NJUPT XCPC Templates

Yunhai Bian

2020 年 12 月 15 日

目录

1 数据结构 **3**
1.1 树状数组 3
1.2 线段树 4
1.2.1 HDU1540（lmax, rmax） 4
1.2.2 HDU4578（区间加/乘/置数，区间和/平方和/立方和） 6
1.2.3 HDU4553（维护两棵有优先关系的线段树） 10
1.2.4 HDU1542（扫描线求矩形面积并） 13
1.2.5 HDU1255（矩形 2 次覆盖面积并） 15
1.3 分块 18
1.4 ST 表 18
1.5 Splay 19
1.6 Treap 22
1.7 树链剖分 25
1.7.1 点权 25
1.7.2 边权 29
1.8 莫队 33
1.8.1 离线询问排序方式 33
1.8.2 洛谷 P4396（基础莫队） 34
1.8.3 AcWing2521（带修莫队） 36
1.8.4 AcWing2523（回滚莫队） 38
1.8.5 树上莫队 40

2 图论 **41**
2.1 最短路 41
2.1.1 朴素 Dijkstra 41
2.1.2 堆优化 Dijkstra 41
2.1.3 Bellman-Ford 42
2.1.4 SPFA 43
2.1.5 Floyd 44

2.2 最小生成树 45
2.2.1 Prim 45
2.2.2 Kruskal 45
2.3 次小生成树 46
2.4 有向图的强连通分量 46
2.5 无向图的双连通分量 46
2.6 最近公共祖先 46
2.6.1 倍增 46
2.6.2 Tarjan 48
2.6.3 树剖 49
2.7 网络流 50
2.7.1 Dinic 50
2.7.2 EK 52
2.8 二分图 53
2.8.1 染色法 53
2.8.2 匈牙利算法 54
2.8.3 最大流之二分图最大匹配 55
2.8.4 最大流之二分图多重匹配 56

3 字符串 58

3.1 Manacher 58
3.2 KMP 60
3.3 AC 自动机 60

4 其他 62

4.1 离散化 62
4.2 高精度 63

1 数据结构

1.1 树状数组

```
// 注意树状数组不能处理下标0开始
// 一维
int c[N];

inline int lowbit(int x) {
   return x & -x;
}

int add(int x, int y) {
   for (int i = x; i <= n; i += lowbit(i)) c[i] += y;
}

int sum(int x) {
   int res = 0;
   for (int i = x; i; i -= lowbit(i)) res += c[i];
   return res;
}

// 二维
LL c[N][N];

inline int lowbit(int x) {
   return x & -x;
}

void add(int x, int y, LL v) {
   for (int i = x; i <= n;i += lowbit(i))
      for (int j = y; j <= m;j += lowbit(j))
         c[i][j] += v;
}

LL query(int x, int y) {
   LL ans = 0;
   for (int i = x; i; i -= lowbit(i))
      for (int j = y; j; j -= lowbit(j))
         ans += c[i][j];
   return ans;
}
```

1.2 线段树

1.2.1 HDU1540（lmax, rmax）

题意：有 n 个点连成一条线，编号从左至右为 1~n，有三种操作：1. 摧毁一个点 2. 查询某个点能到的所有点数（包括自己）3. 重建上一次被摧毁的点。

分析：用一个栈 stk 来存放被摧毁的点，摧毁点 x 就 stk[++top] = x，重建上一个点就只需要取出栈顶 x = stk[top–]。线段树每个节点维护区间左侧连续最大长度（点数）lmax 以及右侧最大连续长度 rmax。摧毁一个点就是在线段树中找到该点并将其 lmax=rmax=0，重建就是 lmax=rmax=1，然后 pushup 上去。难点在于 号查询操作，如果点 x 在当前结点的左孩子，分两种情况来看，如果点 x 被左孩子的右侧最大连续区间包含了，那么 x 能到达的所有点数就是左孩子的 rmax + 右孩子的 lmax，否则递归直接递归左孩子即可。剩余情况类似。

```
#include <iostream>

using namespace std;

const int N = 50010;

int n, m;
struct Tree {
   int l, r;
   int lmax, rmax;
} tr[N << 2];
int stk[N], top;

void pushup(Tree &root, Tree &left, Tree &right) {
   root.lmax = left.lmax, root.rmax = right.rmax;
   if (left.r - left.l + 1 == left.lmax) root.lmax += right.lmax;
   if (right.r - right.l + 1 == right.rmax) root.rmax += left.rmax;
}

void pushup(int u) {
   pushup(tr[u], tr[u << 1], tr[u << 1 | 1]);
}

void build(int u, int l, int r) {
   if (l == r) {
      tr[u] = {l, r, 1, 1};
   } 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(int u, int x, int y) {
    if (tr[u].l == x && tr[u].r == x) {
        tr[u].lmax = tr[u].rmax = y;
    } else {
        int mid = tr[u].l + tr[u].r >> 1;
        if (x <= mid) modify(u << 1, x, y);
        else modify(u << 1 | 1, x, y);
        pushup(u);
    }
}

int query(int u, int x) {
    if (tr[u].l == x && tr[u].r == x) {
        return tr[u].lmax;
    } else {
        int mid = tr[u].l + tr[u].r >> 1;
        if (x <= mid) {
            if (tr[u << 1].r - tr[u << 1].rmax + 1 <= x) {
                return tr[u << 1].rmax + tr[u << 1 | 1].lmax;
            } else {
                return query(u << 1, x);
            }
        } else {
            if (tr[u << 1 | 1].l + tr[u << 1 | 1].lmax - 1 >= x) {
                return tr[u << 1 | 1].lmax + tr[u << 1].rmax;
            } else {
                return query(u << 1 | 1, x);
            }
        }
    }
}

int main() {
    while (scanf("%d%d", &n, &m) != EOF) {
        build(1, 1, n);
        while (m--) {
```

```
        char op[2]; int x;
        scanf("%s", op);
        if (*op == 'D') {
           scanf("%d", &x);
           modify(1, x, 0);
           stk[++top] = x;
        } else if (*op == 'R') {
           int x = stk[top--];
           modify(1, x, 1);
        } else {
           scanf("%d", &x);
           printf("%d\n", query(1, x));
        }
     }
  }
  return 0;
}
```

