b.t;
}
void add(int v, int & res)
{
    if (!cnt[v]) res++;
    cnt[v]++;
}
void del(int v, int & res)
{
    cnt[v]--;
    if (!cnt[v]) res--;
}
int main()
{
    for (int i = 0; i < m; i++)
    {
        char op[2];
        int a, b;
        cin >> op >> a >> b;
        if (*op=='R') modify[++cntm] = {a, b};
        else query[++cntq] = {a, b, cntm, cntq};
    }
    len = cbrt((double)n * max(1, cntm)) + 1;
    for (int i = 1; i <= n; i++) id[i] = (i - 1) / len + 1;
    sort(query + 1, query + 1 + cntq, cmp);
    for (int i = 0, j = 1, t = 0, k = 1, res = 0; k <= cntq; k++)
    {
        int l = query[k].l, r = query[k].r, tt = query[k].t;
        while (i < r) add(w[++i], res);
        while (i > r) del(w[i--], res);
        while (j < l) del(w[j++], res);
        while (j > l) add(w[--j], res);
        while (t < tt)
        {
            t++;
            if (modify[t].x >= j && modify[t].x <= i)
            {
                del(w[modify[t].x], res);
                add(modify[t].v, res);
            }
            swap(w[modify[t].x], modify[t].v);
        }
        while (t > tt)
        {
```

```
            if (modify[t].x >= j && modify[t].x <= i)
            {
                del(w[modify[t].x], res);
                add(modify[t].v, res);
            }
            swap(w[modify[t].x], modify[t].v);
            t--;
        }
        ans[query[k].id] = res;
    }
}
```

1.12.2 回滚莫队

一般用于处理一些删去后不能统计的信息, 如最大值, 删去后无法知道新的最大值, 则使用回滚莫队

```
struct Query
{
    int id, l, r;
}query[N];
int n, m, len, w[N], id[N], ans[N];
bool cmp(Query a, Query b)
{
    if (id[a.l] != id[b.l]) return id[a.l] < id[b.l];
    return a.r < b.r; //回滚莫队不能用奇偶排序优化
}
int main()
{
    for (int x = 0; x < m;)
    {
        int y = x; //设立一个指针 查找后面的询问中有多少个的左端点与当前询问的左端点在一个块中
        int right = id[query[x].l] * len; //当前这个询问的左端点所在块的最右端
        while (y < m && id[query[y].l] == id[query[x].l]) y++; //如果左端点都在一个块中 移动指针
        while (x < y && query[x].r <= right)//优先处理完左右端点都在一个块中的询问
        {
            int res = 0;
            int l = query[x].l, r = query[x].r, id = query[x].id;
            for (int k = l; k <= r; k++) add(k, w[k], res); //暴力统计答案
            ans[id] = res; //记录答案
            for (int k = l; k <= r; k++) cntl[w[k]] = 0; //暴力清空数组
            x++; //询问指针向后移动
        }
        int i = right, j = right + 1, res = 0; //i 为右指针（在左端点所在的块之后移动）
        while (x < y)//j 是左指针（在左端点所在的块中移动）
        {
            int l = query[x].l, r = query[x].r, id = query[x].id;
            while (i < r)//先移动右指针统计答案
            {
                i++;
                int u = w[i];
                if (!cntl[u]) cntl[u] = i;
                cntr[u] = i;
                res = max(res, cntr[u] - cntl[u]);
            }
            int backup = res; //备份一下当前的 res
            while (j > l)//移动左指针统计答案
            {
                j--;
                int u = w[j];
                if (cntr[u] >= right + 1) prer[u] = cntr[u];
                if (cntl[u] >= right + 1) prel[u] = cntl[u];
                if (!cntr[u]) cntr[u] = j;
                cntl[u] = j;
                res = max(res, cntr[u] - cntl[u]);
            }
            ans[id] = res; //记录答案
            res = backup; //还原 res
            while (j < right + 1)//左指针向右移动并恢复部分信息
            {
                cntl[w[j]] = prel[w[j]];
                cntr[w[j]] = prer[w[j]];
                j++;
            }
            x++;
```

```
        } // 把用到的数组都清空一下
        memset(cntl, 0, sizeof cntl), memset(cntr, 0, sizeof cntr);
        memset(prer, 0, sizeof prer), memset(prel, 0, sizeof prel);
    }
}
```

例子： 区间询问 $\max\{cnt_x \times x\}$

```
struct Query
{
    int id, l, r;
}query[N];
int n, m, len, w[N], id[N], cnt[N], ans[N];
vector<int> nums;
bool cmp(Query a, Query b)
{
    if (id[a.l] != id[b.l]) return id[a.l] < id[b.l];
    return a.r < b.r;
}
void add(int x, LL & res)
{
    cnt[x]++;
    res = max(res, cnt[x] * nums[x]);
}
int main()
{
    len = sqrt(n);
    for (int i = 1; i<= n; i++) cin >> w[i], lsh.push_back(w[i]);
    for (int i = 1; i <= n; i++) id[i] = (i - 1) / len + 1;
    for (int i = 0; i < m; i++) query[i] = {i, l, r};
    sort(query, query + m, cmp);
    for (int x = 0; x < m;)
    {
        int y = x;
        while (y < m && id[query[y].l] == id[query[x].l]) y++;
        int right = id[query[x].l] * len;
        while (x < y && query[x].r <= right)
        {
            int res = 0;
            int l = query[x].l, r = query[x].r, id = query[x].id;
            for (int i = l; i <= r; i++) add(w[i], res);
            ans[id] = res;
            for (int i = l; i <= r; i++) cnt[w[i]]--;
            x++;
        }
        int res = 0;
        int i = right, j = right + 1;
        while (x < y)
        {
            int l = query[x].l, r = query[x].r, id = query[x].id;
            while (i < r) add(w[++i], res);
            int backup = res;
            while (j > l) add(w[--j], res);
            ans[id] = res;
            res = backup;
            while (j < right + 1) cnt[w[j++]]--;
            x++;
        }
        memset(cnt, 0, sizeof cnt);
    }
}
```

1.12.3 树上莫队

如果将树换成一个序列, 那么就是一个经典的基础莫队问题, 等于是要在树上做莫队算法, 这就是一个树上莫队问题。有一个通用的做法能将树上的问题统一变成区间中的问题。我们将这棵树按照欧拉序列的形式写成一个序列。欧拉序列就是按照深度优先遍历的方式遍历整个树, 每个节点在进入和出去的时候都需要加入序列一次。因此欧拉序列中每个节点应该出现两次。这样就能将树上的任意一段路径 (x, y) 转化为欧拉序列中的一段区间

```
struct Query
{
    int id, l, r, p;
}query[N];
```

```cpp
int h[N], e[N], ne[N], idx;
int n, m, len, w[N];
int first[N], last[N], seq[N], top; //第一次出现的位置 最后一次出现的位置 欧拉序中对应的点
int depth[N], f[N][16]; //lca 的数组
int id[N], cnt[N], st[N], ans[N];
void dfs(int u, int fa)//处理出树的欧拉序
{
    seq[++top] = u;
    first[u] = top;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j != fa) dfs(j, u);
    }
    seq[++top] = u;
    last[u] = top;
}
bool cmp(Query & a, Query & b)
{
    if (id[a.l] != id[b.l]) return id[a.l] < id[b.l];
    return a.r < b.r;
}
void add(int x, int & res)
{
    st[x] ^= 1; //若在区间中出现一次, 则区间中存在这个数 若在区间中出现两次, 则区间中不存在这个数
    if (st[x] == 0)//删掉这个数
    {
        cnt[w[x]]--;
        if (!cnt[w[x]]) res--;
    }
    else//加上这个数
    {
        if (!cnt[w[x]]) res++;
        cnt[w[x]]++;
    }
}
int main()
{
    dfs(1, -1);
    for (int i = 0; i < m; i++)
    {
        int a, b;
        if (first[a] > first[b]) swap(a, b); //first 在在前的最先在欧拉序中出现
        int p = lca(a, b); //求一下 lca
        if (a == p) query[i] = {i, first[a], first[b]}; //如果 a 是 b 的祖先 那么从 first[a] 到 first[b]
        else query[i] = {i, last[a], first[b], p}; //如果 a 不是 b 的祖先 那么从 last[a] 到 first[b]
        //并且注意 若 a 不是 b 的祖先 则欧拉序中不存在二者的 lca 但是要查询的是包括二者的 lca 的 故需记录
    }
    sort(query, query + m, cmp);
    for (int i = 0, j = 1, k = 0, res = 0; k < m; k++)
    {
        int l = query[k].l, r = query[k].r, id = query[k].id, p = query[k].p;
        while (i < r) add(w[++i], res); //先扩展区间
        while (j > l) add(w[--j], res); //先扩展区间
        while (i > r) del(w[i--], res); //后缩小区间
        while (j < l) del(w[j++], res); //后缩小区间
        if (p) add(p, res); //如果存在公共祖先 那么添加
        ans[id] = res; //记录答案
        if (p) add(p, res); //如果添加过 则删去 (由于欧拉序的性质 再添加一遍即可)
    }
}
```

例题: 查询树上路径的结点不同颜色数

```cpp
struct Query
{
    int id, l, r, p;
}query[N];
int h[N], e[N], ne[N], idx;
int n, m, len, w[N];
int first[N], last[N], seq[N], top;
int depth[N], f[N][16];
```

```cpp
int id[N], cnt[N], st[N], ans[N];
vector<int> lsh;
void dfs(int u, int fa)
{
    seq[++top] = u, first[u] = top;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j != fa) dfs(j, u);
    }
    seq[++top] = u, last[u] = top;
}
bool cmp(Query & a, Query & b)
{
    if (id[a.l] != id[b.l]) return id[a.l] < id[b.l];
    return a.r < b.r;
}
void add(int x, int & res)
{
    st[x] ^= 1;
    if (st[x] == 0)
    {
        cnt[w[x]]--;
        if (!cnt[w[x]]) res--;
    }
    else
    {
        if (!cnt[w[x]]) res++;
        cnt[w[x]]++;
    }
}
int main()
{
    len = sqrt(2 * n);
    for (int i = 1; i<= n; i++) cin >> w[i], lsh.push_back(w[i]);
    for (int i = 1; i <= 2 * n; i++) id[i] = (i - 1) / len + 1;
    for (int i = 1; i < n; i++) add_edge(a, b), add_edge(b, a);
    dfs(1, -1), bfs();
    for (int i = 0; i < m; i++)
    {
        int a, b;
        if (first[a] > first[b]) swap(a, b);
        int p = lca(a, b);
        if (a == p) query[i] = {i, first[a], first[b]};
        else query[i] = {i, last[a], first[b], p};
    }
    sort(query, query + m, cmp);
    for (int i = 0, j = 1, k = 0, res = 0; k < m; k++)
    {
        int l = query[k].l, r = query[k].r, id = query[k].id, p = query[k].p;
        while (i < r) add(seq[++i], res);
        while (i > r) add(seq[i--], res);
        while (j < l) add(seq[j++], res);
        while (j > l) add(seq[--j], res);
        if (p) add(p, res);
        ans[id] = res;
        if (p) add(p, res);
    }
}
```

1.13 偏序问题

1.13.1 二维偏序

```cpp
vector<array<int, 4>> v;
for (int i = 1; i <= n; i++) v.push_back({x, -2, y}); //先按照 x 排序 查询排前面
for (int i = 1; i <= m; i++)
{
    int x1, y1, x2, y2;
    v.push_back({x2, 1, y2, i}), v.push_back({x2, -1, y1 - 1, i});
    v.push_back({x1 - 1, -1, y2, i}), v.push_back({x1 - 1, 1, y1 - 1, i});
}
```

```
sort(v.begin(), v.end());
for (auto & [x, sign, y, id]:v)//利用数据结构解决第二维偏序
    if (sign == -2) add(y, 1);
    else ans[id] += sign * query(y);
```

1.13.2 三维偏序

在三维空间上有一些点形如 (x_i, y_i, z_i), 再给出一些查询 $(x_1, x_2, y_1, y_2, z_1, z_2)$。求 $\sum_{i=1}^{n}[x_1 \leqslant x_i \leqslant x_2, y_1 \leqslant y_i \leqslant y_2, z_1 \leqslant z_i \leqslant z_2]$

先利用二维差分将询问拆成四个询问, 即求 $\sum_{i=1}^{n}[x_i \leqslant x_2, y_i \leqslant y_2, z_1 \leqslant z_i \leqslant z_2] - \sum_{i=1}^{n}[x_i \leqslant x_1 - 1, y_i \leqslant y_2, z_1 \leqslant z_i \leqslant z_2] - \sum_{i=1}^{n}[x_i \leqslant x_2, y_i \leqslant y_1 - 1, z_1 \leqslant z_i \leqslant z_2] + \sum_{x=1}^{n}[x_i \leqslant x_1 - 1, y_i \leqslant y_1 - 1, z_1 \leqslant z_i \leqslant z_2]$

那么对于每个询问, 相当于求一个形式的和 $\sum_{i=1}^{n}[x_i \leqslant X, y_i \leqslant Y, z_i \leqslant Z]$

对于每个原始点, 插入一个事件 $(x_i, y_i, 0, z_i, val_i)$

对于每个查询点, 插入一个事件 $(X, Y, 1, sign, Z_1, Z_2)$

正确性: 先按照 x 坐标排序保证 x 的顺序是对的, 再通过归并排序保证 y 的顺序是对的, 通过 $0/1$ 区分原始点和查询点保证原始点都在查询点之前插入然后 (Z_1, Z_2) 的贡献通过树状数组相减得到。

即 $\sum_{z_i \leqslant Z_2} - \sum_{z_i \leqslant Z_1 - 1}$ 注意这个式子是通过树状数组算出来的, 所以要保证原始点和查询点的 z 坐标都在树状数组的值域内, 若不满足则要先进行离散化。

```
void cdq(int l, int r)
{
    if (l >= r) return ;
    int mid = (l + r) >> 1;
    cdq(l, mid), cdq(mid + 1, r);
    int i = l, j = mid + 1, k = 0;
    while (i <= mid && j <= r)
    {
        if (v[i][1] <= v[j][1])
        {
            if (!v[i][2]) add(v[i][3], v[i][4]);
            w[k++] = v[i++];
        }
        else
        {
            if (v[j][2]) ans += v[j][3] * (query(v[j][5]) - query(v[j][4] - 1));
            w[k++] = v[j++];
        }
    }
    while (i <= mid)
    {
        if (!v[i][2]) add(v[i][3], v[i][4]);
        w[k++] = v[i++];
    }
    while (j <= r)
    {
        if (v[j][2]) ans += v[j][3] * (query(v[j][5]) - query(v[j][4] - 1));
        w[k++] = v[j++];
    }
    for (i = l; i <= mid; i++) if (!v[i][2]) add(v[i][3], -v[i][4]);
    for (i = l, j = 0; i <= r; i++, j++) v[i] = w[j];
}
```

1.13.3 四维偏序

四维偏序优化 dp

```
int dp[N];
bool cmp1(array<int, 7> a, array<int, 7> b)
{
    if (a[1] != b[1]) return a[1] < b[1];
    if (a[2] != b[2]) return a[2] < b[2];
    return a[3] < b[3];
}
bool cmp2(array<int, 7> a, array<int, 7> b)
{
    if (a[2] != b[2]) return a[2] < b[2];
    return a[3] < b[3];
```

```
}
void cdq2(int l, int r)
{
    if (l >= r) return ;
    int mid = (l + r) >> 1;
    cdq2(l, mid);
    stable_sort(w + l, w + 1 + mid, cmp2);
    stable_sort(w + mid + 1, w + 1 + r, cmp2);
    int i = l, j = mid + 1;
    while (i <= mid && j <= r)
    {
        if (w[i][2] <= w[j][2])
        {
            if (!w[i][6]) bit.add(w[i][3], dp[w[i][5]]);
            i++;
        }
        else
        {
            if (w[j][6]) dp[w[j][5]] = max(dp[w[j][5]], bit.query(w[j][3]) + w[j][4]);
            j++;
        }
    }
    while (i <= mid)
    {
        if (!w[i][6]) bit.add(w[i][3], dp[w[i][5]]);
        i++;
    }
    while (j <= r)
    {
        if (w[j][6]) dp[w[j][5]] = max(dp[w[j][5]], bit.query(w[j][3]) + w[j][4]);
        j++;
    }
    for (int j = l; j < i; j++)
        if (!w[j][6]) bit.clear(w[j][3]);
    stable_sort(w + 1 + l, w + 1 + r, cmp1);
    cdq2(mid + 1, r);
}
void cdq1(int l, int r)
{
    if (l >= r) return ;
    int mid = (l + r) >> 1;
    cdq1(l, mid);
    for (int i = l; i <= mid; i++) w[i][6] = 0;
    for (int i = mid + 1; i <= r; i++) w[i][6] = 1;
    stable_sort(w + l, w + 1 + r, cmp1);
    cdq2(l, r);
    stable_sort(w + l, w + 1 + r);
    cdq1(mid + 1, r);
}
void solve()
{
    for (int i = 0; i < M; i++) bit.tr[i] = -INF;
    for (int i = 1; i <= n; i++)
    {
        cin >> w[i][0] >> w[i][1] >> w[i][2] >> w[i][3] >> w[i][4];
        w[i][5] = i, dp[i] = w[i][4];
    }
    sort(w + 1, w + 1 + n);
    cdq1(1, n);
    int ans = -INF;
    for (int i = 1; i <= n; i++) ans = max(ans, dp[i]);
    cout << ans << "\n";
}
```

1.13.4 高维偏序

bitset 解决高维偏序 n 个向量的 m 维偏序

```
bitset<N> bs[N], temp;
int id[N], w[M][N], n, m, cur;
bool cmp(int a, int b) { return w[cur][a] < w[cur][b]; }
int main()
{
```

```
    for (int i = 1; i <= n; i++) bs[i].set(), id[i] = i;
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++)
            cin >> w[i][j];
    for (int j = 1; j <= m; j++)
    {
        cur = j; //现在按照哪一维排序
        temp.reset();
        sort(id + 1, id + 1 + n, cmp);
        for (int i = 1; i <= n; i++)
        {
            int k = i;
            while (k + 1 <= n && w[j][id[k]] == w[j][id[k + 1]]) k++;
            for (int u = i; u <= k; u++) bs[id[u]] &= temp;
            for (int u = i; u <= k; u++) temp.set(id[u]);
            i = k;
        }
    }
} // 算完之后 假如 bitset[i] 中 j 为 1 的话 代表 i 完全大于 j
```

1.14 整体二分

用途：离线解决带修改的区间第 k 小、区间 rank 等问题。

核心思想：将所有操作按时间序离线，在值域上二分，同时处理一批询问。

关键：修改操作影响后续询问，询问根据当前值域范围决定贡献。

优势：支持带修改，时间复杂度 $O((n+m)\log^2 n)$，比线段树套线段树更优。

整体二分本身就是支持带修改的, 因为是按照时间序把操作离线下来的

以解决区间第 k 小为例:

```
struct Query
{
    int type, x, y, k, id;
}q[N << 1], temp1[N << 1], temp2[N << 1];
void solve(int l, int r, int ql, int qr)
{
    if (ql > qr) return ;
    if (l == r)
    {
        for (int i = ql; i <= qr; i++)
        {
            if (q[i].type == 1)
                ans[q[i].id] = l;
        }
        return ;
    }
    int mid = (l + r) >> 1, p1 = 0, p2 = 0;
    for (int i = ql; i <= qr; i++)
        if (!q[i].type)
        {
            if (q[i].x <= mid)
            {
                add(q[i].y, 1);
                temp1[++p1] = q[i];
            }
            else temp2[++p2] = q[i];
        }
        else
        {
            int cnt = query(q[i].y) - query(q[i].x - 1);
            if (cnt >= q[i].k) temp1[++p1] = q[i];
            else
            {
                q[i].k -= cnt;
                temp2[++p2] = q[i];
            }
        }
    for (int i = 1; i <= p1; i++)
    {
        q[i + ql - 1] = temp1[i];
        if (!q1[i].type) add(q1[i].y, -1); //清空贡献
    }
    for (int i = 1; i <= p2; i++) q[i + p1 + ql - 1] = temp2[i];
```

```
    solve(l, mid, ql, ql + p1 - 1);
    solve(mid + 1, r, ql + p1, qr);
}
int main()
{
    for (int i = 1, x; i <= n; i++)
    {
        cin >> x;
        q[cnt++] = {0, x, i};
    }
    for (int i = 0, l, r, k; i < m; i++) q[cnt++] = {1, l, r, k, i};
    solve(-INF, INF, 0, cnt - 1);
    for (int i = 0; i < m; i++) cout << ans[i] <<"\n";
}
```

例子：

Q l r k 表示查询下标在区间 $[l, r]$ 中的第 k 小的数

C x y 表示将 a_x 改为 y

```
const int N = 100010, INF = 1e9;
struct Query
{
    int type, x, y, k, id;
}q[N << 2], q1[N << 2], q2[N << 2];
int n, m, cnt, tr[N], ans[N], w[N];
void solve(int l, int r, int ql, int qr)
{
    if (ql > qr) return ;
    if (l == r)
    {
        for (int i = ql; i <= qr; i++)
            if (q[i].type == 1) ans[q[i].id] = l;
        return ;
    }
    int mid = (l + r) >> 1, p1 = 0, p2 = 0;
    for (int i = ql; i <= qr; i++)
        if (q[i].type == 0)
        {
            if (q[i].x <= mid)
            {
                add(q[i].y, 1);
                q1[++p1] = q[i];
            }
            else q2[++p2] = q[i];
        }
        else if (q[i].type == 1)
        {
            int cnt = query(q[i].y) - query(q[i].x - 1);
            if (cnt >= q[i].k) q1[++p1] = q[i];
            else
            {
                q[i].k -= cnt;
                q2[++p2] = q[i];
            }
        }
        else
        {
            if (q[i].x <= mid)
            {
                add(q[i].y, q[i].k);
                q1[++p1] = q[i];
            }
            else q2[++p2] = q[i];
        }
    for (int i = 1; i <= p1; i++)
    {
        q[i + ql - 1] = q1[i];
        if (!q1[i].type) add(q1[i].y, -1);
        else if (q1[i].type == 2) add(q1[i].y, -q1[i].k);
    }
    for (int i = 1; i <= p2; i++) q[i + p1 + ql - 1] = q2[i];
    solve(l, mid, ql, ql + p1 - 1);
    solve(mid + 1, r, ql + p1, qr);
}
```

```
int main()
{
    int qcnt = 0;
    for (int i = 1; i <= n; i++)
    {
        cin >> w[i];
        q[cnt++] = {0, w[i], i};
    }
    for (int i = 0, l, r, k; i < m; i++)
    {
        char op[2];
        cin >> op;
        if (*op=='Q')
        {
            cin >> l >> r >> k;
            q[cnt++] = {1, l, r, k, qcnt};
            qcnt++;
        }
        else
        {
            cin >> l >> r;
            q[cnt++] = {2, w[l], l, -1};
            q[cnt++] = {2, r, l, 1};
            w[l] = r;
        }
    }
    solve(0, INF, 0, cnt - 1);
}
```

1.15 猫树分治

猫树可以支持不修改的高速查询区间信息。

先考虑普通猫树, 普通猫树是线段树的拓展。线段树将询问区间拆成 $\log n$ 个区间, 然后再进行合并。猫树通过增加预处理的时间和空间, 将询问区间拆成两个区间, 加快了合并效率。

在查询 $[l,r]$ 这段区间的信息时, 将线段树上代表 $[l,l]$ 和 $[r,r]$ 的区间的节点的 LCA 求出来, 设这个节点 p 所代表的区间是 $[L,R]$。可以知道 $[l,r]$ 区间一定跨越了 $[L,R]$ 的中点 mid, 且 $l \in [L,mid], r \in [mid+1,r]$。在建树的时候预处理出 $[L,mid]$ 的后缀信息和 $[mid+1,R]$ 的前缀信息, 查询的时候拼凑起来即可。

现在瓶颈在于快速求得 LCA，不需要用 ST 表的 $O(n\log n)-O(1)$ 的技巧，可以利用堆式建树，将这个序列补成 2 的整次幂，然后建线段树。此时, 线段树上的两个节点的 LCA 编号, 就是两个节点二进制编号的 LCP。且有结论 $lcp(x,y)=x>>\log(x\oplus y)$, 即可快速查询 LCA。

若猫树的空间开销过大，难以承受，且问题可以离线，可以采用猫树分治，即将询问分配到猫树上的每个节点，查询该节点上的询问时才需要将猫树这个节点建出来，整个过程和整体二分较为类似。

```
int h[N], w[N], n, m, ans[M], f[N][K];
array<int, 4> q[M], tl[M], tr[M];
void solve(int l, int r, int ql, int qr)
{
    if (ql > qr) return ;
    if (l == r)
    {
        for (int i = ql; i <= qr; i++)
            if (q[i][3] >= h[l]) ans[q[i][0]] = w[l];
        return ;
    }
    int mid = (l + r) >> 1;
    memset(f[mid], 0, sizeof f[mid]);
    for (int i = h[mid]; i <= 200; i++) f[mid][i] = w[mid];
    for (int i = mid - 1; i >= l; i--)
    {
        memcpy(f[i], f[i + 1], sizeof f[i]);
        for (int j = 200; j >= h[i]; j--)
            f[i][j] = max(f[i][j], f[i][j - h[i]] + w[i]);
    }
    memset(f[mid + 1], 0, sizeof f[mid + 1]);
    for (int i = h[mid + 1]; i <= 200; i++) f[mid + 1][i] = w[mid + 1];
    for (int i = mid + 2; i <= r; i++)
    {
        memcpy(f[i], f[i - 1], sizeof f[i]);
        for (int j = 200; j >= h[i]; j--)
            f[i][j] = max(f[i][j], f[i][j - h[i]] + w[i]);
    }
```

```
    int cnt1 = 0, cnt2 = 0;
    for (int i = ql; i <= qr; i++)
    {
        auto [id, l, r, t] = q[i];
        if (r <= mid) tl[++cnt1] = q[i];
        else if (l > mid) tr[++cnt2] = q[i];
        else
        {
            for (int j = 0; j <= t; j++)
                ans[id] = max(ans[id], f[l][j] + f[r][t - j]);
        }
    }
    for (int i = 1; i <= cnt1; i++) q[ql + i - 1] = tl[i];
    for (int i = 1; i <= cnt2; i++) q[ql + cnt1 + i - 1] = tr[i];
    solve(l, mid, ql, ql + cnt1 - 1);
    solve(mid + 1, r, ql + cnt1, ql + cnt1 + cnt2 - 1);
}
void solve()
{
    for (int i = 1; i<= n; i++) cin >> h[i];
    for (int i = 1; i<= n; i++) cin >> w[i];
    for (int i = 1; i <= m; i++) q[i] = {i, l, r, t};
    solve(1, n, 1, m);
}
```

2 树

2.1 最近公共祖先

2.1.1 树剖求 LCA

```
int lca(int a, int b)
{
    while (top[a] != top[b])
    {
        if (depth[top[a]] < depth[top[b]]) swap(a, b);
        a = fa[top[a]];
    }
    if (depth[a] > depth[b]) return b;
    return a;
}
```

2.1.2 树剖求 k 级祖先

需要根据深度特判一下 k 级祖先是否存在

```
int query(int x, int k)
{
    while (k >= dfn[x] - dfn[top[x]] + 1)
    {
        k -= dfn[x] - dfn[top[x]] + 1;
        x = fa[top[x]];
    }
    return yingshe[dfn[x] - k];
} // yingshe 是 dfn 的反函数
```

2.1.3 DFS 序求 LCA

```
int h[N], ne[N << 1], e[N << 1], idx, dfn[N], minv[N][19], cnt;
void dfs(int u, int fa)
{
    minv[dfn[u] = ++cnt][0] = fa;
    for (int i = h[u]; ~i; i = ne[i]) if (e[i] != fa) dfs(e[i], u);
}
int get(int x, int y){ return dfn[x] < dfn[y]?x:y; }
int lca(int u, int v)
{
    if (u == v) return u;
    if ((u = dfn[u]) >(v = dfn[v])) swap(u, v);
```

```
    int d = __lg(v - u++);
    return get(minv[u][d], minv[v - (1 << d) + 1][d]); //做完之后要记得处理 ST 表
}
void solve()//要记得处理 ST 表
{
    dfs(root, 0);
    for (int k = 1; k <= 19; k++)
        for (int i = 1; i + (1 << k) - 1 <= n; i++)
            minv[i][k] = get(minv[i][k - 1], minv[i + (1 << k - 1)][k - 1]);
}
```

2.2 最小生成树

2.2.1 Prim

```
int n, g[N][N], dist[N];
bool st[N];
int prim()
{
    memset(dist, 0x3f, sizeof dist);
    int res = 0;
    for (int i = 0; i < n; i++)
    {
        int t = -1;
        for (int j = 1; j <= n; j++)
            if (!st[j] && (t == -1 || dist[t] > dist[j])) t = j;
        st[t] = true;
        if (i && dist[t] == INF) return INF;
        if (i) res += dist[t];
        for (int j = 1; j <= n; j++) dist[j] = min(dist[j], g[t][j]);
    }
    return res;
}
```

2.2.2 Kruskal 重构树

以大根堆为例:

Kruskal 重构树的叶子是原点其余点是虚点原点之间的 lca 的值就是使得这两点联通的最大花费的最小值

如果给定 u, 再给定一个阈值 v, 问经过的边权值不能超过 v 的条件下, 就是倍增跳祖先, 子树就是能到的点

注意点数是 $2 \times n - 1$ 边数是 $2 \times n$

很有可能要在重构树上面跳所以求 lca 一般用的是倍增而不是树剖

```
void kruskal()
{
    sort(edge, edge + m);
    for (int i = 1; i <= 2 * n; i++) p[i] = i;
    for (int i = 0; i < m; i++)
    {
        int a = edge[i].a, b = edge[i].b;
        if (find(a) == find(b)) continue;
        cnt++; //开一个新节点
        add(cnt, p[a]), add(cnt, p[b]);
        p[p[a]] = p[p[b]] = p[cnt] = cnt;
        v[cnt] = edge[i].c;
    }
    for (int i = cnt; i >= 1; i--)//因为形成的可能是一个森林 所以需要遍历
        if (!sz[i]) dfs1(i, -1, 1), dfs2(i, i);
}
int query(int a, int b)//前面利用树剖求 lca
{
    if (find(a) != find(b)) return - 1;
    while (top[a] != top[b])
    {
        if (depth[top[a]] < depth[top[b]]) swap(a, b);
        a = fa[top[a]];
    }
    if (depth[a] > depth[b]) return v[b];
    else return v[a];
}
```

例子：归程 (NOI2018)

```
int h[N], ne[M], w[M], e[M], idx;
int T, n, m, q, k, s, dist[N], p[N], v[N], fa[N][20];
bool st[N];
void dfs(int u)
{
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        fa[j][0] = u;
        for (int k = 1; k < 20; k++) fa[j][k] = fa[fa[j][k - 1]][k - 1];
        dfs(j);
        dist[u] = min(dist[u], dist[j]);
    }
}
int query(int s, int p)
{
    for (int k = 19; k >= 0; k--)
        if (v[fa[s][k]] > p) s = fa[s][k];
    return dist[s];
}
void scutsky()
{
    for (int i = 1; i <= n; i++) h[i] = -1, p[i] = i, st[i] = false;
    for (int i = 0, u, v, l, a; i < m; i++)
    {
        add(u, v, l), add(v, u, l);
        edges[i] = {u, v, a};
    }
    dijkstra();
    sort(edges, edges + m);
    for (int i = 1; i <= 2 * n; i++) h[i] = -1;
    idx = 0;
    int cnt = n;
    for (int i = 0; i < m; i++)
    {
        int a = edges[i].a, b = edges[i].b;
        int pa = find(a), pb = find(b);
        if (pa == pb) continue;
        cnt++;
        v[cnt] = edges[i].c;
        add(cnt, pa, 0), add(cnt, pb, 0);
        p[pa] = p[pb] = p[cnt] = cnt;
    }
    for (int i = n + 1; i <= cnt; i++) dist[i] = INF;
    dfs(cnt);
    int ans = 0;
    while (q--)
    {
        int v, p;
        v = (v + k * ans - 1) % n + 1, p = (p + k * ans) % (s + 1);
        ans = query(v, p);
    }
}
```

上面解决的问题都是边权, 如果是点权, 有两种解决方法:

1. 为原图每条边巧妙赋值, 将点权转化为边权。若限制经过的点权最大值, 因为走一条边 $u \to v$ 需满足 w_u, w_v 都不超过限制, 所以 $w(u,v) = \max\{w_u, w_v\}$。类似地, 若限制最小值则 $w(u,v) = \min\{w_u, w_v\}$。

2(**点权多叉重构树**). 实际上我们几乎用不到重构树是二叉树这一性质, 因此存在更高妙的做法。不妨设题目限制点权最大值。将节点按权值从小到大排序, 按序遍历每个点 i 及其所有出边 $i \to u$。若 u 已经遍历过, 则 $w_i \geq w_u$,$\max\{w_i, w_u\}$ 取到 w_i, 此时若 i, u 不连通则从 i 向 u 的代表元连边。

上述做法与一般 kruskal 重构树几乎等价: 普通重构树中点权相同的虚点, 仅有深度最小的有用: 按权值从小到大枚举节点相当于对所有边排序, 因为边权即 $\max\{w_i, w_u\}$。这样做不用新建虚点, 有效减小了常数。

例:IOI2018 狼人

给一个 n 个点, m 条边的无向图, 再给定 Q 个询问, 每次询问, 给定起点 s 和 t, 再给定两个限制 L, R, 要求在 1 状态下不能经过编号 $\leqslant L$ 的点, 在 2 状态下不能经过编号 $\geq R$ 的点, 起始状态是 1, 可以在中途变为 2, 问 s 能否到

建出两个重构树, 对 s 和 t 分别在两个重构树上倍增, 二维数点两个子树是否有交集

```
struct KTree
{
    vector<int> e[N];
```

```
    int p[N], fa[N][M];
    int dfn[N], sz[N], cnt;
    int find(int x)
    {
        if (x != p[x]) p[x] = find(p[x]);
        return p[x];
    }
    void init() { for (int i = 1; i <= n; i++) p[i] = i; }
    void add(int a, int b) { e[a].push_back(b); }
    void dfs(int u)
    {
        dfn[u] = ++cnt, sz[u] = 1;
        for (auto j:e[u])
        {
            fa[j][0] = u;
            for (int k = 1; k < M; k++) fa[j][k] = fa[fa[j][k - 1]][k - 1];
            dfs(j);
            sz[u] += sz[j];
        }
    }
    int find(int x, int val, int ty)
    {
        if (ty == 0 && x < val) return x;
        else if (ty == 1 && x > val) return x;
        for (int k = 18; k >= 0; k--)
            if (ty == 0)
            {
                if (fa[x][k] && fa[x][k] >= val) x = fa[x][k];
            }
            else if (ty == 1)
            {
                if (fa[x][k] && fa[x][k] <= val) x = fa[x][k];
            }
        return x;
    }
}L, R;
void solve()
{
    while (m--) g[a].push_back(b), g[b].push_back(a);
    L.init(), R.init();
    for (int i = 1; i <= n; i++)
        for (auto j:g[i])
            if (i > j && L.find(i) != L.find(j))
            {
                L.add(L.find(i), L.find(j));
                L.p[L.find(j)] = L.find(i);
            } // 限制了点权的 max 不能太大
    for (int i = n; i >= 1; i--)
        for (auto j:g[i])
            if (i < j && R.find(i) != R.find(j))
            {
                R.add(R.find(i), R.find(j));
                R.p[R.find(j)] = R.find(i);
            } // 限制了点权的 min 不能太小
    L.dfs(n), R.dfs(1);
    vector<array<int, 5>> q;
    for (int i = 0; i < Q; i++)
    {
        int s, t, l, r;
        int u = R.find(s, l, 0), v = L.find(t, r, 1);
        int x1 = R.dfn[u], x2 = R.dfn[u] + R.sz[u] - 1;
        int y1 = L.dfn[v], y2 = L.dfn[v] + L.sz[v] - 1;
        //二维数点 看看有无交集
        q.push_back({x2, 1, 1, y2, i});
        q.push_back({x1 - 1, 1, -1, y2, i});
        q.push_back({x2, 1, -1, y1 - 1, i});
        q.push_back({x1 - 1, 1, 1, y1 - 1, i});
    }
    for (int i = 1; i <= n; i++) q.push_back({R.dfn[i], 0, L.dfn[i]});
    sort(q.begin(), q.end());
    for (auto [x, ty, sign, y, id]:q)
    {
        if (!ty) add(sign, 1);
        else ans[id] += sign * query(y);
```

```
    }
    for (int i = 0; i < Q; i++) cout << (ans[i] >0)<<"\n";
}
```

2.2.3 Boruvka

Boruvka 算法的执行流程是：对于当前每个连通块，都找到其伸出去的最小的边，然后进行合并。每次合并，连通块数量至少减半，故合并次数是 $O(\log n)$ 的。每次对于所有集合寻找最小边的时间假如是 $O(f)$，那么这个算法时间复杂度是 $O(f \times \log n)$

当边权由点的点权决定时, 可以考虑 Boruvka 算法

```cpp
int p[N], choose[N];
bool used[M];
void solve()
{
    for (int i = 1; i <= n; i++) p[i] = i;
    vector<array<int, 3>> op(m + 1);
    for (int i = 1; i <= m; i++) op[i] = {a, b, c};
    int cnt = 0, ans = 0;
    bool flag = true;
    while (flag)
    {
        flag = false;
        memset(choose, 0, sizeof choose);
        for (int i = 1; i <= m; i++)
        {
            if (used[i]) continue;
            auto [a, b, v] = op[i];
            int pa = find(a), pb = find(b);
            if (pa == pb) continue;
            if (!choose[pa] || op[choose[pa]][2] > v) choose[pa] = i;
            if (!choose[pb] || op[choose[pb]][2] > v) choose[pb] = i;
        }
        for (int i = 1; i <= n; i++)
        {
            if (find(i) != i) continue;
            if (choose[i]&&!used[choose[i]])
            {
                flag = true;
                auto t = op[choose[i]];
                cnt++, ans += t[2];
                used[choose[i]] = true;
                p[find(t[0])] = find(t[1]);
            }
        }
    }
    if (cnt == n - 1) cout << ans << "\n";
    else cout << "orz\n";
}
```

2.3 重链剖分

