1. 双指针

[双指针 - OI wiki]

使用一前一后两个指针，根据一定规则依次移动来遍历数组并进行处理。

双指针常用来进行**维护区间信息**，**子序列匹配**等操作，是比较常用的基础方法，经常用来配合其他算法进行求解问题。

2. 离散化

[离散化 - OI wiki]

概念

给定一组数字，进行操作时不需关心这组每个数的绝对大小，只需要关心相对大小。这时可以根据他们的相对大小来重新分配编号，之后利用编号进行操作，如：

```
原数组：
[3, 10, 4, 8, 1000]
离散化后：
[1, 4, 2, 3, 5]
```

有时给定的数据能达到1e9，但明显数组不能开这么大，就可以使用离散化的技巧。

实现

利用STL排序和去重函数进行离散化。

```
// 数组
int arr[N], tmp[N];
for (int i = 1; i <= n; ++i)  // step 1 创建原数组的副本
    tmp[i] = arr[i];
std::sort(tmp + 1, tmp + n + 1);  // step 2 将副本中的值从小到大排序
int len = std::unique(tmp + 1, tmp + n + 1) - (tmp + 1);  // step 3 将排序好的副本去重
for (int i = 1; i <= len; ++i)   // step 4 查找原数组的每一个元素在副本中的位置，位置即为排名，将其作为离散化后的值
    arr[i] = std::lower_bound(tmp + 1, tmp + len + 1, arr[i]) - tmp;


// vector
int n;
vector<int> v(n);
for(int i = 0; i < n; i++) cin >> v[i];
sort(v.begin(), v.end()); // 排序
v.erase(unique(v.begin(), v.end()), v.end()); // 去重
n = v.size(); // 去重后的大小
```

```
// int index[N];
map<int, int> index; // 经常数据范围到1e9
int cnt = 0;
for(int i : v) index[i] = ++cnt; // 利用map进行编号



// 把相同的元素跟据输入顺序离散化为不同的数据
struct Data {
    int idx, val;

    bool operator<(const Data& o) const { // 将副本按值从小到大排序，当值相同时，按出现顺序从小到大排序
        if (val == o.val)
            return idx < o.idx;
        return val < o.val;
    }
} tmp[maxn];
int a[N];

for (int i = 1; i <= n; ++i) tmp[i] = (Data){ i, a[i] }; // 创建原数组的副本，同时记录每个元素出现的位置
std::sort(tmp + 1, tmp + n + 1);
for (int i = 1; i <= n; ++i) arr[tmp[i].idx] = i; // 将离散化后的数字放回原数组
```

线段树、树状数组等数据结构会用到。

3. 莫队

普通莫队算法

带修改莫队

树上莫队

离线算法

离线算法是**非交互式**的，可以先读取完所有询问，再一起回答。因为考虑到了所有的询问，所以可以获得比在线情况下更好的复杂度。

概念

莫队 = 离线 + 暴力 + 分块

莫队算法的关键在于，根据问题的类型，对每个询问进行**排序**，每个询问中暴力处理，每次得到的答案可以对下一次询问做出贡献，得以优化。

莫队算法可以解决一类离线区间询问问题，适用性极为广泛。同时将其加以扩展，便能轻松处理树上路径询问以及支持修改操作。

具体实现

首先将数组进行分块，把查询的区间按**左端点所在块的编号**进行排序，如果相同就按**右端点**排序。

普通莫队

```
// 关键代码
struct Q {
    int l, r, id; // 利用id来记录对应询问的答案
} q[N];
int n, m, a[N];
int pos[N]; // 每个数所在的块的编号
int ans[N];

void add(int x) {
    // 添加操作
}
void del(int x) {
    // 删除操作
}

void solve() {
    cin >> n >> m;
    int len = sqrt(n); // 块长度
    for(int i = 1; i <= n; i++) {
        cin >> a[i];
        pos[i] = (i - 1) / len + 1;
    }
    for(int i = 1; i <= m; i++) {
        cin >> q[i].l >> q[i].r;
        q[i].id = i;
    }
    sort(q + 1, q + 1 + m, [](Q a, Q b) {
        if(pos[a.l] == pos[b.l]) return a.r < b.r;
        return pos[a.l] < pos[b.l];
    });
    int l = 1, r = 0;
    for(int i = 1; i <= m; i++) {
        while (l > q[i].l) add(--l); // [l,r] -> [l-1,r]
        while (r < q[i].r) add(++r); // [l,r] -> [l,r+1]
        while (l < q[i].l) del(l++); // [l,r] -> [l+1,r]
        while (r > q[i].r) del(r--); // [l,r] -> [l,r-1]
        ans[q[i].id] = tot;
    }
    for(int i = 1; i <= m; i++) cout << ans[i] << " \n"[i == m];
}
```