1.2.2 HDU4578（区间加/乘/置数，区间和/平方和/立方和）

题意：线段树区间加，区间乘，区间置数，区间和，平方和，立方和。

分析：需要维护，置数标记 same，乘法标记 mul，加法标记 add，区间和标记 s[0~2] 分别表示和，平方和，立方和。

首先确定前三个标记维护优先级，same > mul > add，然后就是三个和的维护需要推导一下。

1. 区间置数，三个和很好维护不说了
2. 区间乘 k，三个和分别乘以 k, k^2, k^3
3. 区间加 a，初始有 $s[0] = \sum x$, $s[1] = \sum x^2$, $s[2] = \sum x^3$，区间长度为 len

$$\sum(x+a) = \sum x + \sum a = s[0] + len * a \tag{1}$$

$$\sum(x+a)^2 = \sum x^2 + 2a\sum x + \sum a^2 = s[1] + 2a * s[0] + len * a^2 \tag{2}$$

$$\sum(x+a)^3 = \sum x^3 + 3a\sum x^2 + 3a^2\sum x + \sum a^3 = s[2] + 3a*s[1] + 3a^2*s[0] + len*a^3 \tag{3}$$

注意维护和的时应该倒序维护（立方和，平方和，和），防止要用的值被先更新了。

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

using namespace std;
```

```

typedef pair<int, int> PII;
typedef long long LL;

const int N = 100010, mod = 10007;

int n, m;
struct Tree {
    int l, r;
    LL same, mul, add, s[3];
 } tr[N << 2];

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

void pushup(int u) {
    for (int i = 0; i < 3; i++) {
        tr[u].s[i] = (tr[u << 1].s[i] + tr[u << 1 | 1].s[i]) % mod;
    }
}

void pushdown(int u) {
    auto &root = tr[u], &left = tr[u << 1], &right = tr[u << 1 | 1];
    if (root.same) {
        left.same = right.same = root.same;
        left.mul = right.mul = 1, left.add = right.add = 0;
        LL base = 1;
        for (int i = 0; i < 3; i++) {
            (base *= root.same) %= mod;
            left.s[i] = (left.r - left.l + 1) * base % mod;
            right.s[i] = (right.r - right.l + 1) * base % mod;
        }
        root.same = 0;
    }
    if (root.mul != 1) {
        LL k = root.mul;
        (left.mul *= k) %= mod, (right.mul *= k) %= mod;
        (left.add *= k) %= mod, (right.add *= k) %= mod;
        LL base = 1;
```

```cpp
        for (int i = 0; i < 3; i++) {
            (base *= k) %= mod;
            (left.s[i] *= base) %= mod, (right.s[i] *= base) %= mod;
        }
        root.mul = 1;
    }
    if (root.add) {
        LL a = root.add;
        (left.add += a) %= mod, (right.add += a) %= mod;
        (left.s[2] += 3 * a * left.s[1] + 3 * a * a * left.s[0] + (left.r - left.l + 1) * a * a * a) %= mod;
        (right.s[2] += 3 * a * right.s[1] + 3 * a * a * right.s[0] + (right.r - right.l + 1) * a * a * a) %= mod;
        (left.s[1] += 2 * a * left.s[0] + (left.r - left.l + 1) * a * a) %= mod;
        (right.s[1] += 2 * a * right.s[0] + (right.r - right.l + 1) * a * a) %= mod;
        (left.s[0] += (left.r - left.l + 1) * a) %= mod, (right.s[0] += (right.r - right.l + 1) * a) %= mod;
        root.add = 0;
    }
}

// [l,r]乘k再加a
void modify_mul_add(int u, int l, int r, LL k, LL a) {
    if (tr[u].l >= l && tr[u].r <= r) {
        if (k != 1) {
            (tr[u].mul *= k) %= mod;
            (tr[u].add *= k) %= mod;
            LL base = 1;
            for (int i = 0; i < 3; i++) {
                (base *= k) %= mod;
                (tr[u].s[i] *= base) %= mod;
            }
        }
        if (a) {
            (tr[u].add += a) %= mod;
            (tr[u].s[2] += 3 * a * tr[u].s[1] + 3 * a * a * tr[u].s[0] + (tr[u].r - tr[u].l + 1) * a * a * a) %= mod;
            (tr[u].s[1] += 2 * a * tr[u].s[0] + (tr[u].r - tr[u].l + 1) * a * a) %= mod;
            (tr[u].s[0] += (tr[u].r - tr[u].l + 1) * a) %= mod;
        }
```

```
    } else {
        pushdown(u);
        int mid = tr[u].l + tr[u].r >> 1;
        if (l <= mid) modify_mul_add(u << 1, l, r, k, a);
        if (r > mid) modify_mul_add(u << 1 | 1, l, r, k, a);
        pushup(u);
    }
}

void modify_assign(int u, int l, int r, int c) {
    if (tr[u].l >= l && tr[u].r <= r) {
        tr[u].same = c, tr[u].mul = 1, tr[u].add = 0;
        LL base = 1;
        for (int i = 0; i < 3; i++) {
            (base *= tr[u].same) %= mod;
            tr[u].s[i] = (tr[u].r - tr[u].l + 1) * base % mod;
        }
    } else {
        pushdown(u);
        int mid = tr[u].l + tr[u].r >> 1;
        if (l <= mid) modify_assign(u << 1, l, r, c);
        if (r > mid) modify_assign(u << 1 | 1, l, r, c);
        pushup(u);
    }
}

LL query(int u, int l, int r, int type) {
    if (tr[u].l >= l && tr[u].r <= r) {
        return tr[u].s[type];
    } else {
        pushdown(u);
        LL res = 0;
        int mid = tr[u].l + tr[u].r >> 1;
        if (l <= mid) (res += query(u << 1, l, r, type)) %= mod;
        if (r > mid) (res += query(u << 1 | 1, l, r, type)) %= mod;
        return res;
    }
}

int main() {
     while (cin >> n >> m && n && m) {
         build(1, 1, n);
         while (m--) {
```