```cpp
void dfs1(int u, int father, int dep) //如果重复调用 要清空 son 调用 dfs1(root, -1, 1)
{
    sz[u] = 1, fa[u] = father, depth[u] = dep;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j == father) continue;
        dfs1(j, u, dep + 1);
        sz[u] += sz[j];
        if (sz[j] > sz[son[u]]) son[u] = j;
    }
}
void dfs2(int u, int t)//调用 dfs2(root, root)
{
    dfn[u] = ++cnt, nw[cnt] = w[u], top[u] = t; //把原来的权值赋值给新序列的权值
    if (!son[u]) return ;
    dfs2(son[u], t);
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
```

```
        if (j == fa[u] || j == son[u]) continue;
        dfs2(j, j);
    }
}
void modify_path(int u, int v, int k)//修改从 u 到 v 路径上的值
{
    while (top[u] != top[v])
    {
        if (depth[top[u]] < depth[top[v]]) swap(u, v);
        modify(1, dfn[top[u]], dfn[u], k);
        u = fa[top[u]];
    }
    if (depth[u] < depth[v]) swap(u, v);
    modify(1, dfn[v], dfn[u], k);
}
int query_path(int u, int v)//查询从 u 到 v 的路径上的和
{
    int res = 0;
    while (top[u] != top[v])
    {
        if (depth[top[u]] < depth[top[v]]) swap(u, v);
        res += query(1, dfn[top[u]], dfn[u]);
        u = fa[top[u]];
    }
    if (depth[u] < depth[v]) swap(u, v);
    res += query(1, dfn[v], dfn[u]);
    return res;
}
```

例子：洛谷模板题

```
int n, m, r, p, w[N], h[N], ne[M], e[M], idx;
int fa[N], son[N], dep[N], sz[N], dfn[N], top[N], nw[N], cnt;
struct Node
{
    int l, r, sum, add;
}tr[N * 4];
void dfs1(int u, int father, int depth)
{
    fa[u] = father, dep[u] = depth, sz[u] = 1;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j == father) continue;
        dfs1(j, u, depth + 1);
        sz[u] += sz[j];
        if (sz[son[u]] < sz[j]) son[u] = j;
    }
}
void dfs2(int u, int t)
{//注意是在第二次 dfs 的时候求 dfs 序
    dfn[u] = ++cnt, nw[cnt] = w[u], top[u] = t;
    if (!son[u]) return ;
    dfs2(son[u], t);
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j == fa[u] || j == son[u]) continue;
        dfs2(j, j);
    }
}
void modify_path(int u, int v, int k)
{
    while (top[u] != top[v])
    {
        if (dep[top[u]] < dep[top[v]]) swap(u, v);
        modify(1, dfn[top[u]], dfn[u], k);
        u = fa[top[u]];
    }
    if (dep[u] < dep[v]) swap(u, v);
    modify(1, dfn[v], dfn[u], k);
}
int query_path(int u, int v)
```

```
{
    int res = 0;
    while (top[u] ≠ top[v])
    {
        if (dep[top[u]] < dep[top[v]]) swap(u, v);
        res += query(1, dfn[top[u]], dfn[u]);
        u = fa[top[u]];
    }
    if (dep[u] < dep[v]) swap(u, v);
    res += query(1, dfn[v], dfn[u]);
    return res;
}
int main()
{
    for (int i = 0; i < n - 1; i++) add(a, b), add(b, a);
    dfs1(r, -1, 1), dfs2(r, r), build(1, 1, n);
    while (m--)
    {
        int t, x, y, z;
        cin >> t >> x;
        if (t == 1)
        {
            cin >> y >> z;
            modify_path(x, y, z);
        }
        else if (t == 2)
        {
            cin >> y;
            cout << query_path(x, y) % p <<"\n";
        }
        else if (t == 3)
        {
            cin >> z;
            modify_tree(x, z);
        }
        else cout << query_tree(x) % p <<"\n";
    }
}
```

树链剖分一般是维护点权, 假如是边权, 则需要将边权转化为点权

画一棵树观察一下, 可以发现, 对于每一条边的两端的节点, 只能选一个映射, 选择深度较深的点进行映射, 否则如果这个深度较浅的节点有多个儿子, 这些儿子都会映射到这个节点上, 是不允许的

```
void dfs1(int u, int father, int dep)
{
    fa[u] = father, depth[u] = dep, sz[u] = 1;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j == father) continue;
        v[j] = w[i]; //边权映射为点权
        dfs1(j, u, dep + 1);
        sz[u] += sz[j];
        if (sz[son[u]] < sz[j]) son[u] = j;
    }
}
void change(int k, int w)//修改第 k 条边
{
    int u = e[k * 2 - 1], v = e[k * 2 - 2];
    if (depth[u] < depth[v]) swap(u, v);
    Change(1, dfn[u], dfn[u], w);
}
void change_path(int u, int v, int w)
{
    while (top[u] ≠ top[v])
    {
        if (depth[top[u]] < depth[top[v]]) swap(u, v);
        Change(1, dfn[top[u]], dfn[u], w);
        u = fa[top[u]];
    }
    if (depth[u] < depth[v]) swap(u, v); //此时 v 的高度较高 v 是 u, v 两点的 lca
    Change(1, dfn[v] + 1, dfn[u], w); //lca 映射的边不是路径上的边 故修改 id[v] + 1
```

```
}
int query_path(int u, int v)
{
    int res = 0;
    while (top[u] ≠ top[v])
    {
        if (depth[top[u]] < depth[top[v]]) swap(u, v);
        res = max(res, query(1, dfn[top[u]], dfn[u]));
        u = fa[top[u]];
    }
    if (depth[u] < depth[v]) swap(u, v); //此时 v 的高度较高 v 是 u, v 两点的 lca
    res = max(res, query(1, dfn[v] + 1, dfn[u])); //lca 映射的边不是路径上的边 故修改 id[v] + 1
    return res;
}
```

例子： Qtree1

```
int h[N], ne[M], e[M], w[M], idx;
int fa[N], son[N], sz[N], depth[N], top[N], id[N], cnt, n, v[N], nv[N];
struct Node
{
    int l, r, max;
}tr[N << 2];
void dfs1(int u, int father, int dep)
{
    fa[u] = father, depth[u] = dep, sz[u] = 1;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j == father) continue;
        v[j] = w[i];
        dfs1(j, u, dep + 1);
        sz[u] += sz[j];
        if (sz[son[u]] < sz[j]) son[u] = j;
    }
}
void dfs2(int u, int t)
{
    dfn[u] = ++cnt, nv[cnt] = v[u], top[u] = t;
    if (!son[u]) return ;
    dfs2(son[u], t);
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j == fa[u] || j == son[u]) continue;
        dfs2(j, j);
    }
}
int query_path(int u, int v)
{
    int res = 0;
    while (top[u] ≠ top[v])
    {
        if (depth[top[u]] < depth[top[v]]) swap(u, v);
        res = max(res, query(1, dfn[top[u]], dfn[u]));
        u = fa[top[u]];
    }
    if (depth[u] < depth[v]) swap(u, v);
    res = max(res, query(1, dfn[v] + 1, dfn[u]));
    return res;
}
void modify_edge(int x, int t)
{
    int u = e[x * 2 - 1], v = e[x * 2 - 2];
    if (depth[u] < depth[v]) swap(u, v);
    modify(1, dfn[u], t);
}
int main()
{
    for (int i = 0; i < n - 1; i++) add(a, b, c), add(b, a, c);
    dfs1(1, -1, 1), dfs2(1, 1);
    build(1, 1, n);
    char op[10];
```

```
    while (cin >> op, *op≠'D')
    {
        int a, b;
        cin >> a >> b;
        if (*op=='C') modify_edge(a, b);
        else
        {
            if (a == b) cout <<"0\n";
            else cout << query_path(a, b) <<"\n";
        }
    }
}
```

2.4 点分治

主要变化的就是 cal, getpath, get 三个函数

点分治就是求出所有儿子的子树路径, 然后路径组合答案

允许单独的子树路径, 也允许两个不同的子树之间的路径组合, 不允许同一子树的路径组合

可能需要用容斥算答案

例: 求树上距离在 k 以内的点对数量

```
int h[N], ne[M], e[M], w[M], idx, p[N], q[N], n, k;
bool st[N];
int get_size(int u, int fa)
{
    if (st[u]) return 0;
    int res = 1;
    for (int i = h[u]; ~i; i = ne[i])
        if (e[i] ≠ fa) res += get_size(e[i], u);
    return res;
}
int get_wc(int u, int fa, int tot, int & wc)//求重心 返回子树大小
{
    if (st[u]) return 0;
    int sum = 1, ms = 0;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j == fa) continue;
        int t = get_wc(j, u, tot, wc);
        ms = max(ms, t), sum += t;
    }
    ms = max(ms, tot - sum);
    if (ms <= tot / 2) wc = u;
    return sum;
}
void get_path(int u, int fa, int dist, int & qt)//得到路径信息
{
    if (st[u]) return ;
    q[qt++] = dist;
    for (int i = h[u]; ~i; i = ne[i])
        if (e[i] ≠ fa) get_path(e[i], u, dist + w[i], qt);
}
int get(int a[], int n)//算路径之间对答案的贡献
{
    sort(a, a + n);
    int res = 0;
    for (int i = n - 1, j = -1; i >= 0; i--)
    {
        while (j + 1 < i && a[j + 1] + a[i] <= k) j++;
        j = min(j, i - 1), res += j + 1;
    }
    return res;
}
int cal(int u)
{
    if (st[u]) return 0;
    int res = 0, pt = 0;
    get_wc(u, -1, get_size(u, -1), u);
    st[u] = true;
    for (int i = h[u]; ~i; i = ne[i])
    {
```

```
        int j = e[i], qt = 0;
        get_path(j, -1, w[i], qt);
        res -= get(q, qt);
        for (int t = 0; t < qt; t++)
        {
            if (q[t] <= k) res++;
            p[pt++] = q[t];
        }
    }
    res += get(p, pt);
    for (int i = h[u]; ~i; i = ne[i]) res += cal(e[i]);
    return res;
}
int main()
{
    for (int i = 0; i < n - 1; i++) add(a, b, c), add(b, a, c);
    cout << cal(0) <<"\n";
}
```

2.5 树上启发式合并

用途：处理子树统计问题，避免对每个节点重新统计整个子树。

核心思想：保留重儿子的贡献，重新计算轻儿子的贡献，利用重链剖分的性质优化时间复杂度。

关键技巧：先算轻儿子再清空，再算重儿子保留信息，最后补上轻儿子贡献。

时间复杂度：$O(n \log n)$，每个节点最多被计算 $\log n$ 次（重链层数）。

经典例题: 求树上以 i 为根的子树中颜色出现次数最多的编号和

考虑对每棵子树暴力去做，时间复杂度是 $O(n^2)$

对儿子处理 Info 的时候，是可以保留一个儿子的 Info 的，如果保留重儿子的 Info，时间复杂度是 $O(n \log n)$

具体流程是:

1. 先遍历轻儿子, 去算轻儿子的答案, 同时清空轻儿子的 Info
2. 遍历重儿子, 算重儿子的答案, 保留重儿子的 Info
3. 遍历所有轻儿子, 计算轻儿子的贡献带来的 Info
4. 加上自己这个点, 再算答案

```
int color[N], sz[N], son[N], res[N], cnt[N];
void dfs(int u, int fa)//先预处理出重儿子
{
    sz[u] = 1;
    for (int j:g[u])
    {
        if (j == fa) continue;
        dfs(j, u);
        sz[u] += sz[j];
        if (sz[son[u]] < sz[j]) son[u] = j;
    }
}
array<int, 2> ans;
void add(int u)//加入一个点 Info 的变化 对答案的影响
{
    cnt[color[u]]++;
    if (cnt[color[u]] > ans[0]) ans = {cnt[color[u]], color[u]};
    else if (cnt[color[u]] == ans[0]) ans[1] += color[u];
}
void del(int u)//删除一个点 Info 的变化 一般来说答案直接清空 不会影响答案
{
    cnt[color[u]]--;
}
void add_tree(int u, int fa)//遍历加点
{
    add(u);
    for (int j:g[u]) if (j != fa) add_tree(j, u);
}
void del_tree(int u, int fa)//遍历删点
{
    del(u);
    for (int j:g[u]) if (j != fa) del_tree(j, u);
}
void dfs(int u, int fa, int ty)//ty 是类型 0 代表轻儿子 1 代表重儿子
{
    for (int j:g[u]) if (j != fa && j != son[u]) dfs(j, u, 0); //先遍历轻儿子
```

```
    if (son[u]) dfs(son[u], u, 1); //再遍历重儿子
    for (int j:g[u]) if (j ≠ fa && j ≠ son[u]) add_tree(j, u); //加入轻儿子贡献
    add(u); //加入自己的贡献
    res[u] = ans[1]; //算答案
    if (!ty)//如果自己是轻儿子 清空 Info 和答案
    {
        ans = {0, 0};
        del_tree(u, fa);
    }
}
dfs(1, 0), dfs(1, 0, 1);
```

2.6 虚树

用途：多次询问树上少量关键点的问题，避免每次在整棵树上 DP。

核心思想：只保留关键点和它们的 LCA 构成的必要结构，大大减少 DP 的状态数。

构建步骤：1) 按 DFS 序排序关键点 2) 加入相邻关键点的 LCA 3) 去重并按 DFS 序连边。

时间复杂度：建树 $O(k \log k)$，DP $O(k)$，其中 k 是关键点数。

虚树一般都是在树上做 dp, 然后给你多次询问, 每次询问都会给你不多的关键点, 故每次询问都建出虚树, 减小复杂度

建出的点有关键点，关键点之间的 LCA，根节点 (一般为 1)

如果关键点有重复, 要记得去重

```
void build(vector<int> v)//建虚树
{
    int n = v.size();
    sort(v.begin(), v.end(), [](int p1, int p2) { return dfn[p1] < dfn[p2]; });
    for (int i = 0; i + 1 < n; i++) v.push_back(lca(v[i], v[i + 1]));
    v.push_back(1);
    sort(v.begin(), v.end(), [](int p1, int p2) { return dfn[p1] < dfn[p2]; });
    v.erase(unique(v.begin(), v.end()), v.end());
    for (int i = 0; i + 1 < v.size(); i++) g[lca(v[i], v[i + 1])].push_back(v[i + 1]);
}
void clear(int u)//清空虚树
{
    for (int j:g[u]) clear(j);
    g[u].clear();
}
```

例题: 消耗战 (边有边权, 给出一些关键点, 要求断开 1 和这些关键点, 求最小代价)

```
int h[N >>1], ne[N << 1], e[N << 1], w[N << 1], idx;
int dfn[N], fa[N], son[N], sz[N], Top[N], depth[N], cnt;
int stk[N], top, st[N], f[N], val[N];
void dfs(int u)
{
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        dfs(j);
        if (st[j]) f[u] += w[i];
        else f[u] += min(f[j], (LL)w[i]);
    }
}
int solve(vector<int>& v)
{
    for (int t : v) st[t] = 1;
    sort(v.begin(), v.end(), [&](int x, int y) {return dfn[x] < dfn[y]; });
    top = 0;
    vector<int> clr;
    stk[++top] = 1, stk[++top] = v[0];
    clr.push_back(1), clr.push_back(v[0]);
    for (int i = 1; i < v.size(); i++)
    {
        int p = lca(stk[top], v[i]);
        clr.push_back(p), clr.push_back(v[i]);
        while (top > 1 && depth[stk[top - 1]] >= depth[p])
            add(stk[top - 1], stk[top], val[stk[top]]), top--;
        if (p ≠ stk[top]) add(p, stk[top], val[stk[top]]), stk[top] = p;
        stk[++top] = v[i];
    }
```

```
    while (top) add(stk[top - 1], stk[top], val[stk[top]]), top--;
    dfs(1);
    int ans = f[1];
    for (int t : clr) st[t] = 0, h[t] = -1, f[t] = 0;
    idx = 0;
    return ans;
}
void solve()
{
    val[1] = 1ll << 62;
    dfs1(1, 0, 1), dfs2(1, 1);
    while (m--) cout << solve(v)<<"\n";
}
```

有一棵有 n 个节点的树,q 次询问（询问互相独立）, 每次给定 k_i 个颜色, 每个颜色有一个起始点 v_j 和移动速度 s_j, 每一个颜色在每一次操作中会使它周围没有被染色的连通块上与它的距离不超过 s_j 的点全部染为这一个颜色, 每一轮中, 颜色从 1 到 k_i 依次开始操作, 一直到所有点全部被染色为止, 再询问 m_i 个关键点的颜色。

对关键点建出虚树, 考虑最短路, 把边权设置为 {轮数, 编号, 走的路程} 即可。

```
vector<array<int, 2>> g[N];
void build(vector<int> v)
{
    sort(v.begin(), v.end());
    v.erase(unique(v.begin(), v.end()), v.end());
    int n = v.size();
    sort(v.begin(), v.end(), [](int p1, int p2) { return dfn[p1] < dfn[p2]; });
    for (int i = 0; i + 1 < n; i++) v.push_back(lca(v[i], v[i + 1]));
    v.push_back(1);
    sort(v.begin(), v.end(), [](int p1, int p2) { return dfn[p1] < dfn[p2]; });
    v.erase(unique(v.begin(), v.end()), v.end());
    for (int i = 0; i + 1 < v.size(); i++)
    {
        int p = lca(v[i], v[i + 1]);
        g[p].push_back({v[i + 1], depth[v[i + 1]] - depth[p]});
        g[v[i + 1]].push_back({p, depth[v[i + 1]] - depth[p]});
    }
}
array<int, 3> dist[N]; //round id route
void dfs_(int u, int fa)
{
    dist[u] = {INF, INF, INF};
    for (auto [j, w]:g[u]) if (j != fa) dfs_(j, u);
}
bool chu[N], st[N];
void clear(int u, int fa)
{
    chu[u] = false;
    for (auto [j, w]:g[u]) if (j != fa) clear(j, u);
    g[u].clear();
}
int s[N], id[N];
priority_queue<array<int, 4>, vector<array<int, 4>>, greater<>> heap;
void addp(int u, int fa)
{
    st[u] = false;
    heap.push({dist[u][0], dist[u][1], dist[u][2], u});
    for (auto [j, w]:g[u]) if (j != fa) addp(j, u);
}
void dij()
{
    while (heap.size())
    {
        auto [round, ID, route, u] = heap.top();
        heap.pop();
        if (st[u] || round > N) continue;
        st[u] = true;
        for (auto [j, w]:g[u])
        {
            if (chu[j]) continue;
            array<int, 3> nd {};
            if (route + w <= s[ID]) nd = {round, ID, route + w};
            else
            {
```

```
                w- = (s[ID] - route);
                int t1 = (w + s[ID] - 1) / s[ID], t2 = (w - 1) % s[ID] + 1;
                nd = {round + t1, ID, t2};
            }
            if (nd < dist[j])
            {
                dist[j] = nd;
                heap.push({dist[j][0], dist[j][1], dist[j][2], j});
            }
        }
    }
}
void solve()
{
    while (q--)
    {
        int k, m;
        vector<int> v, op;
        for (int i = 1, x; i <= k; i++)
        {
            cin >> x >> s[i];
            op.push_back(x), v.push_back(x);
            id[x] = i;
        }
        vector<int> Q;
        while (m--)
        {
            int x;
            cin >> x;
            Q.push_back(x), v.push_back(x);
        }
        build(v), dfs_(1, 0);
        for (int x : op) chu[x] = true, dist[x] = {1, id[x], 0};
        addp(1, 0), dij();
        for (int x : Q) cout << dist[x][1] << " ";
        cout << "\n";
        clear(1, 0);
    }
}
```

2.7 笛卡尔树

```
struct DT
{
    int root, ls[N], rs[N], f[N][M], path[N][M];
    void init(int w[])
    {
        for (int i = 0; i < M; i++)
            for (int l = 0; l + (1 << i) - 1 <= n; l++)
                if (!i) f[l][i] = w[l], path[l][i] = l;
                else if (f[l][i - 1] >= f[l + (1 << (i - 1))][i - 1])
                    f[l][i] = f[l][i - 1], path[l][i] = path[l][i - 1];
                else
                    f[l][i] = f[l + (1 << (i - 1))][i - 1], path[l][i] = path[l + (1 << (i -
                    ↪  1))][i - 1];
    }
    int query(int l, int r)
    {
        int k = __lg(r - l + 1);
        if (f[l][k] >= f[r - (1 << k) + 1][k]) return path[l][k];
        else return path[r - (1 << k) + 1][k];
    }
    void build(int w[])
    {
        for (int i = 1; i <= n; i++) ls[i] = rs[i] = 0;
        init(w);
        root = build(1, n);
    }
    int build(int l, int r)
    {
        if (l > r) return 0;
        int x = query(l, r);
```

```
        ls[x] = build(l, x - 1), rs[x] = build(x + 1, r);
        return x;
    }
}dt;
```

2.8 树哈希

给定两棵树, 第一棵树有 n 个点, 第二棵树有 $n+1$ 个点, 已知第二棵树是第一棵树加上一个叶节点, 然后将节点的编号打乱得到的, 求这个多余的叶节点的编号

树哈希 + 换根, 求出每个点为根时的哈希值, 对于第二棵树枚举删去哪个叶子节点, 就在其父节点的哈希值中减去叶子的哈希值, 若第一棵树存在这个哈希值, 那么这个叶子节点符合条件

```
const ULL mask = chrono::steady_clock::now().time_since_epoch().count();
int h[N], ne[M], e[M], idx, d[N];
ULL f[N], v[N];
ULL shift(ULL x)//映射函数
{
    x ^= mask, x ^= x << 19, x ^= x >>27, x ^= x << 13, x ^= mask;
    return x;
}
void dfs1(int u, int fa)
{
    f[u] = 1;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j == fa) continue;
        dfs1(j, u);
        f[u] += shift(f[j]);
    }
}
void dfs2(int u, int fa)
{
    if (u != 1) v[u] = shift(f[fa] + v[fa] - shift(f[u])); //要特判一下不能是原来的根节点
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j == fa) continue;
        dfs2(j, u);
    }
}
void solve()
{
    for (int i = 0; i < n - 1; i++) add(a, b), add(b, a);
    dfs1(1, 0), dfs2(1, 0);
    set<ULL> S;
    for (int i = 1; i <= n; i++) S.insert(v[i] + f[i]);
    for (int i = 0; i < n; i++) add(a, b), add(b, a), d[a]++, d[b]++;
    dfs1(1, 0), dfs2(1, 0);
    for (int i = 1; i <= n + 1; i++)
        if (d[i] == 1)
        {
            int p = e[h[i]];
            if (S.find(f[p] + v[p] - shift(1)) != S.end())
            {
                cout << i << "\n";
                return ;
            }
        }
}
```

给你一棵 n 个节点的无根树, 现在要你定一个根节点, 满足其不同构的子树的数量最大。

两次 dfs, 第一次 dfs 先求出平凡情况, 第二次 dfs 利用换根计算这个点作为根时整棵树的贡献

```
ULL f[N], v[N];
map<ULL, int> mp;
int cnt, ans, maxcnt;
void dfs1(int u, int fa)
{
    f[u] = 1;
    for (int i = h[u]; ~i; i = ne[i])
```

```
    {
        int j = e[i];
        if (j == fa) continue;
        dfs1(j, u);
        f[u] += shift(f[j]);
    }
    if (mp[f[u]] == 0) cnt++;
    mp[f[u]]++;
}
void dfs2(int u, int fa)
{
    int t; //存一下父节点作为子树的贡献
    if (u != 1)
    {
        mp[v[fa] + f[fa]]--; //减去父节点为根
        if (mp[v[fa] + f[fa]] == 0) cnt--;
        t = v[u] = v[fa] + f[fa] - shift(f[u]); //存一下父节点作为子树的贡献
        if (mp[v[u]] == 0) cnt++;
        mp[v[u]]++; //加上父节点为子树
        mp[f[u]]--; //减去当前点为子树
        if (mp[f[u]] == 0) cnt--;
        v[u] = shift(v[u]);
        if (mp[f[u] + v[u]] == 0) cnt++;
        mp[f[u] + v[u]]++; //加上当前点为根
        if (cnt > maxcnt) maxcnt = cnt, ans = u;
    }
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j == fa) continue;
        dfs2(j, u);
    }
    if (u != 1)//回溯时清空操作
    {
        mp[v[fa] + f[fa]]++; //加上父节点为根
        if (mp[v[fa] + f[fa]] == 1) cnt++;
        if (mp[t] == 1) cnt--;
        mp[t]--; //减去父节点为子树
        mp[f[u]]++; //加上当前点为子树
        if (mp[f[u]] == 1) cnt++;
        if (mp[f[u] + v[u]] == 1) cnt--;
        mp[f[u] + v[u]]--; //减去当前点为根
    }
}
```

3 图论

3.1 拓扑排序

一般来说做 dp 的时候再开一个 pre 数组记录前驱进行 dp

若要求字典序最小/大的拓扑序, 则将队列换成小根堆/大根堆的优先队列, 通过归纳法可知每一步都取到了最优解

若要求 1 尽量靠前, 1 尽量靠前的情况下 2 尽量靠前 2 尽量靠前的情况下 3 尽量靠前…

对原图建反图, 求一个字典序最大的拓扑序, 拓扑序的反序即为所求, 因为尽量把大的都放在后面了, 留出前面的位置小的

注意拓扑 dp 的时候不一定是入队才能更新

3.2 最短路

3.2.1 Dijkstra

朴素 Dijkstra (邻接矩阵版本)

```
int g[N][N], dist[N], n;
bool vis[N];
void dijkstra(int start)
{
    memset(dist, 0x3f, sizeof dist);
    memset(vis, false, sizeof vis);
    dist[start] = 0;

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

```
        }
        vis[u] = true;

        for (int v = 1; v <= n; v++) {
            if (g[u][v] != INF) {
                dist[v] = min(dist[v], dist[u] + g[u][v]);
            }
        }
    }
}
```

最短路径树

最短路径树可以用来描述从源点 S 开始跑单源最短路到每个点的最短路的路径。

常用在: 删边最短路, 对边进行微调求最短路。

例子: CF545E 求边权和最小的最短路径树并输出树边

```
int h[N], w[M], ne[M], e[M], idx, n, m, start, pre[N], dist[N];
bool vis[N];
vector<int> tree_edges;
void dijkstra(int src)
{
    memset(dist, 0x3f, sizeof dist);
    memset(vis, false, sizeof vis);
    dist[src] = 0;

    priority_queue<PII, vector<PII>, greater<PII>> pq;
    pq.push({0, src});

    while (!pq.empty()) {
        int d = pq.top().first, u = pq.top().second;
        pq.pop();

        if (vis[u]) continue;
        vis[u] = true;

        if (u != src) tree_edges.push_back(pre[u]);

        for (int i = h[u]; ~i; i = ne[i]) {
            int v = e[i];
            if (dist[v] > dist[u] + w[i]) {
                dist[v] = dist[u] + w[i];
                pre[v] = i;
                pq.push({dist[v], v});
            }
        }
    }
}
void solve()
{
    while (m--) add(a, b, c), add(b, a, c);
    cin >> root;
    dijkstra();
    int v = 0;
    for (int t : ans) v += w[t];
    cout << v <<"\n";
    for (int t : ans) cout << t / 2 + 1 << " ";
}
```

例题: 每次修改一条边的权值, 求 1-n 的最短路 (询问之间独立)

```
int h[N], ne[N << 1], e[N << 1], w[N << 1], idx;
int n, m, q, cnt;
int dist[2][N], path[N], L[N], R[N], edgeid[N];
bool st[N], on_path[N];
array<int, 3> edge[N];
struct Node
{
    int l, r, minv, tag;
}tr[N << 2];
void dijkstra(int dist[], int s, int ty)
{
```

```
    for (int i = 1; i <= n; i++) dist[i] = INF;
    memset(st, false, sizeof st);
    dist[s] = 0;
    priority_queue<PII, vector<PII>, greater<PII>> heap;
    heap.push({0, s});
    while (heap.size())
    {
        auto [d, t] = heap.top();
        heap.pop();
        if (st[t]) continue;
        st[t] = true;
        for (int i = h[t]; ~i; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                path[j] = i;
                if (ty == 2)
                {
                    if (s == 1&&!on_path[j]) L[j] = L[t];
                    if (s == n&&!on_path[j]) R[j] = R[t];
                }
                heap.push({dist[j], j});
            }
        }
    }
}
void trace()
{
    for (int i = 1; i != n;)
    {
        on_path[i] = true;
        L[i] = cnt++, R[i] = cnt;
        int edge = path[i], from = e[edge ^ 1];
        edgeid[edge / 2 + 1] = cnt;
        i = from;
    }
    on_path[n] = true, L[n] = cnt++, R[n] = cnt;
}
void solve()
{
    for (int i = 1; i <= m; i++)
    {
        edge[i] = {a, b, c};
        add(a, b, c), add(b, a, c);
    }
    dijkstra(dist[1], n, 1);
    trace();
    dijkstra(dist[0], 1, 2), dijkstra(dist[1], n, 2);
    build(1, 1, cnt);
    for (int i = 1; i <= m; i++)
        if (!edgeid[i])
        {
            auto [a, b, c] = edge[i];
            modify(1, L[a] + 1, R[b] - 1, c + dist[0][a] + dist[1][b]);
            modify(1, L[b] + 1, R[a] - 1, c + dist[1][a] + dist[0][b]);
        }
    while (q--)
    {
        int id, v;
        auto [a, b, c] = edge[id];
        if (edgeid[id])
        {
            if (v < c) cout << min(dist[0][a] + dist[1][b], dist[1][a] + dist[0][b]) +
            ↪  v<<"\n";
            else cout << min(dist[0][n] - c + v, query(1, edgeid[id]))<<"\n";
        }
        else
        {
            if (v > c) cout << dist[0][n]<<"\n";
            else cout << min({dist[0][n], dist[1][a] + dist[0][b] + v, dist[0][a] + dist[1][b]
            ↪  + v})<<"\n";
        }
```

```
    }
}
```

3.2.2 Bellman-Ford

```
const int INF = 0x3f3f3f3f;
int n, m, dist[N];
struct Edge { int a, b, w; } edges[M];
int bellman_ford()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
        {
            int a = edges[j].a, b = edges[j].b, w = edges[j].w;
            dist[b] = min(dist[b], dist[a] + w);
        }
    if (dist[n] > INF / 2) return - 1;
    else return dist[n];
}
```

3.2.3 SPFA

SPFA 求最短路

```
int n, h[N], w[N], e[N], ne[N], idx, dist[N];
bool st[N];
void spfa()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    queue<int> q;
    q.push(1);
    while (q.size())
    {
        int t = q.front();
        q.pop();
        st[t] = false;
        for (int i = h[t]; ~i; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                if (!st[j]) q.push(j), st[j] = true;
            }
        }
    }
}
```

SPFA 判断负环

```
int n, h[N], w[N], e[N], ne[N], idx, dist[N], cnt[N];
bool st[N];
bool spfa()
{
    queue<int> q;
    for (int i = 1; i <= n; i++) q.push(i);
    memset(st, true, sizeof st);
    while (q.size())
    {
        int t = q.front();
        q.pop();
        st[t] = false;
        for (int i = h[t]; ~i; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > dist[t] + w[i])
            {
```

```
                dist[j] = dist[t] + w[i];
                cnt[j] = max(cnt[j], cnt[t] + 1);
                if (cnt[j] >= n + 1) return true;
                if (!st[j]) q.push(j), st[j] = true;
            }
        }
    }
    return false;
}
```

3.2.4 Floyd

Floyd 求最小环 (输出最小环的长度)

```
int g[N][N], d[N][N], n, m, res = INF; //g 为原图的边 d 为最短路数组
void floyd()
{
    memset(g, 0x3f, sizeof g);
    for (int i = 1; i <= n; i++) g[i][i] = 0;
    while (m--)
    {
        int a, b, c;
        g[a][b] = g[b][a] = min(g[a][b], c);
    }
    memcpy(d, g, sizeof g);
    for (int k = 1; k <= n; k++)//假设 k 点是环中的编号最大的点
    {
        for (int i = 1; i < k; i++)
            for (int j = i + 1; j < k; j++)
                if (g[i][k] + g[k][j] + d[j][i] < res)//这里可能爆 int 要开 longlong 或者移项
                    res = g[i][k] + g[k][j] + d[j][i];
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
    }
    if (res == 1e9) cout <<"No solution.\n";
    else cout << res;
}
```

Floyd 倍增

实际上 Floyd 的转移就是一个矩阵乘法, 可以根据题目不同的特性构造矩阵乘法, 用矩阵快速幂解决

例题: 一个正边权有向图, 最多可以选择 k 次使得使得通过下一条道路时, 需要的费用变为原来的相反数, 求 $1 \to n$ 最短路

首先用 Floyd 算法求出两两之间最短路, 然后在 m 条边中枚举一条边修改, 得到一个矩阵 M 。其中 $M_{i,j}$ 表示从 i 出发到 j 最多修改 1 次道路费用的最短路。

重新定义矩阵乘法 $C = A \times B$ 为 $C_{i,j} = \min_k\{A_{i,k} + B_{k,j}\}$, 在这种定义下, $M \times M$ 就是任意点到任意点最多修改两次道路费用的最短路。求出 M^k 即可。该运算满足结合律，可以用快速幂优化。时间复杂度为 $O(n^3 + n^2m + n^3 \log k)$。

注意矩阵乘法的时候不需要的点初始化为幺元，否则出现 $(1, 0)$*$(0, 4)$->$(1, 4)$ 的非法转移

```
template<int row, int col>
struct Matrix
{
    int r, c, ele[row][col];
    Matrix():r(row), c(col) {}
    int & operator()(int a, int b) { return ele[a][b]; }
};
template<int m, int n, int p>
auto operator * (Matrix<m, n> m1, Matrix<n, p> m2)
{
    Matrix<m, p> res;
    memset(res.ele, 0x3f, sizeof res.ele);
    for (int i = 0; i < m; i++)
        for (int k = 0; k < n; k++)
            for (int j = 0; j < p; j++)
                res(i, j) = min(res(i, j), m1(i, k) + m2(k, j));
    return res;
}
int n, m, k, g[N][N];
array<int, 3> edge[N * 25];
Matrix<N, N> P;
void solve()
```

```
{
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) P(i, j) = INF;
    cin >> n >> m >> k;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            g[i][j] = i == j?0:INF;
    for (int i = 1, a, b, w; i <= m; i++)
    {
        cin >> a >> b >> w;
        edge[i] = {a, b, w};
        g[a][b] = w;
    }
    for (int k = 1; k <= n; k++)
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                g[i][j] = min(g[i][j], g[i][k] + g[k][j]);
    if (!k)
    {
        cout << g[1][n]<<"\n";
        return ;
    }
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
        {
            P(i, j) = g[i][j];
            for (int k = 1; k <= m; k++)
            {
                auto [a, b, w] = edge[k];
                P(i, j) = min(P(i, j), g[i][a] - w + g[b][j]);
            }
        }
    Matrix<N, N> res = P;
    k--;
    while (k)
    {
        if (k & 1) res = res * P;
        k>>=1;
        P = P * P;
    }
    cout << res(1, n)<<"\n";
}
```

3.2.5 分层图最短路

求 $1 \to n$ 的最短路。(允许走 k 条免费边 $(k \leqslant 10)$)

建出 $k+1$ 层的分层图, 每层图都正常连边, 层与层之间连边, 代价为 0。从第 i 层走到第 $i+1$ 层的含义是用了一条免费边。要注意的是, 答案不一定在最后一层, 因为不一定需要走 k 条免费边。

3.2.6 同余最短路

给定 $n, a_{1\ldots n}, l, r$, 求出有多少 $b \in [l, r]$ 可以使等式 $\sum_{i=1}^{n} a_i x_i = b$ 存在非负整数解

经典前缀和转化问题, 将问题转化为有多少 $b \in [1, x]$ 可以使等式成立。

首先将 a 数组排序, 然后在 a_1 的剩余系下求解。

对于 $i \in [0, a_1 - 1]$, 枚举 $j (j \geq 2)$, 连边 $i \to (i + a_j) \bmod a_1$, 边权为 a_j, 意义是 i 这个剩余系下花费 a_j 的代价就可以转移到 $(i + a_j) \bmod a_1$ 这个剩余系。

最后 $dist_i$ 的意义是到达 i 这个剩余系所花费的最小代价。

对于 $x \equiv i \pmod{a_1} \wedge x \geq dist_i$ 的 x 都能够被 $a_{1\ldots n}$ 利用非负正数组 $\{x_{1\ldots n}\}$ 表示出来的。

$$ans = \sum_{i=0}^{a_1-1} (\frac{x - dist_i}{a_1} + 1)[x \geqslant dist_i]$$

```
void dijkstra()
{
    memset(dist, 0x3f, sizeof dist);
    dist[0] = 0;
    priority_queue<PII, vector<PII>, greater<PII>> heap;
    heap.push({0, 0});
    while (heap.size())
    {
        auto [d, t] = heap.top();
```

```
        heap.pop();
        if (st[t]) continue;
        st[t] = true;
        for (int i = h[t]; ~i; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                heap.push({dist[j], j});
            }
        }
    }
}
int cal(int n)
{
    int res = 0;
    for (int i = 0; i < a[1]; i++)
        if (n >= dist[i]) res += (n - dist[i]) / a[1] + 1;
    return res;
}
void solve()
{
    cin >> n >> l >> r;
    int cnt = 0;
    for (int i = 1; i <= n; i++)
    {
        int x;
        if (x) a[++cnt] = x;
    }
    n = cnt;
    sort(a + 1, a + 1 + n);
    for (int i = 0; i < a[1]; i++)
        for (int j = 2; j <= n; j++)
            add(i, (i + a[j]) % a[1], a[j]);
    dijkstra();
    cout << cal(r) - cal(l - 1)<<"\n";
}
```

3.3 差分约束

(1) 求不等式组的可行解:

1. 将每个不等式 $x_i \leqslant x_j + c$ 转化为一条从 x_j 走到 x_i 且长度为 c 的边 (求最短路)
2. 建立一个超级源点, 使得该源点一定可以遍历到所有边 (每条不等式都要符合)
3. 若图中出现负环, 则原不等式组无解; 否则 $dist_i$ 就是原不等式组的一个可行解

(2) 如何求最大值或最小值:

如果求最小值, 则求最长路; 如果求最大值, 则求最短路:

将形如 $x_i \leqslant c(c$ 为常数) 的不等式转化为从超级源点向 x_i 连一条长度为 c 的边再按照上述方法做即可

将题目的条件转化为不等式关系

特别地:$a = b \Leftrightarrow a \leqslant b \wedge a \geqslant b$

```
int main()
{
    while (m--)
    {
        int k, a, b;
        if (k == 1) add(a, b, 0), add(b, a, 0); //a 和 b 要一样
        else if (k == 2) add(a, b, 1); //a 必须少于 b
        else if (k == 3) add(b, a, 0); //a 必须不少于 b
        else if (k == 4) add(b, a, 1); //a 必须多于 b
        else add(a, b, 0); //a 必须不多于 b
    }
    for (int i = 1; i <= n; i++) add(0, i, 1); //每个数都至少为 1 故连边
    if (!spfa()) cout <<"-1\n"; // 如果出现正环则 spfa 返回 false
    else//最短路是负环 最长路是正环
    {
        int sum = 0;
        for (int i = 1; i <= n; i++) sum += dist[i];
        cout << sum;
    }
}
```

(3) 差分约束的优化:
玄学优化: 若 spfa 的总入队次数大于 $k \times n$(k 为一个较小的常数如 10 左右) 则说明图中很有可能存在负环
若图中的边都为非正/非负边可以先利用 tarjan 缩点判断是否存在正环/负环再在拓扑图上做 dp 递推 (即可保证线性的复杂度)

```
for (int i = 0; i <= n; i++) if (!dfn[i]) tarjan(i);
for (int i = 0; i <= n; i++)
    for (int j = h[i]; ~j; j = ne[j])
    {
        int k = e[j], a = id[i], b = id[k];
        if (a != b) add(hs, a, b, w[j]);
        else if (w[j])
        {
            cout <<"-1\n";
            return 0;
        }
    }
for (int i = scc_cnt; i >= 1; i--)
    for (int j = hs[i]; ~j; j = ne[j])
    {
        int k = e[j];
        f[k] = max(f[k], f[i] + w[j]);
    }
for (int i = 1; i <= scc_cnt; i++) ans += sz[i] * f[i];
```

有时可以利用一些转换, 避免图中出现负权边
例: 构造一个长度为 n 的 01 序列, 使得 1 的数量最少, 给出 m 个限制, 是 $[l, r]$ 中至少有 v 个 1
正常建边会有负权边, 但是假如把问题转换为 0 的数量最多, 就不会有负权边

```
int n, m, h[N], w[M], ne[M], e[M], idx, dist[N];
bool st[N];
void solve()
{
    for (int i = 1; i <= n; i++)
    {
        add(i, i - 1, 0); //s[i] + 0 >= s[i - 1]
        add(i - 1, i, 1); //s[i] <= 1 + s[i - 1]
    }
    while (m--)
    {
        int l, r, v;
        //l~r 的 1 至少为 v 个 l~r 的 0 至多为 r - l + 1 - v 个
        //s[r] <= r - l + 1 - v + s[l - 1]
        add(l - 1, r, r - l + 1 - v);
    }
    int S = N - 1;
    for (int i = 0; i <= n; i++) add(S, i, i); //前缀为 i 的 0 的个数不能超过 i 而且要让 S 连接到所有
    ↪ 点
    dijkstra(S);
    for (int i = 0; i <= n; i++) dist[i] = i - dist[i];
    for (int i = 1; i <= n; i++) cout << dist[i] - dist[i - 1]<<" ";
}
```

3.4 强连通分量

DAG 上的性质:
1. 问在 DAG 上最少选择多少个点能够使得从这些点出发可以到达所有点, 那么答案就是入度为 0 的点的个数。
2. 假设 DAG 上出度为 0 的结点有 a 个, 入度为 0 的结点有 b 个, 那么再加 $\max\{a, b\}$ 条边可以使得该有向图强连通。
强连通分量编号的逆序是一个拓扑序, 可以直接倒序枚举做拓扑操作

```
int h[N], hs[N], ne[M], e[M], idx;
int dfn[N], low[N], ts, stk[N], top, id[N], scc_cnt; //按 scc_cnt 递减的顺序是拓扑序
bool ins[N];
void tarjan(int u)
{
    dfn[u] = low[u] = ++ts;
    stk[++top] = u, ins[u] = true;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
```

```
        if (!dfn[j])
        {
            tarjan(j);
            low[u] = min(low[u], low[j]);
        }
        else if (ins[j]) low[u] = min(low[u], dfn[j]);
    }
    if (low[u] == dfn[u])
    {
        scc_cnt++;
        int y;
        do {
            y = stk[top--];
            ins[y] = false;
            id[y] = scc_cnt;
            //sz[scc_cnt]++; 像这样可以用该点的信息维护强连通分量的信息（如点数，权值）
        } while (y != u);
    }
} // 注意建新图的时候要注意需不需要判掉重边
void build_graph()//建新图
{
    for (int i = 1; i <= n; i++)
        for (int j = h[i]; ~j; j = ne[j])
        {
            int k = e[j], a = id[i], b = id[k];
            if (a != b) add(hs, a, b);
        }
}// for (int i = 1; i <= n; i++) if (!dfn[i]) tarjan(i);
```

3.5 边双连通分量

```
void tarjan(int u, int fa)
{
    dfn[u] = low[u] = ++ts, stk[++top] = u;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (!dfn[j])
        {
            tarjan(j, i);
            low[u] = min(low[u], low[j]);
        }
        else if (i != (fa ^ 1)) low[u] = min(low[u], dfn[j]);
    }
    if (dfn[u] == low[u])
    {
        dcc_cnt++;
        int y;
        do {
            y = stk[top--];
            dcc[dcc_cnt].push_back(y);
            id[y] = dcc_cnt;
        } while (y != u);
    }
}
```

给边双定向使得成为强连通

```
void dfs(int u, int fa)
{
    st[u] = true;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j == fa) continue;
        if (!use[i]&&!use[i ^ 1]) use[i ^ 1] = true;
        if (!st[j]) dfs(j, u);
    }
}
```

3.6 点双连通分量

点双一般用圆方树解决。同一个点有可能属于多个点双, 为了更好地刻画点双的性质, 引入圆方树解决点双问题。圆方树是一棵方点和圆点交替的一棵树。每一个方点代表着一个点双, 每一个圆点是原始的点。夹在若干个方点之间的圆点是割点, 否则若一个圆点是叶子, 则这个点不是割点

例题: 每次查询 a, b 两点间的必经点数量

点双的性质: 若两个点在同一个点双内, 则这两点间的简单路径不唯一。

对于该图, 建出一个圆方树后, 两点之间的必经点的数量, 即为割点的数量。即求在圆方树上两点的路径上有多少个圆点 (包括本身) 因为圆方树是圆方交替的, 很容易得到答案是 $\frac{dist_{a,b}}{2}+1$(两个圆点距离一定为偶数)

```cpp
vector<int> g[M];
int h[N], ne[M << 1], e[M << 1], idx;
int dfn[N], low[N], ts, tot;
int stk[N], tt; //处理完之后 栈可能有剩余的 但是可能不影响
int sz[M], fa[M], top[M], son[M], depth[M]; //注意圆方树点数开两倍
void tarjan(int u)
{
    dfn[u] = low[u] = ++ts, stk[++tt] = u;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (!dfn[j])
        {
            tarjan(j);
            low[u] = min(low[u], low[j]);
            if (low[j] >= dfn[u])
            {
                tot++;
                while (true)
                {
                    int x = stk[tt--];
                    g[tot].push_back(x);
                    if (x == j) break;
                }
                g[u].push_back(tot);
            }
        }
        else low[u] = min(low[u], dfn[j]);
    }
}
void solve()
{
    while (m--) add(a, b), add(b, a);
    tot = n; //记得对 tot 赋值 图可能不连通 是一个圆方树森林 注意处理
    tarjan(1), dfs(1, 0, 1), dfs(1, 1);
    while (q--)
    {
        int p = lca(a, b);
        cout << (depth[a] + depth[b] - 2 * depth[p]) / 2 + 1<<"\n";
    }
}
```

3.6.1 圆方树

点双

仙人掌和点双可以转化为圆方树解决, 可以很好刻画它们的性质

例题: 给定一张图 (保证连通), 每个点有点权。现在有两种操作:

1. $C\ a\ w$: 把 a 的点权改为 w
2. $A\ a\ b$: 询问从 a 到 b 的所有简单路径 (不经过重复点) 中, 点权最小的点的点权

当到达一个点双时, 一定存在一条简单路径, 从外部进入这个点双, 然后经过点双里面的权值最小点, 然后再走出这个点双。所以一个点双对答案的贡献必然是点双里面的最小权值

于是可以建立圆方树, 方点的权值为点双中的最小圆点权值。然后原图就变成了一棵树, 询问时就可以直接树剖套线段树求路径最小值了

一个圆点的权值变动, 可能会引发与之相连的方点权值变动 (当这个圆点是点双中的的最小权值点时会发生这件事情)。那么我们可以对每个方点维护一个 multiset, 里面存所有与之相邻的圆点权值, 然后权值就是 multiset 中的最小值, 每次修改就删掉原来的权值, 插入新的权值。然后我们每修改一个圆点的权值, 我们就遍历与之相邻的所有方点并按上述方法修改 multise

对于一个方点, multiset 里面存它所有子节点的权值。然后修改一个圆点时, 就只需要动它父亲的 multiset(它的父亲必然是一个方点)。询问时, 仍然可以正常询问, 只不过如果 lca 是一个方点, 那还要额外计算 fa_{lca} 的权值对答案的贡献

```
void tarjan(int u)
{
    dfn[u] = low[u] = ++ts;
    stk[++tt] = u;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (!dfn[j])
        {
            tarjan(j);
            low[u] = min(low[u], low[j]);
            if (low[j] >= dfn[u])
            {
                tot++;
                while (true)
                {
                    int x = stk[tt--];
                    g[tot].push_back(x);
                    if (x == j) break;
                }
                g[u].push_back(tot);
            }
        }
        else low[u] = min(low[u], dfn[j]);
    }
}
int w[N];
multiset<int> S[N];
void solve()
{
    memset(h, -1, sizeof h);
    cin >> n >> m >> q;
    for (int i = 1; i<= n; i++) cin >> w[i];
    while (m--)
    {
        int a, b;
        cin >> a >> b;
        add(a, b), add(b, a);
    }
    tot = n;
    tarjan(1);
    memset(dfn, 0, sizeof dfn);
    dfs(1, 0, 1), dfs(1, 1);
    tr.build(1, 1, tot);
    for (int i = 1; i <= n; i++)
    {
        tr.modify(1, dfn[i], w[i]);
        if (fa[i]) S[fa[i]].insert(w[i]);
    }
    for (int i = n + 1; i <= tot; i++)
    {
        S[i].insert(INF);
        tr.modify(1, dfn[i], *S[i].begin());
    }
    while (q--)
    {
        char opt;
        int a, b;
        cin >> opt >> a >> b;
        if (opt=='C')
        {
            tr.modify(1, dfn[a], b);
            if (fa[a])
            {
                int p = fa[a];
                S[p].extract(w[a]);
                w[a] = b;
                S[p].insert(w[a]);
                tr.modify(1, dfn[p], *S[p].begin());
            }
            else w[a] = b;
        }
        else
        {
```

```
            int p = lca(a, b), res = query_path(a, b);
            if (p > n && fa[p]) res = min(res, w[fa[p]]);
            cout << res << "\n";
        }
    }
}
```

仙人掌

```
//仙人掌转化为圆方树（有向树）
//任取一点作为根
//将环变形 选定环上离根最近的点（特殊点）从这个点向一个新建的方点连边（边权为 0）
//再从这个方点向环上其余点连边（边权为这个点到特殊点的最短距离）
//要求 x 和 y 的距离 先求 p = lca(x, y)
//当 p 是圆点时 满足 dist(x, y) = d[x] + d[y] - 2 * d[p]
//当 p 是方点时 x 和 y 必然在某一个环上相遇 找到 Xc 和 Yc(x 和 y 均差一步跳到 p)
//则 dist(x, y) = dist(x, Xc) + dist(y, Yc) + min(dist(Xc, Yc))
//转化为圆方树 边数不变 变成一棵树 所以最后点数 = 边数 + 1
int h1[N], h2[N], ne[N], e[N], w[N], idx;
int dfn[N], low[N], ts;
int s[N], stot[N], fa[N][15], fu[N], fw[N], fe[N];
int d[N], depth[N], U, V, nid, n, m, q;
void add(int h[], int a, int b, int c) { e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = ++idx; }
void build_circle(int x, int y, int z)
{
    int sum = z;
    for (int k = y; k != x; k = fu[k])
    {
        s[k] = sum;
        sum += fw[k];
    }
    s[x] = stot[x] = sum;
    add(h2, x, ++nid, 0);
    for (int k = y; k != x; k = fu[k])
    {
        stot[k] = sum;
        add(h2, nid, k, min(s[k], sum - s[k]));
    }
}
void tarjan(int u, int from)
{
    dfn[u] = low[u] = ++ts;
    for (int i = h1[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (!dfn[j])
        {
            fu[j] = u, fw[j] = w[i], fe[j] = i;
            tarjan(j, i);
            low[u] = min(low[u], low[j]);
            if (dfn[u] < low[j]) add(h2, u, j, w[i]);
        }
        else if (i != (from ^ 1)) low[u] = min(low[u], dfn[j]);
    }
    for (int i = h1[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (dfn[u] < dfn[j] && fe[j] != i)
            build_circle(u, j, w[i]);
    }
}
void dfs(int u, int father, int dep)
{
    fa[u][0] = father, depth[u] = dep;
    for (int k = 1; k < 15; k++)
        fa[u][k] = fa[fa[u][k - 1]][k - 1];
    for (int i = h2[u]; ~i; i = ne[i])
    {
        int j = e[i];
        d[j] = d[u] + w[i];
        dfs(j, u, dep + 1);
```

```
    }
}
int lca(int u, int v)
{
    if (depth[u] < depth[v]) swap(u, v);
    for (int k = 14; k >= 0; k--)
        if (depth[fa[u][k]] >= depth[v])
            u = fa[u][k];
    if (u == v) return u; //lca 一定是圆点
    for (int k = 14; k >= 0; k--)
        if (fa[u][k] != fa[v][k])
            u = fa[u][k], v = fa[v][k];
    U = u, V = v;
    return fa[u][0];
}
int query(int u, int v)
{
    int p = lca(u, v);
    if (p <= n) return d[u] + d[v] - 2 * d[p]; //p 是圆点
    int du = d[u] - d[U], dv = d[v] - d[V];
    int l = abs(s[U] - s[V]);
    return du + dv + min(l, stot[U] - l);
}
void solve()
{
    memset(h1, -1, sizeof h1);
    memset(h2, -1, sizeof h2);
    cin >> n >> m >> q;
    nid = n;
    while (m--)
    {
        int a, b, c;
        cin >> a >> b >> c;
        add(h1, a, b, c), add(h1, b, a, c);
    }
    tarjan(1, -1);
    dfs(1, 0, 1);
    while (q--)
    {
        int u, v;
        cin >> u >> v;
        cout << query(u, v)<<"\n";
    }
}
```

3.7 2-SAT

将 $a \vee b$ 转化为 $\neg a \to b$ 和 $\neg b \to a$(建边)

将 $a = 1$ 转化为 $a \vee a$, 再转化为 $\neg a \to a$(建边)

将 $a = 0$ 转化为 $\neg a \vee \neg a$, 再转化为 $a \to \neg a$(建边)

```
int h[N], ne[N], e[N], idx, dfn[N], low[N], ts, scc_cnt, id[N], stk[N], top;
bool ins[N];
void tarjan(int u)
{
    dfn[u] = low[u] = ++ts;
    stk[++top] = u, ins[u] = true;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (!dfn[j])
        {
            tarjan(j);
            low[u] = min(low[u], low[j]);
        }
        else if (ins[j]) low[u] = min(low[u], dfn[j]);
    }
    if (low[u] == dfn[u])
    {
        scc_cnt++;
        int y;
        do {
            y = stk[top--];
```

```
            ins[y] = false;
            id[y] = scc_cnt;
        } while (y != u);
    }
}
void solve()
{
    while (m--)//给出 i 为 a 或者 j 为 b 的条件 x 点为 false x + n 为 true
    {
        int i, a, j, b;
        if (!a&&!b) add(i + n, j), add(j + n, i);
        else if (!a && b) add(i + n, j + n), add(j, i);
        else if (a&&!b) add(i, j), add(j + n, i + n);
        else if (a && b) add(i, j + n), add(j, i + n);
    }
    for (int i = 1; i <= 2 * n; i++) if (!dfn[i]) tarjan(i);
    for (int i = 1; i <= n; i++)
        if (id[i] == id[i + n])//在同一个 scc 里说明能互相推导, 矛盾
        {
            cout << "IMPOSSIBLE\n";
            return ;
        }
    cout << "POSSIBLE\n";
    for (int i = 1; i <= n; i++)//用 tarjan, id 小说明拓扑序大, 取拓扑序大的
        if (id[i] < id[i + n]) cout << "0 ";
        else cout << "1 ";
}
```

前后缀连边优化建图 (一定要记得把空间开大)

```
const int N = 4000010, M = 8000010; //这题总边数为 6n + 2m
int h[N], ne[M], e[M], idx;
int dfn[N], low[N], ts, scc_cnt;
int id[N], stk[N], top;
bool ins[N];
void tarjan(int u)
{
    dfn[u] = low[u] = ++ts;
    stk[++top] = u, ins[u] = true;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (!dfn[j])
        {
            tarjan(j);
            low[u] = min(low[u], low[j]);
        }
        else if (ins[j]) low[u] = min(low[u], dfn[j]);
    }
    if (low[u] == dfn[u])
    {
        scc_cnt++;
        int y;
        do {
            y = stk[top--];
            ins[y] = false;
            id[y] = scc_cnt;
        } while (y != u);
    }
}
void solve()
{
    while (m--)
    {
        int a, b;
        add(a, b + n), add(b, a + n); //2m 条
    }
    int cnt = 2 * n;
    while (k--)
    {
        int w;
        vector<int> v(w), pre(w), suf(w);
        for (int i = 0; i<w; i++) cin >> v[i];
```

```
        for (int i = 0; i < w; i++)
        {
            pre[i] = ++cnt;
            add(pre[i], v[i]); //n 条 向反面连边
            if (i) add(pre[i], pre[i - 1]); //n 条 建立前缀关系
        }
        for (int i = w - 1; i >= 0; i--)
        {
            suf[i] = ++cnt;
            add(suf[i], v[i]); //n 条 向反面连边
            if (i != w - 1) add(suf[i], suf[i + 1]); //n 条 建立后缀关系
        }
        for (int i = 0; i < w; i++)
        {
            int p = i - 1, s = i + 1;
            if (p != -1) add(v[i] + n, pre[p]); //n 条
            if (s != w) add(v[i] + n, suf[s]); //n 条
        }
    }
    for (int i = 1; i <= 2 * n; i++) if (!dfn[i]) tarjan(i);
    for (int i = 1; i <= n; i++)
        if (id[i] == id[i + n])
        {
            cout << "NIE\n";
            return ;
        }
    cout << "TAK\n";
}
```

例:abc210f, 给你一些卡片, 卡片可以选择正反, 最后所有卡片两两互质, 求有无合法方案

```
const int N = 1000010, M = 2000010, K = M << 2; //点数和边数特别不好数 大概估计一下 尽量开大点
int h[N], ne[K], e[K], idx;
int dfn[N], low[N], ts, scc_cnt;
int id[N], stk[N], top;
bool ins[N];
int a[N], b[N];
vector<int> pr[M];
int primes[M], minp[M], pcnt;
void tarjan(int u)
{
    dfn[u] = low[u] = ++ts;
    stk[++top] = u, ins[u] = true;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (!dfn[j])
        {
            tarjan(j);
            low[u] = min(low[u], low[j]);
        }
        else if (ins[j]) low[u] = min(low[u], dfn[j]);
    }
    if (low[u] == dfn[u])
    {
        scc_cnt++;
        int y;
        do {
            y = stk[top--];
            ins[y] = false;
            id[y] = scc_cnt;
        } while (y != u);
    }
}
void get(int x, int id)
{
    while (x >1)
    {
        int p = minp[x];
        pr[p].push_back(id);
        while (x % p == 0) x /= p;
    }
}
```

```cpp
inline int dif(int x)//求卡片的反面
{
    if (x <= n) return x + n;
    else return x - n;
}
void solve()
{
    get_primes(M - 1);
    for (int i = 1; i <= n; i++)
    {
        cin >> a[i] >> b[i];
        get(a[i], i), get(b[i], i + n);
    }
    int cnt = 2 * n;
    for (int i = 2; i < M; i++)
    {
        if (!pr[i].size()) continue;
        int m = pr[i].size();
        vector<int> pre(m), suf(m);
        for (int j = 0; j < m; j++)
        {
            pre[j] = ++cnt;
            add(pre[j], dif(pr[i][j]));
            if (j) add(pre[j], pre[j - 1]);
        }
        for (int j = m - 1; j >= 0; j--)
        {
            suf[j] = ++cnt;
            add(suf[j], dif(pr[i][j]));
            if (j != m - 1) add(suf[j], suf[j + 1]);
        }
        for (int j = 0; j < m; j++)
        {
            int p = j - 1, s = j + 1;
            if (p != -1) add(pr[i][j], pre[p]);
            if (s != m) add(pr[i][j], suf[s]);
        }
    }
    for (int i = 1; i <= 2 * n; i++) if (!dfn[i]) tarjan(i);
    for (int i = 1; i <= n; i++)
        if (id[i] == id[i + n])
        {
            cout << "No\n";
            return ;
        }
    cout << "Yes\n";
}
```

例子： 给你一些无向边, 边是有颜色和权值的, 要求最小化选出最大的边的权值
并且选出的边是一个匹配, 对于同一种颜色, 未选出的边分别形成匹配

```cpp
int h[N], ne[M], pre[M], e[M], idx;
int dfn[N], low[N], ts, scc_cnt;
int id[N], stk[N], top;
bool ins[N];
vector<array<int, 4>> op;
int n, m;
map<int, int> color[N];
vector<int> group[N];
inline bool check(int mid)
{
    memset(h, -1, sizeof h);
    memset(dfn, 0, sizeof dfn);
    idx = ts = top = scc_cnt = 0;
    map<PII, int> edge;
    for (int i = 1; i <= n; i++) color[i].clear();
    int cnt = 0;
    for (int i = 1; i <= m; i++)
    {
        int a = op[i][0], b = op[i][1], c = op[i][2], t = op[i][3];
        if (t > mid) add(i + m, i);
        color[a][c]++, color[b][c]++;
        if (!edge.count({a, c})) edge[{a, c}] = ++cnt;
```

```
        group[edge[{a, c}]].push_back(i);
        if (!edge.count({b, c})) edge[{b, c}] = ++cnt;
        group[edge[{b, c}]].push_back(i);
    }
    int pcnt = 2 * m;
    for (int i = 1; i <= n; i++)
    {
        vector<int> all;
        for (auto [x, y]:color[i])
        {
            vector<int> part;
            for (auto t:group[edge[{i, x}]])
            {
                all.push_back(t);
                part.push_back(t);
            }
            int len = part.size(); //如果不选 其他都要选
            vector<int> pre(len), suf(len);
            for (int j = 0; j < len; j++)
            {
                pre[j] = ++pcnt;
                add(pre[j], part[j] + m);
                if (j) add(pre[j], pre[j - 1]);
            }
            for (int j = len - 1; j >= 0; j--)
            {
                suf[j] = ++pcnt;
                add(suf[j], part[j] + m);
                if (j != len - 1) add(suf[j], suf[j + 1]);
            }
            for (int j = 0; j < len; j++)
            {
                int p = j - 1, s = j + 1;
                if (p != -1) add(part[j], pre[p]);
                if (s != len) add(part[j], suf[s]);
            }
        }
        int len = all.size(); //如果选了 其他都不能选
        vector<int> pre(len), suf(len);
        for (int j = 0; j < len; j++)
        {
            pre[j] = ++pcnt;
            add(pre[j], all[j]);
            if (j) add(pre[j], pre[j - 1]);
        }
        for (int j = len - 1; j >= 0; j--)
        {
            suf[j] = ++pcnt;
            add(suf[j], all[j]);
            if (j != len - 1) add(suf[j], suf[j + 1]);
        }
        for (int j = 0; j < len; j++)
        {
            int p = j - 1, s = j + 1;
            if (p != -1) add(all[j] + m, pre[p]);
            if (s != len) add(all[j] + m, suf[s]);
        }
    }
    for (int i = 1; i <= cnt; i++) group[i].clear();
    for (int i = 1; i <= 2 * m; i++)
        if (!dfn[i]) tarjan(i);
    for (int i = 1; i <= m; i++)
        if (id[i] == id[i + m]) return false;
    return true;
}
void solve()
{
    op.push_back({0, 0, 0, 0});
    vector<int> val(m);
    for (int i = 0; i < m; i++)
    {
        op.push_back({a, b, c, d});
        val[i] = d;
    }
```

```
    val.push_back(0);
    sort(val.begin(), val.end());
    if (!check(val[m])) cout<<"No\n";
    else
    {
        cout << "Yes\n";
        int l = 0, r = m;
        while (l < r)
        {
            int mid = (l + r) >> 1;
            if (check(val[mid])) r = mid;
            else l = mid + 1;
        }
        l = val[l], check(l);
        vector<int> ans;
        for (int i = 1; i <= m; i++)
            if (id[i] > id[i + m]) ans.push_back(i);
        cout << l << " " << ans.size() << "\n";
        for (int t : ans) cout << t << " ";
    }
}
```

3.8 欧拉路径

欧拉路径: 经过连通图中所有边恰好一次的迹称为欧拉路径。

欧拉回路: 经过连通图中所有边恰好一次的回路称为欧拉回路。

欧拉路的问题一般都是一个点，有前后两个部位，使得自己的前面与别人的后面相连，自己的后面与别人的前面相连

一般来说，是找一个特殊的属性，对于同一个点，前面的属性和后面的属性连边

因为欧拉路一般都是 n 条边，所以这样会构造出 n 条边，合理

因为是同一个点，所以不能拆开，所以可以假想它们是两个部份，已经用一条边连上了．那么建的边可以使得，从一个点的前面走到一个点的后面，然后一个点的后面和一个点的前面，通过某种性质关联起来，使得他们成为一个点，所以现在相当于走到另一个点的前面，再通过上一个点一样的方式从前面走到后面

所以解决欧拉路看似是要在点之间连边，但是实质上是在点内连边，点与点之间是根据题目的性质联系在一起的

type=1 代表无向图, type=0 代表有向图

最后输出的走过的边的编号, 如果是无向图, 负数代表走的是给出的边的反向边

```
int h[N], ne[M], e[M], idx, ans[M >>1];
int n, m, type, din[N], dout[N], cnt;
bool used[M];
void dfs(int u)
{
    for (int & i = h[u]; ~i;)
    {
        if (used[i])
        {
            i = ne[i];
            continue;
        }
        used[i] = true;
        if (type == 1) used[i ^ 1] = true;
        int t;
        if (type == 1)
        {
            t = i / 2 + 1;
            if (i & 1) t = -t;
        }
        else t = i + 1;
        int j = e[i];
        i = ne[i];
        dfs(j);
        ans[++cnt] = t;
    }
}
int main()
{
    cin >> type;
    cin >> n >> m;
    for (int i = 0; i < m; i++)
    {
        add(a, b);
        if (type == 1) add(b, a);
```

```
        din[b]++, dout[a]++;
    }
    if (type == 1)
    {
        for (int i = 1; i <= n; i++)
            if (din[i] + dout[i] & 1)
            {
                cout <<"NO\n";
                return 0;
            }
    }
    else
    {
        for (int i = 1; i <= n; i++)
            if (din[i] != dout[i])
            {
                cout <<"NO\n";
                return 0;
            }
    }
    for (int i = 1; i <= n; i++)
        if (~h[i])
        {
            dfs(i);
            break;
        }
    if (cnt < m) cout <<"NO\n";
    else
    {
        cout <<"YES\n";
        for (int i = cnt; i; i--) cout << ans[i] <<" ";
    }
}
```

例题: 给你两个串集合, 都有 n 个长度为 m 的串。问是否存在方案将两个集合重新排列拼接, 使得他们循环同构。

先枚举循环同构的偏移量 x, 那么就是要求每个第一个集合中的串长度为 x 的后缀和第二个串中长度为 x 的前缀匹配, 第二个集合中长度为 n - x 的后缀和第一个集合中串长度为 n - x 的前缀匹配。将对应长度的前后缀看成点, 字符串看成边, 那么就是要求解欧拉回路。时间复杂度 $O(nm)$。

```
int h[N << 2], ne[N << 2], e[N << 2], idx, tot, din[N << 2], dout[N << 2];
bool used[N << 2];
void add(int a, int b) { e[idx] = b, ne[idx] = h[a], h[a] = ++idx, din[b]++, dout[a]++; }
void dfs(int u, vector<int>&a ans)
{
    for (int & i = h[u]; ~i;)
    {
        if (used[i])
        {
            i = ne[i];
            continue;
        }
        used[i] = true;
        int t = i + 1, j = e[i];
        i = ne[i];
        dfs(j, ans);
        ans.push_back(t);
    }
}
void solve()
{
    cin >> n >> m;
    for (int i = 1; i <= 2 * n; i++)
    {
        string s;
        cin >> s;
        hs[i].init(s);
    }
    for (int i = 0; i < m; i++)
    {
        map < array<int, 2>, int > pre, suf;
        tot = idx = 0;
        for (int j = 1; j <= n; j++)
        {
```

```
            auto v1 = hs[j].get(1, i);
            auto v2 = hs[j].get(i + 1, m);
            if (!pre.contains(v1)) pre[v1] = ++tot;
            if (!suf.contains(v2)) suf[v2] = ++tot;
            add(pre[v1], suf[v2]);
        }
        for (int j = n + 1; j <= n * 2; j++)
        {
            auto v1 = hs[j].get(1, m - i);
            auto v2 = hs[j].get(m - i + 1, m);
            if (!suf.contains(v1)) suf[v1] = ++tot;
            if (!pre.contains(v2)) pre[v2] = ++tot;
            add(suf[v1], pre[v2]);
        }
        bool ok = true;
        for (int j = 1; j <= tot; j++)
            if (din[j] != dout[j])
            {
                ok = false;
                break;
            }
        if (!ok)
        {
            for (int i = 1; i <= tot; i++) h[i] = -1, din[i] = dout[i] = 0;
            for (int i = 0; i <= idx; i++) used[i] = false;
            continue;
        }
        vector<int> ans;
        dfs(1, ans);
        if (ans.size() < 2 * n)
        {
            for (int i = 1; i <= tot; i++) h[i] = -1, din[i] = dout[i] = 0;
            for (int i = 0; i <= idx; i++) used[i] = false;
            continue;
        }
        reverse(ans.begin(), ans.end()); //注意找到的路径反过才是正确的欧拉环路
        vector<int> ans1, ans2;
        for (int x : ans)
            if (x <= n) ans1.push_back(x);
            else ans2.push_back(x - n);
        for (int x : ans1) cout << x << " ";
        cout << "\n";
        for (int x : ans2) cout << x << " ";
        cout << "\n";
        for (int i = 1; i <= tot; i++) h[i] = -1, din[i] = dout[i] = 0;
        for (int i = 0; i <= idx; i++) used[i] = false;
        return ;
    }
    cout<<"-1\n";
}
```

现在有一个长度为 n 的数组 a 和一个长度为 $n-1$ 的排列 p。分别构造四个数列 b,c,b',c', 其定义如下: $b_i = \min\{a_i, a_{i+1}\}$,$c_i = \max\{a_i, a_{i+1}\}$,$b'_i = b_{p_i}$,$c'_i = c_{p_i}$

如果没有任何一个可能的序列, 那么输出 -1。

若 $b_i > c_i$ 显然无解

否则题目相当于告诉你 b_i 和 c_i 相邻, 要求构造出一个欧拉路径

现给定序列 b' 和 c' , 求出任何一个合法的序列 a。

```
void dfs(int u)
{
    for (int & i = h[u]; ~i;)
    {
        if (used[i])
        {
            i = ne[i];
            continue;
        }
        used[i] = used[i ^ 1] = true;
        int t = i, j = e[i];
        i = ne[i];
        dfs(j);
        ans[++cnt] = t;
```

```
    }
}
int main()
{
    memset(h, -1, sizeof h);
    cin >> n;
    for (int i = 1; i<n; i++) cin >> b[i], lsh.push_back(b[i]);
    for (int i = 1; i<n; i++) cin >> c[i], lsh.push_back(c[i]);
    sort(lsh.begin(), lsh.end());
    lsh.erase(unique(lsh.begin(), lsh.end()), lsh.end());
    for (int i = 1; i < n; i++) b[i] = find(b[i]), c[i] = find(c[i]);
    for (int i = 1; i < n; i++)
    {
        if (b[i] > c[i])
        {
            cout <<"-1\n";
            return 0;
        }
        add(b[i], c[i]), add(c[i], b[i]);
        d[b[i]]++, d[c[i]]++;
    }
    int start = 0, y = 0;
    for (int i = 0; i < lsh.size(); i++)
        if (d[i] & 1)
        {
            y++;
            start = i;
        }
    if (y != 0 && y != 2) cout <<"-1\n";
    else
    {
        dfs(start);
        if (cnt < n - 1)
        {
            cout <<"-1\n";
            return 0;
        }
        cout << lsh[start] <<" ";
        for (int i = cnt; i; i--) cout << lsh[e[ans[i]]] <<" ";
    }
}
```

n 对珍珠由 n 条线所连起来, 共 $2n$ 颗。现在你可以在任意两个未被线连起来的珍珠之间连一条线, 共可连 n 条, 使得这 $2n$ 颗珍珠形成一个环。设一条线所连的两颗珍珠权值为 u, v, 则该线的权值为最大的整数 k 满足 $2^k | u \operatorname{xor} v$。如果 $u = v$, 则 $k = 20$。求所有新连的线的权值最小值的最大值并给出方案, 即 $2n$ 颗珍珠所形成的的环。

```
const int N = (1 << 20) + 5, INF = 2e9;
int h[N], ne[N << 1], e[N << 1], idx, n, din[N], dout[N], a[N], b[N];
bool used[N];
void add(int a, int b) { e[idx] = b, ne[idx] = h[a], h[a] = ++idx, dout[a]++, din[b]++; }
vector<int> path;
void dfs(int u)
{
    for (int & i = h[u]; ~i;)
    {
        if (used[i])
        {
            i = ne[i];
            continue;
        }
        used[i] = used[i ^ 1] = true;
        int t = i / 2 + 1;
        if (i & 1) t = -t;
        int j = e[i];
        i = ne[i];
        dfs(j);
        path.push_back(t);
    }
}
bool check(int mid)
{
    memset(din, 0, sizeof din), memset(dout, 0, sizeof dout);
    memset(h, -1, sizeof h), memset(used, false, sizeof used);
```

```
    idx = 0;
    for (int i = 1; i <= n; i++)
    {
        int x = a[i] % (1 << mid), y = b[i] % (1 << mid);
        add(x, y), add(y, x);
        din[y]++, din[x]++;
    }
    for (int i = 0; i < (1 << mid); i++)
        if (din[i] + dout[i] & 1) return false;
    path.clear();
    for (int i = 0; i < (1 << mid); i++)
        if (~h[i])
        {
            dfs(i);
            break;
        }
    return path.size() == n;
}
void solve()
{
    cin >> n;
    for (int i = 1; i<= n; i++) cin >> a[i] >> b[i];
    int l = 0, r = 20;
    while (l < r)
    {
        int mid = l + r + 1 >>1;
        if (check(mid)) l = mid;
        else r = mid - 1;
    }
    cout << l<<"\n";
    check(l);
    reverse(path.begin(), path.end());
    for (int x:path)
    {
        if (x < 0) cout << -2 * x <<" "<<-2 * x - 1<<" ";
        else cout << 2 * x - 1<<" "<<2 * x<<" ";
    }
    cout<<"\n";
}
```

3.9 无向图的三元环计数

```
vector<int> e[N];
int n, m, din[N], st[N];
void solve()
{
    vector<array<int, 2>> op;
    while (m--)
    {
        int a, b;
        op.push_back({a, b});
        din[a]++, din[b]++;
    }
    for (auto [a, b]:op)
    {
        if (din[a] > din[b]) swap(a, b);
        else if (din[a] == din[b] && a > b) swap(a, b);
        e[a].push_back(b);
    }
    int ans = 0;
    for (int u = 1; u <= n; u++)
    {
        for (auto v:e[u]) st[v] = u;
        for (auto v:e[u])
            for (auto w:e[v])
                if (st[w] == u) ans++;
    }
    cout << ans << "\n";
}
```