关于四个循环位置的讨论

莫队区间的移动过程，就相当于加入了 $[1,r]$ 的元素，并删除了 $[1,l-1]$ 的元素。因此，

- 对于 $l\le r$ 的情况，$[1,l-1]$ 的元素相当于被加入了一次又被删除了一次，$[l,r]$ 的元素被加入一次，$[r+1,+\infty)$ 的元素没有被加入。这个区间是合法区间。
- 对于 $l=r+1$ 的情况，$[1,r]$ 的元素相当于被加入了一次又被删除了一次，$[r+1,+\infty)$ 的元素没有被加入。这时这个区间表示空区间。
- 对于 $l>r+1$ 的情况，那么 $[r+1,l-1]$（这个区间非空）的元素被删除了一次但没有被加入，因此这个元素被加入的次数是负数。

因此，如果某时刻出现 $l>r+1$ 的情况，那么会存在一个元素，它的加入次数是负数。这在某些题目会出现问题，例如我们如果用一个 `set` 维护区间中的所有数，就会出现「需要删除 `set` 中不存在的元素」的问题。

循环顺序	正确性	反例或注释
`l--,l++,r--,r++`	错误	$l<r<l'<r'$
`l--,l++,r++,r--`	错误	$l<r<l'<r'$
`l--,r--,l++,r++`	错误	$l<r<l'<r'$
`l--,r--,r++,l++`	正确	证明较繁琐
`l--,r++,l++,r--`	正确	
`l--,r++,r--,l++`	正确	
`l++,l--,r--,r++`	错误	$l<r<l'<r'$
`l++,l--,r++,r--`	错误	$l<r<l'<r'$
`l++,r++,l--,r--`	错误	$l<r<l'<r'$
`l++,r++,r--,l--`	错误	$l<r<l'<r'$
`l++,r--,l--,r++`	错误	$l<r<l'<r'$
`l++,r--,r++,l--`	错误	$l<r<l'<r'$

全部 24 种排列中只有 6 种是正确的，其中有 2 种的证明较繁琐，这里只给出其中 4 种的证明。

这 4 种正确写法的共同特点是，前两步先扩大区间（`l--` 或 `r++`），后两步再缩小区间（`l++` 或 `r--`）。这样写，前两步是扩大区间，可以保持 $l\le r+1$；执行完前两步后，$l\le l'\le r'\le r$ 一定成立，再执行后两步只会把区间缩小到 $[l',r']$，依然有 $l\le r+1$，因此这样写是正确的。

例题：P3901 数列找不同

现有数列 $A_1,A_2,\ldots,A_N$，Q 个询问 (L_i,R_i)，询问 $A_{L_i},A_{L_i+1},\ldots,A_{R_i}$ 是否互不相同。