```
        int type, x, y, c;
        scanf("%d%d%d%d", &type, &x, &y, &c);
        if (type == 1) {
          modify_mul_add(1, x, y, 1ll, c);
        } else if (type == 2) {
          modify_mul_add(1, x, y, c, 0ll);
        } else if (type == 3) {
          modify_assign(1, x, y, c);
        } else {
          printf("%lld\n", query(1, x, y, c - 1));
        }
      }
    }
    return 0;
}
```

1.2.3 HDU4553（维护两棵有优先关系的线段树）

题意：有一个长度为 n 的时间轴，有两种操作：1. DS QT 表示屌丝申请第一段长度为 QT 的空闲时间，能申请到就输出起始时间。2. NS QT 表示女神申请第一段长度为 QT 的空闲时间，如果能申请到输出起始时间，如果找不到，可以无视屌丝已经申请的时间，再找到一个第一个连续空闲时间大于等于 QT 的起始位置。3. STUDY!! L R 表示清空这段时间的所有申请用于学习，由于三分钟热度，之后再有人申请到 STUDY 的时间还是会分配出去。

分析：线段树维护两个时间轴的信息，分别表示屌丝时间轴的分配情况，还有女神时间轴的分配情况。详见代码，下标 0 表示屌丝，下标 1 表示女神。same 为区间相同的值的标记，lmax, rmax, tmax 分别表示区间左侧最长连续空闲时间，右侧最长连续空闲时间，区间内最长连续空闲时间，用 1 表示空闲。然后根据题目要求操作即可。

```
// 1表示空闲
#include <bits/stdc++.h>

using namespace std;

typedef pair<int, int> PII;
typedef long long LL;

const int N = 100010;

int T, Case, n, m;
```

```
struct Tree {
   int l, r;
   int same[2], lmax[2], rmax[2], tmax[2]; // 下标0维护分配给屌丝的时间，下标1维护分配给女神的时间
} tr[N << 2];

void pushup(int u, int type) {
   auto &root = tr[u], &left = tr[u << 1], &right = tr[u << 1 | 1];
   root.lmax[type] = left.lmax[type];
   if (left.lmax[type] == left.r - left.l + 1) root.lmax[type] += right.lmax[type];
   root.rmax[type] = right.rmax[type];
   if (right.rmax[type] == right.r - right.l + 1) root.rmax[type] += left.rmax[type];
   root.tmax[type] = max(max(left.tmax[type], right.tmax[type]), left.rmax[type] + right.lmax[type]);
}

void build(int u, int l, int r) {
   tr[u] = {l, r, -1, -1, 1, 1, 1, 1, 1, 1};
   if (l == r) return;
   int mid = l + r >> 1;
   build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
   pushup(u, 0), pushup(u, 1);
}

void pushdown(int u, int type) {
   auto &root = tr[u], &left = tr[u << 1], &right = tr[u << 1 | 1];
   if (root.same[type] != -1) {
      left.same[type] = right.same[type] = root.same[type];
      left.lmax[type] = left.rmax[type] = left.tmax[type] = root.same[type] * (left.r - left.l + 1);
      right.lmax[type] = right.rmax[type] = right.tmax[type] = root.same[type] * (right.r - right.l + 1);
      root.same[type] = -1;
   }
}

void modify(int u, int l, int r, int x, int type) {
   if (tr[u].l >= l && tr[u].r <= r) {
      tr[u].same[type] = x;
      tr[u].lmax[type] = tr[u].rmax[type] = tr[u].tmax[type] = x * (tr[u].r - tr[u].l + 1);
```

```
    } else {
        pushdown(u, type);
        int mid = tr[u].l + tr[u].r >> 1;
        if (l <= mid) modify(u << 1, l, r, x, type);
        if (r > mid) modify(u << 1 | 1, l, r, x, type);
        pushup(u, type);
    }
}

// 找到第一段长度为x的连续空闲区间的左端点
int query(int u, int x, int type) {
    if (tr[u].tmax[type] < x) return -1; // 不存在
    pushdown(u, type);
    if (tr[u << 1].tmax[type] >= x) return query(u << 1, x, type);
    if (tr[u << 1].rmax[type] + tr[u << 1 | 1].lmax[type] >= x) return
        tr[u << 1].r - tr[u << 1].rmax[type] + 1;
    return query(u << 1 | 1, x, type);
}

int main() {
    for (cin >> T; T--; ) {
        printf("Case %d:\n", ++Case);
        scanf("%d%d", &n, &m);
        build(1, 1, n);
        while (m--) {
            char op[10];
            int x, y;
            scanf("%s", op);
            if (*op == 'N') {
                scanf("%d", &x);
                int st = query(1, x, 0); // 先在屌丝时间轴查询是否存在长度为x的
                    连续空闲(1)区间
                if (st != -1) { // 在屌丝时间轴查到了,同时修改两个时间轴的区间[st
                    ,st+x-1]置为忙碌状态
                    modify(1, st, st + x - 1, 0, 0);
                    modify(1, st, st + x - 1, 0, 1);
                    printf("%d,don't put my gezi\n", st);
                } else { // 屌丝时间轴中没有这样的区间
                    st = query(1, x, 1); // 在女神时间轴中查
                    if (st != -1) { // 在女神时间轴查到,就同时修改两个时间轴的区间
                        [st,st+x-1]置为忙碌状态
                        modify(1, st, st + x - 1, 0, 0);
                        modify(1, st, st + x - 1, 0, 1);
```