3.10 优化建图

3.10.1 线段树优化建图

边有: 点向点连边 点向区间连边 区间向点连边 三种类型, 求最短路

```cpp
const int N = 100010, INF = 1e18;
const int B = N << 2, K = N << 3, M = 2 * __lg(N) * N; //线段树大小（偏移量）图的点数 图的边数:2lg(n) * m
struct SgtG
{
    struct Node { int l, r; } tr[N << 2];
    int n, h[K], ne[M], e[M], w[M], idx, leaf[N], dist[K]; //节点对应的是 in 树中的哪个叶子
    void add(int a, int b, int c) { e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = ++idx; }
    void build(int u, int l, int r)//建的是 in 树 顺便规划好 out 树
    {
        tr[u] = {l, r};
        if (l == r) leaf[l] = u; //记录每个点对应的在 in 树上的叶子的坐标
        else
        {
            int mid = (l + r) >> 1;
            add(u, u << 1, 0), add(u, u << 1 | 1, 0); //in 树的父亲连儿子
            add(u * 2 + B, u + B, 0), add(u * 2 + 1 + B, u + B, 0); //out 树的儿子连父亲
            build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r); //正常递归建树
        }
    }
    void init(int _n)
    {
        n = _n, idx = 0, memset(h + 1, -1, n << 5), memset(h + 1 + B, -1, n << 5); //初始化 4 * n 和 4 * n + B
        build(1, 1, n);
        for (int i = 1; i <= n; i++) add(leaf[i], leaf[i] + B, 0); //in 树向 out 树连边
    }
    void add(int u, int l, int r, int x, int w, int tp)
    {
        if (l<= tr[u].l && r >= tr[u].r)
            if (tp) add(u + B, leaf[x], w); //区间向点连边 所以是 out 树的区间向 in 树的叶子连边
            else add(leaf[x] + B, u, w); //点向区间连边 所以是 out 树的叶子向 in 树的区间连边
        else
        {
            int mid = (tr[u].l + tr[u].r) >> 1;
            if (l <= mid) add(u << 1, l, r, x, w, tp);
            if (r > mid) add(u << 1 | 1, l, r, x, w, tp);
        }
    }
    void point_to_segment(int x, int l, int r, int v) { add(1, l, r, x, v, 0); }
    void segment_to_point(int l, int r, int x, int v) { add(1, l, r, x, v, 1); }
    bool st[K];
    void dijkstra(int s)
    {
        s = leaf[s];
        memset(dist + 1, 0x3f, n << 5), memset(dist + 1 + B, 0x3f, n << 5);
        memset(st + 1, false, n << 2), memset(st + 1 + B, false, n << 2);
        priority_queue<array<int, 2>, vector<array<int, 2>>, greater<>> heap;
        dist[s] = 0;
        heap.push({0, s});
        while (heap.size())
        {
            auto [d, t] = heap.top();
            heap.pop();
            if (st[t]) continue;
            st[t] = true;
            for (int i = h[t]; ~i; i = ne[i])
            {
                int j = e[i];
                if (dist[j] > dist[t] + w[i])
                {
                    dist[j] = dist[t] + w[i];
                    heap.push({dist[j], j});
                }
            }
        }
    }

    //区间向区间连边 建立两个虚点即可 注意边数和点数都要变 还要分配编号
```

```
    void segment_to_segment(int a, int b, int c, int d, int v)
    {
        idcnt++;
        segment_to_point(a, b, idcnt, 0);
        idcnt++;
        add(leaf[idcnt - 1] + B, leaf[idcnt], v);
        point_to_segment(idcnt, c, d, 0);
    }
    void init(int _n, int rn)
    {
        n = rn, idcnt = _n, idx = 0, memset(h + 1, -1, n << 5), memset(h + 1 + B, -1, n << 5);
        ↪ //初始化 4 * n 和 4 * n + B
        build(1, 1, n);
        for (int i = 1; i <= n; i++) add(leaf[i], leaf[i] + B, 0); //in 树向 out 树连边
    }
}graph;
void solve()
{
    int n, q, s;
    cin >> n >> q >> s;
    graph.init(n);
    while (q--)
    {
        int t, a, b, x, l, r, w;
        cin >> t;
        if (t == 1)
        {
            cin >> a >> b >> w; //点向点连边 为了方便可以理解为点向区间连边
            graph.point_to_segment(a, b, b, w);
        }
        else if (t == 2)
        {
            cin >> x >> l >> r >> w; //点向区间连边
            graph.point_to_segment(x, l, r, w);
        }
        else
        {
            cin >> x >> l >> r >> w; //区间向点连边
            graph.segment_to_point(l, r, x, w);
        }
    }
    graph.dijkstra(s);
    for (int i = 1; i <= n; i++)
        if (graph.dist[graph.leaf[i]] >= INF) cout<<"-1 ";
        else cout << graph.dist[graph.leaf[i]]<<" ";
}
```

给你一个无向图, 你需要求出一个点覆盖使得选出点编号之间的最小差距最大化。

形式化的, 如果你选出的点集为 $a_1, a_2, \ldots, a_k (a_1 < a_2 < \cdots < a_k)$, 那么最小差距即为 $\min\limits_{i=1}^{k-1} (a_{i+1} - a_i)$, 特别的, 当 $k = 1$ 时最小差距为 n。

二分之后用 2-SAT 解决即可, 注意选一个点的时候一个区间的点都不能选, 所以线段树优化建图。

```
const int N = 200010, B = N << 2, K = N << 3, M = 2 * __lg(N) * N;
struct SgtG
{
    struct Node { int l, r; } tr[N << 2];
    int n, h[K], ne[M], e[M], idx, leaf[N];
    void add(int a, int b) { e[idx] = b, ne[idx] = h[a], h[a] = ++idx; }
    void build(int u, int l, int r)
    {
        tr[u] = {l, r};
        if (l == r) leaf[l] = u;
        else
        {
            int mid = (l + r) >> 1;
            add(u, u << 1), add(u, u << 1 | 1);
            add(u * 2 + B, u + B), add(u * 2 + 1 + B, u + B);
            build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
        }
    }
    void init(int _n)
    {
```

```
        n = _n, idx = 0;
        memset(h + 1, -1, n << 4), memset(h + 1 + B, -1, n << 4);
        build(1, 1, n);
        for (int i = 1; i <= n; i++) add(leaf[i], leaf[i] + B);
    }
    void add(int u, int l, int r, int x)
    {
        if (l<= tr[u].l && r >= tr[u].r) add(leaf[x] + B, u);
        else
        {
            int mid = (tr[u].l + tr[u].r) >> 1;
            if (l <= mid) add(u << 1, l, r, x);
            if (r > mid) add(u << 1 | 1, l, r, x);
        }
    }
    void point_to_segment(int x, int l, int r) { add(1, l, r, x); }
}graph;
int n, m, dfn[K], low[K], stk[K], id[K], top, ts, scc_cnt;
bool ins[K];
array<int, 2> e[N];
void tarjan(int u)
{
    dfn[u] = low[u] = ++ts, stk[++top] = u, ins[u] = true;
    for (int i = graph.h[u]; ~i; i = graph.ne[i])
    {
        int j = graph.e[i];
        if (!dfn[j])
        {
            tarjan(j);
            low[u] = min(low[u], low[j]);
        }
        else if (ins[j]) low[u] = min(low[u], dfn[j]);
    }
    if (low[u] == dfn[u])
    {
        scc_cnt++;
        int y;
        do {
            y = stk[top--];
            ins[y] = false;
            id[y] = scc_cnt;
        } while (y != u);
    }
}
bool check(int mid)
{
    graph.init(2 * n), ts = scc_cnt = 0;
    memset(dfn + 1, 0, n << 5), memset(dfn + 1 + B, 0, n << 5);
    for (int i = 1; i <= m; i++)
    {
        auto [a, b] = e[i];
        graph.add(graph.leaf[a], graph.leaf[b + n]);
        graph.add(graph.leaf[b], graph.leaf[a + n]);
    }
    for (int i = 1; i <= n; i++)
    {
        if (i != 1) graph.point_to_segment(i + n, max(1, i - mid + 1), i - 1);
        if (i != n) graph.point_to_segment(i + n, i + 1, min(n, i + mid - 1));
    }
    for (int i = 1; i <= 2 * n; i++)
        if (!dfn[graph.leaf[i]]) tarjan(graph.leaf[i]);
    for (int i = 1; i <= n; i++)
        if (id[graph.leaf[i]] == id[graph.leaf[i + n]]) return false;
    return true;
}
void solve()
{
    cin >> n >> m;
    for (int i = 1; i<= m; i++) cin >> e[i][0] >> e[i][1];
    int l = 1, r = n;
    while (l < r)
    {
        int mid = l + r + 1 >>1;
        if (check(mid)) l = mid;
```

```
        else r = mid - 1;
    }
    cout << l<<"\n";
}
```

3.10.2 前后缀优化建图

某个点向一个区间的前缀/后缀连边时可以用前后缀优化建图, 如拓扑排序, 2-SAT, 详见 2—SAT 的前后缀优化建图。

3.10.3 ST 表优化建图

```
const int N = 100010, M = 17, K = N * M * 5;
struct ST_graph
{
    int in[N][M], out[N][M], n, node;
    int h[K], ne[K], e[K], w[K], idx;
    void add(int a, int b, int c) { e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = ++idx; }
    void init(int _n)
    {
        memset(h, -1, sizeof h);
        n = _n, node = n << 1;
        for (int i = 1; i <= n; i++) in[i][0] = i, out[i][0] = i + n, add(i, i + n, 0);
        for (int k = 1; k < M; k++)
            for (int i = 1; i + (1 << k) - 1 <= n; i++)
            {
                in[i][k] = ++node;
                add(node, in[i][k - 1], 0), add(node, in[i + (1 << k - 1)][k - 1], 0);
                out[i][k] = ++node;
                add(out[i][k - 1], node, 0), add(out[i + (1 << k - 1)][k - 1], node, 0);
            }
    }
    void add(int l1, int r1, int l2, int r2, int v)
    {
        int np = ++node, k1 = __lg(r1 - l1 + 1), k2 = __lg(r2 - l2 + 1);
        add(out[l1][k1], np, v), add(out[r1 - (1 << k1) + 1][k1], np, v);
        add(np, in[l2][k2], 0), add(np, in[r2 - (1 << k2) + 1][k2], 0);
    }
}st;
```

3.11 网络流/二分图

二分图的性质:

最大匹配数 = 最小点覆盖 = 总点数 - 最大独立集 = 总点数 - 最小路径覆盖

最小顶点覆盖是指最少的顶点数使得二分图 G 中的每条边都至少与其中一个点相关联, 二分图的最小顶点覆盖数 = 二分图的最大匹配数

最小路径覆盖也称为最小边覆盖, 是指用尽量少的不相交简单路径覆盖二分图中的所有顶点。二分图的最小路径覆盖数 =|V|-二分图的最大匹配数

最大独立集是指寻找一个点集, 使得其中任意两点在图中无对应边。对于一般图来说, 最大独立集是一个 NP 完全问题, 对于二分图来说最大独立集 =|V|-二分图的最大匹配数。

在网络流中, 给二分图赋予值, 二分图的性质可以扩展为:

最大流 = 最小割 = 最小权点覆盖

最大权独立集 = 总点权和 - 最小权点覆盖

3.11.1 二分图染色

```
int n, h[N], e[N], ne[N], idx, color[N]; //多测的时候 color 要清空
bool dfs(int u, int c)
{
    color[u] = c;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (!color[j])
        {
            if (!dfs(j, c ^ 3)) return false;
        }
        else if (color[j] == c) return false;
    }
    return true;
}
bool check()
```

```
{
    bool flag = true;
    for (int i = 1; i <= n; i++)
        if (!color[i])
            if (!dfs(i, 1))
            {
                flag = false;
                break;
            }
    return flag;
}
```

3.11.2 二分图最大匹配

匈牙利算法

```
int n1, n2, match[N], h[N], e[N], ne[N], idx;
bool st[N];
bool find(int x)
{
    for (int i = h[x]; ~i; i = ne[i])
    {
        int j = e[i];
        if (!st[j])
        {
            st[j] = true;
            if (match[j] == 0 || find(match[j]))
            {
                match[j] = x;
                return true;
            }
        }
    }
    return false;
}
int Match()
{
    int res = 0;
    for (int i = 1; i <= n; i++)
    {
        memset(st, false, sizeof st);
        if (find(i)) res++;
    }
    return res;
}
```

3.11.3 二分图博弈

仿照求解二分图最大匹配可行边必经边的思路, 将跑完 $Dinic$ 算法后的残量网络 G_0 进行 SCC 缩点。

定理: 左部点 u 是最大匹配的必经点, 当且仅当 u 是匹配点且 u 和源点 s 在 G_0 中不属于同一个 SCC。右部点同理, 当且仅当其为匹配点且和 T 不在一个 SCC 中。

```
int h[N], e[M], ne[M], f[M], idx, d[N], cur[N], n, m, S, T;
int dfn[N], low[N], ts, stk[N], top, id[N], scc_cnt;
bool ins[N], win[K][K];
char g[K][K];
int dx[] = {-1, 0, 1, 0}, dy[] = {0, 1, 0, -1};
void tarjan(int u)
{
    dfn[u] = low[u] = ++ts;
    stk[++top] = u, ins[u] = true;
    for (int i = h[u]; ~i; i = ne[i])
    {
        if (!f[i]) continue;
        int j = e[i];
        if (!dfn[j])
        {
            tarjan(j);
            low[u] = min(low[u], low[j]);
        }
        else if (ins[j]) low[u] = min(low[u], dfn[j]);
    }
```

```
    if (low[u] == dfn[u])
    {
        scc_cnt++;
        int y;
        do {
            y = stk[top--];
            ins[y] = false;
            id[y] = scc_cnt;
        } while (y != u);
    }
}
inline int gid(int x, int y) { return x * m + y; }
void solve()
{
    cin >> n >> m;
    S = n * m, T = n * m + 1;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
        {
            cin >> g[i][j];
            if (g[i][j]=='.')
            {
                if ((i + j) & 1) add(S, gid(i, j), 1);
                else add(gid(i, j), T, 1);
            }
        }
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            if (g[i][j]=='.')
                for (int k = 0; k < 4; k++)
                {
                    int x = i + dx[k], y = j + dy[k];
                    if (x < 0 || y<0 || x >= n || y >= m || g[x][y]=='#') continue;
                    if ((i + j) & 1) add(gid(i, j), gid(x, y), 1);
                    else add(gid(x, y), gid(i, j), 1);
                }
    int maxflow = dinic();
    for (int i = 0; i <= T; i++)
        if (!dfn[i]) tarjan(i);
    for (int i = 0; i < idx; i += 2)
    {
        if (f[i]) continue;
        int a = e[i ^ 1], b = e[i];
        if (a == S && id[S] != id[b])
        {
            int x = b / m, y = b % m;
            win[x][y] = true;
        }
        if (b == T && id[T] != id[a])
        {
            int x = a / m, y = a % m;
            win[x][y] = true;
        }
    }
    vector<array<int, 2>> ans;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            if (!win[i][j] && g[i][j]=='.') ans.push_back({i + 1, j + 1});
    if (ans.size() == 0) cout<<"LOSE\n";
    else
    {
        cout<<"WIN\n";
        for (auto [a, b]:ans) cout << a << " " << b << "\n";
    }
}
```

3.11.4 最大流

用途：求网络最大流，常用于二分图最大匹配、最小割等问题。

Dinic 算法：分层图 + 多路增广，时间复杂度 $O(n^2m)$，二分图上 $O(m\sqrt{n})$。

关键优化：当前弧优化（cur 数组）避免重复遍历无用边。

Dinic 算法 (利用 Dinic 跑二分图最大匹配, 可以优化到 $m\sqrt{n}$)

```cpp
int h[N], e[M], ne[M], f[M], idx, d[N], cur[N]; //cur 是当前弧优化
void add(int a, int b, int c)
{
    e[idx] = b, f[idx] = c, ne[idx] = h[a], h[a] = ++idx;
    e[idx] = a, f[idx] = 0, ne[idx] = h[b], h[b] = ++idx;
}
bool bfs()
{
    memset(d, -1, sizeof d);
    d[S] = 0, cur[S] = h[S];
    queue<int> q;
    q.push(S);
    while (q.size())
    {
        int t = q.front();
        q.pop();
        for (int i = h[t]; ~i; i = ne[i])
        {
            int j = e[i];
            if (d[j] == -1 && f[i])
            {
                d[j] = d[t] + 1;
                cur[j] = h[j];
                q.push(j);
                if (j == T) return true;
            }
        }
    }
    return false;
}
int find(int u, int limit)
{
    if (u == T) return limit;
    int flow = 0;
    for (int i = cur[u]; ~i && flow < limit; i = ne[i])
    {
        int j = e[i];
        cur[u] = i;
        if (d[j] == d[u] + 1 && f[i])
        {
            int t = find(j, min(f[i], limit - flow));
            if (!t) d[j] = -1;
            f[i] -= t, f[i ^ 1] += t, flow += t;
        }
    }
    return flow;
}
int dinic()
{
    int r = 0, flow;
    while (bfs()) while (flow = find(S, INF)) r += flow;
    return r;
}
```

3.11.5 无源汇上下界可行流

```cpp
int h[N], ne[M], e[M], f[M], l[M], d[N], cur[N], idx, n, m, S, T, C[N];
void add(int a, int b, int c, int d) { add(a, b, d - c), C[a] -= c, C[b] += c; }
void solve()
{
    S = 0, T = n + 1;
    for (int i = 0; i < m; i++)
    {
        int a, b, c, d;
        cin >> a >> b >> c >> d;
        l[i] = c;
        add(a, b, c, d);
        //add(a, b, d - c); //上界 - 下界
        //C[a] -= c, C[b] += c; //流量守恒的情况
    }
    int tot = 0;
    for (int i = 1; i <= n; i++)//用流量守恒平衡一下
```

```
        if (C[i] >0) add(S, i, C[i]), tot += C[i];
        else if (C[i] < 0) add(i, T, -C[i]);
    if (tot != dinic()) cout<<"NO\n";
    else
    {
        cout<<"YES\n";
        for (int i = 0; i < m; i++)
            cout << f[i << 1 | 1] + l[i]<<"\n"; //加上下界
    }
}
```

若是有源汇上下界可行流, 则正常建图就行, 源汇为 s 和
再新建一个 S, T 用来平衡流量
对于上下界可行流来说, S, T 都是用于平衡流量的, 方案和正常的方案找法一样

```
for (int i = 1; i <= n; i++)//用流量守恒平衡一下
    if (C[i] >0) add(S, i, C[i]), tot += C[i];
    else if (C[i] < 0) add(i, T, -C[i]);
```

3.11.6 有源汇上下界最大流

```
int h[N], ne[M], e[M], f[M], l[M], d[N], cur[N], C[N], idx;
int n, m, S, T;
void add(int a, int b, int c)
{
    e[idx] = b, f[idx] = c, ne[idx] = h[a], h[a] = ++idx;
    e[idx] = a, f[idx] = 0, ne[idx] = h[b], h[b] = ++idx;
}
void add(int a, int b, int c, int d) { add(a, b, d - c), C[a] -= c, C[b] += c; }
bool bfs()
{
    memset(d, -1, sizeof d);
    d[S] = 0, cur[S] = h[S];
    queue<int> q;
    q.push(S);
    while (q.size())
    {
        int t = q.front();
        q.pop();
        for (int i = h[t]; ~i; i = ne[i])
        {
            int j = e[i];
            if (d[j] == -1 && f[i])
            {
                d[j] = d[t] + 1;
                cur[j] = h[j];
                q.push(j);
                if (j == T) return true;
            }
        }
    }
    return false;
}
int find(int u, int limit)
{
    if (u == T) return limit;
    int flow = 0;
    for (int i = cur[u]; ~i && flow < limit; i = ne[i])
    {
        int j = e[i];
        cur[u] = i;
        if (d[j] == d[u] + 1 && f[i])
        {
            int t = find(j, min(f[i], limit - flow));
            if (!t) d[j] = -1;
            f[i] -= t, f[i ^ 1] += t, flow += t;
        }
    }
    return flow;
}
```

```
int dinic()
{
    int r = 0, flow;
    while (bfs()) while (flow = find(S, INF)) r += flow;
    return r;
}
void solve()
{
    int s, t;
    cin >> n >> m >> s >> t;
    S = 0, T = n + 1;
    while (m--)
    {
        int a, b, c, d;
        add(a, b, c, d);
    }
    int tot = 0;
    for (int i = 1; i <= n; i++)//这里是操作所有点 包括 s 和 t 一定要注意有没有包含 s,
        if (C[i] >0) add(S, i, C[i]), tot += C[i];
        else if (C[i] < 0) add(i, T, -C[i]);
    add(t, s, INF);
    if (dinic() < tot) cout<<"No Solution\n";
    else
    {
        int res = f[idx - 1]; //注意 f[idx - 1] =/= dinic() 不要乱写
        S = s, T = t;
        f[idx - 1] = f[idx - 2] = 0;
        cout << res + dinic()<<"\n";
    }
}
```

3.11.7 有源汇上下界最小流

大部分和最大流一样, 最后是退流

```
if (dinic() < tot) cout<<"No Solution\n";
else
{
    int res = f[idx - 1];
    S = t, T = s; //注意这里不一样!!!
    f[idx - 1] = f[idx - 2] = 0;
    cout << res - dinic()<<"\n"; //可行流 - 最大流
}
```

3.11.8 最小割

最小割 = 最大流 (最大流最小割定理)

最大流 = 最小割

最小割的求解: 从 S 出发, 沿着有流量的边走, 能走到的点就是 s 集, 走不到的点就是 t 集, 连接两个集合之间的边就是最小割中的边。

例题: 求解代价最小的割点集, 使得 S 和 T 不连通

拆点为入点和出点, 以点的代价相连, 原来的边边权设为正无穷, 求一次最大流, 即求一次最小割, 能得到最小割, 再通过 dfs 找到方案

```
void dfs(int u)
{
    st[u] = 1;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (!st[j] && f[i]) dfs(j);
    }
}
void solve()
{
    T += n;
    for (int i = 1; i <= n; i++)
    {
        int v;
        add(i, i + n, v);
    }
    while (m--)
```

```
    {
        int a, b;
        add(a + n, b, INF), add(b + n, a, INF);
    }
    dinic();
    dfs(S);
    for (int i = 1; i <= n; i++)
        if (st[i] ^ st[i + n]) cout << i<<" ";
}
```

3.11.9 最小费用流

整数版

```
int h[N], ne[M], e[M], f[M], w[M], idx, d[N], pre[N], incf[N], n, m, S, T;
bool st[N];
void add(int a, int b, int c, int d)
{
    e[idx] = b, f[idx] = c, w[idx] = d, ne[idx] = h[a], h[a] = ++idx;
    e[idx] = a, f[idx] = 0, w[idx] = -d, ne[idx] = h[b], h[b] = ++idx;
}
bool spfa()
{
    memset(d, 0x3f, sizeof d);
    queue<int> q;
    q.push(S);
    d[S] = 0, incf[S] = INF, incf[T] = 0;
    while (q.size())
    {
        int t = q.front();
        q.pop();
        st[t] = false;
        for (int i = h[t]; ~i; i = ne[i])
        {
            int j = e[i];
            if (d[j] > d[t] + w[i] && f[i])
            {
                d[j] = d[t] + w[i], pre[j] = i;
                incf[j] = min(incf[t], f[i]);
                if (!st[j]) q.push(j), st[j] = true;
            }
        }
    }
    return incf[T] >0;
}
void EK(int & flow, int & cost)
{
    flow = cost = 0;
    while (spfa())
    {
        int t = incf[T];
        flow += t, cost += t * d[T];
        for (int i = T; i != S; i = e[pre[i] ^ 1]) f[pre[i]] -= t, f[pre[i] ^ 1] += t;
    }
}
int main()
{
    while (m--) add(a, b, c, d);
    int flow, cost;
    EK(flow, cost);
    cout << flow <<" " << cost;
}
```

若花费为实数

```
int h[N], ne[M], e[M], f[M], idx, incf[N], pre[N], n, m, S, T;
bool st[N];
double d[N], w[M];
void add(int a, int b, int c, double d)
{
    e[idx] = b, f[idx] = c, w[idx] = d, ne[idx] = h[a], h[a] = ++idx;
    e[idx] = a, f[idx] = 0, w[idx] = -d, ne[idx] = h[b], h[b] = ++idx;
```

```
}
bool spfa()
{
    for (int i = 0; i < N; i++) d[i] = INF;
    queue<int> q;
    q.push(S);
    d[S] = 0, incf[S] = INF, incf[T] = 0;
    while (q.size())
    {
        int t = q.front();
        q.pop();
        st[t] = false;
        for (int i = h[t]; ~i; i = ne[i])
        {
            int j = e[i];
            if (d[j] > d[t] + w[i] && f[i])
            {
                d[j] = d[t] + w[i];
                pre[j] = i;
                incf[j] = min(incf[t], f[i]);
                if (!st[j])
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }
    return incf[T] >0;
}
void EK(int & flow, double & cost)
{
    flow = cost = 0;
    while (spfa())
    {
        int t = incf[T];
        flow += t, cost += t * d[T];
        for (int i = T; i ≠ S; i = e[pre[i] ^ 1]) f[pre[i]] -= t, f[pre[i] ^ 1] += t;
    }
}
```

3.11.10 有源汇上下界费用流

注意上下界网络流是强行让下界流的, 所以没有算下界的流量花费, 所以最后结果要加上每一个下界 × 单位流量。

3.11.11 KM 算法 (二分图最大权完美匹配)

第一行输出是最大权的匹配结果, 第二行是完美匹配下右部第 i 个点相匹配的左部点的编号

```
const int N = 510;
int e[N][N], A[N], B[N], slack[N], n, m, mch[N], pre[N], vis[N];
void bfs(int id)
{
    memset(vis, 0, sizeof vis), memset(slack, 0×3f, sizeof slack);
    int x = mch[0] = id, y = 0;
    while (true)
    {
        vis[y] = 1;
        int d = 1e18, _y = 0;
        for (int i = 1; i <= n; i++)
        {
            if (vis[i]) continue;
            LL D = A[x] + B[i] - e[x][i];
            if (D < slack[i]) slack[i] = D, pre[i] = y;
            if (slack[i] < d) d = slack[i], _y = i;
        }
        A[id] -= d;
        for (int i = 1; i <= n; i++)
        {
            if (vis[i]) B[i] += d, A[mch[i]] -= d;
            else slack[i] -= d;
        }
        if (!mch[y = _y]) break;
        x = mch[y];
```

```
    }
    while (y) mch[y] = mch[pre[y]], y = pre[y];
}
int main()
{
    cin >> n >> m;
    memset(e, 0×cf, sizeof e);
    for (int i = 1, y, c, h; i <= m; i++)
    {
        cin >> y >> c >> h;
        e[y][c] = h;
    }
    memset(A, 0×cf, sizeof A);
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            A[i] = max(A[i], e[i][j]);
    for (int i = 1; i <= n; i++) bfs(i);
    int ans = 0;
    for (int i = 1; i <= n; i++) ans += A[i] + B[i];
    cout << ans << "\n";
    for (int i = 1; i <= n; i++) cout << mch[i]<<" ";
}
```

$\frac{n}{2}$ 个选第一种权值,$\frac{n}{2}$ 选第二个权值, 求此情况下的最大权完美匹配

```
int w[2][N][N], e[N][N], A[N], B[N], slack[N], n, mch[N], pre[N], vis[N];
void bfs(int id)
{
    memset(vis, 0, sizeof vis), memset(slack, 0×3f, sizeof slack);
    int x = mch[0] = id, y = 0;
    while (true)
    {
        vis[y] = 1;
        int d = 2e9, _y = 0;
        for (int i = 1; i <= n; i++)
        {
            if (vis[i]) continue;
            int D = A[x] + B[i] - e[x][i];
            if (D < slack[i]) slack[i] = D, pre[i] = y;
            if (slack[i] < d) d = slack[i], _y = i;
        }
        A[id] -= d;
        for (int i = 1; i <= n; i++)
        {
            if (vis[i]) B[i] += d, A[mch[i]] -= d;
            else slack[i] -= d;
        }
        if (!mch[y = _y]) break;
        x = mch[y];
    }
    while (y) mch[y] = mch[pre[y]], y = pre[y];
}
void solve()
{
    cin >> n;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            cin >> w[0][i][j];
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            cin >> w[1][i][j];
    int ans = 0;
    vector<int> v;
    for (int i = 0; i < (1 << n); i++)
        if (__builtin_popcount(i) == n / 2) v.push_back(i);
    random_shuffle(v.begin(), v.end());
    for (int u = 0; u < v.size(); u++)
    {
        int k = v[u];
        memset(A, 0×cf, sizeof A); //注意这些要清空的
        memset(B, 0, sizeof B), memset(mch, 0, sizeof mch);
        for (int j = 1; j <= n; j++)
            for (int i = 0; i < n; i++)
```

```
                e[i + 1][j] = w[k >> i & 1][i + 1][j];
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                A[i] = max(A[i], e[i][j]);
        for (int i = 1; i <= n; i++) bfs(i);
        int res = 0;
        for (int i = 1; i <= n; i++) res += A[i] + B[i];
        ans = max(ans, res);
        if (u % 1000 == 0 && clock() >= 1.9 * CLOCKS_PER_SEC) break; //卡时
    }
    cout << ans << "\n";
}
```

3.11.12 网络流的有关模型

最大流 = 最小割

最小割的求解: 从 S 出发, 沿着有流量的边走, 能走到的点就是 s 集, 走不到的点就是 t 集, 连接两个集合之间的边就是最小割中的边。

例题: 求解代价最小的割点集, 使得 S 和 T 不连通

拆点为入点和出点, 以点的代价相连, 原来的边边权设为正无穷, 求一次最大流, 即求一次最小割, 能得到最小割, 再通过 dfs 找到方案

```
void dfs(int u)
{
    st[u] = 1;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (!st[j] && f[i]) dfs(j);
    }
}
void solve()
{
    T += n;
    for (int i = 1; i <= n; i++)
    {
        int v;
        add(i, i + n, v);
    }
    while (m--)
    {
        int a, b;
        add(a + n, b, INF), add(b + n, a, INF);
    }
    dinic();
    dfs(S);
    for (int i = 1; i <= n; i++)
        if (st[i] ^ st[i + n]) cout << i<<" ";
}
```

最小割的可行边和必须边

求出最大流后对于一条边来说显然有最小割 $\rightarrow$ 满流。考虑现有的满流边 $u \rightarrow v$ 被替代: 残流网络中有包含 $u \rightarrow v$ 的环, 让流沿着环流动一圈, 最大流不变, 但是该边不再满流。

故: 两个端点在同一 SCC 内的边必然总不是最小割 (即不可能为可行边或必须边) 将当前残流网络缩成 DAG, DAG 上的边才有可能成为最小割。在这些边里, 直接将 S 和 T 相连的边必须要割, 故这些边为必须边。对于其他边都能构造割与不割的方案, 则它们是可行边。

对于可行边, 可割可不割。割的构造: 把这条边左端点到 S 的路径钦定为 S 集合, 其余为 T 集合, 然后把所有 S, T 之间的边割断。不割的构造: 如果右端点不是 T , 把这条边右端点到 S 的路径钦定为 S 集合。否则左端点必然不是 S , 把这条边左端点到 T 的路径钦定为 T 集合即可。这样整条边总是被完整地包含在 S 或 T 集中。

具体实现, 需要先跑最大流, 然后 Tarjan 缩强连通分量, 条件是:

可行边: 两端不在一个强连通分量内。

必须边: 一端在 S 的分量内, 另一端在 T 的分量内。

```
void tarjan(int u)
{
    dfn[u] = low[u] = ++ts;
    stk[++top] = u, ins[u] = true;
    for (int i = h[u]; ~i; i = ne[i])
    {
```

```
        if (!f[i]) continue; //有流量才能走
        int j = e[i];
        if (!dfn[j])
        {
            tarjan(j);
            low[u] = min(low[u], low[j]);
        }
        else if (ins[j]) low[u] = min(low[u], dfn[j]);
    }
    if (low[u] == dfn[u])
    {
        scc_cnt++;
        int y;
        do {
            y = stk[top--];
            ins[y] = false;
            id[y] = scc_cnt;
        } while (y != u);
    }
}
void solve()
{
    dinic();
    for (int i = 1; i <= n; i++) if (!dfn[i]) tarjan(i);
    for (int i = 0; i < idx; i += 2)
    {
        int a = e[i ^ 1], b = e[i];
        if (f[i] || id[a] == id[b]) cout<<"0 0\n"; //不满流或者在一个 SCC 里必然不是最小割
        else
        {
            cout<<"1 "; //下面根据条件判断
            if (id[a] == id[S] && id[b] == id[T]) cout<<"1\n";
            else cout<<"0\n";
        }
    }
}
```

基本概念:
图的匹配: 选出某些边, 使得每两个边没有公共端点。
图的独立集: 选出某些点, 使得每两个点没有连边。
图的点覆盖: 选出某些点, 每条边都至少有一个端点被选择。
闭合子图: 有向图的子图, 满足没有指向子图外的出边。
图的路径覆盖: 在 DAG 上用 (边) 不相交的简单路径覆盖所有的节点。
二分图: 分为两个部分的图 (称为左部和右部), 同一个部分内没有边。
二分图判定: 充要条件: 没有奇环。dfs 染色判定即可。
二分图最大匹配: 二分图中边最多的匹配 (可能有多种方案)。
对一个二分图来说, 最小点覆盖 = 最大匹配数
对所有图来说, 最大独立集 = 总点数-最小点覆盖
团: 两两之间有连边
最大团 = 补图的最大独立集
Hall 定理
二分图完美匹配: 若两侧点集为 X, Y , 匹配数达到 $\min(|X|,|Y|)$ 称之为完美匹配。
定理: 不妨设 $|X| \leqslant |Y|$。
二分图存在完美匹配 $\iff$ 对于 $1 \leqslant k \leqslant |X|$, 均满足从 X 选出 k 个不同点, 连向 Y 的点集大小 $\geqslant k$
闭合子图问题
最大权闭合子图: 点有点权 (可正可负), 选出点权和最大的闭合子图。
考虑最小割建模。对于正价点, 连源, 边权为点权。对应地, 负价点连汇, 边权为点权绝对值。
原图中的有向边保留, 边权置为 INF, 此时有:
最大权闭合子图 = 正点权和 − 最小割 (构造) 最大权闭合子图 = 正点权和 − 最小割 (构造)
方案: 在闭合子图中的点为: 未割过的正权点 和 已经割过的负权点
最大密度子图
考虑 01 分数规划, 转化为 $\sum$(点权-二分的密度) 之和最大, 将其视为新点权可用最大权闭合子图去做。
最小路径覆盖
有定理: 最小路径覆盖 = 总点数 − 拆点二分图最大匹配
采用调整的思想。首先将每个点用单独一个路径覆盖, 这是合法的。
每合并两条路径, 的最终路径数就减小 1
拆出入点变成二分图, 对于图中原有的边, 从出点连向入点即可。

某条边被选中, 就相当于“串起”了两条已有的路径。(按照最大匹配合并路径即可得知最小路径覆盖)

注意到“单条路径”的限制: 不能分叉。这跟匹配的“左端点不重复”限制是等价的。

不能多条汇聚, 这跟匹配的“右端点不重复”限制是等价的。

这样就把最小路径覆盖转化成了二分图最大匹配。

偏序集的最小链覆盖问题

定义:

偏序集: 元素之间存在大小关系, 允许不可比 (无定义) 的情况出现。但是比较一定有传递性, 且不能互相矛盾。

链: 一个子图, 任意两个点都可比。(可以跳过一些中间点, 不是图论中的链)

反链: 一个子图, 任意两个点都不可比。

能够发现, 如果把 DAG 的有向边看做 $'<'$, 那么一定不会互相矛盾。一般来讲, 偏序集通常是以 DAG 的形式给出的。

现在, 需要用尽量少且不重的链, 覆盖图上的每一个点。

我们先 $O(n^3)$ 求出传递闭包 (充分利用传递性)。此时两个点之间的比较可以跳过中间点。

不难发现, 新图中的一条连续路径, 就对应着原图的一条链。我们求出最小路径覆盖即可。

最长反链问题

由 Dilworth 定理可知道, 最长反链大小 = 总点数 − 最小链覆盖

例题: 给一个 DAG, 求最长反链, 要求构造方案, 并考虑每个点是否能出现在最长反链中

最长反链: 一张有向无环图的最长反链为一个集合 $S \subseteq V$, 满足对于 S 中的任意两个不同的点 $u, v \in S(u \neq v)$,u 不能到达 v,v 也不能到达 u, 且 S 的大小尽量大。

根据 Dilworth 定理, 一个 DAG 中最长反链的大小, 等于其最小链划分的大小。

最小链划分: 在 DAG 中选出若干条链, 每个点恰好属于其中一条链, 且链数尽量少。

需要注意的是, 这里对“链”的定义, 不需要是 DAG 中连续的一条链, 只需要前一个点能通过路径到达后一个点即可。你也可以理解为: 一条连续的链, 挖掉中间的一些点, 形成的点集也算作“链”。

另一个理解方式是: 这里的“链划分”要求每个点不重不漏地属于一条链, 但也可以理解为, 链一定要是连续的, 并且允许多条链重复经过同一个点, 只需要保证每个点都被经过即可。(以这种角度去理解, 可以称之为最小“可重链覆盖”。)

这两种理解方式 (即 (1) 最小链划分和 (2) 最小可重链覆盖) 是等价的, 只不过前者的链可以跳过中间的点。

在后文中, 给出一种使用二分图匹配求 DAG 中的最小不可重链覆盖的方法。为了别求错, 我们需要先对 DAG 求一次传递闭包, 把 u 能间接到达 v 的点对之间的边 $u \to v$ 建出来, 这样就把“可重链覆盖”转化为“不可重链覆盖”了。

这里给出使用二分图匹配求最小不可重链覆盖的方法。

考虑从每个点自成一条链的形态出发, 此时恰好有 n 条链。

可以发现最终答案一定是合并 (首尾相接) 若干条链形成的。考虑重新描述这个过程:

对于一个点, 它在最终的链上, 一定只有最多一个前驱, 和最多一个后继。

我们考虑把每个点拆成入点和出点, 那么入点和出点应该只能匹配上最多一个点 (表示前驱或者后继)。

这似乎是二分图匹配的形式, 具体地, 我们考虑:

把一个点 x 拆成两个点:x_{out} 和 x_{in}, 表示出点和入点。

对于一条边 $x \to y$, 连接 x_{out} 与 y_{in}, 表示原图中 x 的出边指向 y(这条边是 y 的入边)。

那么最终形成了一个二分图, 左侧是所有 x_{out}, 右侧是所有 x_{in}。而且所有边都是连接左侧的点和右侧的点的。

在这个二分图 $G = \langle\langle V_{out}, V_{in}\rangle, E'\rangle$ 上做二分图最大匹配:

每一个匹配边 $x_{out} \leftrightarrow y_{in}$ 都可以还原原图中链的一条边 $x \to y$。

每匹配 1 条边, 链的个数就减少 1, 则有最小链覆盖的大小等于 n 减去最大匹配的大小。

从右侧 (in) 的非匹配点 (这里为 B, 可能有多个) 开始 DFS, 右侧的点只能走非匹配边向左访问, 左侧的点 (out) 只能走匹配边向右访问:

我们取左侧被 DFS 到的点, 以及右侧没被 DFS 到的点, 记做集合 S, 可以证明 S 是一个最小点覆盖。

证明:

1. 首先有: 最小点覆盖等于最大匹配。我们可以证明 $|S| = m$。这是因为: 右侧的非匹配点一定都被 DFS 到了, 所以在右侧选取的必然是匹配点。如果一个右侧的匹配点没被选取, 即它被 DFS 到了, 而这只有可能是因为它在左侧匹配到的点被 DFS 到了, 那么左侧匹配到的点就会被选上。即是: 每条匹配边的两端点恰好会被选一个。而左侧的非匹配点一定不会被 DFS 到, 这是因为如果被 DFS 到了, 必然会形成一条交错路 (匈牙利算法中的), 不满足最大匹配的条件。所以有且仅有匹配边的端点会被选上, 而且每条匹配边的两端点恰好被选一个, 所以 $\boldsymbol{|S| = m}$。

2. S 可以覆盖所有的边。我们把边按照左右端点是否被 DFS 到, 分成 $2 \times 2 = 4$ 类。那么如果出现了左端点没被 DFS 到, 但是右端点被 DFS 到了的边, 它才不会被覆盖。然而这是不可能的, 这是因为对于一个右侧被 DFS 到的点, 与它相连的左侧的点一定都被 DFS 到了。

然后有最大独立集等于最小点覆盖的补集。也就是只要选出左侧没被 DFS 到的点和右侧被 DFS 到的点就行了。

回到 DAG 的情况 (注意到我们举的例子并不是 DAG 导出的二分图, 所以这个例子不能用来解释最长反链):

令最大独立集为 I, 考虑选出所有 x_{out} 和 x_{in} 都属于 I 的点, 记做集合 A, 它们构成一个最长反链。

证明: 先证 A 的确是一个反链: 这是容易的, 因为任取 $x \in A$,x_{in} 就一定是被 DFS 到的点, 而 x_{out} 一定是没被 DFS 到的点, 任何两个 $x, y \in A$ 之间若是有连边就和 DFS 的过程冲突了。

首先有 $|I| = 2n - |S| = 2n - m$, 而 $|I| - |A|$ 可以看作是满足「x_{out} 或 x_{in} 属于 I」的 x 的个数, 显然这样的 x 不会超过 n 个, 所以 $|I| - |A| \leqslant n$, 所以 $|A| \geqslant |I| - n = n - m$。

但是 A 再大, 也不能大过 $n - m$, 所以 $|A| = n - m$, 也就是一个最长反链。

总结: 只要选出 x_{out} 没被 DFS 到, 且 x_{in} 被 DFS 到了的点, 这些点就组成一个最长反链。

然后是第三问, 这只要默认该点被选中, 也就是删除这个点和与其有偏序关系的所有点后, 再求一次最长反链, 如果最长反链的大小只减小了 1, 那么这个点就能在最长反链中, 否则不能。

```cpp
bitset<N> bs[N];
int cal(vector<array<int, 2>>& e)
{
    memset(h, -1, sizeof h); idx = 0;
    S = 0, T = N - 1;
    for (int i = 1; i <= n; i++) add(S, i, 1), add(i + n, T, 1);
    for (auto [a, b]:e) add(a, b + n, 1);
    return dinic();
}
bool st[N], del[N];
void dfs(int u)
{
    if (st[u]) return ;
    st[u] = true;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j<1 || j >2 * n) continue;
        if (!f[i]) dfs(j);
    }
}
array<int, 2> g[M];
void solve()
{
    cin >> n >> m;
    vector<array<int, 2>> edge;
    for (int i = 1, a, b; i <= m; i++) g[i] = {a, b}, bs[a][b] = 1;
    for (int k = 1; k <= n; k++)
        for (int i = 1; i <= n; i++)
            if (bs[i][k]) bs[i] |= bs[k];
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            if (i != j && bs[i][j]) edge.push_back({i, j});
    int longest_invlink = n - cal(edge);
    cout << longest_invlink<<"\n";
    for (int i = 0; i < idx; i += 2)
    {
        int a = e[i ^ 1], b = e[i], c = f[i];
        if (b == T && c) dfs(a);
    }
    for (int i = 1; i <= n; i++) cout << (!st[i] && st[i + n]);
    cout<<"\n";
    for (int i = 1; i <= n; i++)
    {
        edge.clear();
        memset(del, false, sizeof del);
        del[i] = true;
        for (int j = 1; j <= n; j++)
            if (bs[i][j] || bs[j][i]) del[j] = true;
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                if (i != j && bs[i][j]&&!del[i]&&!del[j]) edge.push_back({i, j});
        int cnt = 0;
        for (int j = 1; j <= n; j++) cnt+=!del[j];
        cout << (cnt - cal(edge) == longest_invlink - 1);
    }
}
```

3.12 退流操作

在网络流中, 若动态加边可以通过 dinic 很好的保证时间, 如果是删边, 则需要通过退流操作完成, 并删去边.