第一行，两个整数N,Q。
第二行，N 个整数$A_1,A_2,\ldots,A_N$。
接下来 Q 行，每行两个整数 L_i,R_i。

对每个询问输出一行，`Yes` 或 `No`。

对于 50% 的数据，$N, Q \le 10^3$。
对于 100% 的数据，$1 \le N, Q \le 10^5$，$1 \le A_i \le N$，$1 \le L_i \le R_i \le N$。

```
#include <bits/stdc++.h>
#define ll long long
using namespace std;
const int N = 1e5 + 5;

int n, m;
int a[N], cnt[N], ans[N], tot;
struct Q {
    int l, r, id;
};
int pos[N];

void add(int x) {
    if (!cnt[a[x]]) tot++;
    cnt[a[x]]++;
}
void del(int x) {
    cnt[a[x]]--;
    if (!cnt[a[x]]) tot--;
}

void solve() {
    cin >> n >> m;
    int len = sqrt(n);
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        pos[i] = (i - 1) / len + 1;
    }
    vector<Q> v(m);
    for (int i = 0; i < m; i++) {
        cin >> v[i].l >> v[i].r;
        v[i].id = i;
    }
    sort(v.begin(), v.end(), [&](Q a, Q b) {
        if (pos[a.l] == pos[b.l]) return a.r < b.r;
        return pos[a.l] < pos[b.l];
        });
    int l = 1, r = 0;
    for (int i = 0; i < m; i++) {
        while (l > v[i].l) add(--l);
        while (r < v[i].r) add(++r);
        while (l < v[i].l) del(l++);
        while (r > v[i].r) del(r--);
        ans[v[i].id] = (tot == v[i].r - v[i].l + 1);
    }
    for (int i = 0; i < m; i++) {
        if (ans[i]) cout << "Yes\n";
        else cout << "No\n";
    }
}
int main()
{
```

```
    ios::sync_with_stdio(false), cin.tie(0), cout.tie(0);
    int _T = 1;
    // cin >> _T;
    while (_T--)
        solve();
    return 0;
}
```

带修改莫队

普通莫队是不能带修改的。

我们可以强行让它可以修改，就像 DP 一样，可以强行加上一维 **时间维**, 表示这次操作的时间。

时间维表示经历的修改次数。

即把询问 $[l,r]$ 变成 $[l,r,time]$。这三个维度都可以前后移动：

- $[l-1,r,time]$
- $[l+1,r,time]$
- $[l,r-1,time]$
- $[l,r+1,time]$
- $[l,r,time-1]$
- $[l,r,time+1]$

但是带修改莫队的**分块长度取值与普通莫队不同**，在 $len=n^{2/3}$ 的时候达到最优复杂度 $O(n^{5/3})$。

此外，排序的第二关键字是 r 所在块的编号，而不是 r 的大小。

例题：

P1903 [国家集训队] 数颜色 / 维护队列

题目大意：给你一个序列，M 个操作，有两种操作：

1. 修改序列上某一位的数字
2. 询问区间 $[l,r]$ 中数字的种类数（多个相同的数字只算一个）

```
#include <bits/stdc++.h>
#define ll long long
using namespace std;
const int N = 1e6 + 5;

int n, m;
struct Q { // 记录查询操作，t为这次查询之前经过的修改次数
    int id, t, l, r;
} q[N];
struct RE { // 记录修改操作
    int p, c;
} r[N];
int cnt[N];
int a[N], pos[N], ans[N], tot;
int qcnt, rcnt;

inline void add(int x) {
    if (cnt[x] == 0) tot++;
    cnt[x]++;
```

```cpp
}
inline void del(int x) {
    if (cnt[x] == 1) tot--;
    cnt[x]--;
}

inline void solve() {
    cin >> n >> m;
    int len = pow(n, 2.0 / 3);
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        pos[i] = (i - 1) / len + 1;
    }
    for (int i = 0; i < m; i++) {
        char ch;
        int x, y;
        cin >> ch >> x >> y;
        if (ch == 'Q') {
            q[++qcnt] = { qcnt, rcnt, x, y };
        }
        else {
            r[++rcnt] = { x, y };
        }
    }
    sort(q + 1, q + 1 + qcnt, [&](Q a, Q b) {
        if (pos[a.l] != pos[b.l]) return pos[a.l] < pos[b.l];
        if (pos[a.r] != pos[b.r]) return pos[a.r] < pos[b.r];
        return a.t < b.t;
        });
    int L = 1, R = 0, last = 0; // last记录最后一次修改的时间点
    for (int i = 1; i <= qcnt; i++) {
        while (R < q[i].r) add(a[++R]);
        while (R > q[i].r) del(a[R--]);
        while (L > q[i].l) add(a[--L]);
        while (L < q[i].l) del(a[L++]);
        // 时间维度的变化
        while (last < q[i].t) { // 进行修改
            last++;
            if (L <= r[last].p && r[last].p <= R) {
                add(r[last].c);
                del(a[r[last].p]);
            }
            swap(a[r[last].p], r[last].c); // 更新，回退时会用到
        }
        while (last > q[i].t) { // 回退修改
            if (L <= r[last].p && r[last].p <= R) {
                add(r[last].c);
                del(a[r[last].p]);
            }
            swap(a[r[last].p], r[last].c);
            last--; // 先修改再last--
        }
        ans[q[i].id] = tot;
    }

    for (int i = 1; i <= qcnt; i++) {
        cout << ans[i] << '\n';
    }
```

```
}
int main()
{
    ios::sync_with_stdio(false), cin.tie(0), cout.tie(0);
    int _T = 1;
    // cin >> _T;
    while (_T--)
        solve();
    return 0;
}
```