```
                    printf("%d,don't put my gezi\n", st);
                } else { // 没有空闲时间
                    puts("wait for me");
                }
            }
        } else if (*op == 'D') {
            scanf("%d", &x);
            int st = query(1, x, 0);
            if (st != -1) { // 在屌丝时间轴查到了，区间修改为忙碌状态
                modify(1, st, st + x - 1, 0, 0);
                printf("%d,let's fly\n", st);
            } else { // 没有空闲时间
                puts("fly with yourself");
            }
        } else { // 由于是三分钟热度，应该是将区间置为空闲状态
            scanf("%d%d", &x, &y);
            modify(1, x, y, 1, 0);
            modify(1, x, y, 1, 1);
            puts("I am the hope of chinese chengxuyuan!!");
        }
    }
  }
  return 0;
}
```

1.2.4 HDU1542（扫描线求矩形面积并）

题意：线段树扫描线求矩形面积并。

分析：注意线段树的每一个叶子结点表示的不是单个点，而是一个区间，其中的标记含义如注释。

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

using namespace std;

const int N = 200010;

int n, Case;
struct Segment {
  double x, y1, y2;
  int k;
  bool operator < (const Segment &W) const {
```

```
        return x < W.x;
    }
} seg[N];
// 线段树的每一个叶子结点（假设下标为i），表示一个区间[y_i, y_{i+1}]
struct Node {
    int l, r, cnt; // cnt表示[l,r]区间被完全覆盖的次数，cnt>0就表示要算上[l,r]这一整段，其表示实际的区间为[ys[l],ys[r+1]]
    double len; // len表示当前线段树的区间[l,r]内，cnt>0（即被覆盖的实际区间）的合并长度之和。
} tr[N << 2]; // 比如 y1, y2, y3 离散化后为k_y1,k_y2,k_y3。其区间[k_y1,k_y2],[k_y1,k_y2],[k_y2,k_y3]是线段树中的3个叶子结点。
vector<double> ys;

int find(double y) {
    return lower_bound(ys.begin(), ys.end(), y) - ys.begin();
}

void pushup(int u) {
    if (tr[u].cnt) {
        tr[u].len = ys[tr[u].r + 1] - ys[tr[u].l];
    } else if (tr[u].l != tr[u].r) {
        tr[u].len = tr[u << 1].len + tr[u << 1 | 1].len;
    } else {
        tr[u].len = 0;
    }
}

void build(int u, int l, int r) {
    tr[u] = {l, r, 0, 0};
    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 k) {
    if (tr[u].l >= l && tr[u].r <= r) {
        tr[u].cnt += k;
        pushup(u);
    } else {
        int mid = tr[u].l + tr[u].r >> 1;
        if (l <= mid) modify(u << 1, l, r, k);
        if (r > mid) modify(u << 1 | 1, l, r, k);
        pushup(u);
```

```
    }
}

int main() {
    while (cin >> n, n) {
        ys.clear();
        for (int i = 0; i < n; i ++ ) {
            double x1, y1, x2, y2;
            scanf("%lf%lf%lf%lf", &x1, &y1, &x2, &y2);
            seg[i * 2] = {x1, y1, y2, 1};
            seg[i * 2 + 1] = {x2, y1, y2, -1};
            ys.push_back(y1), ys.push_back(y2);
        }
        sort(ys.begin(), ys.end());
        ys.erase(unique(ys.begin(), ys.end()), ys.end());
        build(1, 0, ys.size() - 2); // 共ys.size()个y，那么相邻之间就有ys.size()-1个区间，就有ys.size()-1个线段树的叶子节点。
        sort(seg, seg + n * 2);
        double res = 0;
        for (int i = 0; i < n * 2; i++){
            if (i) res += tr[1].len * (seg[i].x - seg[i - 1].x);
            int l = find(seg[i].y1), r = find(seg[i].y2) - 1;
            // 右端点注意要减去1，假设实际区间为[L,R]，那么对应线段树中的区间就是[L,R-1]
            modify(1, l, r, seg[i].k);
        }
        printf("Test case #%d\n", ++Case);
        printf("Total explored area: %.2lf\n\n", res);
    }
    return 0;
}
```

1.2.5 HDU1255（矩形 2 次覆盖面积并）

题意：线段树扫描线求至少被覆盖 2 次的矩形面积并。

分析：与上一个题目类似，这里需要分别维护 len1, len2，其中 len1 含义与上一个题的 len 一样，len2 表示线段树区间内被覆盖至少 2 次的实际区间的合并的长度。只需要求改 pushup 函数，更新 len2 时候需要分情况讨论，如果区间被完全覆盖了至少 2 次，len2 就是区间长度；否则，如果当前是叶子结点，那么此时最多会被完全覆盖 1 次，对 len2 没有贡献；否则，如果不是叶子结点并且恰好被覆盖 1 次，那么想要求该区间内至少被覆

盖 2 次的长度，就需要计算当前结点的左右子结点中被覆盖至少 1 次的长度，如果不是叶子结点并且没有被完全覆盖过，直接用子结点的 len2 之和来更新当前结点的 len2 即可。有点绕，但是并不难理解。

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

using namespace std;

const int N = 2010;

int T, n;
struct Segment {
   double x, y1, y2;
   int k;
   bool operator < (const Segment &W) const {
      return x < W.x;
   }
} seg[N];
struct Node {
   int l, r, cnt; // cnt表示[l,r]区间被完全覆盖的次数
   double len1, len2; // len1表示线段树区间[l,r]中cnt>0的区间合并后的长度,
      len2对应cnt>1
} tr[N << 2];
vector<double> ys;

int find(double y) {
   return lower_bound(ys.begin(), ys.end(), y) - ys.begin();
}

void pushup(int u) {
   // 更新len1
   if (tr[u].cnt > 0) {
      tr[u].len1 = ys[tr[u].r + 1] - ys[tr[u].l];
   } else if (tr[u].l == tr[u].r) {
      tr[u].len1 = 0;
   } else {
      tr[u].len1 = tr[u << 1].len1 + tr[u << 1 | 1].len1;
   }
   // 更新len2
   if (tr[u].cnt > 1) {
      tr[u].len2 = ys[tr[u].r + 1] - ys[tr[u].l];
   } else if (tr[u].l == tr[u].r) {
      tr[u].len2 = 0;
```

```
    } else {
        if (tr[u].cnt == 1) { // 被完全覆盖了1次
            // 如果子区间有恰好被覆盖至少1次的，那么合在一起就是至少覆盖2次的面积了
            tr[u].len2 = tr[u << 1].len1 + tr[u << 1 | 1].len1; // 加上子
                区间至少覆盖1次的面积
        } else { // cnt=0
            tr[u].len2 = tr[u << 1].len2 + tr[u << 1 | 1].len2;
        }
    }
}

void build(int u, int l, int r) {
    tr[u] = {l, r, 0, 0};
    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 k) {
    if (tr[u].l >= l && tr[u].r <= r) {
        tr[u].cnt += k;
        pushup(u);
    } else {
        int mid = tr[u].l + tr[u].r >> 1;
        if (l <= mid) modify(u << 1, l, r, k);
        if (r > mid) modify(u << 1 | 1, l, r, k);
        pushup(u);
    }
}

int main() {
    for (cin >> T; T--; ) {
        cin >> n;
        ys.clear();
        for (int i = 0; i < n; i++) {
            double x1, y1, x2, y2;
            scanf("%lf%lf%lf%lf", &x1, &y1, &x2, &y2);
            seg[i * 2] = {x1, y1, y2, 1};
            seg[i * 2 + 1] = {x2, y1, y2, -1};
            ys.push_back(y1), ys.push_back(y2);
        }
        sort(ys.begin(), ys.end());
        ys.erase(unique(ys.begin(), ys.end()), ys.end());
```