例题: [SDOI2014] LIS

给定序列 A, 序列中的每一项 A_i 有删除代价 B_i 和附加属性 C_i。请删除若干项, 使得 A 的最长上升子序列长度减少至少 1, 且付出的代价之和最小, 并输出方案。

首先 DP 求出 LIS。

画出最优转移的 DAG, 我们的目标就是把这个 DAG 割了。

每个位置拆点, 点权为 $B[i]$, 最优转移边权为 inf , 求最小割, 能得到第一问的答案。

考虑: 最小割的必须边和可行边。

我们可以先安排一个 c 尽量小的可行边, 然后排除掉等价的其他可行边。

一种暴力的想法是, 删掉这条边, 重新跑一次最大流。这需要 $O(n)$ 次最大流, 仍然很低效。

我们考虑在原网络信息的基础上进行调整。

接下来引入一种重要的操作: **退流**。
观察删去某条边 $(u \to v)$ 之后的网络, 该边的两个端点的流量就不守恒了。
考虑调整, 不难发现从 $u \to S$ 跑一次最大流, 从 $T \to v$ 跑一次最大流, 把这条边的流量退回去。
这样仍然需要 $O(n)$ 次退流, 但是肯定比 $O(n)$ 次完整的最大流快。
前面提到, 可以缩 SCC 求可行边, 但是这题的图在不断变化, 只好直接暴力搜索, 查看是否有增广路。
一条边是最小割的可行边, 就表明这条边存在于某一种最小割中
一条边 (u, v) 是最小割的可行边需要满足两个条件
1. (u, v) 满流
1. 不存在从 u 到 v 的增广路径
我们只需要看看从 u 到 v 能不能 bfs 就好了

```cpp
int n, a[N], b[N], c[N], dp[N];
bool st[N];
bool check(int a, int b)
{
    memset(st, false, sizeof st);
    queue<int> q;
    q.push(a), st[a] = true;
    while (q.size())
    {
        int t = q.front();
        q.pop();
        for (int i = h[t]; ~i; i = ne[i])
        {
            int j = e[i];
            if (st[j] || !f[i]) continue;
            st[j] = true, q.push(j);
        }
    }
    return st[b];
}
void solve()
{
    cin >> n;
    for (int i = 1; i<= n; i++) cin >> a[i];
    for (int i = 1; i<= n; i++) cin >> b[i];
    for (int i = 1; i<= n; i++) cin >> c[i];
    memset(h, -1, sizeof h); idx = 0;
    int maxlen = 0;
    for (int i = 1; i <= n; i++)
    {
        dp[i] = 1;
        for (int j = 1; j < i; j++)
            if (a[j] < a[i]) dp[i] = max(dp[i], dp[j] + 1);
        maxlen = max(maxlen, dp[i]);
    }
    S = 0, T = N - 1;
    vector<array<int, 4>> edge;
    for (int i = 1; i <= n; i++)
    {
        add(i, i + n, b[i]);
        edge.push_back({c[i], i, i + n, idx - 2});
        if (dp[i] == 1) add(S, i, INF);
        if (dp[i] == maxlen) add(i + n, T, INF);
    }
    for (int i = 1; i <= n; i++)
        for (int j = 1; j < i; j++)
            if (dp[i] == dp[j] + 1 && a[j] < a[i]) add(j + n, i, INF);
    int maxflow = dinic();
    vector<int> ans;
    sort(edge.begin(), edge.end());
    for (auto [_, u, v, id]:edge)
    {
        if (check(u, v)) continue; //检查有没有 u-> v 的增广路
        ans.push_back(u);
        S = u, T = 0; //退流
        dinic();
        S = N - 1, T = v; //退流
        dinic();
        f[id] = f[id ^ 1] = 0; //删边
    }
    cout << maxflow<<" "<<ans.size()<<"\n";
```

```
    sort(ans.begin(), ans.end());
    for (int t:ans) cout << t<<" ";
}
```

4 字符串

4.1 字符串哈希

双哈希

```
struct HASH
{
    const int P1 = 1109, P2 = 3307, mod1 = 3036999473, mod2 = 3037000493;
    int h1[N], p1[N], h2[N], p2[N];
    void init(string & s)
    {
        p1[0] = p2[0] = 1;
        int n = s.size() - 1; //把" "+s 传进来
        for (int i = 1; i <= n; i++)
        {
            h1[i] = (h1[i - 1] * P1 + s[i]) % mod1, p1[i] = p1[i - 1] * P1 % mod1;
            h2[i] = (h2[i - 1] * P2 + s[i]) % mod2, p2[i] = p2[i - 1] * P2 % mod2;
        }
    }
    int get1(int l, int r) {return (h1[r] - h1[l - 1] * p1[r - l + 1] % mod1 + mod1) % mod1; }
    int get2(int l, int r) {return (h2[r] - h2[l - 1] * p2[r - l + 1] % mod2 + mod2) % mod2; }
    array<int, 2> get(int l, int r)
    {
        if (l > r) return {0, 0};
        return {get1(l, r), get2(l, r)};
    }
    array<int, 2> combine(array<int, 2> a, array<int, 2> b, int len)
    {//把两个串拼起来 前者哈希值 后者哈希值 后者的长度
        return { (a[0] * p1[len] + b[0]) % mod1, (a[1] * p2[len] + b[1]) % mod2 };
    }
}h;
```

int128 的单哈希

```
const int P = 1709;
const i128 mod = (9e18L) + 32007943;
void init(string s)
{
    p[0] = 1;
    int n = s.size() - 1; //把" "+s 传进来
    for (int i = 1; i <= n; i++) h[i] = ((i128)h[i - 1] * P + s[i]) % mod, p[i] = (i128)p[i -
    ↪  1] * P % mod;
}
int get(int l, int r) {return (h[r] - (i128)h[l - 1] * p[r - l + 1] % mod + mod) % mod; }
```

4.2 KMP

KMP 算法用于字符串匹配，其中 p 串（长度 m）去匹配 s 串（长度 n）。

匹配的本质是求以 s_i 结尾的子串中能匹配到的 p 串的最大前缀。

```
int s[N], p[M], ne[M];
void init()
{
    for (int i = 2, j = 0; i <= m; i++)
    {
        while (j && p[i] ≠ p[j + 1]) j = ne[j];
        if (p[i] == p[j + 1]) j++;
        ne[i] = j;
    }
}
void match()
{
    for (int i = 1, j = 0; i <= n; i++)
    {
```

```
        while (j && s[i] ≠ p[j + 1]) j = ne[j];
        if (s[i] == p[j + 1]) j++;
        if (j == m) j = ne[j];
    }
}
```

4.2.1 KMP 自动机

```
int jump[N][M], ne[N];
string s;
void solve()
{
    cin >> s;
    int n = s.size(), q;
    s = " "+s;
    int fail = 0;
    for (int i = 1; i <= n; i++)
    {
        fail = jump[fail][s[i]-'a'];
        ne[i] = fail;
        jump[i - 1][s[i]-'a'] = i;
        for (int j = 0; j < 26; j++) jump[i][j] = jump[fail][j];
    }
    cin >> q;
    while (q--)
    {
        string str;
        cin >> str;
        int len = n + str.size(), pre_fail = fail;
        s += str;
        int pre_jumpn = jump[n][s[n + 1]-'a'];
        for (int i = n + 1; i <= len; i++)
        {
            fail = jump[fail][s[i]-'a'];
            ne[i] = fail;
            jump[i - 1][s[i]-'a'] = i;
            for (int j = 0; j < 26; j++) jump[i][j] = jump[fail][j];
            cout << ne[i]<<" ";
        }
        cout << "\n";
        fail = pre_fail, jump[n][s[n + 1]-'a'] = pre_jumpn; //注意还原
        for (int i = 1; i <= str.size(); i++) s.pop_back();
    }
}
```

4.3 Z 函数 (exKMP)

```
void Z(string s, string t)
{
    int n = s.size(), m = t.size();
    s = " "+s, t = " "+t;
    z[1] = n;
    for (int i = 2, l = 0, r = 0; i <= n; i++)
    {
        z[i] = i > r?0:min(z[i - l + 1], r - i + 1);
        while (s[1 + z[i]] == s[i + z[i]]) z[i]++;
        if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
    }
    for (int i = 1, l = 0, r = 0; i <= m; i++)
    {
        p[i] = i > r?0:min(z[i - l + 1], r - i + 1);
        while (p[i] < n && s[1 + p[i]] == t[i + p[i]]) p[i]++;
        if (i + p[i] - 1 > r) l = i, r = i + p[i] - 1;
    }
} // z[i] 是 s 与后缀 s[i..n] 的 lcp p[i] 是 s 与 t 的后缀 t[i..n] 的 lcp
```

4.4 Trie（字典树）

```cpp
int tr[N][26], cnt[N], idx;
void insert(char str[])
{
    int p = 0;
    for (int i = 0; str[i]; i++)
    {
        int u = str[i]-'a';
        if (!tr[p][u]) tr[p][u] = ++idx; //如果不存在 则开新节点
        p = tr[p][u];
    }
    cnt[p]++;
}
int query(char str[])
{
    int p = 0;
    for (int i = 0; str[i]; i++)
    {
        int u = str[i]-'a';
        if (!tr[p][u]) return 0; //如果不存在当前节点 返回 0
        p = tr[p][u];
    }
    return cnt[p];
}
```

4.4.1 01Trie

01trie 一定要记得清空的时候从 $[0, idx]$ 清空

```cpp
void insert(int v)
{
    int p = 0;
    for (int i = 29; i >= 0; i--)
    {
        int t = v >> i & 1;
        if (!tr[p][t]) tr[p][t] = ++idx;
        p = tr[p][t];
        cnt[p]++;
    }
}
void del(int v)
{
    int p = 0;
    for (int i = 29; i >= 0; i--)
    {
        int t = v >> i & 1;
        p = tr[p][t];
        cnt[p]--;
    }
}
int query(int v)
{
    int res = 0, p = 0;
    for (int i = 29; i >= 0; i--)
    {
        int t = v >> i & 1;
        if (tr[p][t ^ 1] && cnt[tr[p][t ^ 1]]) p = tr[p][t ^ 1], res |= 1 << i;
        else p = tr[p][t];
    }
    return res;
}
void solve()
{
    while (q--)
    {
        char op[2]; int x;
        cin >> op >> x;
        if (*op=='+') insert(x);
        else if (*op=='-') del(x);
        else cout << query(x) <<"\n";
    }
}
```

例题: 最小化 $a_i \bigoplus x$ 的最大值

```
void insert(int x)
{
    int p = 0;
    for (int k = 29; k >= 0; k--)
    {
        int t = x >> k & 1;
        if (!tr[p][t]) tr[p][t] = ++idx;
        p = tr[p][t];
        cnt[p]++;
    }
}
int dfs(int u, int k)
{
    if (!k)
    {
        if (cnt[tr[u][0]] && cnt[tr[u][1]]) return 1;
        else return 0;
    }
    if (!tr[u][0]) return dfs(tr[u][1], k - 1);
    else if (!tr[u][1]) return dfs(tr[u][0], k - 1);
    else return (1 << k) + min(dfs(tr[u][0], k - 1), dfs(tr[u][1], k - 1));
}
void solve()
{
    for (int i = 1; i <= n; i++) insert(x);
    cout << dfs(0, 29)<<"\n";
}
```

01Trie 全局 +1

给定一棵树, 要求支持以下操作:

1. 对于一个点, 和它相邻地所有点权值 $+1$
2. 单点修改一个点的权值
3. 查询与一个点相邻所有点的异或和

邻域修改和查询不可做, 所以转化为维护儿子和维护父亲去做, 自己的 Trie 里存的是儿子的权值, 对父亲单独修改, 再维护一些修改标记即可

```
struct Trie
{
    int tr[N * M][2], xor_sum[N * M], cnt[N * M], root[N], idx;
    void pushup(int p)
    {
        cnt[p] = cnt[tr[p][0]] + cnt[tr[p][1]]; //节点个数
        xor_sum[p] = (xor_sum[tr[p][0]] << 1) ^ (xor_sum[tr[p][1]] << 1) ^ (cnt[tr[p][1]] &
        ↪  1);
    } // 左儿子异或值左移一位 右儿子异或值左移一位 如果右儿子有奇数个值 这一位是 1
    int insert(int p, int x, int d)
    {
        if (!p) p = ++idx;
        if (d >= M)
        {
            cnt[p]++;
            return p;
        }
        tr[p][x & 1] = insert(tr[p][x & 1], x >>1, d + 1);
        pushup(p);
        return p;
    }
    void add1(int p)
    {
        swap(tr[p][0], tr[p][1]); //交换左右儿子
        if (tr[p][0]) add1(tr[p][0]); //继续往 0 走
        pushup(p);
    }
}tr;
vector<int> g[N];
int n, m, a[N], fa[N], tag[N];
void dfs(int u, int father)
{
    fa[u] = father;
```

```
    if (father) tr.root[father] = tr.insert(tr.root[father], a[u], 0);
    for (int j:g[u])
    {
        if (j == father) continue;
        dfs(j, u);
    }
}
void solve()
{
    cin >> n >> m;
    for (int i = 0, a, b; i < n - 1; i++)
    {
        cin >> a >> b;
        g[a].push_back(b), g[b].push_back(a);
    }
    for (int i = 1; i<= n; i++) cin >> a[i];
    dfs(1, 0);
    while (m--)
    {
        int t, u, v;
        cin >> t >> u;
        if (t == 1)
        {
            tag[u]++;
            tr.add1(tr.root[u]);
            if (fa[u])
            {
                int p = fa[u];
                a[p]++;
                if (fa[p])
                {
                    tr.root[fa[p]] = tr.insert(tr.root[fa[p]], a[p] - 1 + tag[fa[p]], 0);
                    tr.root[fa[p]] = tr.insert(tr.root[fa[p]], a[p] + tag[fa[p]], 0);
                }
            }
        }
        else if (t == 2)
        {
            cin >> v;
            a[u] -= v;
            if (fa[u])
            {
                tr.root[fa[u]] = tr.insert(tr.root[fa[u]], a[u] + v + tag[fa[u]], 0);
                tr.root[fa[u]] = tr.insert(tr.root[fa[u]], a[u] + tag[fa[u]], 0);
            }
        }
        else cout << (tr.xor_sum[tr.root[u]] ^ (a[fa[u]] + tag[fa[fa[u]]]))<<"\n";
    }
}
```

给定一棵 n 个结点的有根树 T, 结点从 1 开始编号, 根结点为 1 号结点, 每个结点有一个正整数权值 v_i。

设 x 号结点的子树内 (包含 x 自身) 的所有结点编号为 $c_1, c_2, \ldots, c_k$, 定义 x 的价值为:

$val(x) = (v_{c_1} + d(c_1, x)) \oplus (v_{c_2} + d(c_2, x)) \oplus \ldots \oplus (v_{c_k} + d(c_k, x))$

其中 $d(x, y)$ 表示树上 x 号结点与 y 号结点间唯一简单路径所包含的边数,$d(x, x) = 0$。$\oplus$ 表示异或运算。

求出 $\sum_{i=1}^{n} val(i)$ 的结果。

问题可以转化为: Trie 的合并 Trie 全体 $+1$ 查询 Trie 内所有数的异或值

```
const int N = 530010, M = 21;
struct Trie
{
    int tr[N * (M + 1)][2], xor_sum[N * (M + 1)], cnt[N * (M + 1)], root[N], idx;
    void pushup(int p)
    {
        cnt[p] = cnt[tr[p][0]] + cnt[tr[p][1]];
        xor_sum[p] = (xor_sum[tr[p][0]] << 1) ^ (xor_sum[tr[p][1]] << 1) ^ (cnt[tr[p][1]] &
        ↪  1);
    }
    int insert(int p, int x, int d)
    {
        if (!p) p = ++idx;
        if (d >= M)
        {
```

```
            cnt[p]++;
            return p;
        }
        tr[p][x & 1] = insert(tr[p][x & 1], x >>1, d + 1);
        pushup(p);
        return p;
    }
    void add1(int p)
    {
        swap(tr[p][0], tr[p][1]);
        if (tr[p][0]) add1(tr[p][0]);
        pushup(p);
    }
    int merge(int x, int y)//x += y
    {
        if (!x||!y) return x | y;
        xor_sum[x] ^= xor_sum[y];
        cnt[x] += cnt[y];
        tr[x][0] = merge(tr[x][0], tr[y][0]);
        tr[x][1] = merge(tr[x][1], tr[y][1]);
        return x;
    }
}tr;
vector<int> g[N];
int n, m, a[N], fa[N], tag[N];
long long ans;
void dfs(int u)
{
    for (int j:g[u])
    {
        dfs(j);
        tr.add1(tr.root[j]);
        tr.root[u] = tr.merge(tr.root[u], tr.root[j]);
    }
    ans += tr.xor_sum[tr.root[u]];
}
void solve()
{
    cin >> n;
    for (int i = 1; i <= n; i++)
    {
        cin >> a[i];
        tr.root[i] = tr.insert(tr.root[i], a[i], 0);
    }
    for (int i = 2; i <= n; i++)
    {
        int p;
        cin >> p;
        g[p].push_back(i);
    }
    dfs(1);
    cout << ans << "\n";
}
```

4.4.2 可持久化 Trie

和可持久化线段树比较像, 操作也基本一样

```
const int N = 50005, M = 30, INF = 2e9;
int tr[N * (M + 1)][2], cnt[N * (M + 1)], root[N], idx; //注意空间 要比 logn 多 1
int insert(int p, int k, int v)
{
    int q = ++idx;
    if (k < 0)
    {
        cnt[q]++;
        return q;
    }
    int u = v >> k & 1;
    tr[q][u ^ 1] = tr[p][u ^ 1];
    tr[q][u] = insert(tr[p][u], k - 1, v);
    cnt[q] = cnt[tr[q][0]] + cnt[tr[q][1]]; //pushup
    return q;
}
```

```cpp
int query(int p, int q, int k, int v)//在第 p + 1 - q 版本的 Trie 中寻找异或 v 能得到的最大值
{
    if (k < 0) return 0;
    int u = v >> k & 1;
    if (cnt[tr[q][u ^ 1]] - cnt[tr[p][u ^ 1]]) return (1 << k) + query(tr[q][u ^ 1], tr[p][u ^ 1], k - 1, v);
    else return query(tr[q][u], tr[p][u], k - 1, v);
}
//root[i] = insert(root[i - 1], M - 1, w[i]);
//query(root[l - 1], root[r], M - 1, w[x])
```

4.5 AC 自动机

AC 自动机类似于多模式串同时匹配的 KMP，首先建出 Trie，然后再建出失配指针加快匹配速度。

注意：如果涉及到 dp 或计算价值等问题，要考虑清楚是否需要上传子串贡献。

```cpp
struct ACAM
{
    int trie[N][M], fail[N], cnt[N], tot;
    vector<int> e[N];
    void clear()
    {
        for (int i = 0; i <= tot; i++)
        {
            memset(trie[i], 0, sizeof trie[i]);
            e[i].clear(), fail[i] = cnt[i] = 0;
        }
        tot = 0;
    }
    void insert(string & s)//如果想知道串长 就记录一下深度
    {
        int p = 0;
        for (auto c:s)
        {
            int t = c-'a';
            if (!trie[p][t]) trie[p][t] = ++tot;
            p = trie[p][t];
        }
        cnt[p] = 1;
    }
    void build()
    {
        queue<int> q;
        for (int i = 0; i < 26; i++) if (trie[0][i]) q.push(trie[0][i]);
        while (q.size())
        {
            int t = q.front();
            q.pop();
            for (int i = 0; i < 26; i++)
            {
                int p = trie[t][i];
                if (!p) trie[t][i] = trie[fail[t]][i];
                else fail[p] = trie[fail[t]][i], q.push(p);
            }
        }
    }
    void build_graph()
    {
        for (int i = 1; i <= tot; i++) e[fail[i]].push_back(i);
    }
    void dfs(int u)
    {
        for (int j:e[u])
        {
            dfs(j);
             cnt[u] += cnt[j];
        }
    }
}acam; //插入完字符串记得调用 build 函数
```

例: 给定 n 个模式串 s_i 和一个文本串 t, 求有多少个不同的模式串在文本串里出现过。两个模式串不同当且仅当他们编号不同。

对文本串进行匹配, 对每个出现过的状态打个标记, 最后在 Fail 树上上传标记并统计答案

```
while (n--)
{
    string s;
    cin >> s;
    acam.insert(s);
}
acam.build();
string s;
cin >> s;
int p = 0;
for (auto c:s)
{
    int t = c-'a';
    p = acam.trie[p][t];
    acam.ex[p] = true;
}
acam.build_graph();
acam.dfs(0);
int ans = 0;
for (int i = 1; i <= acam.idx; i++)
    if (acam.ex[i]) ans += acam.cnt[i];
```

例子: 给你一个文本串 S 和 n 个模式串 $T_1 \sim T_n$, 请你分别求出每个模式串 T_i 在 S 中出现的次数。

在每个状态上计数, 最后在 fail 树上上传给父亲即可。每个单词结尾时在 Trie 树上对应节点记录编号。

AC 自动机上计数

给定 n 个字符串, 再给定 q 个询问, 每次询问字符串 s_i 在字符串 s_j 中出现了多少次

若 s_i 在 s_j 中出现, 则说明 s_i 是 s_j 的一个前缀的后缀

考虑前缀信息, Trie 树上从根到 s_j 所代表节点的路径上的每一个点都是 s_j 的前缀

考虑后缀信息, 对于一个字符串 s 来说, 其最大后缀能匹配的字符串由 fail 指针刻画

把询问挂到 j 上, 问题变为 j 在 Trie 上有多少个祖先 k, 满足在 fail 树上,i 是 k 的祖先。

转换一下视角, 即求在 fail 树上,i 的子树中有多少个 j 的前缀 k

在 Trie 树上 dfs, 利用 $+1-1$ 的技巧在 fail 树上进行单点加, 遇到询问就进行子树查即可。

给定包含 k 个字符串的集合 S, 有 n 个操作, 操作有三种类型:

询问操作: 给出一个字符串 t, 询问当前字符串集 S 中的每一个字符串匹配询问字符串 t 的次数之和

添加操作: 表示将编号为 i 的字符串加入到集合中

删除操作: 表示将编号为 i 的字符串从集合中删除。

对于询问, 可以将 t 放在 ACAM 上面跑, 求 t 的每一个前缀 $t[1,i]$ 的所有后缀能匹配的次数, 就是匹配的总次数。设 $t[1,i]$ 在 ACAM 上的状态为 u, 那么就是在 fail 树上求根到 u 求和。对于添加和删除操作, 就是在 fail 树进行单点加。对于单点加链求和, 可以转化为子树加, 单点查。

AC 自动机上 DP

例子: 给定若干模式串，要求构造一个长度为 k 的字符串，使其包含的模式串个数最多（可重复）

ACAM 的状态转移为 dp 刻画了转移方向, 考虑在 ACAM 的节点上 dp,

有状态转移方程 $f_{i,tr_{j,u}} = \max\{f_{i-1,j} + cnt_{tr_{j,u}}\}(u \in \Sigma)$

4.5.1 二进制分组 AC 自动机 (动态)

维护一个字符串集合, 支持三种操作:

1. 加字符串
2. 删字符串
3. 查询集合中的所有字符串在给出的模板串中出现的次数

强制在线

二进制分组合并 AC 自动机, 删除的话就考虑贡献是可差分的, 对添加和删除分别建 AC 自动机即可

```
int trie[N][M], ch[N][M], ne[N], idx;
int ed[N], sum[N];
int root[N], sz[N], tot;
void insert(int p, string & s, int v)
{
    for (auto c:s)
    {
        int t = c-'a';
        if (!trie[p][t]) trie[p][t] = ++idx;
        p = trie[p][t];
    }
    ed[p] += v;
}
int merge(int p, int q)
```

```
{
    if (!p||!q) return p | q;
    ed[p] += ed[q];
    for (int i = 0; i < M; i++) trie[p][i] = merge(trie[p][i], trie[q][i]);
    return p;
}
void build(int p)
{
    queue<int> q;
    for (int i = 0, v; i < M; i++)
        if (v = trie[p][i])
        {
            ne[v] = p, ch[p][i] = trie[p][i];
            q.push(v);
        }
        else ch[p][i] = p;
    while (q.size())
    {
        int t = q.front(); q.pop();
        sum[t] = sum[ne[t]] + ed[t];
        for (int i = 0; i < M; i++)
        {
            int x = trie[t][i];
            if (!x) ch[t][i] = ch[ne[t]][i];
            else
            {
                ch[t][i] = trie[t][i];
                ne[x] = ch[ne[t]][i];
                q.push(x);
            }
        }
    }
}
void modify(string & s, int v)
{
    root[++tot] = ++idx;
    sz[tot] = 1;
    insert(root[tot], s, v);
    while (sz[tot] == sz[tot - 1])
    {
        tot--;
        sz[tot] += sz[tot + 1];
        root[tot] = merge(root[tot], root[tot + 1]);
    }
    build(root[tot]);
}
int query(string & s)
{
    int res = 0;
    for (int i = 1; i <= tot; i++)
    {
        int p = root[i];
        for (auto c:s)
        {
            p = ch[p][c-'a'];
            res += sum[p];
        }
    }
    return res;
}
void solve()
{
    int n, opt;
    cin >> n;
    while (n--)
    {
        string s;
        cin >> opt >> s;
        if (opt == 1) modify(s, 1);
        else if (opt == 2) modify(s, -1);
        else
        {
            cout << query(s)<<"\n";
            fflush(stdout);
```

```
        }
    }
}
```

4.6 后缀数组

rk_i : 第 i 个后缀的排名

sa_i : 排名为 i 的是哪个后缀

$height_i$: 排名为 i 的后缀与排名为 $i-1$ 的后缀的 lcp 长度, 即 lcp_{sa_i,sa_j}

```cpp
struct SA
{
    int n, sa[N], rk[N], ork[N], buc[N], id[N], height[N], st[N][M];
    char s[N];
    void get_sa()
    {
        int m = 1 << 7, p = 0;
        memset(buc, 0, sizeof buc), memset(ork, 0, sizeof ork); //多测清空可能是 min(N - 1, n + m)
        for (int i = 1; i <= n; i++) buc[rk[i] = s[i]]++;
        for (int i = 1; i <= m; i++) buc[i] += buc[i - 1];
        for (int i = n; i; i--) sa[buc[rk[i]]--] = i;
        for (int w = 1; ; m = p, p = 0, w<<=1)
        {
            for (int i = n - w + 1; i <= n; i++) id[++p] = i;
            for (int i = 1; i<= n; i++) if (sa[i] > w) id[++p] = sa[i] - w;
            memset(buc, 0, m + 1 << 3), memcpy(ork, rk, n + 1 << 3);
            p = 0;
            for (int i = 1; i <= n; i++) buc[rk[i]]++;
            for (int i = 1; i <= m; i++) buc[i] += buc[i - 1];
            for (int i = n; i; i--) sa[buc[rk[id[i]]]--] = id[i];
            for (int i = 1; i <= n; i++) rk[sa[i]] = ork[sa[i - 1]] == ork[sa[i]]
                &&ork[sa[i - 1] + w] == ork[sa[i] + w]?p:++p;
            if (p == n) break;
        }
        //sa[rk[i]] = i, 需要保证 s[0] 和 s[n + 1] 为空字符 (多测清空), 否则可能出错
        s[0] = s[n + 1] = 0;
        for (int i = 1, k = 0; i <= n; i++)
        {
            if (k) k--;
            while (s[i + k] == s[sa[rk[i] - 1] + k]) k++;
            height[rk[i]] = k;
        }
    }
    void init()
    {
        for (int i = 0; i < M; i++)
            for (int l = 1; l + (1 << i) - 1 <= n; l++)
                if (!i) st[l][i] = height[l];
                else st[l][i] = min(st[l][i - 1], st[l + (1 << (i - 1))][i - 1]);
    }
    void init(string str)
    {
        n = str.size();
        for (int i = 1; i <= n; i++) s[i] = str[i - 1];
        get_sa(), init();
    }
    int query(int l, int r)
    {
        int k = __lg(r - l + 1);
        return min(st[l][k], st[r - (1 << k) + 1][k]);
    }
    int lcp(int a, int b)
    {
        if (a == b) return n - a + 1;
        a = rk[a], b = rk[b];
        if (a > b) swap(a, b);
        return query(a + 1, b);
    }
};
```

SAIS

```cpp
int n, s[N << 1], s1[N], sa[N], cnt[N], P[N << 1], rk[N], idx[N], height[N];
bool t[N << 1];
#define PL(x) sa[idx[s[x]]++] = x
#define PS(x) sa[idx[s[x]]--] = x
void IS(int * s, bool * t, int * sa, int * cnt, int n, int m, int * v, int top)
{
    fill(sa + 1, sa + n + 1, 0);
    memcpy(idx, cnt, sizeof(int) * (m + 1));
    for (int i = top; i; --i) PS(v[i]);
    for (int i = 1; i <= m; ++i) idx[i] = cnt[i - 1] + 1;
    for (int i = 1; i <= n; ++i)
    {
        int x = sa[i] - 1;
        if (x && t[x]) PL(x);
    }
    memcpy(idx, cnt, sizeof(int) * (m + 1));
    for (int i = n; i; --i)
    {
        int x = sa[i] - 1;
        if (x&&!t[x]) PS(x);
    }
}
void SAIS(int * s, bool * t, int * p, int * cnt, int n, int m)
{
    int top = 0, *s1 = s + n + 1;
    t[n] = 0;
    for (int i = 1; i <= n; ++i) ++cnt[s[i]];
    for (int i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
    for (int i = n - 1; i; --i) t[i] = (s[i] > s[i + 1] || (s[i] == s[i + 1] && t[i + 1]));
    for (int i = 2; i <= n; ++i) rk[i] = ((t[i - 1]&&!t[i])?(p[++top] = i, top):0);
    IS(s, t, sa, cnt, n, m, p, top);
    int nm = 0;
    for (int i = 1, x = 0, y = 0; i <= n; ++i) if (x = rk[sa[i]])
    {
        if (!m || p[x + 1] - p[x] != p[y + 1] - p[y]) ++nm;
        else
        {
            for (int l = p[x], r = p[y]; l <= p[x + 1]; ++l, ++r)
                if (s[l] != s[r]) { ++nm; break; }
        }
        y = x;
        s1[x] = nm;
    }
    if (nm < top) SAIS(s1, t + n + 1, p + n + 1, cnt + m + 1, top, nm);
    else for (int i = 1; i <= top; ++i) sa[s1[i]] = i;
    for (int i = 1; i <= top; ++i) s1[i] = p[sa[i]];
    IS(s, t, sa, cnt, n, m, s1, top);
}
void SA(string & ss)
{
    n = ss.size();
    for (int i = 1; i <= n; ++i) s[i] = ss[i - 1];
    s[++n] = 1;
    SAIS(s, t, P, cnt, n, 'z'+2);
    for (int i = 1; i < n; ++i) sa[i] = sa[i + 1];
    sa[n--] = 0;
    for (int i = 1; i <= n; ++i) rk[sa[i]] = i;
    for (int i = 1, k = 0; i <= n; i++)
    {
        if (k) k--;
        while (s[i + k] == s[sa[rk[i] - 1] + k]) k++;
        height[rk[i]] = k;
    }
}
```

不同子串个数:$ans = \sum_{i=1}^{n} (len_{sa_i} - height_i) = \frac{n\times(n+1)}{2} - \sum_{i=1}^{n} height_i$。

给出一个字符串 s, 对于 s $s[1,i]$, 输出其本质不同的子串个数。

如果考虑正常做, 每添加一个字符, 所有的后缀都会增加一个字符, 并不好做。考虑把字符串倒过来, 每次在前面添加一个字符, 那么所有后缀都没有变, 只是增加了一个新的后缀。考虑新增加的这个后缀对答案的影响。

$\Delta = len_{new} - lcp_{pre_{new},next_{new}} + lcp_{pre_{new},new} + lcp_{new,next_{new}}$

用链表实现前后继的操作, 用 ST 表实现 $O(1)$ 查询两个后缀的 lcp

给出一个字符串 s, 每个字符带权, 后缀 i 和后缀 j 满足 $lcp_{i,j}=r$, 则称后缀 i 和后缀 j 满足 k 相似 $(k=0,1,2,\ldots,r)$, 对于 $r=0,1,2,\ldots,n-1$, 求有多少组后缀 $\{i,j\}$ 满足 r 相似, 且求 r 相似下的 $\max\{a_i\times a_j\}$。

进行后缀排序。利用并查集对信息进行维护。对 r 从小到大进行枚举并合并集合。在同一个集合里, 则说明这些后缀是互相 r 相似的, 即 $lcp_{i,j}\geq r(i,j\in S)$, 若当前枚举到 $r=k$, 则将满足 $height_i=k$ 的所有 i, 将 $i-1$ 和 i 进行合并, 合并时维护集合大小和集合最大值。

求 $\sum\limits_{1\leqslant i<j\leqslant n} lcp_{i,j}$

求出后缀数组。问题转化为求 $\sum\limits_{1\leqslant i\leqslant j\leqslant n}\min\limits_{k=i}^{j} height_k$, 即求所有区间的最小值的和, 这是一个单调栈的经典问题。求出每个值作为最小值的贡献区间, 利用乘法原理即可得到答案。

给定字符串 s,t 求 $lcp_{s,t}$ 允许失配 $k(k\leqslant 10)$ 次

将两个字符串拼在一起, 跑后缀数组。利用 $lcp_{i,j}=\min\limits_{k=rk_i+1}^{rk_j} height_k$ 不断求 lcp。失配一次就往后跳, 继续求 lcp, 直到失配 k 次。

4.7 后缀自动机

用途：处理字符串所有子串的问题，如子串去重、字典序第 k 小子串等。

核心性质：从初始状态到任意状态的路径与原串的所有子串一一对应。

空间复杂度：状态数不超过 $2n-1$，边数不超过 $3n-4$，数组要开两倍空间。

关键概念：endpos 等价类：结束位置集合相同的子串为一类，SAM 每个状态对应一个等价类。

SAM 定义: 字符串 s 的 SAM 是一个接受 s 的所有后缀的最小的有限状态自动机。具体地, SAM 有状态集合 Q, 每个状态是有向无环图上的一个节点。从每个状态出发有若干条或零条转移边, 每条转移边都对应一个字符 (因此, 一条路径表示一个字符串), 且从一个状态出发的转移互不相同。根据 DFA 的定义, SAM 还存在终止状态集合 F, 表示从初始状态 T 到任意终止状态的任意一条路径与 s 的每一个后缀一一对应。

SAM 最重要, 也是最基本的一个性质: 从 T 到任意状态的所有路径与 s 的所有子串一一对应。我们称状态 p 表示字符串 t_p, 当且仅当存在一条 $T\to p$ 的路径使得该路径所表示的字符串为 t_p。根据上述性质,t_p 是 s 的子串。

SAM 一般分为两部分:$DWAG$(有向无环单词图) 和 $Parent$ 树

$endpos$: 字符串 t 在 s 中所有出现的结束位置的集合。

$endpos$ 等价类: 若 t_i,t_j 均为 s 的子串, 且 $endpos_i=endpos_j$, 则称 t_i,t_j 同属于一个 $endpos$ 等价类。一个 $endpos$ 等价类是极大的。SAM 上的每个节点代表了一个 $endpos$ 等价类。