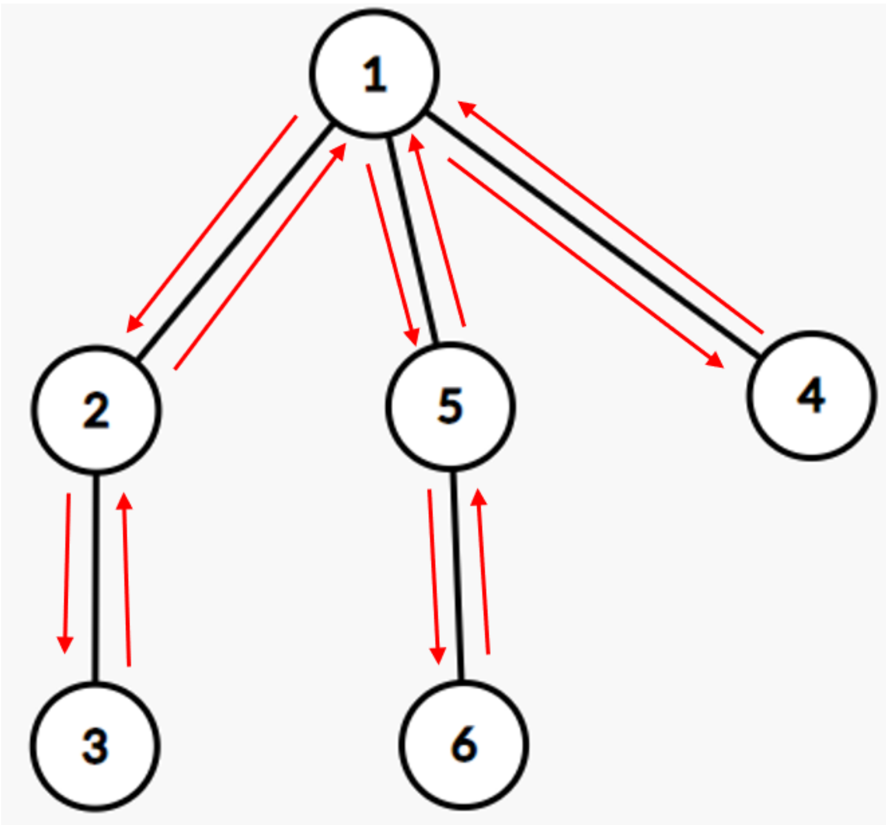
树上莫队

普通莫队和带修改莫队都是在一维数组上进行的操作，如果其他数据结构也转换成一维数组的形式，那么也可以用莫队来求解区间问题。

对于树来说，可以根据他的**欧拉序**来将整棵树转换为一维数组表示，然后在欧拉序上分块、跑莫队。

欧拉序指的是，从树的根节点出发，依次dfs每个节点。每次进入这个节点的dfs过程时，将这个节点 push_back 到欧拉序列的后面，完成这个点的dfs过程后，将这个点再次加入到欧拉序列的最后。最后得到的欧拉序的长度为 $2n$

比如下面这个例子，得到的欧拉序就是 $[1,2,3,3,2,5,6,6,5,4,4,1]$



获取欧拉序：

```
vector<int> ed[N];
bool vis[N];
vector<int> d; // 存放欧拉序
void dfs(int u, int f) {
    d.push_back(u); // 开始dfs时加入一次
    for(int v : ed[u]) {
        if(v == f || vis[v]) continue;
        vis[v] = 1;
```