```cpp
        build(1, 0, ys.size() - 2);
        sort(seg, seg + n * 2);
        double res = 0;
        for (int i = 0; i < n * 2; i++){
            if (i) res += tr[1].len2 * (seg[i].x - seg[i - 1].x);
            int l = find(seg[i].y1), r = find(seg[i].y2) - 1;
            modify(1, l, r, seg[i].k);
        }
        printf("%.2lf\n", res);
    }
    return 0;
}
```

1.3 分块

```cpp
/*
 * belong[i]表示下标i所属于的块编号
 * B表示每一块的大小,sz表示一共有多少块
 * L[i],R[i]分别表示块i的左闭边界和右闭边界
*/
int n, B, sz;
int belong[N], L[N], R[N];

void build() {
    B = sqrt(n), sz = (n - 1) / B + 1;
    for (int i = 1; i <= n; i++) belong[i] = (i - 1) / B + 1;
    for (int i = 1; i <= sz; i++) L[i] = (i - 1) * B + 1, R[i] = L[i] +
         B - 1;
    R[sz] = n;
}
```

1.4 ST 表

```cpp
// ST表可维护区间最值/区间gcd
int n, a[N];
int f[N][M]; // f[i][j]表示区间[i, i+2^j-1]区间的最大值
int Log2[N];

void ST_pre() {
    Log2[2] = 1;
    for (int i = 3; i < N; i++) Log2[i] = Log2[i >> 1] + 1;
```

```
    for (int i = 1; i <= n; i++) f[i][0] = a[i];
    for (int j = 1; j < M; j++) {
        for (int i = 1; i + (1 << j) - 1 <= n; i++)
            f[i][j] = max(f[i][j- 1], f[i + (1 << j - 1)][j - 1]);
    }
}

int query(int l, int r) {
    int k = Log2[r - l + 1];
    return max(f[l][k], f[r - (1 << k) + 1][k]);
}
```

1.5 Splay

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

using namespace std;

const int N = 100010, INF = 1e9;

int n, m;
int w[N];
struct Splay {
    int ch[2], fa, sz, val, flag; // flag=1为逆序标记
    void init(int _val, int _fa) {
        val = _val, fa = _fa;
        sz = 1, flag = 0;
        // ch[0] = ch[1] = 0; // 多测
    }
} tr[N];
int root;
int stk[N], top;

int get(int u) {
    return tr[tr[u].fa].ch[1] == u;
}

void pushup(int u) {
    tr[u].sz = tr[tr[u].ch[0]].sz + tr[tr[u].ch[1]].sz + 1;
}

void pushdown(int u) {
```

```
    if (tr[u].flag) {
        swap(tr[u].ch[0], tr[u].ch[1]);
        tr[tr[u].ch[0]].flag ^= 1;
        tr[tr[u].ch[1]].flag ^= 1;
        tr[u].flag = 0;
    }
}

void rotate(int u) {
    int fa = tr[u].fa, gfa = tr[fa].fa;
    int k = get(u);
    tr[gfa].ch[get(fa)] = u, tr[u].fa = gfa;
    tr[fa].ch[k] = tr[u].ch[k ^ 1], tr[tr[u].ch[k ^ 1]].fa = fa;
    tr[u].ch[k ^ 1] = fa, tr[fa].fa = u;
    pushup(fa), pushup(u);
}

// 把u转到v的下面
void splay(int u, int v) {
    while (tr[u].fa != v) {
        int fa = tr[u].fa, gfa = tr[fa].fa;
        if (gfa != v) {
            if (get(u) ^ get(fa)) rotate(u);
            else rotate(fa);
        }
        rotate(u);
    }
    if (!v) root = u; // v=0，说明u被转到根位置
}

// 将w[l~r]建成一棵splay，其根节点的父节点为fa
int build(int l, int r, int fa) {
    int u = stk[top--];
    int mid = l + r >> 1;
    tr[u].init(w[mid], fa);
    if (l < mid) tr[u].ch[0] = build(l, mid - 1, u);
    if (r > mid) tr[u].ch[1] = build(mid + 1, r, u);
    pushup(u);
    return u;
}

int kth(int k) {
    int u = root;
```

```
    while (u) {
        pushdown(u);
        if (tr[tr[u].ch[0]].sz >= k) u = tr[u].ch[0];
        else if (tr[tr[u].ch[0]].sz + 1 == k) return u;
        else k -= tr[tr[u].ch[0]].sz + 1, u = tr[u].ch[1];
    }
    return -1;
}

// // 结点u内存回收
// void dfs(int u) {
// stk[++top] = u;
// if (tr[u].ch[0]) dfs(tr[u].ch[0]);
// if (tr[u].ch[1]) dfs(tr[u].ch[1]);
// }

void print(int u) { // 中序遍历输出序列
    pushdown(u);
    if (tr[u].ch[0]) print(tr[u].ch[0]);
    if (tr[u].val != INF && tr[u].val != -INF) printf("%d ", tr[u].val)
        ;
    if (tr[u].ch[1]) print(tr[u].ch[1]);
}

int main() {
    for (int i = 1; i < N; i++) stk[++top] = i; // 内存复用
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i++) w[i] = i; // w[1~n]为需要维护的序列，一般
        为题目输入
    w[0] = -INF, w[n + 1] = INF; // 哨兵
    root = build(0, n + 1, 0);
    while (m--) {
        int l, r; scanf("%d%d", &l, &r);
        // 找到[l+1,r+1]的前后
        l = kth(l), r = kth(r + 2);
        splay(l, 0), splay(r, l); // 将l转到根，r转到l下面，此时r的左孩子就表
            示需要维护的区间
        tr[tr[r].ch[0]].flag ^= 1;
    }
    print(root);
    return 0;
}
```

1.6 Treap

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

using namespace std;

const int N = 100010, INF = 1e8;

int n;
struct Node {
   int l, r;
   int key, val;
   int cnt, sz;
} tr[N];
int root, idx;

int new_node(int key) {
   tr[++idx].key = key;
   tr[idx].val = rand();
   tr[idx].cnt = tr[idx].sz = 1;
   return idx;
}

void pushup(int p) {
   tr[p].sz = tr[tr[p].l].sz + tr[tr[p].r].sz + tr[p].cnt;
}

void build() {
   new_node(-INF), new_node(INF);
   root = 1, tr[1].r = 2;
   pushup(root);
}

void RR(int &p) {
   int q = tr[p].l;
   tr[p].l = tr[q].r, tr[q].r = p, p = q;
   pushup(tr[p].r), pushup(p);
}

void LR(int &p) {
   int q = tr[p].r;
   tr[p].r = tr[q].l, tr[q].l = p, p = q;
   pushup(tr[p].l), pushup(p);
```