两个字符串 t_1,t_2 的 $endpos$ 可能相等。例如当 $s=$"abab" 时, endpos("b")=endpos("ab")。这样, 我们可以将 s 的子串划分为若干等价类, 用一个状态表示。SAM 的每个状态 p 对应若干 $endpos$ 集合相同的子串。换句话说,$\forall t\in substr(p)$,$endpos(t)$ 相等。因此, SAM 的状态数等于所有子串的等价类个数 (初始状态对应空串)。

$endpos$ 等价类子串的关系: 它们的出现次数, 出现的结束位置都是一样的。这些子串是一系列的后缀一样的子串, 且长度是连续的, 如 $\{abcc,bcc,cc\}$, 相当于在最长的子串中不断删去首字符, 而 $endpos$ 集合不发生变化。若 $endpos$ 集合发生变化, 因为串越来越短, 所以 $|endpos|$ 必定增大, 即子串出现位置增多。此时出现一条有向边 $endpos_p\to endpos_q$, 这些边组成了 $parent$ 树上的边。

len_p: 状态 p 所表示的所有子串中, 长度最长的那一个子串的长度。则长度最短的那一个字串的长度为 $len_{fa_p}+1$。可以推出, 等价类 $endpos_p$ 中的子串有 $len_p-len_{fa_p}$ 种

要注意后缀自动机的点数要开成串的两倍

一般来说, return 谁, 谁的 sz 就可以设为 1

```cpp
struct SAM
{
    int len, fa;
    int ch[26];
}sam[N]; //注意 SAM 空间要开两倍
int tot = 1, last = 1;
char s[N];
void extend(int c)
{
    int p = last, np = last = ++tot;
    sz[tot] = 1, sam[np].len = sam[p].len + 1; //如果不需要记录 sz 可以不记录
    for (; p&&!sam[p].ch[c]; p = sam[p].fa) sam[p].ch[c] = np;
    if (!p) sam[np].fa = 1;
    else
    {
        int q = sam[p].ch[c];
        if (sam[q].len == sam[p].len + 1) sam[np].fa = q;
        else
        {
            int nq = ++tot;
            sam[nq] = sam[q], sam[nq].len = sam[p].len + 1;
            sam[q].fa = sam[np].fa = nq;
            for (; p && sam[p].ch[c] == q; p = sam[p].fa) sam[p].ch[c] = nq;
        }
    }
}
--------------------------------
for (int i = 0; s[i]; i++) extend(s[i]-'a');
```

```
//radixsort(); 基数排序计算完贡献后, 每个点的 sz 就是 | endpos|
```

```
void radixsort()
{
    for (int i = 1; i <= tot; i++) buc[sam[i].len]++;
    for (int i = 1; i <= tot; i++) buc[i] += buc[i - 1];
    for (int i = tot; i; i--) id[buc[sam[i].len]--] = i;
    for (int i = tot; i; i--) sz[sam[id[i]].fa] += sz[id[i]];
}
-------------------------------------------------
for (int i = 1; i <= n; i++) extend(s[i]-'a'); //要转化一下
```

如果字符集大小比较大可以利用 map 进行转移

```
struct SAM
{
    int len, fa;
    map<int, int> ch;
}sam[N];
int tot = 1, last = 1;
int s[N];
void extend(int c)
{
    int p = last, np = last = ++tot;
    sam[np].len = sam[p].len + 1;
    for (; p && sam[p].ch.find(c) == sam[p].ch.end(); p = sam[p].fa) sam[p].ch[c] = np;
    if (!p) sam[np].fa = 1;
    else
    {
        int q = sam[p].ch[c];
        if (sam[q].len == sam[p].len + 1) sam[np].fa = q;
        else
        {
            int nq = ++tot;
            sam[nq] = sam[q], sam[nq].len = sam[p].len + 1;
            sam[q].fa = sam[np].fa = nq;
            for (; p && sam[p].ch[c] == q; p = sam[p].fa) sam[p].ch[c] = nq;
        }
    }
}
```

求本质不同子串个数:$ans = \sum_{i=2}^{tot}(len_i - len_{fa_i})$

SAM 用于字符串匹配

用文本串 t 在 s 的 SAM 上跑匹配时, 我们可以得到对于 t 的每个前缀 $t[1,i]$, 其作为出现 s 的子串出现的最长后缀 L_i。若当前状态 p,$t[i-L_{i-1},i-1]$ 所表示的状态不能匹配 t_i(即 $\delta(p,t_i)$ 不存在), 就需要跳 fa, 令 $p \leftarrow fa_p$, 并更新此时的匹配长度。若可以匹配, 则令 $p \leftarrow \delta(p,t_i)$, 并更新匹配长度。

```
for (int i = 1, p = 1, cnt = 0; i <= n; i++)
{
    int t = s[i]-'a';
    while (p >1&&!sam[p].ch[t]) p = sam[p].fa, cnt = sam[p].len;
    if (sam[p].ch[t]) p = sam[p].ch[t], cnt++;
}
```

线段树合并维护 $endpos$ 集合

对于部分题目, 我们需要维护每个状态的 $endpos$ 集合, 以刻画每个子串在字符串中所有出现位置的信息。为此, 我们在 $s[1,i]$ 对应状态的 $endpos$ 集合里插入位置 i, 再根据 $endpos$ 集合构造出来的树本质上就是后缀链接树这一事实, 在 $parent$ 树上进行线段树合并即可得到每个状态的 $endpos$ 集合。这是一个非常有用且常见的技巧。

注意, 线段树合并时会破坏原有线段树的结构, 因此若需要在线段树合并后保留每个状态的 $endpos$ 集合对应的线段树的结构, 需要在线段树合并时新建节点。即可持久化线段树合并。SAM 相关问题的线段树合并通常均需要可持久化。

特别的, 如果仅为了得到 $endpos$ 集合大小, 那么只需求出每个状态在 $parent$ 树上的子树有多少个表示 s 的前缀的状态。前缀状态即所有曾作为 cur 的节点。对此, 有两种解决方法: 直接建图 dfs 和基数排序计算。

快速定位子串

给定区间 $[l,r]$, 求 $s[l,r]$ 在 SAM 上的对应状态: 在构建 SAM 时容易预处理 $s_{1,i}$ 所表示的状态 pos_i。从 pos_r 开始在 $parent$ 树上倍增找到最浅的,$len \geq r-l+1$ 的状态 p 即为所求。

4.7.1 广义后缀自动机

```
//在线构造 GSAM
struct SAM
{
    int len, fa;
    int ch[26];
}sam[N];
int tot = 1;
int extend(int c, int last, int id)//如果不需要维护 id 不需要加入
{
    if (sam[last].ch[c])//这个状态之前已经存在过了
    {
        int p = last, x = sam[p].ch[c];
        if (sam[p].len + 1 == sam[x].len)
        {
            sz[id][x] = 1; //如果维护 sz 的话
            return x;
        }
        else
        {
            int y = ++tot;
            sam[y].len = sam[p].len + 1;
            for (int i = 0; i < 26; i++) sam[y].ch[i] = sam[x].ch[i];
            while (p && sam[p].ch[c] == x) sam[p].ch[c] = y, p = sam[p].fa;
            sam[y].fa = sam[x].fa, sam[x].fa = y;
            sz[id][y] = 1;
            return y;
        }
    }
    int z = ++tot, p = last;
    sam[z].len = sam[last].len + 1;
    while (p&&!sam[p].ch[c]) sam[p].ch[c] = z, p = sam[p].fa;
    if (!p) sam[z].fa = 1;
    else
    {
        int x = sam[p].ch[c];
        if (sam[p].len + 1 == sam[x].len) sam[z].fa = x;
        else
        {
            int y = ++tot; //下面四行可以直接复制上面的四行
            sam[y].len = sam[p].len + 1;
            for (int i = 0; i < 26; i++) sam[y].ch[i] = sam[x].ch[i];
            while (p && sam[p].ch[c] == x) sam[p].ch[c] = y, p = sam[p].fa;
            sam[y].fa = sam[x].fa, sam[z].fa = sam[x].fa = y;
        }
    }
    sz[id][z] = 1; //如果维护 sz 的话
    return z;
}
----------------------------------
for (int i = 1; i <= n; i++)
{
    string s;
    cin >> s;
    int last = 1;
    for (auto c:s) last = insert(c-'a', last);
}
----------------------------------
//离线构造 GSAM
struct SAM
{
    int len, fa;
    int ch[26];
}sam[N];
int tot = 1, idx = 1; //注意这里 idx 一定要从 1 开始 和普通 trie 不同
int tr[N][26], pos[N];
int extend(int p, int c)
{
    int np = ++tot;
    sz[tot] = 1, sam[np].len = sam[p].len + 1;
    for (; p&&!sam[p].ch[c]; p = sam[p].fa) sam[p].ch[c] = np;
    if (!p) sam[np].fa = 1;
    else
    {
        int q = sam[p].ch[c];
```

```
        if (sam[q].len == sam[p].len + 1) sam[np].fa = q;
        else
        {
            int nq = ++tot;
            sam[nq] = sam[q], sam[nq].len = sam[p].len + 1;
            sam[q].fa = sam[np].fa = nq;
            for (; p && sam[p].ch[c] == q; p = sam[p].fa) sam[p].ch[c] = nq;
        }
    }
    return np;
}
void insert(string s)
{
    int p = 1;
    for (auto c:s)
    {
        int t = c-'a';
        if (!tr[p][t]) tr[p][t] = ++idx;
        p = tr[p][t];
    }
}
void build()
{
    queue<int> q;
    q.push(pos[1] = 1);
    while (q.size())
    {
        int t = q.front();
        q.pop();
        for (int i = 0, p; i < 26; i++)
            if (p = tr[t][i])
            {
                pos[p] = extend(pos[t], i);
                q.push(p);
            }
    }
}
-------------------------------------
for (int i = 1; i <= n; i++)
{
    string s;
    cin >> s;
    insert(s);
}
build();
```

线段树合并维护 SAM 的 endpos

例子： CF666E。给定一个串 s 以及一个字符串数组 $T_{1\dots m}$,q 次询问, 每次问 s 的子串 $s[pl,pr]$ 在 $T[l,r]$ 中的哪个串里的出现次数最多, 并输出出现次数。

```
const int N = 500010, M = 50010, K = M * 2;
struct SAM
{
    int len, fa;
    int ch[26];
}sam[K];
struct Node
{
    int l, r;
    int cnt, maxv, maxvfrom;
}tr[K << 6];
int h[K], ne[K], e[K], edgeidx;
int fa[K][17], minlen[N][17];
int ed[N], maxlen[N];
int n, m, tot = 1;
char s[N], t[M];
int root[K], idx;
void pushup(int p)
{
    if (tr[tr[p].l].maxv >= tr[tr[p].r].maxv)
    {
        tr[p].maxv = tr[tr[p].l].maxv;
        tr[p].maxvfrom = tr[tr[p].l].maxvfrom;
```

```
    }
    else
    {
        tr[p].maxv = tr[tr[p].r].maxv;
        tr[p].maxvfrom = tr[tr[p].r].maxvfrom;
    }
}
int modify(int p, int l, int r, int x, int val)
{
    if (!p) p = ++idx;
    if (l == r)
    {
        tr[p].maxv = tr[p].cnt += val;
        tr[p].maxvfrom = x;
        return p;
    }
    int mid = (l + r) >> 1;
    if (x <= mid) tr[p].l = modify(tr[p].l, l, mid, x, val);
    else tr[p].r = modify(tr[p].r, mid + 1, r, x, val);
    pushup(p);
    return p;
}
int merge(int p, int q, int l, int r)
{
    if (!p||!q) return p | q;
    int nw = ++idx;
    if (l == r)
    {
        tr[nw].cnt = tr[nw].maxv = tr[p].cnt + tr[q].cnt;
        tr[nw].maxvfrom = tr[p].maxvfrom;
        return nw;
    }
    int mid = (l + r) >> 1;
    tr[nw].l = merge(tr[p].l, tr[q].l, l, mid);
    tr[nw].r = merge(tr[p].r, tr[q].r, mid + 1, r);
    pushup(nw);
    return nw;
}
array<int, 2> query(int p, int l, int r, int ql, int qr)
{
    if (!p) return {0, 0};
    if (ql<= l && qr >= r) return {tr[p].maxv, tr[p].maxvfrom};
    int mid = (l + r) >> 1;
    array<int, 2> res = {0, 0};
    if (ql <= mid)
    {
        auto t = query(tr[p].l, l, mid, ql, qr);
        if (t[0] > res[0]) res = t;
    }
    if (qr > mid)
    {
        auto t = query(tr[p].r, mid + 1, r, ql, qr);
        if (t[0] > res[0]) res = t;
    }
    return res;
}
int extend(int c, int last, int id)//如果不需要维护 id 不需要加入
{
    if (sam[last].ch[c])//这个状态之前已经存在过了
    {
        int p = last, x = sam[p].ch[c];
        if (sam[p].len + 1 == sam[x].len)
        {
            root[x] = modify(root[x], 1, m, id, 1);
            return x;
        }
        else
        {
            int y = ++tot;
            sam[y].len = sam[p].len + 1;
            for (int i = 0; i < 26; i++) sam[y].ch[i] = sam[x].ch[i];
            while (p && sam[p].ch[c] == x) sam[p].ch[c] = y, p = sam[p].fa;
            sam[y].fa = sam[x].fa, sam[x].fa = y;
            root[y] = modify(root[y], 1, m, id, 1);
            return y;
```

```
        }
    }
    int z = ++tot, p = last;
    sam[z].len = sam[last].len + 1;
    while (p&&!sam[p].ch[c]) sam[p].ch[c] = z, p = sam[p].fa;
    if (!p) sam[z].fa = 1;
    else
    {
        int x = sam[p].ch[c];
        if (sam[p].len + 1 == sam[x].len) sam[z].fa = x;
        else
        {
            int y = ++tot;
            sam[y].len = sam[p].len + 1;
            for (int i = 0; i < 26; i++) sam[y].ch[i] = sam[x].ch[i];
            while (p && sam[p].ch[c] == x) sam[p].ch[c] = y, p = sam[p].fa;
            sam[y].fa = sam[x].fa, sam[z].fa = sam[x].fa = y;
        }
    }
    root[z] = modify(root[z], 1, m, id, 1);
    return z;
}
void dfs(int u)
{
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        fa[j][0] = u, minlen[j][0] = sam[u].len;
        for (int k = 1; k <= 16; k++)
        {
            fa[j][k] = fa[fa[j][k - 1]][k - 1];
            minlen[j][k] = min(minlen[j][k - 1], minlen[fa[j][k - 1]][k - 1]);
        }
        dfs(j);
        root[u] = merge(root[u], root[j], 1, m);
    }
}
int find(int l, int r)
{
    int p = ed[r], len = r - l + 1;
    for (int k = 16; k >= 0; k--)
        if (minlen[p][k] >= len)
            p = fa[p][k];
    return p;
}
void solve()
{
    n = strlen(s + 1);
    cin >> m;
    for (int i = 1; i <= m; i++)
    {
        cin >> t + 1;
        int j = strlen(t + 1), last = 1;
        for (int k = 1; k <= j; k++) last = extend(t[k]-'a', last, i);
    }
    for (int i = 1, p = 1, cnt = 0; i <= n; i++)
    {
        int t = s[i]-'a';
        while (p >1&&!sam[p].ch[t]) p = sam[p].fa, cnt = sam[p].len;
        if (sam[p].ch[t]) p = sam[p].ch[t], cnt++;
        ed[i] = p, maxlen[i] = cnt;
    }
    memset(h, -1, sizeof h);
    for (int i = 2; i <= tot; i++) add(sam[i].fa, i);
    dfs(1);
    while (q--)
    {
        int l, r, pl, pr;
        if (pr - pl + 1 > maxlen[pr]) cout << l<<" 0\n";
        else
        {
            int u = find(pl, pr);
            auto t = query(root[u], 1, m, l, r);
            if (!t[0]) cout << l<<" 0\n";
```

```
            else cout << t[1]<<" "<<t[0]<<"\n";
        }
    }
}
```

例子： [HEOI2016/TJOI2016] 字符串

对于一个字符串，每次询问 $s[a\dots b]$ 的所有子串和 $s[c,d]$ 的 $\max_{\text{lcp}}$。

考虑二分答案 mid，那么问题转化为是否存在 $s[a\dots b]$ 的子串包含子串 $s[c,c+\text{mid}-1]$。在 parent 树上倍增找到 $s[c,c+\text{mid}-1]$ 所在的节点，只需判断该节点的 endpos 集合中是否存在 $[a+\text{mid}-1,b]$ 中的数，线段树合并即可。

```
struct SAM
{
    int len, fa;
    int ch[26];
}sam[N];
struct Node
{
    int l, r, sum;
}tr[N << 5];
int tot = 1, last = 1;
int n, q, root[N], idx;
vector<int> g[N];
int fa[N][M], minlen[N][M];
int ed[N];
void pushup(int p)
{
    tr[p].sum = tr[tr[p].l].sum + tr[tr[p].r].sum;
}
int merge(int p, int q, int l, int r)
{
    if (!p||!q) return p | q;
    int nw = ++idx;
    if (l == r)
    {
        tr[nw].sum = tr[p].sum + tr[q].sum;
        return nw;
    }
    int mid = (l + r) >> 1;
    tr[nw].l = merge(tr[p].l, tr[q].l, l, mid);
    tr[nw].r = merge(tr[p].r, tr[q].r, mid + 1, r);
    pushup(nw);
    return nw;
}
int modify(int p, int l, int r, int x, int v)
{
    if (!p) p = ++idx;
    if (l == r)
    {
        tr[p].sum += v;
        return p;
    }
    int mid = (l + r) >> 1;
    if (x <= mid) tr[p].l = modify(tr[p].l, l, mid, x, v);
    else tr[p].r = modify(tr[p].r, mid + 1, r, x, v);
    pushup(p);
    return p;
}
void extend(int c, int id)
{
    int p = last, np = last = ++tot;
    root[tot] = modify(root[tot], 1, n, id, 1), ed[id] = tot;
    sam[np].len = sam[p].len + 1;
    for (; p&&!sam[p].ch[c]; p = sam[p].fa) sam[p].ch[c] = np;
    if (!p) sam[np].fa = 1;
    else
    {
        int q = sam[p].ch[c];
        if (sam[q].len == sam[p].len + 1) sam[np].fa = q;
        else
        {
            int nq = ++tot;
            sam[nq] = sam[q], sam[nq].len = sam[p].len + 1;
            sam[q].fa = sam[np].fa = nq;
```

```
            for (; p && sam[p].ch[c] == q; p = sam[p].fa) sam[p].ch[c] = nq;
        }
    }
}
void dfs(int u)
{
    for (int j:g[u])
    {
        fa[j][0] = u, minlen[j][0] = sam[u].len;
        for (int k = 1; k < M; k++)
        {
            fa[j][k] = fa[fa[j][k - 1]][k - 1];
            minlen[j][k] = min(minlen[j][k - 1], minlen[fa[j][k - 1]][k - 1]);
        }
        dfs(j);
        root[u] = merge(root[u], root[j], 1, n);
    }
}
int query(int p, int l, int r, int ql, int qr)
{
    if (ql<= l && qr >= r) return tr[p].sum;
    int mid = (l + r) >> 1, res = 0;
    if (ql <= mid) res += query(tr[p].l, l, mid, ql, qr);
    if (qr > mid) res += query(tr[p].r, mid + 1, r, ql, qr);
    return res;
}
bool check(int a, int b, int c, int d, int mid)
{
    int x = ed[c + mid - 1];
    for (int k = M - 1; k >= 0; k--)
        if (minlen[x][k] >= mid)
            x = fa[x][k];
    return query(root[x], 1, n, a + mid - 1, b) >0;
}
void solve()
{
    cin >> n >> q;
    string s;
    cin >> s;
    for (int i = 0; i < n; i++) extend(s[i]-'a', i + 1);
    for (int i = 2; i <= tot; i++) g[sam[i].fa].push_back(i);
    dfs(1);
    while (q--)
    {
        int a, b, c, d;
        int l = 0, r = min(b - a + 1, d - c + 1);
        while (l < r)
        {
            int mid = l + r + 1 >>1;
            if (check(a, b, c, d, mid)) l = mid;
            else r = mid - 1;
        }
        cout << l<<"\n";
    }
}
```

4.8 Manacher

统一奇偶回文串,R_i 是扩展后字符串的最长回文半径, 那么以当前位置为回文中心的回文串长度是 $R_i - 1$

```
char t[N]; //记得开两倍空间
int n, m, R[N];
void manacher(string & s)
{
    t[0] = '!', t[m = 1] = '@';
    for (auto & c:s) t[++m] = c, t[++m] = '@';
    t[++m] = '#';
    for (int i = 1, c = 0, r = 0; i < m; i++)
    {
        R[i] = r < i?1:min(R[c * 2 - i], r - i + 1);
        while (t[i - R[i]] == t[i + R[i]]) R[i]++;
        if (i + R[i] - 1 > r) c = i, r = i + R[i] - 1;
    }
```

```
}
```

4.9 回文自动机

```
struct PAM
{
    int sz, tot, last;
    //pam 的节点数 pam 加入的字符数 当前匹配到哪个指针
    int cnt[N], tr[N][M], len[N];
    //节点所代表的不同的回文串的个数（其实就是深度）pam 的指针 当前节点（回文串）的长度
    int ex[N], fail[N]; //当前状态所表示回文串的出现次数 指向当前状态的最长回文真后缀
    char s[N]; //pam 的字符
    bool st[N];
    int node(int l)//建一个长度为 l 的新节点
    {
        sz++;
        memset(tr[sz], 0, sizeof tr[sz]);
        len[sz] = l, fail[sz] = cnt[sz] = 0;
        return sz;
    }
    void init()
    {
        sz = -1, last = 0; //初始化
        s[tot = 0] = '$'; //插入一个特殊字符
        node(0), node(-1); //先建偶根 再建奇根
        fail[0] = 1; //偶根的 fail 指针指向奇根
    }
    int getfail(int x)
    {
        while (s[tot - len[x] - 1] != s[tot]) x = fail[x];
        return x; //跳到可以匹配当前字符的最长回文真后缀
    }
    void add(int c)
    {
        s[++tot] = c; //加入当前字符
        int now = getfail(last); //跳到可以匹配当前字符的最长回文真后缀
        if (!tr[now][c])//如果说不存在 cXXXc 这样的回文子串的状态
        {
            int x = node(len[now] + 2); //那就新建一个
            fail[x] = tr[getfail(fail[now])][c]; //更新一下 fail
            tr[now][c] = x; //更新一下转移
        }
        last = tr[now][c]; //更新一下当前的指针
        if (!st[last]) cnt[last] = cnt[fail[last]] + 1, st[last] = true;
        ex[last]++;
    }
}pam; //树形 dp 上传每个串的出现次数
void pushup(){ for (int i = pam.sz; i >= 1; i--) pam.ex[pam.fail[i]] += pam.ex[i]; }
void solve()
{
    pam.init();
    for (int i = 1; i <= n; i++)
    {//动态加入字符，求得以第 i 个字符结尾的回文子串个数
        pam.add(s[i]-'a');
        cout << pam.cnt[pam.last]<<" ";
    }
}
```

求最长双倍回文串

```
vector<int> g[N];
int fa[N][19];
void dfs(int u, int father)
{
    fa[u][0] = father;
    for (int i = 1; i <= 18; i++) fa[u][i] = fa[fa[u][i - 1]][i - 1];
    for (int j:g[u]) dfs(j, u);
}
void solve()
{
    int n;
```

```
    pam.init();
    string s;
    for (auto c:s) pam.add(c-'a');
    for (int i = 1; i <= pam.sz; i++) g[pam.fail[i]].push_back(i);
    dfs(0, 0);
    int ans = 0;
    for (int i = 1; i <= pam.sz; i++)
    {
        if (pam.len[i] % 4) continue;
        int tar = pam.len[i] / 2;
        int t = i;
        for (int k = 18; k >= 0; k--)
            if (pam.len[fa[t][k]] > tar)
                t = fa[t][k];
        t = fa[t][0];
        if (pam.len[t] == tar) ans = max(ans, pam.len[i]);
    }
    cout << ans << '\n';
}
```

4.10 可撤销回文自动机

可撤销回文自动机支持在字符串末尾添加和删除字符，动态维护所有回文子串信息。

核心优化：使用 $mem[u][c]$ 数组预计算从节点 u 开始匹配字符 c 时应该跳转的位置，避免重复的 $fail$ 跳转，使得每次操作摊还时间复杂度为 $O(1)$。

设 $mem[u][c] = v$ 表示从节点 u 开始，当匹配字符 c 失败时，应该跳转到的节点 v，满足：

$$mem[u][c] = \begin{cases} u & \text{如果} s[|s| - len[u] - 1] = c \\ mem[fail[u]][c] & \text{否则} \end{cases}$$

例题：牛客 14055F - 给定模板字符串 S，生成长度为 n 的字符串 T - 对 T 的每个回文子串 P，得分为 $OCC_S(P)$（P 在 S 中的出现次数）- 目标：计算 $\sum_{T\in\Sigma^n} V(T)$，其中 $V(T) = \sum_{\substack{1\le i\le j\le n \\ T[i..j]\text{是回文}}} OCC_S(T[i..j])$ - 支持对 S 末尾动态添加/删除字符

```
struct Palindromic_AutoMaton
{
    vector<int> str; // 存储字符串
    int nxt[N][A], fail[N], len[N]; // 转移边，失配边，回文串长度
    int num[N], w[N], fa[N], last, tot; // 出现次数，权值，父节点，当前节点，总节点数
    int mem[N][A]; // 优化查找 fail 的数组
    stack<pair<int, bool>> ops; // 操作栈：(上一个节点，是否新建节点)

    void clear()
    {
        len[fail[1] = 0] = 0; // 偶根
        len[fail[0] = 1] = -1; // 奇根
        str.clear();
        str.push_back(-1); // 边界字符
        last = 0, tot = 1;
        while (!ops.empty()) ops.pop();
        w[0] = w[1] = 0;
        memset(mem[1], 0, sizeof mem[1]);
        for (int ch = 0; ch < A; ch++) mem[0][ch] = 1; // 奇根的 mem 全指向奇根
        memset(nxt[0], 0, sizeof nxt[0]);
        memset(nxt[1], 0, sizeof nxt[1]);
    }

    int newnode(int _len)
    {
        len[++tot] = _len;
        fail[tot] = num[tot] = w[tot] = 0;
        memset(mem[tot], 0, sizeof mem[tot]);
        memset(nxt[tot], 0, sizeof nxt[tot]);
        return tot;
    }

    void append(int ch)
    {
        str.push_back(ch);
        int fat = last, strlen = str.size() - 1;
        // 使用 mem 数组优化：如果当前节点不能匹配，直接跳到预计算的位置
        if (str[strlen - len[last] - 1] != str[strlen]) fat = mem[fat][ch];
        bool create_new = false;
```

```
        if (!nxt[fat][ch]) // 需要新建节点
        {
            create_new = true;
            int cur = newnode(len[fat] + 2);
            fa[cur] = fat; // 记录父节点便于撤销
            fail[cur] = nxt[mem[fat][ch]][ch]; // 设置失配指针
            // 更新 mem 数组: 继承 fail 节点的 mem, 并修正当前字符
            for (int i = 0; i < A; i++) mem[cur][i] = mem[fail[cur]][i];
            mem[cur][str[strlen - len[fail[cur]]]] = fail[cur];
            nxt[fat][ch] = cur;
            // 根据题目要求计算权值
            if (len[cur] <= L) w[cur] = 1ll * pw[L - len[cur]] * (L - len[cur] + 1) % mod;
            w[cur] = (w[cur] + w[fail[cur]]) % mod; // 继承 fail 节点权值
        }
        ops.push({last, create_new}); // 记录操作
        last = nxt[fat][ch];
        ++num[last];
        ans = (ans + w[last]) % mod;
    }

    void cancel() // 撤销操作
    {
        if (ops.empty()) return;
        auto [pos, flag] = ops.top();
        ops.pop();
        ans = (ans + mod - w[last]) % mod; // 减去贡献
        if (flag) // 如果创建了新节点, 需要删除
        {
            --tot;
            nxt[fa[last]][str.back()] = 0; // 删除父节点到该节点的边
        }
        last = pos; // 恢复上一个状态
        str.pop_back(); // 删除字符
    }
} pam;
void solve()
{
    ans = 0;
    cin >> L >> m >> s;
    pam.clear();
    for (auto c : s) pam.append(c - 'a');
    cout << ans << "\n";
    while (m--)
    {
        int opt;
        cin >> opt;
        if (opt == 2) pam.cancel();
        else
        {
            string t;
            cin >> t;
            for (auto c : t) pam.append(c - 'a');
        }
        cout << ans << "\n";
    }
}
```

4.11 bitset 优化字符串匹配

例题 CF914F

给你一个字符串 s, 共有 q 次操作, 每个都是下面两种形式的一种。

1 i c: 将字符串 s 的第 i 项变为字符 c。

2 l r y: 求字符串 y 在 s 的子串 $s[l,r]$ 中作为子串出现的次数。

```
bitset<N> c[26], ans;
char s[N];
void solve()
{
    cin >> s + 1;
    int n = strlen(s + 1), q;
    for (int i = 1; i <= n; i++) c[s[i]-'a'][i] = 1;
    //记录下每个字符在 s 中出现的位置
    cin >> q;
```

```
    while (q--)
    {
        if (t == 1)
        {
            int x;
            char ch;
            c[s[x]-'a'][x] = 0, c[ch-'a'][x] = 1, s[x] = ch;
        }
        else
        {
            int l, r;
            string y;
            if (y.size() > r - l + 1)
            {
                cout<<"0\n";
                continue;
            }
            ans.set(); //一开始所有位置都可能是答案
            for (int i = 0; i<y.size(); i++) ans s&= = c[y[i]-'a'] >> i;
            //若当前位置为合法的位置 那么当前位置 + i 处必须为字符 y[i] 所以 & 一下判不合法
            int L = l, R = r - y.size() + 1;
            //bitset 给出的后缀和 要查 sum[L~R] 即查 suf[L] - suf[R + 1]
            cout << (ans >> L).count() - (ans >> R + 1).count()<<"\n";
        }
    }
}
```

例题 CF963D

给你一个字符串 s, 有 n 个询问, 第 i 个询问包含一个整数 k 和一个字符串 m。要求找到一个字符串 t , 使得 t 是 s 的子串并且 m 至少在 t 中出现了 k 次。你只需要求出 t 的最短长度。

```
bitset<N> c[26], ans;
char s[N];
int p[N];
void solve()
{
    cin >> s + 1;
    int n = strlen(s + 1), q;
    for (int i = 1; i <= n; i++) c[s[i]-'a'][i] = 1;
    cin >> q;
    while (q--)
    {
        int k;
        string t;
        cin >> k >> t;
        ans.set();
        for (int i = 0; i<t.size(); i++) ans s&= = c[t[i]-'a'] >> i;
        int cnt = 0;
        //一定要利用这两个函数 如果遍历去找 复杂度是错的
        for (int i = ans._Find_first(); i <= n; i = ans._Find_next(i)) p[++cnt] = i;
        if (cnt < k)
        {
            cout<<"-1\n";
            continue;
        }
        int ans = 1e9;
        for (int i = k; i <= cnt; i++) ans = min(ans, p[i] - p[i - k + 1]);
        cout << ans + t.size()<<"\n";
    }
}
```

4.12 字符串问题中的根号分治思想

例子： CF710F, 三种操作, 加字符串, 删字符串, 查询集合中所有字符串在给定串中出现的次数, 强制在线。

考虑哈希优化匹配。查询的时候, 只需要枚举集合中字符串的长度作为查询串的子串长度即可。设 $len = \sum |s|$。那么不同长度最多只有 $\sqrt{len}$ 种, 利用桶 + 哈希优化。

例子： 给定 n 个模板串，m 个查询串，依次查询每一个查询串是多少个模板串的子串。

正解:GSAM+ 线段树合并。

设置一个阈值 B，长度大于 B 的查询串不超过 $\frac{\sum |q_i|}{B}$ 个，对这些串直接跑 KMP 暴力，复杂度是 $O(\frac{\sum |q_i|}{B} \sum |s_i|)$。对于长度不大于 B 的串，考虑预处理答案。枚举模板串中长度为 $[1, B]$ 的所有子串，记录下它们的出现次数，利用哈希优化到 $O(B \sum |s_i|)$。故总复杂度为 $O(\frac{\sum |q_i|}{B} \sum |s_i| + B \sum |s_i|)$，当 B 取 $\sqrt{\sum |q_i|}$ 时，复杂度最优，为 $O(\sqrt{\sum |q_i|} \sum |s_i|)$。

4.13 Border Series

一个串 S 的所有 Border 按长度排序后，可以被划分为 $O(\log n)$ 个等差数列

例题：对于每一个位置 i，求 $a_i \times \sum b[i - \text{border}_i + 1]$，强制在线

考虑根号分治，设置一个阈值 B，当公差小于 B 时，维护一个等差数列的前缀和，当公差大于 B 时，暴力在 Border 树上跳

```
int n, s[N], a[N], b[N], ne[N];
int diff[N], sum[M][N], depth[N], anc[N];
void add(int u, int fa)
{
    diff[u] = u - fa, depth[u] = depth[fa] + 1;
    if (diff[u] == diff[fa]) anc[u] = anc[fa];
    else anc[u] = fa;
}
int query(int u, int x)//b[x - border + 1]
{
    int res = 0;
    while (u)
    {
        int d = diff[u];
        if (d < M)
        {
            int len = depth[u] - depth[anc[u]];
            res += sum[d][x - u + 1 + d * (len - 1)] - sum[d][max(0, x - u + 1 - d)];
            u = anc[u];
        }
        else res += b[x - u + 1], u = ne[u];
    }
    return res;
}
void solve()
{
    cin >> n;
    long long ans = 0;
    for (int i = 1, j = 0; i <= n; i++)
    {
        cin >> s[i] >> a[i] >> b[i];
        s[i] = (ans + s[i]) % n;
        while (j && s[j + 1] != s[i]) j = ne[j];
        j += s[j + 1] == s[i];
        if (i == 1) j = 0;
        ne[i] = j;
        add(i, ne[i]);
        for (int k = 1; k < M; k++)
            if (i < k) sum[k][i] = b[i];
            else sum[k][i] = sum[k][i - k] + b[i];
        ans += 1ll * a[i] * query(i, i);
        cout << ans << "\n";
    }
}
```

4.13.1 Palindrome series

用途：利用回文串的周期性质，将回文后缀划分为 $O(\log n)$ 个等差数列，优化 DP 转移。

核心思想：类似 Border series，回文后缀也具有等差数列的结构特性。

应用：加速回文划分 DP，将 $O(n^2)$ 的转移优化到 $O(n \log n)$。

回文后缀和 Border 也有着强联系，所以回文后缀也可以被划分为 $O(\log)$ 个等差数列

例: 给你一个长度为偶数的字符串，求它的偶数长度的回文划分方案数

有 dp 方程:$dp_i = \sum_{0 \leqslant j < i} dp_j (s[j+1, i] \text{ 是偶回文串})$

利用等差数列的性质加速转移

```
struct PAM
{
    int sz, tot, last;
    int tr[N][M], len[N], fail[N], jump[N], diff[N];
    char s[N];
    int node(int l)
    {
        sz++;
        memset(tr[sz], 0, sizeof tr[sz]);
        len[sz] = l, fail[sz] = 0;
        return sz;
```

```
    }
    void init()
    {
        sz = -1, last = 0;
        s[tot = 0] = '$';
        node(0), node(-1);
        fail[0] = 1;
    }
    int getfail(int x)
    {
        while (s[tot - len[x] - 1] ≠ s[tot]) x = fail[x];
        return x;
    }
    void add(int c)
    {
        s[++tot] = c;
        int now = getfail(last);
        if (!tr[now][c])
        {
            int x = node(len[now] + 2);
            fail[x] = tr[getfail(fail[now])][c];
            tr[now][c] = x;
            diff[x] = len[x] - len[fail[x]];
            jump[x] = diff[x] == diff[fail[x]]?jump[fail[x]]:fail[x];
        }
        last = tr[now][c];
    }
}pam;
int dp[N], g[N];
int cal(int i)
{
    int res = 0;
    for (int p = pam.last; p >1; p = pam.jump[p])
    {
        g[p] = dp[i - pam.len[pam.jump[p]] - pam.diff[p]];
        if (pam.diff[p] == pam.diff[pam.fail[p]]) g[p] = (g[p] + g[pam.fail[p]]) % mod;
        res += g[p];
    }
    return res % mod;
}
void solve()
{
    string tt, s = " ";
    cin >> tt;
    string t = string(tt.rbegin(), tt.rend());
    int n = tt.size();
    for (int i = 0; i < n / 2; i++) s += tt[i], s += t[i];
    pam.init();
    dp[0] = 1;
    for (int i = 1; i <= n; i++)
    {
        pam.add(s[i]-'a');
        dp[i] = cal(i) * (i % 2 == 0);
    }
    cout << dp[n]<<"\n";
}
```

5 动态规划

5.1 背包

5.1.1 01 背包

```
for (int i = 1; i <= n; i++)
    for (int j = v; j >= vi; j--) f[j] = max(f[j], f[j - vi] + wi);
```

5.1.2 完全背包

```
for (int i = 1; i <= n; i++)
    for (int j = vi; j <= v; j++) f[j] = max(f[j], f[j - vi] + wi);
```

5.1.3 二进制优化多重背包

```cpp
vector<array<int, 2>> item;
for (int i = 1; i <= n; i++)
{
    cin >> v >> w >> s;
    for (int j = 1; j <= s; s -= j, j<<=1) item.push_back({v * j, w * j});
    if (s) item.push_back({v * s, w * s});
}
for (auto [v, w]:item)
    for (int i = m; i >= v; i--)
        f[i] = max(f[i], f[i - v] + w);
```

5.1.4 分组背包

```cpp
for (int i = 1; i <= n; i++)
{
    int count; //这组有 count 个物品
    for (int j = m; j >= 0; j--)
        for (int k = 1; k <= count; k++)
            if (j >= v[k]) f[j] = max(f[j], f[j - v[k]] + w[k]);
}
```