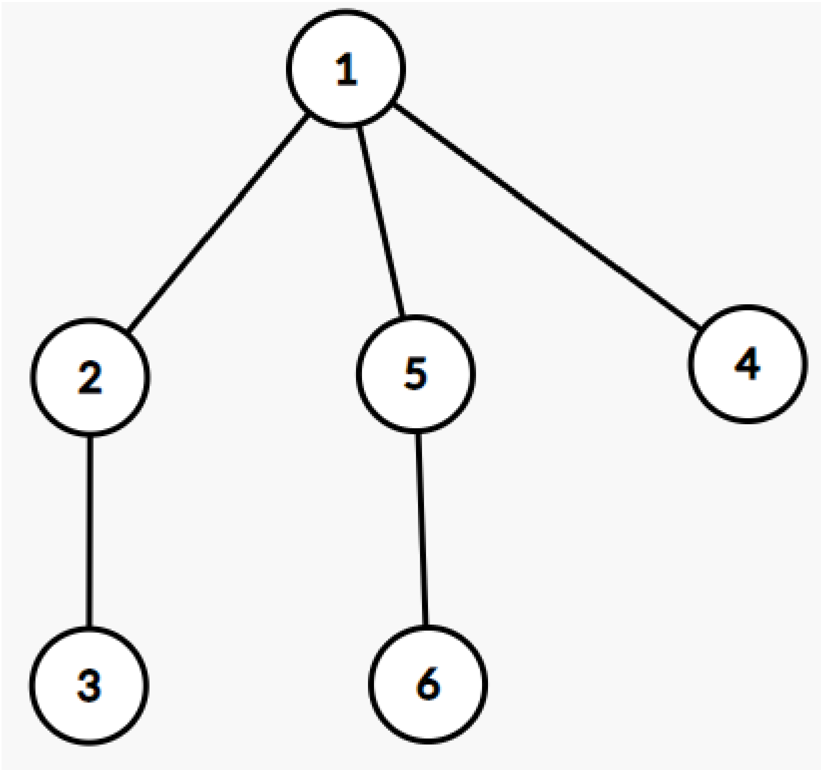
```
        dfs(v);
    }
    d.push_back(u); // 结束dfs时加入一次
}

dfs(1, 0);
```

例题 SPOJ COT2 - Count on a tree II

题意：

- 给定 n 个结点的树，每个结点有一种颜色。
- m 次询问，每次询问给出 u, v ，回答 u, v 之间的路径上的结点的不同颜色数。
- $1 \le n \le 4 * 10^4, 1 \le m \le 10^5$ ，颜色是不超过 $2 * 10^9$ 的非负整数。

要知道 u 到 v 路径上有哪些点，需要求一下他们的最近公共祖先 lca，然后有两种情况：



这棵树的欧拉序是 $[1, 2, 3, 3, 2, 5, 6, 6, 5, 4, 4, 1]$

1. $lca(u, v) = u || lca(u, v) = v$

   以 $u = 3, v = 1$ 为例，在欧拉序中的路径是 $[3, 2, 5, 6, 6, 5, 4, 4, 1]$，可以发现4、5、6号节点都出现了两次，因为他们不属于 u 到 v 的路径，于是可以删掉他们，变成 $[3, 2, 1]$。

2. $lca(u, v) \neq u || lca(u, v) \neq v$

   以 $u = 3, v = 4$ 为例，他们的 lca 是 1，路径是 $[3, 2, 5, 6, 6, 5, 4]$，去掉出现两次的点得到 $[3, 2, 4]$，但是还需要加上他们的 $lca = 1$ 这个点，也就是 $[3, 2, 4, 1]$。