```
}

void insert(int &p, int key) {
   if (!p) p = new_node(key);
   else if (tr[p].key == key) tr[p].cnt++;
   else if (tr[p].key > key) {
      insert(tr[p].l, key);
      if (tr[tr[p].l].val > tr[p].val) RR(p);
   } else {
      insert(tr[p].r, key);
      if (tr[tr[p].r].val > tr[p].val) LR(p);
   }
   pushup(p);
}

void remove(int &p, int key) {
   if (!p) return;
   if (tr[p].key == key) {
      if (tr[p].cnt > 1) tr[p].cnt--;
      else if (tr[p].l || tr[p].r) {
         if (!tr[p].r || tr[tr[p].l].val > tr[tr[p].r].val) {
            RR(p);
            remove(tr[p].r, key);
         } else {
            LR(p);
            remove(tr[p].l, key);
         }
      } else {
         p = 0;
      }
   } else if (tr[p].key > key) {
      remove(tr[p].l, key);
   } else {
      remove(tr[p].r, key);
   }
   pushup(p);
}

int get_rank_by_key(int p, int key) {
   if (!p) return 0; // never occur in this problem
   if (tr[p].key == key) return tr[tr[p].l].sz + 1;
   if (tr[p].key > key) return get_rank_by_key(tr[p].l, key);
   return tr[tr[p].l].sz + tr[p].cnt + get_rank_by_key(tr[p].r, key);
```

```
}

int get_key_by_rank(int p, int rank) {
   if (!p) return INF; // never occur in this problem
   if (tr[tr[p].l].sz >= rank) return get_key_by_rank(tr[p].l, rank);
   if (tr[tr[p].l].sz + tr[p].cnt >= rank) return tr[p].key;
   return get_key_by_rank(tr[p].r, rank - tr[tr[p].l].sz - tr[p].cnt);
}

// find max that smaller than key
int get_prev(int p, int key) {
   if (!p) return -INF;
   if (tr[p].key >= key) return get_prev(tr[p].l, key);
   return max(tr[p].key, get_prev(tr[p].r, key));
}

// find min that bigger than key
int get_next(int p, int key) {
   if (!p) return INF;
   if (tr[p].key <= key) return get_next(tr[p].r, key);
   return min(tr[p].key, get_next(tr[p].l, key));
}

int main() {
   build();
   cin >> n;
   while (n--) {
      int op, x;
      cin >> op >> x;
      if (op == 1) insert(root, x);
      else if (op == 2) remove(root, x);
      else if (op == 3) printf("%d\n", get_rank_by_key(root, x) - 1);
         // 注意之前插入了-INF的哨兵
      else if (op == 4) printf("%d\n", get_key_by_rank(root, x + 1));
      else if (op == 5) printf("%d\n", get_prev(root, x));
      else printf("%d\n", get_next(root, x));
   }
   return 0;
}
```

1.7 树链剖分

1.7.1 点权

题意：给定一棵树，树中包含 n 个结点，有点权。初始时，1 号节点为树的根节点。现在要对该树进行 m 次操作，操作分为以下 4 种类型：

1. u v k，修改路径上节点权值，将节点 u 和节点 v 之间路径上的所有节点（包括这两个节点）的权值增加 k。

2. u k，修改子树上节点权值，将以节点 u 为根的子树上的所有节点的权值增加 k。

3. u v，询问路径，询问节点 u 和节点 v 之间路径上的所有节点（包括这两个节点）的权值和。

4. u，询问子树，询问以节点 u 为根的子树上的所有节点的权值和。

分析：树剖后以 DFS 序建线段树，子树 u 对应线段树上的区间 [dfn[u], dfn[u]+sz[u]-1]，u v 之间路径的区间构成详见代码。

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

using namespace std;

typedef long long LL;

const int N = 100010, M = N << 1;

int n, m;
int a[N], na[N]; // na表示新编号
int h[N], e[M], ne[M], idx;
struct Tree {
   int l, r;
   LL sum, add;
} tr[N << 2];
int dfn[N], ts; // dfn表示dfs序（优先遍历重儿子）
int dep[N], sz[N], top[N], fa[N], son[N];
// dep[i]表示i的深度（根节点的深度为1）,sz[i]表示以i为根的子树的大小
// top[i]表示i所在重链的顶点，fa[i]表示i的父结点，son[i]表示子树i的重儿子

void add(int a, int b) {
   e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

void dfs1(int u, int father, int depth) {
   dep[u] = depth, fa[u] = father, 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;
    }
}

// 点u所属的重链的顶点为t
void dfs2(int u, int t) {
    dfn[u] = ++ts, na[ts] = a[u], top[u] = t;
    if (!son[u]) return; // u为叶结点
    dfs2(son[u], t); // 重儿子
    // 处理轻儿子
    for (int i = h[u]; ~i; i = ne[i]) {
        int j = e[i];
        if (j == fa[u] || j == son[u]) continue;
        dfs2(j, j); // 轻儿子所处重链的顶点就是自己
    }
}

void pushup(int u) {
    tr[u].sum = tr[u << 1].sum + tr[u << 1 | 1].sum;
}

void build(int u, int l, int r) {
    if (l == r) {
        tr[u] = {l, r, na[r], 0};
    } 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 pushdown(int u) {
    auto &root = tr[u], &left = tr[u << 1], &right = tr[u << 1 | 1];
    if (root.add) {
        left.add += root.add, left.sum += root.add * (left.r - left.l +
            1);
        right.add += root.add, right.sum += root.add * (right.r - right.
```

```
            l + 1);
        root.add = 0;
    }
}

// 将[l,r]区间加上k
void update(int u, int l, int r, int k) {
    if (tr[u].l >= l && tr[u].r <= r) {
        tr[u].add += k;
        tr[u].sum += k * (tr[u].r - tr[u].l + 1);
        return;
    }
    pushdown(u);
    int mid = tr[u].l + tr[u].r >> 1;
    if (l <= mid) update(u << 1, l, r, k);
    if (r > mid) update(u << 1 | 1, l, r, k);
    pushup(u);
}

// 求[l,r]区间和
LL query(int u, int l, int r) {
    if (tr[u].l >= l && tr[u].r <= r) {
        return tr[u].sum;
    }
    pushdown(u); // 下传add标记
    LL res = 0;
    int mid = tr[u].l + tr[u].r >> 1;
    if (l <= mid) res += query(u << 1, l, r);
    if (r > mid) res += query(u << 1 | 1, l, r);
    return res;
}

// 将树上u->v的路径全部加上k
void update_path(int u, int v, int k) {
    while (top[u] != top[v]) { // u,v不在同一个重链上
        if (dep[top[u]] < dep[top[v]]) swap(u, v);
        // 加上u所在的重链和和
        // 其区间为[dfn[top[u]], dfn[u]]
        update(1, dfn[top[u]], dfn[u], k);
        u = fa[top[u]];
    }
    if (dep[u] < dep[v]) swap(u, v);
    // 加上u-v之间路径的和
```

```
    update(1, dfn[v], dfn[u], k);
}

// 求树上u-v之间的路径和
LL query_path(int u, int v) {
    LL res = 0;
    while (top[u] != top[v]) { // u,v不在同一个重链上
        if (dep[top[u]] < dep[top[v]]) swap(u, v);
        // 加上u所在的重链和和
        // 其区间为[dfn[top[u]], dfn[u]]
        res += query(1, dfn[top[u]], dfn[u]);
        u = fa[top[u]];
    }
    if (dep[u] < dep[v]) swap(u, v);
    // 加上u-v之间路径的和
    res += query(1, dfn[v], dfn[u]);
    return res;
}

// 将树上以u为根的子树全部加上k
void update_tree(int u, int k) {
    // 对应区间 [dfn[u], dfn[u]+sz[u]-1]
    update(1, dfn[u], dfn[u] + sz[u] - 1, k);
}

// 求树上以u为根的子树的和
LL query_tree(int u) {
    // 对应区间 [dfn[u], dfn[u]+sz[u]-1]
    return query(1, dfn[u], dfn[u] + sz[u] - 1);
}

int main() {
    memset(h, -1, sizeof h);
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) scanf("%d", &a[i]);
    for (int i = 0; i < n - 1; i++) {
        int a, b;
        scanf("%d%d", &a, &b);
        add(a, b), add(b, a);
    }
    dfs1(1, -1, 1); // 预处理dep,fa,sz
    dfs2(1, 1); // 求dfs序dfn,top
    build(1, 1, n); // 建立线段树
```

```
    for (scanf("%d", &m); m--; ) {
        int type, u, v, k;
        scanf("%d", &type);
        if (type == 1) {
            scanf("%d%d%d", &u, &v, &k);
            update_path(u, v, k);
        } else if (type == 2) {
            scanf("%d%d", &u, &k);
            update_tree(u, k);
        } else if (type == 3) {
            scanf("%d%d", &u, &v);
            printf("%lld\n", query_path(u, v));
        } else {
            scanf("%d", &u);
            printf("%lld\n", query_tree(u));
        }
    }
    return 0;
}
```