5.1.5 树上背包

树上每个节点都有点权, 最多选 m 个儿子, 求最大价值

树上 dp 时, 加入儿子的贡献, 可能会改变父亲的 dp 值造成重复计数, 所以一般开一个 temp 数组计数, 避免重复计数

```cpp
int h[N], ne[N << 1], e[N << 1], w[N << 1], idx, n, m, val[N], f[N][N], temp[N], sz[N];
void dfs(int u, int fa)
{
    sz[u] = 0;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j == fa) continue;
        val[j] = w[i];
        dfs(j, u);
        for (int k = 0; k <= sz[u] + sz[j] && k <= m; k++) temp[k] = 0;
        for (int k = 0; k <= sz[u] && k <= m; k++)
            for (int x = 0; x <= sz[j] && x + k <= m; x++)
                temp[x + k] = max(temp[x + k], f[u][k] + f[j][x]);
        sz[u] += sz[j];
        for (int k = 0; k <= sz[u] && k <= m; k++) f[u][k] = temp[k];
    }
    sz[u]++;
    for (int i = min(sz[u], m); i >= 1; i--) f[u][i] = f[u][i - 1] + val[u];
}
int main()
{
    for (int i = 0; i < n - 1; i++) add(a, b, c), add(b, a, c);
    dfs(1, -1);
    cout << f[1][m];
}
```

5.2 换根 DP

```cpp
void dfs1(int u, int fa)//第一遍 dfs 求出以某个点为根时的答案
{
    sz[u] = 1;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j == fa) continue;
        dfs1(j, u); //先递归再处理
        sz[u] += sz[j], f[u] += f[j];
    }
```

```
    f[u] += sz[u] - 1;
}
//f[u] 的意思是 在 u 的子树中 u 作为根的时候的答案
//v[u] 的意思是 u 的父亲作为 u 的子树时（此时整棵树以 u 为根）的 u 的答案的贡献
//所以以每个点为根时的答案是 f[u] + v[u]
void dfs2(int u, int fa)//开始换根 dp
{
    if (u != 1) v[u] = (f[fa] + v[fa]) - (f[u] + sz[u]) + (n - sz[u]); //先处理再递归
    //以 u 的父亲为根时的答案减去 u 对 u 的父亲的贡献（u 的其他子树的贡献）
    //再加上 u 的父亲作为 u 的儿子时对答案的贡献
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j == fa) continue;
        dfs2(j, u);
    }
}
void solve()
{
    dfs1(1, 0), dfs2(1, 0);
    for (int i = 1; i <= n; i++)
        if (ans < f[i] + v[i]) ans = f[i] + v[i], id = i;
}
```

例子： 求根使得 $\sum_{i=1}^{n} dist_{i,root} \times a_i$ 最大

```
int h[N], ne[M], e[M], idx, a[N], f[N], v[N], sum[N];
void dfs1(int u, int fa)
{
    sum[u] = a[u];
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j == fa) continue;
        dfs1(j, u);
        f[u] += f[j] + sum[j];
        sum[u] += sum[j];
    }
}
void dfs2(int u, int fa)
{
    if (u != 1) v[u] = (f[fa] + v[fa]) - (f[u] + sum[u]) + (sum[1] - sum[u]);
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j == fa) continue;
        dfs2(j, u);
    }
}
void solve()
{
    for (int i = 1; i<= n; i++) cin >> a[i];
    dfs1(1, -1), dfs2(1, -1);
    for (int i = 1; i <= n; i++)
        if (f[i] + v[i] > ans) ans = f[i] + v[i];
}
```

5.3 状压 DP

$O(3^n)$ 枚举子集

```
for (int i = 1; i < (1 << n); i++)
    for (int j = i; ; j = (j - 1) & i)
    {
        // do something
        if (!j) break;
    }
```

5.4 单调队列优化 dp

若转移范围是一段区间, 而且区间的左右端点是单调移动的, 可以使用单调队列维护

单调队列转移前, 先把合适的点放到单调队列里, 再弹出队列

例: 给定 k 个时间段, 每个时间段可以翻转, 求 $2n$ 时间后, 两面同样时间的最小翻转数是多少

$$f_{i,j} = \{ \ f_{i-1,j} f_{i-1,k} + 1(k \in [\max(0, l_i - j), r_i - j]) f_{i-1,k} + 2(k \in [\max(0, j - len, j)]) \tag{1}$$

第一个方程直接维护即可

第二个方程倒序枚举 j, 那么区间单调向右移动

第三个方程正序枚举 j, 那么区间单调向右移动

```
void solve()
{
    int n, k;
    cin >> n >> k;
    memset(dp[0], 0x3f, sizeof dp[0]);
    dp[0][0] = 0;
    for (int i = 1; i <= k; i++)
    {
        int t = i & 1;
        memset(dp[t], 0x3f, sizeof dp[t]);
        dp[t][0] = 0, hh = 0, tt = -1;
        cin >> l[i] >> r[i];
        int len = r[i] - l[i];
        for (int j = 0; j <= n && j <= r[i]; j++)
        {
            if (j >= 0)//如果右区间端点现在可以被拓展
                while (hh > tt || q[tt] < j)//如果队列为空或者最后一个没有拓展到右端点
                {
                    int x;
                    if (hh > tt) x = 0; //为空就放入可以拓展的第一个点
                    else x = q[tt] + 1; //否则就是下一个点
                    while (hh <= tt && dp[t ^ 1][x] <= dp[t ^ 1][q[tt]]) tt--;
                    q[++tt] = x;
                }
            while (hh <= tt && q[hh] < max(0ll, j - len)) hh++; //把过时的点去掉
            dp[t][j] = dp[t ^ 1][j];
            if (hh <= tt) dp[t][j] = min(dp[t][j], dp[t ^ 1][q[hh]] + 2);
        }
        hh = 0, tt = -1;
        for (int j = r[i]; j >= 0; j--)
        {
            if (r[i] - j >= 0)//如果右区间端点现在可以被拓展
                while (hh > tt || q[tt] < r[i] - j)//如果队列为空或者最后一个没有拓展到右端点
                {
                    int x;
                    if (hh > tt) x = 0; //为空就放入可以拓展的第一个点
                    else x = q[tt] + 1; //否则就是下一个点
                    while (hh <= tt && dp[t ^ 1][x] <= dp[t ^ 1][q[tt]]) tt--;
                    q[++tt] = x;
                }
            while (hh <= tt && q[hh] < max(0ll, l[i] - j)) hh++; //把过时的点去掉
            if (hh <= tt) dp[t][j] = min(dp[t][j], dp[t ^ 1][q[hh]] + 1);
        }
    }
    if (dp[k & 1][n] < 2 * N) cout<<"Full\n"<<dp[k & 1][n]<<'\n';
    else cout<<"Hungry\n";
}
```

如果区间里不是每一个点都可以被拓展, 那么就自己更新的时候加入

例: 高桥君在玩双六棋, 棋盘格由用 0 到 N 编号的共 $N+1$ 个格子构成。每一回合, 高桥君会扔一个点数 1 到 M 的骰子。如果高桥君当前在第 i 格, 骰子扔出 k 点, 高桥君就前进到第 $i+k$ 格。如果此时 $i+k > N$, 高桥君立刻输掉。另外, 棋盘上还有若干个“GameOver 格”, 如果高桥停在这些格子, 也立刻输掉游戏。

假设高桥君可以自由控制骰子的点数, 那么他从 0 号格子出发, 到达 N 号格子, 最短需要多少回合? 输出用最短回合到达 N 格时, 每回合骰子的点数组成的序列; 如果无法到达 N 号格子, 输出-1。

```
void solve()
{
    cin >> n >> m >> s;
    hh = 0, tt = -1;
    //[i - m, i - 1]
```

```
    memset(dp, 0x3f, sizeof dp);
    dp[0] = 0, q[++tt] = 0;
    for (int i = 1; i <= n; i++)
    {
        if (s[i]=='1') continue;
        while (hh <= tt && q[hh] < i - m) hh++;
        if (hh <= tt) dp[i] = dp[q[hh]] + 1;
        while (hh <= tt && dp[q[tt]] >= dp[i]) tt--;
        q[++tt] = i;
    }
    cout << dp[n]<<"\n";
}
```

5.5 基环树 DP

```
struct Ringtree
{
    vector<int> g[N], cir[N];
    int d[N], id[N], q[N], n, cnt, hh, tt;
    bool on_cir[N];
    void init(int x)
    {
        n = x, cnt = 0;
        for (int i = 1; i <= n; i++)
        {
            g[i].clear(), cir[i].clear();
            d[i] = id[i] = on_cir[i] = 0;
        }
    }
    void add(int a, int b) { g[a].push_back(b), g[b].push_back(a); }
    void dfs(int u, int fa)
    {
        id[u] = cnt;
        for (int j:g[u])
            if (!on_cir[j] && j != fa) dfs(j, u);
    }
    void build()
    {
        hh = 0, tt = -1;
        for (int i = 1; i <= n; i++)
            if ((d[i] = g[i].size()) == 1) q[++tt] = i;
        while (hh <= tt)
        {
            int t = q[hh++];
            for (int j:g[t])
                if (--d[j] == 1) q[++tt] = j;
        }
        for (int i = 1, t, flag; i <= n; i++)
            if (d[i] >1)
            {
                cnt++, t = i, flag = 1;
                while (flag)
                {
                    cir[cnt].push_back(t);
                    d[t] = flag = 0, on_cir[t] = true;
                    for (int j:g[t])
                        if (d[j] >1)
                        {
                            flag = 1, t = j;
                            break;
                        }
                }
                for (int x:cir[cnt]) dfs(x, x);
            }
    }
}tr;
//tr.init(n) 初始化 tr.add(a, b) 加边 最后 tr.build() 处理基环树
```

5.6 斜率优化

DP 式子形如 $f_i = \min_{j=0}^{i-1}\{t(j) - g(i) \times r(j) + h(i)\}$ 把 $t(j)$ 视作 y_j, 把 $g(i)$ 视作 k_i, 把 $r(j)$ 视作 x_j, 把 $f_i - h(i)$ 视作 b_i

那么原式可以化为 $y_j = k_i x_j + b_i$ 每一组 $\{x_j, y_j\}$ 都是之前存在的, 而 $\{k_i, b_i\}$ 是对于每一个 i 来说的, 要最小化 f_i 即最小化 b_i
对于之前的点 $\{x_j, y_j\}$ 维护一个下凸壳, 第一个斜率大于 k 的点即为满足条件的 j, 若这样的 j 不存在, 取凸壳最后一个点最优
一般都是把方程化为 $y = kx + b$,(x, y) 都是已知点, k 是每次询问给出的, 要求最大化/最小化 b
例: 当方程为 $f_i = \min_{j=0}^{i-1}\{f_j - (t_i + s)c_j + t_i \times c_i + s \times c_n\}$
变为 $f_j = (t_i + s) \times c_j + (f_i - t_i \times c_i - s \times c_n)$
由于每次加入的点横坐标单调递增, 利用单调栈维护每一个 (c_j, f_j) 形成的下凸壳即可
查询的时候二分出凸壳上第一个斜率大于 $k_i(\ t_i + s)$ 的点, 即是满足条件的点

```
for (int i = 1; i <= n; i++)
{
    int l = hh, r = tt;
    while (l < r)
    {
        int mid = (l + r) >> 1;
        if (f[q[mid + 1]] - f[q[mid]] >= (t[i] + s) * (c[q[mid + 1]] - c[q[mid]])) r = mid;
        else l = mid + 1;
    }
    int j = q[l];
    f[i] = f[j] - (t[i] + s) * c[j] + t[i] * c[i] + s * c[n];
    while (hh<tt && (f[q[tt]] - f[q[tt - 1]]) * (c[i] - c[q[tt]])>=
        (f[i] - f[q[tt]]) * (c[q[tt]] - c[q[tt - 1]])) tt--;
    q[++tt] = i;
}
```

例: 当方程为 $f_i = \min_{j=0}^{i-1}\{f_j + (s_i - s_j - L)^2\}(s_i$ 单调递增) 时
拆式子:$f_i = f_j - 2(s_i - L) \times s_j + s_j^2 + (s_i - L)^2$
整理式子:$f_j + s_j^2 = 2(s_i - L) \times s_j + (f_i - (s_i - L)^2)$
维护每一个 $(s_j, f_j + s_j^2)$ 形成的下凸壳即可。
进一步, 由于斜率单调递增, 可以把单调栈换成单调队列, 优化掉斜率比较小的点

```
for (int i = 1; i <= n; i++)
{
    while (hh < tt)
    {
        int x = q[hh], y = q[hh + 1];
        if (2 * (s[i] - L) * (s[y] - s[x]) > f[y] + s[y] * s[y] - f[x] - s[x] * s[x]) hh++;
        else break;
    }
    int j = q[hh];
    f[i] = f[j] + (s[i] - s[j] - L) * (s[i] - s[j] - L);
    while (hh < tt)
    {
        int x = q[tt - 1], y = q[tt];
        if ((f[i] + s[i] * s[i] - f[y] - s[y] * s[y]) * (s[y] - s[x])<=
            (f[y] + s[y] * s[y] - f[x] - s[x] * s[x]) * (s[i] - s[y])) tt--;
        else break;
    }
    q[++tt] = i;
}
```

5.7 动态 DP

用途: 解决带修改的树上 DP 问题，如动态点权的最大权独立集。
核心思想: 用重链剖分 + 线段树，将树上 DP 转化为矩阵乘法，利用矩阵结合律快速合并。
关键: 定义广义矩阵乘法，$A \otimes B$ 中 $(A \otimes B)_{i,j} = \max_k(A_{i,k} + B_{k,j})$。
考虑没有上司的舞会, 点权带修改, 即点权是动态的树上最大权独立集问题
普通的矩阵乘法满足结合律是因为乘法满足交换律和分配律, 加法满足交换律
考虑广义矩阵乘法运算对于加法和 max 运算都满足交换律接下来考虑分配律
加法对 max 运算的分配律:$a + \max(b, c) = \max(a + b, a + c)$ 成立
max 运算对加法的分配律:$\max(a, b + c) = \max(a, b) + \max(a, c)$ 不成立
故可以把加法视作乘法 max 运算视作加法，定义新的广义矩阵乘法运算，且该运算满足结合律。
广义矩阵乘法定义：对于两个 2×2 矩阵 A 和 B:

$$A \otimes B = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \otimes \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} \max(a+e, b+g) & \max(a+f, b+h) \\ \max(c+e, d+g) & \max(c+f, d+h) \end{pmatrix}$$

对应的单位矩阵为：

$$E = \begin{pmatrix} 0 & -\infty \\ -\infty & 0 \end{pmatrix}$$

验证：$A \otimes E = E \otimes A = A$

示例： 设 $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$，$B = \begin{pmatrix} 0 & 1 \\ 2 & 0 \end{pmatrix}$

则 $A \otimes B = \begin{pmatrix} \max(1+0, 2+2) & \max(1+1, 2+0) \\ \max(3+0, 4+2) & \max(3+1, 4+0) \end{pmatrix} = \begin{pmatrix} 4 & 2 \\ 6 & 4 \end{pmatrix}$

定义 $f_{u,0}$ 和 $f_{u,1}$ 分别为以 u 为根的子树中不选/选当前点的最大权独立集

定义 $g_{u,0}$ 和 $g_{u,1}$ 的意义与上面类似但只考虑轻儿子的贡献不考虑重儿子的贡献

$g_{u,0} = \sum\limits_{j \in lightson_u} \max(f_{j,0}, f_{j,1}), g_{u,1} = \sum\limits_{j \in lightson_u} f_{j,0} + val_u$

$f_{u,0} = g_{u,0} + \max(f_{son,0}, f_{son,1}) = \max(f_{son,0} + g_{u,0}, f_{son,1} + g_{u,0})$

$f_{u,1} = g_{u,1} + f_{son,0} = \max(f_{son,0} + g_{u,1}, f_{son,1} + (-INF))$

考虑矩阵转移，我们希望通过矩阵乘法实现状态转移：

设向量 $V_{son} = \begin{pmatrix} f_{son,0} \\ f_{son,1} \end{pmatrix}$，转移矩阵 $M_u = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$

则转移过程为：$V_u = M_u \times V_{son} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \times \begin{pmatrix} f_{son,0} \\ f_{son,1} \end{pmatrix}$

在广义矩阵乘法下：

$$V_u = \begin{pmatrix} \max(a + f_{son,0}, b + f_{son,1}) \\ \max(c + f_{son,0}, d + f_{son,1}) \end{pmatrix} = \begin{pmatrix} f_{u,0} \\ f_{u,1} \end{pmatrix}$$

根据状态转移方程：

$$f_{u,0} = \max(f_{son,0} + g_{u,0}, f_{son,1} + g_{u,0}) \tag{2}$$

$$f_{u,1} = \max(f_{son,0} + g_{u,1}, f_{son,1} + (-\infty)) \tag{3}$$

比较系数可得转移矩阵：

$$M_u = \begin{pmatrix} g_{u,0} & g_{u,0} \\ g_{u,1} & -\infty \end{pmatrix}$$

故可以在重链上做矩阵乘法转移：$V_u = M_u \times V_{son}$

广义矩阵乘法的时候要注意单位阵不是普通的单位阵, 要重新计算

```
int n, m, w[N], f[N][2];
int son[N], fa[N], top[N], sz[N], depth[N];
int dfn[N], yingshe[N], ed[N], cnt;
template<int row, int col>
struct Matrix
{
    int r, c;
    int ele[row][col];
    Matrix():r(row), c(col) {}
    inline int & operator()(int a, int b) { return ele[a][b]; }
};
template<int m, int n, int p>
inline auto operator * (Matrix<m, n> m1, Matrix<n, p> m2)
{
    Matrix<m, p> res;
    memset(res.ele, -0x3f, sizeof res.ele);
    for (int i = 0; i < m; i++)
        for (int k = 0; k < n; k++)
            for (int j = 0; j < p; j++)
                res(i, j) = max(res(i, j), m1(i, k) + m2(k, j));
    return res;
}
Matrix<2, 2> E, g[N]; //单位矩阵
struct Node
{
    int l, r;
    Matrix<2, 2> mat;
}tr[N << 2];
void add(int a, int b) { e[idx] = b, ne[idx] = h[a], h[a] = ++idx; }
void dfs1(int u, int father, int dep)
{
    sz[u] = 1, fa[u] = father, depth[u] = dep;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j == father) continue;
```

```
        dfs1(j, u, dep + 1);
        sz[u] += sz[j];
        if (sz[son[u]] < sz[j]) son[u] = j;
    }
}
void dfs2(int u, int t)
{
    top[u] = t, dfn[u] = ++cnt, yingshe[cnt] = u;
    if (!son[u])
    {
        f[u][1] = w[u]; //叶子是唯一能确定 f[u][0] 和 f[u][1] 的
        g[u] = E; //初始化对应的 g[u] = E
        ed[u] = u; //重链的尾端是当前点
        return ;
    }
    g[u](0, 1) = w[u], g[u](1, 1) = -INF;
    /*
        转移矩阵 g[u] =
        | g[u][0] g[u][1] |
        | g[u][0] - INF |
    */
    dfs2(son[u], t);
    ed[u] = ed[son[u]]; //重链的尾端可以通过重儿子的求得
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j == son[u] || j == fa[u]) continue;
        dfs2(j, j);
        g[u](0, 0) += max(f[j][0], f[j][1]);
        g[u](0, 1) += f[j][0];
    }
    f[u][0] = g[u](0, 0) + max(f[son[u]][0], f[son[u]][1]);
    f[u][1] = g[u](0, 1) + f[son[u]][0];
}
void pushup(int u)
{//递推的时候先乘右边的 所以顺序要注意
    tr[u].mat = tr[u << 1 | 1].mat * tr[u << 1].mat;
}
void build(int u, int l, int r)
{
    if (l == r) tr[u] = {l, r, g[yingshe[l]]};
    else
    {
        tr[u] = {l, r};
        int mid = (l + r) >> 1;
        build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
        pushup(u);
    }
}
auto query(int u, int l, int r)
{
    if (l<= tr[u].l && r >= tr[u].r) return tr[u].mat;
    else
    {
        int mid = (tr[u].l + tr[u].r) >> 1;
        auto res = E; //先递归右区间
        if (r > mid) res = query(u << 1 | 1, l, r);
        if (l <= mid) res = res * query(u << 1, l, r);
        return res;
    }
}
void modify(int u, int x, int z)
{
    if (tr[u].l == tr[u].r) tr[u].mat = g[z];
    else
    {
        int mid = (tr[u].l + tr[u].r) >> 1;
        if (x <= mid) modify(u << 1, x, z);
        else modify(u << 1 | 1, x, z);
        pushup(u);
    }
}
auto query(int x)
{
```

```
    Matrix<1, 2> temp;
    // f[u][0] f[u][1]
    temp(0, 0) = 0, temp(0, 1) = w[ed[x]];
    return temp * query(1, dfn[x], dfn[ed[x]]); //叶子一直乘递归矩阵得到结果
}
void modify(int x, int y)
{
    /*
        a b       g[u][0]   g[u][1]
              =
        c d       g[u][0]   -INF
    */
    Matrix<1, 2> od, nw;
    // f[u][0] f[u][1]
    g[x](0, 1) += y - w[x];
    od = query(top[x]); //重链的顶端是其父亲的轻儿子 所以其父亲的轻儿子贡献要重算
    w[x] = y;
    while (x)
    {
        modify(1, dfn[x], x); //把线段树上 x 节点的矩阵改为 g[x]
        nw = query(top[x]); //查询重链顶端的新值
        x = fa[top[x]]; //往上跳
        g[x](0, 1) += nw(0, 0) - od(0, 0);
        g[x](0, 0) = g[x](1, 0) += max(nw(0, 0), nw(0, 1)) - max(od(0, 0), od(0, 1));
        od = query(top[x]);
    }
}
void solve()
{
    E(1, 0) = E(0, 1) = -INF;
    /*
        初始化单位矩阵:
        0       -INF
        -INF     0
    */
    cin >> n >> m;
    for (int i = 1; i<= n; i++) cin >> w[i];
    dfs1(1, 0, 1), dfs2(1, 1);
    build(1, 1, n);
    while (m--)
    {
        int x, y;
        modify(x, y);
        auto ans = query(1);
        cout << max(ans(0, 0), ans(0, 1))<<"\n";
    }
}
```

5.8 决策单调性优化转移

给出 N, 以及 $A_{1\dots N}, B_{1\dots N}$。对于每个 $k \in [1, N]$, 找出一个 $1 \dots N$ 的集合 S, 满足 $|S| = k$ 且 $\sum\limits_{i\in S} A_i - \max\limits_{i\in S} B_i$ 最大, 输出这个值。

显然按照 B 排序, 再对每一个 B_i 考虑 $k \in [1, i]$ 的答案
显然对于每个 k_i, 决策点 i 单调不降, 那么可以利用决策树优化转移
决策树的深度是 $\log n$，每一层的总和 $\sum \text{len} = n$，所以时间复杂度是 $O(n \log n)$。

```
int root[N], idx, n, a[N], b[N], ans[N];
array<int, 2> w[N];
int insert(int p, int l, int r, int x)
{
    int q = ++idx;
    tr[q] = tr[p];
    if (l == r)
    {
        tr[q].cnt++, tr[q].sum += x;
        return q;
    }
    int mid = (l + r) >> 1;
    if (x <= mid) tr[q].l = insert(tr[p].l, l, mid, x);
    else tr[q].r = insert(tr[p].r, mid + 1, r, x);
    tr[q].sum = tr[tr[q].l].sum + tr[tr[q].r].sum;
    tr[q].cnt = tr[tr[q].l].cnt + tr[tr[q].r].cnt;
    return q;
```

```
}
int query(int p, int l, int r, int k)
{
    if (l == r) return r * k;
    int cnt = tr[tr[p].r].cnt, mid = l + r >>1;
    if (k <= cnt) return query(tr[p].r, mid + 1, r, k);
    else return query(tr[p].l, l, mid, k - cnt) + tr[tr[p].r].sum;
}
void solve(int l, int r, int pl, int pr)//决策点是 pl pr
{
    if (l > r) return ;
    int mid = (l + r) >> 1, p; //每次选中点保证深度
    ans[mid] = -inf;
    for (int i = max(mid, pl); i <= pr; i++)
    {//主席树查一下前 k 大的和
        int t = query(root[i], -INF, INF, mid) - b[i];
        if (t > ans[mid]) ans[mid] = t, p = i;
    }
    solve(l, mid - 1, pl, p), solve(mid + 1, r, p, pr);
}
void solve()
{
    cin >> n;
    for (int i = 1; i<= n; i++) cin >> w[i][1] >> w[i][0];
    sort(w + 1, w + 1 + n);
    for (int i = 1; i <= n; i++)
    {
        a[i] = w[i][1], b[i] = w[i][0];
        root[i] = insert(root[i - 1], -INF, INF, a[i]);
    }
    solve(1, n, 1, n);
    for (int i = 1; i <= n; i++) cout << ans[i]<<"\n";
}
```

一维 DP 的决策单调性

诗人小 G: 有 DP 方程:$f_i = \min_{j=0}^{i-1}\{f_j + |s_i - s_j|^P\}$

该方程具有决策单调性, 即对于 $i < j$, 则有 i 的最优决策点 p_i 小于等于 j 的最优决策点 p_j。

```
array<int, 3> q[N];
void dp()
{
    int hh = tt = 0;
    q[0] = {0, 1, n}; //最优决策点 这个决策点覆盖的区间左端点和右端点
    for (int i = 1; i <= n; i++)
    {
        while (hh <= tt && q[hh][2] < i) hh++; //覆盖的右端点已经不可能了
        q[hh][1] = i; //左端点更新为现在的点
        int j = q[hh][0];
        best[i] = j; //i 的最优决策点为 j
        f[i] = cal(j, i);
        while (hh <= tt && cal(i, q[tt][1]) <= cal(q[tt][0], q[tt][1])) tt--;
        //如果当前点决策比队尾所有区间的要好 队尾弹出
        int l = q[tt][1], r = q[tt][2] + 1;
        while (l < r)//最后一个区间可能后半部分决策不如 i 好 二分出这个后半部分
        {
            int mid = (l + r) >> 1;
            if (cal(i, mid) <= cal(q[tt][0], mid)) r = mid;
            else l = mid + 1;
        }
        if (l <= n) q[tt][2] = l - 1, q[++tt] = {i, l, n}; //修改最后一个决策 插入新决策
    }
}
```

二维 DP 的决策单调性

给定一个序列 a，要把它分成 k 个子段。每个子段的费用是其中相同元素的对数。求所有子段的费用之和的最小值。($2 \leqslant n \leqslant 10^5, 2 \leqslant k \leqslant \min(n, 20)$)

考虑 $f_{j,i}$ 为以 i 结尾, 分成 j 个子段的最小代价。

显然有 dp 方程 $f_{j,i} = \min\{f_{j-1,k} + cal(k+1, i)\}$

显然对于每一层的 i 来说, 决策点单调不降, 考虑到 cal 的贡献比较难算, 但是在分治的时候, 指针移动是比较单调的, 移动次数不会很多, 可以利用类似莫队指针移动的方式快速计算贡献。

```cpp
int n, k, w[N], cnt[N], f[M][N];
int cal(int ql, int qr)
{
    static int sum = 0, l = 1, r = 0;
    while (l > ql)
    {
        l--;
        sum += cnt[w[l]];
        cnt[w[l]]++;
    }
    while (r < qr)
    {
        r++;
        sum += cnt[w[r]];
        cnt[w[r]]++;
    }
    while (l < ql)
    {
        cnt[w[l]]--;
        sum -= cnt[w[l]];
        l++;
    }
    while (r > qr)
    {
        cnt[w[r]]--;
        sum -= cnt[w[r]];
        r--;
    }
    return sum;
}
void solve(int k, int l, int r, int pl, int pr)
{
    if (l > r) return ;
    int mid = (l + r) >> 1, p = -1;
    for (int i = pl; i < mid && i <= pr; i++)
    {
        int v = f[k - 1][i] + cal(i + 1, mid);
        if (v < f[k][mid]) f[k][mid] = v, p = i;
    }
    solve(k, l, mid - 1, pl, p), solve(k, mid + 1, r, p, pr);
}
void solve()
{
    cin >> n >> k;
    for (int i = 1; i<= n; i++) cin >> w[i];
    memset(f, 0×3f, sizeof f);
    for (int i = 0; i <= k; i++) f[i][0] = 0;
    for (int i = 1; i <= k; i++) solve(i, 1, n, 0, n - 1);
    cout << f[k][n]<<"\n";
}
```

6 数论

6.1 质数/约数

6.1.1 由线性筛求所有约数

```cpp
vector<int> getDiv(int x)
{
    vector<int> v(1, 1);
    while (x >1)
    {
        int p = minp[x];
        int l = 0, r = v.size();
        while (x % p == 0)
        {
            for (int k = l; k < r; k++) v.push_back(v[k] * p);
            x /= p, l = r, r = v.size();
        }
    }
    return v;
}
```

6.1.2 PollardRho

```cpp
struct Pollard_Rho
{
    int qmi_mod(i128 x, int n, int mod)
    {
        int res = 1;
        while (n)
        {
            if (n & 1) res = res * x % mod;
            x = x * x % mod;
            n>>=1;
        }
        return res;
    }
    bool MR(int n)
    {
        if (n == 2) return true;
        if (n <= 1 || n % 2 == 0) return false;
        static int base[7] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022};
        int u = n - 1, k = 0;
        while (u % 2 == 0) u>>=1, k++;
        for (auto x:base)
        {
            if (x % n == 0) continue;
            int v = qmi_mod(x, u, n);
            if (v == 1 || v == n - 1) continue;
            for (int j = 1; j <= k; j++)
            {
                int last = v;
                v = (i128)v * v % n;
                if (v == 1)
                {
                    if (last != n - 1) return false;
                    break;
                }
            }
            if (v != 1) return false;
        }
        return true;
    }
    int PR(int n)
    {
        static mt19937_64 sj(chrono::steady_clock::now().time_since_epoch().count());
        uniform_int_distribution<int> u0(1, n - 1);
        int c = u0(sj);
        auto f = [&](int x) {return ((i128)x * x + c) % n; };
        int x = 0, y = 0, s = 1;
        for (int k = 1; ; k<<=1, y = x, s = 1)
        {
            for (int i = 1; i <= k; i++)
            {
                x = f(x), s = (i128)s * abs(x - y) % n;
                if (i % 127 == 0)
                {
                    int d = gcd(s, n);
                    if (d >1) return d;
                }
            }
            int d = gcd(s, n);
            if (d >1) return d;
        }
        return n;
    }
    vector<int> fac;
    void get_fac(int n)
    {
        if (n == 1) return ;
        else if (n == 4)
        {
            fac.push_back(2);
            fac.push_back(2);
            return ;
        }
```

```
        if (MR(n))
        {
            fac.push_back(n);
            return ;
        }
        int x = n;
        while (x == n) x = PR(n);
        get_fac(x), get_fac(n / x);
    }
    vector<int> get_factor(int n)//返回 n 的所有质因子（可重复）
    {
        fac.clear(), get_fac(n);
        return fac;
    }//如果想得到一个数的所有因数 就需要下面的 dfs
    vector<int> v;
    vector<array<int, 2>> p;
    void dfs(int u, int val)
    {
        if (u >= p.size())
        {
            v.push_back(val);
            return ;
        }
        for (int i = 0; i <= p[u][1]; i++) dfs(u + 1, val), val *= p[u][0];
    }
    vector<int> FAC(int n)//返回 n 的所有约数
    {
        fac.clear(), v.clear(), p.clear();
        get_fac(n);
        sort(fac.begin(), fac.end());
        for (int x:fac)
            if (p.size() && p.back()[0] == x) p.back()[1]++;
            else p.push_back({x, 1});
        dfs(0, 1);
        return v;
    }
}pr;
```

6.2 线性预处理逆元

```
inv[1] = 1;
for (int i = 2; i <= n; i++) inv[i] = (mod - mod / i) * inv[mod % i] % mod;
```

6.3 Exgcd

```
int exgcd(int a, int b, int &x, int &y)
{
    if (b == 0) {
        x = 1, y = 0;
        return a;
    }
    int d = exgcd(b, a % b, y, x);
    y -= a / b * x;
    return d;
}
-------------------------
//对于不定方程 ax + by = c
if (c % d) cout <<"-1\n"; // 没有整数解
else
{
    int s = c / d;
    x *= s, y *= s;
    int k1 = b / d, k2 = a / d, cnt = 0, m1 = (x % k1 + k1) % k1;
    if (m1 == 0) m1 += k1;
    int kmin = (m1 - x) * d / b;
    if (y - kmin * a / d >0) cnt = 1;
    else cnt = 2;
    int m2 = (y % k2 + k2) % k2;
    if (m2 == 0) m2 += k2;
    if (cnt == 1)//有整数解 且有正整数解
    {
```

```
        int kmax = -(m2 - y) * d / a;
        int num = kmax - kmin + 1;
        //num, m1, m2, x + kmax * k1, y - kmin * k2
        //正整数解的数量
        //所有正整数解中 x 的最小值
        //所有正整数解中 y 的最小值
        //所有正整数解中 x 的最大值
        //所有正整数解中 y 的最大值
    }
    else//有整数解 但没有正整数解
    {
        //m1, m2
        //整数解中 x 的最小正整数值 y 的最小正整数值
    }
}
```

6.4 中国剩余定理

给定 n 组非负整数 a_i, b_i, 求解关于 x 的方程组的最小非负整数解。

$$\begin{cases} x \equiv b_1 \pmod{a_1} \\ x \equiv b_2 \pmod{a_2} \\ \dots \\ x \equiv b_n \pmod{a_n} \end{cases}$$

其中 $a_1, a_2, a_3 \dots a_n$ 两两互质

```
int exgcd(int a, int b, int & x, int & y)
{
    if (!b)
    {
        x = 1, y = 0;
        return a;
    }
    int d = exgcd(b, a % b, y, x);
    y -= a / b * x;
    return d;
}
int crt()
{
    int n, res = 0, pi = 1;
    for (int i = 1; i<= n; i++) cin >> a[i] >> b[i], pi *= a[i];
    for (int i = 1; i <= n; i++)
    {
        int x, y, M = pi / a[i];
        exgcd(M, a[i], x, y);
        x = (x % a[i] + a[i]) % a[i];
        res = (res + b[i] * M * x) % pi;
    }
    return res;
}
```

扩展中国剩余定理 a_i 之间不一定两两互质

```
int excrt(int n)
{
    int n, x = 0, m1, a1;
    cin >> m1 >> a1;
    for (int i = 0; i < n - 1; i++)
    {
        int m2, a2;
        cin >> m2 >> a2;
        int k1, k2;
        int d = exgcd(m1, m2, k1, k2);
        if ((a2 - a1) % d) break;
        k1 *= (a2 - a1) / d;
        k1 = (k1 % (m2 / d) + m2 / d) % (m2 / d);
        x = k1 * m1 + a1;
        int m = abs(m1 / d * m2);
        a1 = k1 * m1 + a1;
        m1 = m;
```

```
    }
    return x; //最小非负整数 x
}
```

7 线性代数

7.1 矩阵

定义矩阵类

需要注意广义矩阵乘法的时候单位阵的计算, 如果被卡常了, 观察矩阵有没有特殊不变性

要注意矩阵没有用到的位置要设为幺元, 否则乘法导致错误, 如 $(1, 0)*(0, 4)->(1, 4)$ 是非法转移

```
template<int row, int col>
struct Matrix
{
    int r, c, ele[row][col];
    Matrix():r(row), c(col) {}
    int & operator()(int a, int b) { return ele[a][b]; }
};
template<int m, int n, int p>
auto operator * (Matrix<m, n> m1, Matrix<n, p> m2)
{
    Matrix<m, p> res;
    memset(res.ele, 0, sizeof res.ele);
    for (int i = 0; i < m; i++)
        for (int k = 0; k < n; k++)
            for (int j = 0; j < p; j++)
                res(i, j) = (res(i, j) + m1(i, k) * m2(k, j)) % mod;
    return res;
}//访问矩阵 A[i][j] 就是 A(i, j)
```

7.2 高斯消元

```
double a[N][N];
int gauss()//高斯消元
{
    int c, r; //初始行与列
    for (c = 1, r = 1; c <= n; c++)//初始化 且枚举每一列
    {
        int t = r; //从已处理的行下面找到这一列中绝对值最大的数
        for (int i = r; i <= n; i++)//
        {
            if (fabs(a[i][c]) > fabs(a[t][c]))//枚举 比较
            t = i; //更新最大值的位置
        }
        if (fabs(a[t][c]) < eps) continue;
        //如果最大值都为 0 说明这一列全是 0 跳过且同时不能吃到 r++ 的福利 不是行满秩
        for (int i = c; i <= n + 1; i++) swap(a[t][i], a[r][i]);
        //前面的列都是 0 没有交换的必要 故从 c 列开始交换最上面一行和含有最大绝对值的一行
        for (int i = n + 1; i >= c; i--) a[r][i] /= a[r][c];
        //利用初等行变换把最上面一行的第 c 列变为 1
        //细节: 从后往前消的话可以一直用第 c 列的值 较为方便
        for (int i = r + 1; i <= n; i++)
        {
            if (fabs(a[i][c]) > eps)
            //如果这一行的第 c 列数是 0 那么无需操作了 反之则把这个数消为 0
            {
                for (int j = n + 1; j >= c; j--)
                    a[i][j] -= a[r][j] * a[i][c];
                    //依然是从后往前消 倍数为这一行第 c 列的数的值
            }
        }
        r++; //处理下一行 (也有记录是否达到行满秩的作用)
    }
    if (r <= n)
    //如果行满秩的话 最后 r == n + 1 故 r <= n 的时候是行降秩
    {
        for (int i = r; i <= n; i++)
        {
            if (fabs(a[i][n + 1]) > eps) return 2;
```

```
            //因为是行降秩 所以第 r 行及其以下的行原系数矩阵都为 0
            //对于这些方程 若出现方程组右边的值不为 0
            //则出现了 0 = 非零 矛盾 故线性方程组无解
        }
        return 1; //若没有上述条件 则线性方程组有无穷多组解
    }
    //若行满秩 则开始从下往上开始求解 xi 的值
    for (int i = n - 1; i >= 1; i--)
    {
        for (int j = i + 1; j <= n; j++)
        {
            a[i][n + 1] -= a[i][j] * a[j][n + 1];
            //依然是用下面的行消去上面的行
            //注意: 在每一次初等行变化时 a[n][n] 的值已变为 1
            //*a[j][n + 1] 则是隐蔽地利用了 a[j][j] 为 1 的这个条件 从而方便地消去
            //故若除了每一行除了 a[i][i] 其他的值都为 0 的话 最后第 n + 1 的列的值就是 xi 的值
        }
    }
    return 0; //行满秩 返回有唯一解
}
```

7.2.1 bitset 优化解异或方程组

```
std::bitset<1010> matrix[2010]; // matrix[1~n]: 增广矩阵, 0 位置为常数

std::vector<bool> GaussElimination(
    int n, int m) // n 为未知数个数, m 为方程个数, 返回方程组的解
                  //  (多解 / 无解返回一个空的 vector)
{
  for (int i = 1; i <= n; i++) {
    int cur = i;
    while (cur <= m && !matrix[cur].test(i)) cur++;
    if (cur > m) return std::vector<bool>(0);
    if (cur != i) swap(matrix[cur], matrix[i]);
    for (int j = 1; j <= m; j++)
      if (i != j && matrix[j].test(i)) matrix[j] ^= matrix[i];
  }
  std::vector<bool> ans(n + 1);
  for (int i = 1; i <= n; i++) ans[i] = matrix[i].test(0);
  return ans;
}
```