对于本题，先将树转换为欧拉序，然后求任意两个节点的 lca，还要得到每个节点进入欧拉序和离开欧拉序的时间点。

然后将查询看作一维数组上的查询，进而使用莫队算法。

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

const int N = 1e5 + 5;

int n, m;
int a[N], col[N], ans[N];
int tot, pos1[N], pos2[N], pos[N];
int cnt[N];
std::vector<int> ed[N];
bool vis[N];
int idx[N];

struct Q {
    int l, r, id, lca;
};

int fa[N][20], deep[N];
inline void DFS(int x, int f) { // 倍增求lca
    deep[x] = deep[f] + 1;
    fa[x][0] = f;
    for (int i = 1; (1 << i) <= deep[x]; i++)
        fa[x][i] = fa[fa[x][i - 1]][i - 1];
    for (int i : ed[x])
        if (i != f)
            DFS(i, x);
}
inline int lca(int x, int y) {
    if (deep[x] < deep[y]) std::swap(x, y);
    for (int i = 16; i >= 0; i--)
        if (deep[x] - (1 << i) >= deep[y])
            x = fa[x][i];
    if (x == y) return x;
    for (int i = 16; i >= 0; i--)
        if (fa[x][i] != fa[y][i])
            x = fa[x][i], y = fa[y][i];
    return fa[x][0];
}

inline void update(int u, bool op) {
    if (op)
        u = idx[u];
    if (!vis[u]) {
        cnt[a[u]]++;
        if (cnt[a[u]] == 1) tot++;
    }
    else {
        if (cnt[a[u]] == 1) tot--;
        cnt[a[u]]--;
    }
    vis[u] ^= 1;
}

signed main()
{
    std::ios::sync_with_stdio(false), std::cin.tie(0), std::cout.tie(0);
```

```
    std::cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        std::cin >> a[i];
        col[i] = a[i];
    }
    std::sort(col + 1, col + 1 + n);
    auto getid = [](int c) -> int {
        return std::lower_bound(col + 1, col + 1 + n, c) - (col + 1);
        };
    for (int i = 1; i <= n; i++) { // 离散化，将颜色离散化为下标
        a[i] = getid(a[i]);
    }

    for (int i = 1; i < n; i++) {
        int u, v;
        std::cin >> u >> v;
        ed[u].push_back(v);
        ed[v].push_back(u);
    }

    std::vector<int> d; // 存储欧拉序
    std::vector<bool> vis(n + 1, 0);
    auto dfs = [&](auto&& self, int u, int f) -> void { // 求欧拉序
        d.push_back(u);
        for (int v : ed[u]) {
            if (v == f || vis[v]) continue;
            vis[v] = 1;
            self(self, v, u);
        }
        d.push_back(u);
        };
    dfs(dfs, 1, 0);

    int len = sqrt(n << 1);
    for (int i = 0; i < (n << 1); i++) {
        if (!pos1[d[i]]) {
            pos1[d[i]] = i + 1;
        }
        else {
            pos2[d[i]] = i + 1;
        }
        pos[i + 1] = i / len + 1;
        idx[i + 1] = d[i]; // idx[i]对应欧拉序中第i个节点
    }
    DFS(1, 0); // 预处理lca

    std::vector<Q> q(m);
    for (int i = 0; i < m; i++) {
        int u, v;
        std::cin >> u >> v;
        if (pos1[u] > pos1[v]) std::swap(u, v); // 后出现的先离开，在欧拉序中pos2[u]
<pos1[v]
        int F = lca(u, v);
        if (F == u || F == v) {
            q[i].l = pos1[u];
            q[i].r = pos1[v];
            q[i].lca = 0;
```

```
        }
        else {
            q[i].l = pos2[u];
            q[i].r = pos1[v];
            q[i].lca = F;
        }
        q[i].id = i;
    }

    std::sort(q.begin(), q.end(), [&](Q a, Q b) {
        if (pos[a.l] == pos[b.l]) return a.r < b.r;
        return pos[a.l] < pos[b.l];
        });
    int l = 1, r = 0;
    for (int i = 0; i < m; i++) {
        while (l < q[i].l) update(l++, 1);
        while (l > q[i].l) update(--l, 1);
        while (r > q[i].r) update(r--, 1);
        while (r < q[i].r) update(++r, 1);
        if (q[i].lca) { // 处理第二种情况
            update(q[i].lca, 0);
        }
        ans[q[i].id] = tot;
        if (q[i].lca) { // 还原
            update(q[i].lca, 0);
        }
    }
    for (int i = 0; i < m; i++) std::cout << ans[i] << std::endl;
}
```