1.7.2 边权

题意：有 n 个点的树，有边权。两种操作：1. 0 a b，表示更新第 a 条边权为 b，2. 1 a b 表示询问 a 到 b 的路径上边权之和。

分析：树链剖分后转化为序列问题用线段树维护，由于线段树没法维护边权，因此在原来树中需要将边权下放到点权，由于每个结点（除根结点）只有 1 个父亲，因此可以将父亲 -> 儿子的边权下放到儿子的点权上。求树上两点 u, v 距离时，需要注意不能把 LCA(u, v) 的点权计算进去。

```
/* FZU2082 */
#include <iostream>
#include <algorithm>
#include <cstdio>
#include <cstring>

using namespace std;

typedef pair<int, int> PII;
typedef long long LL;

const int N = 50010, M = N << 1;
```

```

int n, m;
int h[N], e[M], w[M], ne[M], idx;
struct Tree {
   int l, r;
   LL sum;
} tr[N << 2];
int dfn[N], ts; // dfn表示dfs序（优先遍历重儿子）
int dep[N], sz[N], top[N], fa[N], son[N];
// dep[i]表示i的深度（根节点的深度为1）,sz[i]表示以i为根的子树的大小
// top[i]表示i所在重链的顶点，fa[i]表示i的父结点，son[i]表示子树i的重儿子

void add(int a, int b, int c) {
   e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
}

void dfs1(int u, int father, int depth) {
   dep[u] = depth, fa[u] = father, 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;
   }
}

// 点u所属的重链的顶点为t
void dfs2(int u, int t) {
   dfn[u] = ++ts, top[u] = t;
   if (!son[u]) return; // u为叶结点
   dfs2(son[u], t); // 处理重儿子
   // 处理轻儿子
   for (int i = h[u]; ~i; i = ne[i]) {
      int j = e[i];
      if (j == fa[u] || j == son[u]) continue;
      dfs2(j, j); // 轻儿子所处重链的顶点就是自己
   }
}

void pushup(int u) {
   tr[u].sum = tr[u << 1].sum + tr[u << 1 | 1].sum;
}
```

```

void build(int u, int l, int r) {
    if (l == r) {
        tr[u].l = l, tr[u].r = r, tr[u].sum = 0;
    } else {
        tr[u].l = l, tr[u].r = r;
        int mid = l + r >> 1;
        build(u << 1, l, mid), build(u << 1 | 1, mid + 1 , r);
    }
}

// 将x位置修改成y
void update(int u, int x, int y) {
    if (tr[u].l == x && tr[u].r == x) {
        tr[u].sum = y;
    } else {
        int mid = tr[u].l + tr[u].r >> 1;
        if (x <= mid) update(u << 1, x, y);
        else update(u << 1 | 1, x, y);
        pushup(u);
    }
}

// 求线段树中[l,r]区间和
LL query(int u, int l, int r) {
    if (tr[u].l >= l && tr[u].r <= r) {
        return tr[u].sum;
    } else {
        LL res = 0;
        int mid = tr[u].l + tr[u].r >> 1;
        if (l <= mid) res += query(u << 1, l, r);
        if (r > mid) res += query(u << 1 | 1, l, r);
        return res;
    }
}

// 求树上u-v之间的路径和
LL query_path(int u, int v) {
    LL res = 0;
    while (top[u] != top[v]) { // u,v不在同一个重链上
        if (dep[top[u]] < dep[top[v]]) swap(u, v);
        // 加上u所在的重链和
        // 其区间为[dfn[top[u]], dfn[u]]
```

```
        res += query(1, dfn[top[u]], dfn[u]);
        u = fa[top[u]];
    }
    if (u == v) return res;
    if (dep[u] < dep[v]) swap(u, v); // 保证u在下面v在上面
    // 加上u-v之间路径的和，v当前在LCA位置，这个点的点权不能算上。
    res += query(1, dfn[son[v]], dfn[u]);
    return res;
}

int main() {
    while (cin >> n >> m) {
        for (int i = 1; i <= n; i++) {
            h[i] = -1, dfn[i] = 0, son[i] = 0;
        }
        idx = ts = 0;
        for (int i = 0; i < n - 1; i++) {
            int a, b, c;
            scanf("%d%d%d", &a, &b, &c);
            add(a, b, c), add(b, a, c);
        }
        dfs1(1, -1, 1); // 预处理dep,fa,sz
        dfs2(1, 1); // 求dfs序dfn,top
        build(1, 1, n);
        for (int i = 0; i < idx; i += 2) {
            int u = e[i], v = e[i ^ 1];
            if (dep[u] > dep[v]) swap(u, v);
            update(1, dfn[v], w[i]);
        }
        while (m--) {
            int type, x, y;
            cin >> type >> x >> y;
            if (type == 0) { // 更新第x条路的过路费为y
                // 第1条 idx=0,1
                // 第2条 idx=2,3
                // 第x条 idx=2x-2,2x-1
                int v = e[2 * x - 2], u = e[2 * x - 1];
                if (dep[u] > dep[v]) swap(u, v);
                // u是父亲, v是儿子，令点v权值更新为y
                update(1, dfn[v], y);
            } else {
                printf("%lld\n", query_path(x, y));
            }
```

```
        }
    }
    return 0;
}
```