7.3 求行列式

模数非质数的情况下也适用

```
int n, mod, a[N][N];
int cal()
{
    int ans = 1;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            a[i][j] %= mod;
    for (int i = 1; i <= n; i++)
    {
        for (int j = i + 1; j <= n; j++)
        {// 消掉 a[j][i]
            int x = i, y = j;
            while (a[x][i])
            {
                int t = a[y][i] / a[x][i];
                for (int k = i; k <= n; k++) a[y][k] = (a[y][k] - t * a[x][k]) % mod;
                swap(x, y);
            }//a[x][i] = 0
            if (x == i)
            {
                for (int k = i; k <= n; k++) swap(a[i][k], a[j][k]);
                ans = -ans;
            }
        }
        if (!a[i][i]) return 0;
```

```
        ans = ans * a[i][i] % mod;
    }
    return (ans % mod + mod) % mod;
}
```

7.4 线性基

用途：处理异或运算相关问题，如最大/最小异或值、第 k 小异或值等。

核心性质：线性基能表示原集合所有数异或得到的所有可能值，且基中元素线性无关。

关键操作：insert 插入数字并维护线性无关性，query_max/min 查询最值。

优化：对于大数据可用 bitset 优化，时间复杂度从 $O(n \log^2 A)$ 降到 $O(n \log A/w)$。

```
const int N = 64;
int num[N], temp[N], zero;
void insert(int x)//插入一个数
{
    for (int i = N - 1; i >= 0; i--)
        if (x >> i & 1)
            if (!num[i])
            {
                num[i] = x;
                return ;
            }
            else x ^= num[i];
    zero = 1;
}
bool check(int x)//检查一个数 x 是否能被异或出来
{
    for (int i = N - 1; i >= 0; i--)
        if (x >> i & 1)
            if (!num[i]) return false;
            else x ^= num[i];
    return true;
}
int query_max()//线性基中的数互相异或出来的最大值
{
    int res = 0;
    for (int i = N - 1; i >= 0; i--)
        res = max(res, res ^ num[i]);
    return res;
}
int query_min()//线性基中的数互相异或出来的最小值
{
    if (zero) return 0;
    for (int i = 0; i < N; i++)
        if (num[i]) return num[i];
}
int query(int k)//返回线性基中的数互相异或出来的第 k 小值
{
    int res = 0, cnt = 0;
    k -= zero;
    if (!k) return 0;
    for (int i = 0; i < N; i++)
    {
        for (int j = i - 1; j >= 0; j--)
            if (num[i] >> j & 1) num[i] ^= num[j];
        if (num[i]) temp[cnt++] = a[i];
    }//消成行简化阶梯型
    if (k >= (1ll << cnt)) return - 1;
    for (int i = 0; i < cnt; i++)
        if (k >> i & 1) res ^= temp[i];
        return res;
}
```

图上的线性基

一般来说图上的线性基与环有关找出图上的环将其异或和插入线性基

连通图 = 生成树 + 若干非树边形成的环

例题: 求连通图上 $1 \to n$ 的路径的 xor 最大值

```cpp
int h[N], ne[M], e[M], w[M], idx;
int n, m, dist[N], fa[N];
bool st[N];
int num[66];
void dfs(int u)
{
    st[u] = true;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (st[j])
        {
            int t = dist[u] ^ dist[j] ^ w[i];
            if (t) insert(t);
            continue;
        }
        tree[i] = true;
        dist[j] = dist[u] ^ w[i], fa[j] = u;
        dfs(j);
    }
}
void insert(int x)
{
    for (int i = 60; i >= 0; i--)
        if (x >> i & 1)
            if (!num[i])
            {
                num[i] = x;
                return ;
            }
            else x ^= num[i];
}
int query(int x)
{
    for (int i = 60; i >= 0; i--) x = max(x, x ^ num[i]);
    return x;
}
void solve()
{
    while (m--) add(a, b, c), add(b, a, c);
    dfs(1);
    cout << query(dist[n])<<"\n";
}
```

区间线性基

离线构造前缀线性基　把询问挂到右端点上

再用奇怪的手段操作维护时间　查询的时候根据左端点的限制去查

```cpp
int num[22], t[22], w[N], ans[N];
vector<array<int, 2>> Q[N];
void insert(int x, int T)
{
    for (int i = 20; i >= 0; i--)
        if (x >> i & 1)
        {
            if (!num[i])//如果是空的话
            {
                num[i] = x, t[i] = T; //把数和时间插入线性基
                return ;
            }
            else if (t[i] < T)//如果它的时间大于线性基
            {
                swap(t[i], T); //换进去
                swap(num[i], x);
            }
            x ^= num[i]; //异或一下
        }
}
int query(int T)
{
    int res = 0;
    for (int i = 20; i >= 0; i--)
```

```
        if (T <= t[i]) res = max(res, res ^ num[i]); //如果时间大于等于线性基
    return res;
}
void solve()
{
    for (int i = 0; i < q; i++)
    {
        int l, r;
        Q[r].push_back({l, i});
    }
    for (int i = 1; i <= n; i++)
    {
        insert(w[i], i);
        for (auto [l, id]:Q[i]) ans[id] = query(l);
    }
    for (int i = 0; i < q; i++) cout << ans[i]<<"\n";
}
```

Bitset 优化线性基

当数据范围较大时，可以用 bitset 优化高斯消元，显著提升常数。

```
const int N = 5005, M = 64;
bitset<N> base[M], temp[M];
int sz;
void insert(bitset<N>& x)
{
    for (int i = M - 1; i >= 0; i--)
    {
        if (!x[i]) continue;
        if (base[i].none())
        {
            base[i] = x;
            sz++;
            return;
        }
        x ^= base[i];
    }
}
bitset<N> query_max()
{
    bitset<N> res;
    for (int i = M - 1; i >= 0; i--)
        res = max(res, res ^ base[i]);
    return res;
}
// 使用示例：处理大整数的线性基
// bitset<N > x; 先将大整数转换为 bitse
// insert(x); 插入到线性基中
```

8 杂项

8.1 斐波那契结论

$f_{n+m} = f_{m-1} \times f_n + f_m \times f_{n+1}$

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

$$\therefore (F_1 F_0) \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}$$

$$\therefore \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

$$\because \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n+m} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^m$$

$$\therefore \begin{pmatrix} F_{n+m+1} & F_{n+m} \\ F_{n+m} & F_{n+m-1} \end{pmatrix} = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \begin{pmatrix} F_{m+1} & F_m \\ F_m & F_{m-1} \end{pmatrix}$$
$$= \begin{pmatrix} F_{n+1}F_{m+1} + F_nF_m & F_{n+1}F_m + F_nF_{m-1} \\ F_nF_{m+1} + F_{n-1}F_m & F_nF_m + F_{n-1}F_{m-1} \end{pmatrix}$$

$$\therefore F_{n+m} = F_{n+1}F_m + F_nF_{m-1}$$

当 $i<0, f_i=(-1)^{i-1}\times f_{-i}$

8.2 切比雪夫距离与曼哈顿距离转化

欧式距离:$|AB|=\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$

曼哈顿距离:$|AB|=|x_1-x_2|+|y_1-y_2|$

切比雪夫距离:$|AB|=\max(|x_1-x_2|,|y_1-y_2|)$

对于 $(x_1,y_1),(x_2,y_2)$ 曼哈顿距离:

$|AB|=\max\{x_1-x_2+y_1-y_2,x_2-x_1+y_1-y_2,x_1-x_2+y_2-y_1,x_2-x_1+y_2-y_1\}$

$=\max\{|(x_1+y_1)-(x_2+y_2)|,|(x_1-y_1)-(x_2-y_2)|\}$

即为 $(x_1+y_1,x_1-y_1),(x_2+y_2,x_2-y_2)$ 两点之间的切比雪夫距离

反解可得 $(x_1,y_1),(x_2,y_2)$ 两点的切比雪夫距离为 $(\frac{x_1+y_1}{2},\frac{x_1-y_1}{2}),(\frac{x_2+y_2}{2},\frac{x_2-y_2}{2})$ 两点的曼哈顿距离

将每一个点 (x,y) 转化为 $(\frac{x+y}{2},\frac{x-y}{2})$ 再求解曼哈顿距离即可。

相应的, 曼哈顿距离转切比雪夫距离为 $(x,y)\to(x+y,x-y)$

结论: 将每个点逆时针旋转 $45°$, 再缩小到原来的一半, 再求解曼哈顿距离即可

8.3 快速排序求第 k 小

```cpp
int w[N], n;
int qsearch(int l, int r, int k)//在 l 和 r 之间找第 k 大
{
    if (l == r) return w[k];
    int mid = w[l + r >>1], i = l - 1, j = r + 1;
    while (i < j)
    {
        do i++; while (w[i] < mid);
        do j--; while (w[j] > mid);
        if (i < j) swap(w[i], w[j]);
    }
    if (k <= j) return qsearch(l, j, k);
    else return qsearch(j + 1, r, k);
}
//std 写法: nth_element(w + 1, w + k, w + n + 1); //让第 k 小的元素归位 之后输出 w[k] 即可
//这个函数不仅能够完成上述任务 还能使得前 k 小都集中在 [1, k] 在不要求顺序求前 k 小的时候很有用
```

8.4 归并排序

```cpp
int w[N], temp[N], n, ans;
void mergesort(int l, int r)
{
    if (l >= r) return; // 若区间长度为 1, 结束递归
    int mid = (l + r) >> 1, i = l, j = mid + 1, k = 0; //i 为左指针, j 为右指针 (双指针算法)
    mergesort(l, mid), mergesort(mid + 1, r); //递归二分
    while (i <= mid && j <= r)
    {
        if (w[i] <= w[j]) temp[k++] = w[i++];
        else temp[k++] = w[j++], ans += mid - i + 1;
    }//比较大小后存入临时数组
    while (i <= mid) temp[k++] = w[i++]; //将剩余的数存入临时数组
    while (j <= r) temp[k++] = w[j++]; //将剩余的数存入临时数组
    for (i = l, j = 0; i <= r; i++, j++) w[i] = temp[j]; //将临时数组的数复制回原数组
}
```

8.5 二分的 01 分数规划

若要选一些物品 (个数有限制) 满足 $\sum_{i=1}^{k}\frac{a_i}{b_i}$ 最大

考虑二分答案 mid, 使得 $\sum_{i=1}^{k}\frac{a_i}{b_i}\geq mid$ 则增大 mid, 否则减小 mid

转化为选一些物品满足 $\sum_{i=1}^{k}(a_i-\text{mid}\cdot b_i)\geq 0$，接着考虑 DP 或者排序，贪心取最优就好

```cpp
int x[N], b[N], n, L, pre[N];
double f[N];
bool check(double mid)
{
    memset(pre, 0, sizeof pre);
    for (int i = 1; i <= n; i++) f[i] = 1e18;
```

```
    for (int i = 1; i <= n; i++)
        for (int j = 0; j < i; j++)
        {
            double t = f[j] - mid * b[i] + sqrtl(fabs(x[i] - x[j] - L));
            if (t < f[i]) f[i] = t, pre[i] = j;
        }
    return f[n] <= 0;
}
void solve()
{
    double l = 0, r = 1e9;
    while (r - l > eps)
    {
        double mid = (l + r) / 2;
        if (check(mid)) r = mid;
        else l = mid;
    }
}
```

8.6 三分

实数三分

```
double l = -INF, r = INF;
while (r - l > eps)
{ //以开口向下的二次函数为例 画图即可理解
    double lmid = (2 * l + r) / 3.0, rmid = (l + 2 * r) / 3.0;
    if (f(lmid) <= f(rmid)) l = lmid; //每次都舍弃掉 1 / 3 的区间
    else r = rmid;
}
```

整数三分

```
int l = 0, r = 1e9;
while (l + 2 < r)
{
    int lmid = (r - l) / 3 + l, rmid = (r - l) / 3 * 2 + l;
    if (f(lmid) > f(rmid)) l = lmid;
    else r = rmid;
}
int ans = f(l); //答案可能在 l~r 之间
for (int i = l + 1; i <= r; i++) ans = min(ans, f(i));
```

8.7 折半搜索

分成两半搜索出一些状态, 最后再将两半合并起来统计答案, 如下题用的是二分

例:$n = 40, m = 10^{18}$ 的背包, 问: 有多少种不同的取物品方案, 使得物品总价值不超过背包体积

```
void work(vector<int>&v v1, vector<int>&v v2)
{
    int len = v1.size();
    for (int i = 1; i < (1 << len); i++)
    {
        int sum = 0;
        for (int j = 0; j < len; j++)
        {
            if (i >> j & 1) sum += v1[j];
            if (sum > m) break;
        }
        if (sum <= m) v2.push_back(sum);
    }
}
void solve()
{
    cin >> n >> m;
    for (int i = 1; i<= n; i++) cin >> w[i];
    vector<int> v1, v2;
    for (int i = 1; i <= n / 2; i++) v1.push_back(w[i]);
    for (int i = n / 2 + 1; i <= n; i++) v2.push_back(w[i]);
    vector<int> v3, v4;
```

```
    work(v1, v3), work(v2, v4);
    int ans = 0;
    v4.push_back(0), v3.push_back(0);
    sort(v4.begin(), v4.end());
    for (auto t:v3)
    {
        if (v4[0] + t > m)
        {
            ans += 1;
            continue;
        }
        int l = 0, r = v4.size() - 1;
        while (l < r)
        {
            int mid = l + r + 1 >>1;
            if (v4[mid] + t <= m) l = mid;
            else r = mid - 1;
        }
        ans += l + 1;
    }
    cout << ans << "\n";
}
```

8.8 区间合并

```
void merge(vector<PII>&s segs)
{
    vector<PII> res;
    sort(segs.begin(), segs.end()); //排序
    int st = -2e9, ed = -2e9; //起始点 终点
    for (auto seg:segs)//遍历
        if (ed < seg.first)//如果上一个终点比这个点左端点小的话
        {
            if (ed != -2e9) res.push_back({st, ed}); //上一个区间合并
            st = seg.first, ed = seg.second; //如果是第一个区间 就收录
        }
        else ed = max(ed, seg.second); //右端点取最大
    if (ed != -2e9) res.push_back({st, ed}); //收录最后第一个区间
    segs = res; //返回区间合并结果
}
```

8.9 快读快写

```
using u64 = unsigned long long;
static const int BUF_SIZE = 1 << 20;
static char ibuf[BUF_SIZE], obuf[BUF_SIZE];
static int ipos = 0, ilen = 0, opos = 0;
inline char gc()
{
    if (ipos == ilen)
    {
        ipos = 0;
        ilen = fread(ibuf, 1, BUF_SIZE, stdin);
        if (ilen == 0) return EOF;
    }
    return ibuf[ipos++];
}
template<typename T>
inline bool readInt(T & o out)
{
    char c;
    T sign = 1, x = 0;
    c = gc();
    if (c == EOF) return false;
    while (c!='-'&&(c<'0'||c>'9')) c = gc();
    if (c=='-') sign = -1, c = gc();
    for (; c>='0'&&c<='9'; c = gc()) x = x * 10 + c-'0';
    out = x * sign;
    return true;
}
```

```cpp
inline void pc(char c)
{
    if (opos == BUF_SIZE) fwrite(obuf, 1, opos, stdout), opos = 0;
    obuf[opos++] = c;
}
template<typename T>
inline void writeInt(T x)
{
    if (x < 0) pc('-'), x = -x;
    char s[24];
    int len = 0;
    if (x == 0) s[len++] = '0';
    while (x) s[len++] = char('0'+x % 10), x /= 10;
    for (int i = len - 1; i >= 0; --i) pc(s[i]);
}
// 程序结束时自动 flush 输出缓冲
struct FastIOFlush
{
    ~FastIOFlush()
    {
        if (opos) fwrite(obuf, 1, opos, stdout);
    }
} fastIOFlush;
```

```cpp
char gc()
{
    static char now[1 << 20], *S, *T;
    if (T == S)
    {
        T = (S = now) + std::fread(now, 1, 1 << 20, stdin);
        if (T == S) return EOF;
    }
    return * S++;
}
void read(int & x)
{
    int fu = 1;
    x = 0;
    char c = gc();
    while (!isdigit(c))
    {
        if (c=='-') fu = -1;
        c = gc();
    }
    while (isdigit(c))
    {
        x = (x << 1) + (x << 3) + (c-'0');
        c = gc();
    }
    x *= fu;
}
void write(int x)
{
    if (x < 0) putchar('-'), x *= -1;
    if (x >9) write(x / 10);
    putchar(x % 10+'0');
}
```

8.9.1 交互高速 IO

```cpp
namespace io
{
    constexpr int MAXBUFFER = 1024 * 1024 * 8;
    char ibuffer[MAXBUFFER], *iptr, obuffer[MAXBUFFER], *optr;
    inline void start_reading()
    { // 开始读取新的一行
        fgets(ibuffer, sizeof(ibuffer), stdin);
        iptr = ibuffer;
    }
    inline void start_writing()
```

```
    { // 开始输出新的一行
        optr = obuffer;
    }
    inline int read_int()
    { // 读入有符号整数
        char * nxt;
        int ret = strtol(iptr, &n nxt, 10);
        iptr = nxt;
        return ret;
    }
    inline double read_double() noexcep
    { // 读入浮点数
        char * nxt;
        double ret = strtod(iptr, &n nxt);
        iptr = nxt;
        return ret;
    }
    inline void write_int(int val)
    { // 输出有符号整数，输出完一行后需要调用 flush
        char tmp[32], *now = tmp + 20;
        int length = 1;
        if (val < 0)
        {
            *optr++ = '-';
            val *= -1;
        }
        *now = ' ';
        do
        {
            *--now = '0' + val % 10;
            val /= 10;
            length += 1;
        } while (val >0);
        memcpy(optr, now, length);
        optr += length;
    }
    inline void flush()
    {
        if (optr != obuffer)
        {
            optr[-1] = '\n';
        }
        fwrite(obuffer, 1, optr - obuffer, stdout);
        fflush(stdout);
    }
}
int main()
{
    io::start_reading();
    double val = io::read_double();
    cout << fixed << setprecision(6) << val <<"\n";
    return 0;
}
```

8.10 开启 O3 优化

第二行不一定支持，一般来说第一行够用了

```
#pragma GCC optimize("O3, unroll - loops")
#pragma GCC target("avx2, bmi, bmi2, lzcnt, popcnt")
```

8.11 卡时

比如要卡在两秒以内

```
if (clock() >= 1.9 * CLOCKS_PER_SEC) break;
```

8.12 运行脚本

```bash
#!/bin/bash
# 用法: ./run.sh [源文件] [超时时间]
# 例子: ./run.sh a.cpp 5s
set -euo pipefail
# 参数与默认值
src=${1:-a.cpp}
t=${2:-1s}
bin=${src%.cpp}
in=${bin}.in
# 编译（仅在源文件更新时）
if [ ! -f "${bin}_asan" ] || [ "$src" -nt "${bin}_asan" ]; then
    g++ -std=c++20 -Og -g -fsanitize=address
        -o${bin}_asan "$src" || { echo \texttt{CE} >&2; exit 1; }
fi
# 不加检查 速度正常
# g++ -std=c++20 -O2 -Wall -Wextra -o"$bin" "$src" || { echo \texttt{CE} >&2; exit 1; }
# 记录开始时间（毫秒）
s=$(date +%s%3N)
# 运行并限时
set +e; timeout "$t" "./${bin}_asan" <"$in"; st=$?; set -e
# 记录结束时间（毫秒）
e=$(date +%s%3N)
# 输出运行时间
echo "run time: $((e-s)) ms"
# 处理超时
[ $st -eq 124 ] && echo \texttt{TLE}
# 以程序自身退出码退出
exit $s
```

8.13 对拍

```cpp
for (int i = 0; i < 10000; i++)
{
    system("./data >./data.in"); //system("data.exe > data.in");
    system("./test<./data.in >./test.out"); //system("test.exe<data.in > test.out");
    system("./std<./data.in >./std.out"); //system("std.exe<data.in > std.out");
    if (system("diff ./test.out ./std.out"))//system("fc test.out std.out")
    {
        cout<<"\texttt{WA}\n";
        return 0;
    }
    else if (i % 20 == 0) cout<<"\texttt{OK} "<<i<<"\n";
}
```

8.13.1 构造题的对拍

例如构造 n 个数的和为 m

duipai.cpp

```cpp
for (int i = 0; i < 10000; i++)
{
    system("./data >./data.in");
    system("./test<./data.in >./test.out");
    system("./check<test.out > check.out");
    system("./yes > yes.out");
    if (system("diff ./check.out ./yes.out"))
    {
        cout<<"\texttt{WA}\n";
        return ;
    }
    else if (i % 20 == 0) cout<<"\texttt{OK} "<<i << endl;
}
```

data.cpp

```cpp
void solve()
{
    int n = rd(3, 7), m = rd(n, 2 * n);
```

```
    cout << n<<" "<<m<<"\n";
}
```

test.cpp

```
void solve()
{
    int n, m;
    cin >> n >> m;
    cout << n<<" "<<m<<"\n"; //要输出数据 方便 checker 检查
    for (int i = 1; i < n; i++) cout<<"1 ";
    cout << m - n + 1<<"\n";
}
```

yes.cpp (用来对 check 结果作判断)

```
void solve() { cout<<"yes\n"; }
```

check.cpp

```
void solve()
{
    int n, m;
    cin >> n >> m;
    cout << n<<" "<<m<<"\n"; //要输出数据 方便 checker 检查
    for (int i = 1; i < n; i++) cout<<"1 ";
    cout << m - n + 1<<"\n";
}
```

8.13.2 随机生成整数序列

```
mt19937 myrand(chrono::steady_clock::now().time_since_epoch().count());
int rd(int l, int r) { return uniform_int_distribution<int>(l, r)(myrand); }
int main()
{
    int n = rd(1, 10);
    cout << n << endl;
    for (int i = 1; i <= n; i++) cout << rd(1, 100)<<" ";
}
```

8.13.3 生成一棵树

```
mt19937 myrand(chrono::steady_clock::now().time_since_epoch().count());
int rd(int l, int r) { return uniform_int_distribution<int>(l, r)(myrand); }
int main()
{
    int n = rd(1, 10);
    cout << n << endl;
    for (int i = 1; i <= n; i++) p[i] = i;
    for (int i = 2, l, r; i <= n; i++)
    {
        do {
            l = rd(1, n), r = rd(1, n);
        } while (find(l) == find(r));
        cout << l<<" "<<r<<" "<<rd(1, 10)<<"\n";
        p[find(l)] = find(r);
    }
}
```

8.13.4 生成一个简单图

```
array<int, 2> e[N];
mt19937 myrand(chrono::steady_clock::now().time_since_epoch().count());
int rd(int l, int r) { return uniform_int_distribution<int>(l, r)(myrand); }
```

```cpp
int main()
{
    int n = rd(3, 6), m = rd(0, (n - 1) * n / 2);
    map < array<int, 2>, bool > h;
    for (int i = 1, x, y; i <= m; i++)
    {
        do {
            x = rd(1, n), y = rd(1, n);
        } while (x == y || h[{x, y}]);
        e[i] = {x, y};
        h[e[i]] = h[{y, x}] = true;
    }
    random_shuffle(e + 1, e + m + 1);
    cout << n<<" "<<m << endl;
    for (int i = 1; i <= m; i++)
    {
        int val = rd(1, 10);
        cout << e[i][0]<<" "<<e[i][1]<<" "<<val << endl;
    }
}
```

8.13.5 生成一个简单联通图

```cpp
array<int, 2> e[N];
mt19937 myrand(chrono::steady_clock::now().time_since_epoch().count());
int rd(int l, int r) { return uniform_int_distribution<int>(l, r)(myrand); }
int main()
{
    int n = rd(3, 6), m = rd(n, (n - 1) * n / 2);
    map < array<int, 2>, bool > h;
    for (int i = 1; i < n; i++)
    {
        int fa = rd(1, i);
        e[i] = {fa, i + 1}, h[e[i]] = h[{i + 1, fa}] = true;
    }
    for (int i = n, x, y; i <= m; i++)
    {
        do {
            x = rd(1, n), y = rd(1, n);
        } while (x == y || h[{x, y}]);
        e[i] = {x, y}, h[e[i]] = h[{y, x}] = true;
    }
    random_shuffle(e + 1, e + m + 1);
    cout << n<<" "<<m << endl;
    for (int i = 1; i <= m; i++)
    {
        int val = rd(1, 10);
        cout << e[i][0]<<" "<<e[i][1]<<" "<<val << endl;
    }
}
```

8.14 测空间

```cpp
bool st;
long long a[5000000];
bool ed;
cout << fixed << setprecision(3) << (&ed-&st) / 1048576.0 <<" MB\n";
```

8.15 pbds

好用的哈希表 (define int long long 要插在中间)

```cpp
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
const int RANDOM = std::chrono::high_resolution_clock::now().time_since_epoch().count();
struct chash {
    int operator()(int x) const { return x ^ RANDOM; }
};
```

```cpp
#define int long long
typedef gp_hash_table<int, int, chash> hash_t;
```

封装好的平衡树 (貌似每次都要插入一个 pair 如果第二维不需要随便插个大于 0 的数应该就行)

```cpp
#include <ext/pb_ds/assoc_container.hpp>
#include <bits/stdc++.h>
#define pb __gnu_pbds
#define pii pair<int, int>
using namespace std;
pb::tree<pii, pb::null_type, less<pii>,
pb::rb_tree_tag, pb::tree_order_statistics_node_update> tr;
int main()
{
    int n, opt, x;
    while (n--)
    {
        cin >> opt >> x;
        if (opt == 1) tr.insert({x, n}); //插入一个数 x
        if (opt == 2) tr.erase(tr.lower_bound({x, 0})); //删除一个数 x
        if (opt == 3) cout << tr.order_of_key({x, 0}) + 1<<'\n'; //查询 x 的排名
        if (opt == 4) cout << tr.find_by_order(x - 1)-> first<<'\n'; //查询排名为 x 的数
        if (opt == 5) cout << ((--tr.lower_bound({x, 0}))-> first)<<'\n'; //求 x 的前驱
        if (opt == 6) cout << (tr.lower_bound({x, 1 << 30})-> first)<<'\n'; //求 x 的后继
    }
}
```

8.16 bitset

8.16.1 基本操作

```cpp
#include <bitset>
bitset<N> bs;                    // 声明长度为 N 的 bitse
bs.set()                         // 全部变为 1
bs.reset()                       // 全部变为 0
bs.set(x)                        // x 这一位变为 1
bs.reset(x)                      // x 这一位变为 0
bs.flip()                        // 全部取反
bs.flip(x)                       // x 这一位取反
bs[x]                            // 访问第 x 位
bs.count()                       // 返回 1 的个数
bs.size()                        // 返回 bitset 的大小
bs.any()                         // 是否存在 1
bs.none()                        // 是否全为 0
bs.test(x)                       // 测试第 x 位是否为 1
bs._Find_first()                 // 第一个 1 的位置，不存在返回 size()
bs._Find_next(x)                 // x 后面第一个 1 的位置，不存在返回 size()
__builtin_popcount(x)            // 返回 x 的二进制中 1 的个数
__builtin_popcountll(x)          // 64 位版本
```

8.16.2 运算操作

```cpp
bitset<N> a, b;
a & b                            // 按位与
a | b                            // 按位或
a ^ b                            // 按位异或
~a                               // 按位取反
a << k                           // 左移 k 位
a >> k                           // 右移 k 位
a <<= k; a >>= k;                // 原地移位
```

8.16.3 应用场景

- **01 背包优化：**转移方程 $dp| = dp << w_i$，复杂度 $O(\frac{n \cdot W}{w})$
- **完全背包优化：**二进制拆分，$dp| = dp << (2^k \cdot w_i)$
- **集合运算：**两个 bitset 取或可快速判断集合是否有交集
- **状态压缩：**用一位表示一个状态，节省空间

8.16.4 手写 bitset

当需要实现 STL bitset 不支持的操作或动态大小时，可以手写实现：

```cpp
struct Bitset
{
    using u64 = uint64_t;
    static const int BitNum = 64;
    vector<u64> data; int size, n; u64 mask;

    Bitset(int _n = 0) : n(_n)
    {
        size = (n >> 6) + ((n & 63) ? 1 : 0);
        data.assign(size, 0);
        int rem = n & 63;
        mask = rem ? ((1ull << rem) - 1) : ~0ull;
    }
    void reset() { fill(data.begin(), data.end(), 0); }
    void set(int pos, bool val = true)
    {
        int idx = size - (pos >> 6) - 1, bit = pos & 63;
        if (val) data[idx] |= (1ull << bit);
        else data[idx] &= ~(1ull << bit);
    }
    bool test(int pos) const
    {
        int idx = size - (pos >> 6) - 1, bit = pos & 63;
        return data[idx] & (1ull << bit);
    }
    Bitset& operator<<=(int v)
    {
        if (v < 0) { *this >>= -v; return *this; }
        int whole = v >> 6; v &= 63;
        for (int i = 0; i + whole < size; i++) data[i] = data[i + whole];
        for (int i = size - whole; i < size; i++) data[i] = 0;
        if (v > 0)
        {
            u64 carry = 0;
            for (int i = size - 1; i >= 0; i--)
            {
                u64 new_carry = data[i] >> (BitNum - v);
                data[i] = (data[i] << v) | carry;
                carry = new_carry;
            }
        }
        data[0] &= mask; return *this;
    }
    Bitset& operator>>=(int v)
    {
        if (v < 0) { *this <<= -v; return *this; }
        int whole = v >> 6; v &= 63;
        for (int i = size - 1; i >= whole; i--) data[i] = data[i - whole];
        for (int i = whole - 1; i >= 0; i--) data[i] = 0;
        if (v > 0)
        {
            u64 carry = 0;
            for (int i = 0; i < size; i++)
            {
                u64 new_carry = (data[i] & ((1ull << v) - 1)) << (BitNum - v);
                data[i] = (data[i] >> v) | carry;
                carry = new_carry;
            }
        }
        data[0] &= mask; return *this;
    }
    Bitset operator&(const Bitset& other) const
    {
        Bitset result(n);
        for (int i = 0; i < size; i++) result.data[i] = data[i] & other.data[i];
        return result;
    }
    Bitset operator|(const Bitset& other) const
    {
        Bitset result(n);
        for (int i = 0; i < size; i++) result.data[i] = data[i] | other.data[i];
        return result;
```

```
    }
    Bitset operator^(const Bitset& other) const
    {
        Bitset result(n);
        for (int i = 0; i < size; i++) result.data[i] = data[i] ^ other.data[i];
        return result;
    }
    Bitset operator~() const
    {
        Bitset result(n);
        for (int i = 0; i < size; i++) result.data[i] = ~data[i];
        result.data[0] &= mask; return result;
    }
    bool any() const { for (int i = 0; i < size; i++) if (data[i]) return true; return false;
    ↪  }
    bool none() const { return !any(); }
    int count() const
    {
        int cnt = 0;
        for (int i = 0; i < size; i++) cnt += __builtin_popcountll(data[i]);
        return cnt;
    }
    void flip() { *this = ~(*this); }
    void flip(int pos) { set(pos, !test(pos)); }
};
```

8.17 vector

```
vector dp(n, vector<int>(m, 0));
vector<vector<int>> e(n);
```

8.18 VSCode 比赛设置

通用设置:

1. 创建专门的代码文件夹并在 VSCode 中打开
2. File → Auto Save，启用自动保存
3. 创建测试 .cpp 文件，运行后编辑 launch.json，将 externalConsole 设为 true
4. File → Preferences → Keyboard Shortcuts，删除 F11 快捷键，将 F5 的功能改为 F11
5. Windows 系统：右键终端标题栏 → 设置 → 默认值 → 高级 → 关闭行为 → 从不自动关闭
6. 电源设置：高性能模式，屏幕永不变暗
7. 注意：可能需要点击齿轮图标才能找到 launch.json

Ubuntu:

1. 在桌面创建文件夹 xcpc
2. 打开 VSCode，点击侧边栏第一个图标 → Open Folder → Desktop/xcpc → Open → 选择信任
3. 展开文件夹，新建 a.cpp 文件
4. File → Auto Save，启用自动保存
5. File → Preferences → Keyboard Shortcuts，搜索 cph，找到 run-testcases，改为 F11
6. Settings → CPH，设置编译参数：-std=c++17 -O2 -Wall
7. g++ -o a a.cpp 编译，./a 运行执行文件

8.19 比赛注意事项

- 注意数据范围：确保数组大小足够，避免 RE
- 增加有效沟通，减少无效沟通
- 注意运算过程中的溢出问题
- 调试特殊样例，手工构造多组测试数据
- 至少手造两组随机样例和两组特殊样例进行测试，不要盲目提交
- 专注自己的题目，不要看队友写代码，不要等待评测结果
- 如果没有自己的题目，给队友打下手（构造样例、拷贝模板、检查代码）
- 多测时确保所有变量都正确清空
- 检查 n、m 等变量是否在全局和局部均有定义
- 计算几何：精度设为 1e-12，全部使用 long double，避免精度问题

8.20 u 群图

U 群常见问题速查

多项式复杂度

可以除 w 或除 $\log$，很难降次数。$\omega < 2.372$ 为矩阵乘法复杂度指数下界。

logbit-OV、LCS、 两数最大 or/最小 and	$O(n^2)$
nbit-OV	$O(n^\omega)$
链 mex	$O(n^{\omega/2})$
$\min+$ 卷积	$O(n^2)$
3SUM、4SUM、两对数和相等、 三点共线、三线共点、 长为 3 等差数列	$O(n^2)$
3XOR	$O(n^2)$
APSP、$\min+$ 矩阵乘法	$O(n^3)$
$+\times/01$ 矩阵乘/矩阵求逆/高斯消元	$O(n^\omega)$
$\min\max$ 矩阵乘	$O(n^{2-(\omega+3)/2})$
and or 矩阵乘、稠密图传递闭包	$O(n^{2-\omega})$
稀疏图传递闭包	$O(n^2)$
区间相等对/逆序对、链颜色数、 稀疏图三元环计数、行加列求和、 区间 ±1 全局数 0	$O(n^{\omega/2-2\omega/(\omega+1)})$

——From CommonAnts

约数个数/不同质因子个数表

$n\le$	10^1	10^2	10^3	10^4	10^5	10^6	10^7	10^8	10^9
$\max\{\omega(n)\}$	2	3	4	5	6	7	8	8	9
$\max\{d(n)\}$	4	12	32	64	128	240	448	768	1344
$\arg\max\{d(n)\}$	8	96	840	9240	83160	997920	8648640	98017920	931170240
$n\le$	10^{10}	10^{11}	10^{12}	10^{13}	10^{14}	10^{15}	10^{16}	10^{17}	10^{18}
$\max\{\omega(n)\}$	10	10	11	12	12	13	13	14	15
$\max\{d(n)\}$	2304	4032	6720	10752	17280	26880	41472	64512	103680
$\arg\max\{d(n)\}$	9311702400	97772875200	963761198400	9316358251200	97821761637600	978217616376000	9897423832296000	96172766227137600	897612484786617600

图灵奖

参考资料：学会计算模型，复杂性，归约/问题类等计算理论基本思想后，自行搜索精细复杂性理论入门讲解 [Fine-grained complexity]

相关 OI 博客：

- lca 的讲解（bilibili 视频 @蔡德仁）
- lxl/crz 的相关讲解（资料集）
- EI《一些经典问题比暴力快一点点的算法》（博客）
- Ynoi《浅谈归约矩乘》（洛谷博客）
- Futari《归约矩乘》（洛谷博客）
- do-while-true《一小类矩阵乘法相关归约》（博客）
- hqztrue《浅谈矩阵乘法在算法竞赛中的应用》（知乎）
- 杨敏行《浅谈复杂度及其在解决问题方面的应用》
  闫陈效《计算理论与 OI 中的难解问题》（2024 集训队论文集）

——From CommonAnts

NP-Hard

- Karp's 21 NP-Complete Problems
- m 个大小均为 c 的背包能否装下给定物品（强 NPC，即不存在伪多项式算法）
- 简单环计数
- 二分图完美匹配计数
- 2-SAT 计数
- 拓扑序计数
- 无向图欧拉回路/欧拉路径计数

问题与做法

- 单点修改区间 mex
  考虑一个值 v，v 没出现说明 $[l,r]$ 夹在 v 的相邻两次出现之间。因此对于每个值 v 的相邻两次出现 i 和 j，在数据结构中插入点 (i,j)、权值为 v；查询为 $i<l\le r<j$ 的点的权值最小值（单点修改，矩形求 $\min$），使用树套树即可。

网站

- 那个画图的网站?
  csacademy.com/app/graph_editor/
- 那个查原题的网站?
  www.yuantiji.ac/zh
- 那个 AtCoder 评分的网站?
  kenkoooo.com/atcoder/#/table/
- 那个能看 Codeforces 题目整理的网站?
  cftracker.netlify.app/contests
- 那个能看所有 OJ 的比赛的网站?
  clist.by
- 那个看编译得到的汇编的网站?
  godbolt.org
- 那个把 C 语言类型转成人话的网站?
  cdecl.org
- XXX 用 LaTeX 怎么打?
  detexify.kirelabs.org/classify.html
  许多网站使用 KaTeX 渲染公式，相应文档见 katex.org/docs/supported

其他

- vuqa 是什么意思?
  van 能的 UOJ 群啊
- UB (Undefined Behavior，未定义行为) 是什么?
  zh.cppreference.com/w/cpp/language/ub
  - 我的引用怎么失效了（`tr[x].ls=insert()`)?
    在 C++ 17 之前，这段代码可能的求值顺序是：求出 `tr[x]` 的位置 → 插入（此时 `vector` 可能因为扩容而移动 `tr` 的存储的位置） → 已经求出来了的 `tr[x]` 的位置失效了，但我们要往这个失效的位置写入数据 → 寄了。
  - 扩展阅读：求值顺序（**不是运算结合性**）：
    zh.cppreference.com/w/cpp/language/eval_order
- 怎么做 NOI 排版风格的题面?
  官方工具：TUACK + LaTeX
  广告：/Wallbreaker5th/fuzzy-cnoi-statement
  /Wallbreaker5th/OI-statement-LaTeX
- 那个 U 群常见问题速查的图?

Version: 2025-05-08
Visit /Wallbreaker5th/vuqa for the source code and the latest version.