1.8 莫队

1.8.1 离线询问排序方式

一般莫队排序方式：以 belong[l] 为第一关键字，r 为第二关键字升序排序。

```
struct Query {
    int id, l, r;
    bool operator < (const Query &W) const {
        if (belong[l] != belong[W.l]) return belong[l] < belong[W.l];
        return r < W.r;
    }
} q[M];
// 奇偶优化
struct Query {
    int id, l, r;
    bool operator < (const Query &W) const {
        if (belong[l] != belong[W.l]) return belong[l] < belong[W.l];
        return belong[l] & 1 ? r < W.r : r > W.r;
    }
} q[N];
```

带修莫队排序方式：以 belong[l] 为第一关键字，belong[r] 为第二关键字，ts 为第三关键字升序排序。

```
struct Query {
    int id, l, r, ts; // id表示当前询问的编号，ts表示当前询问处于第ts次操作后，第ts+1操作前
    bool operator < (const Query &W) const {
        if (belong[l] != belong[W.l]) return belong[l] < belong[W.l];
        if (belong[r] != belong[W.r]) return belong[r] < belong[W.r];
        return ts < W.ts;
    }
```

```
} q[N];
```

1.8.2 洛谷 P4396（基础莫队）

对于每个询问区间 [l, r] 需要求在该区间内值域在 [a, b] 上的数的个数以及不同的数的个数。

可以考虑用两个树状数组来维护当前区间中出现的数字的个数，和不同数字的个数，然后差分一下就得到某个值域中出现的次数了。但这样插入和查询都是 $O(\log n)$ 。加上莫队总复杂度就达到了 $O(n\sqrt{n}\log n)$，这是无法接受的。

考虑值域分块，然后维护每一块的和。插入就是 $O(1)$ 查询为 $O(\sqrt{n})$，不会影响总复杂度。

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

using namespace std;

const int N = 100010;

int n, m;
int a[N], cnt[N], s[N][2], sum[N][2];
int belong[N], L[N], R[N], B, sz;
struct Query {
   int id, l, r, a, b;
   bool operator < (const Query &W) const {
      if (belong[l] != belong[W.l]) return belong[l] < belong[W.l];
      return r < W.r;
   }
} q[N];
int ans[N][2];

void build() {
   B = sqrt(n), sz = (n - 1) / B + 1;
   for (int i = 1; i <= n; i++) belong[i] = (i - 1) / B + 1;
   for (int i = 1; i <= sz; i++) L[i] = (i - 1) * B + 1, R[i] = L[i] +
       B - 1;
   R[sz] = n;
}

void add(int x) {
```

```
    x = a[x];
    cnt[x]++;
    s[x][0]++, sum[belong[x]][0]++;
    if (cnt[x] == 1) s[x][1]++, sum[belong[x]][1]++;
}

void del(int x) {
    x = a[x];
    cnt[x]--;
    s[x][0]--, sum[belong[x]][0]--;
    if (cnt[x] == 0) s[x][1]--, sum[belong[x]][1]--;
}

int ask(int l, int r, int type) {
    if (belong[l] == belong[r]) {
        int res = 0;
        for (int i = l; i <= r; i++) res += s[i][type];
        return res;
    }
    int res = 0;
    for (int i = l; i <= R[belong[l]]; i++) res += s[i][type];
    for (int i = belong[l] + 1; i < belong[r]; i++) res += sum[i][type
        ];
    for (int i = L[belong[r]]; i <= r; i++) res += s[i][type];
    return res;
}

int main() {
    scanf("%d%d", &n, &m); build();
    for (int i = 1; i <= n; i++) scanf("%d", &a[i]);
    for (int i = 0; i < m; i++) {
        int l, r, a, b;
        scanf("%d%d%d%d", &l, &r, &a, &b);
        q[i] = {i, l, r, a, b};
    }
    sort(q, q + m);
    for (int i = 0, l = 1, r = 0; i < m; i++) {
        while (r < q[i].r) add(++r);
        while (l > q[i].l) add(--l);
        while (r > q[i].r) del(r--);
        while (l < q[i].l) del(l++);
        ans[q[i].id][0] = ask(q[i].a, q[i].b, 0);
        ans[q[i].id][1] = ask(q[i].a, q[i].b, 1);
```

```
    }
    for (int i = 0; i < m; i++) printf("%d %d\n", ans[i][0], ans[i][1]);
    return 0;
}
```

1.8.3 AcWing2521（带修莫队）

题意：两种操作 1. 询问区间不同颜色数量 2. 单点修改颜色

分析：时间轴上的 ts 指针移动，需要一点技巧。如果当前莫队区间的时间戳 ts 比查询区间的时间戳 q[i].ts 小的话，需要将 ts+1~q[i].ts 时刻的修改造成的影响累加到答案上，这点并不难做，反之如果 ts > q[i].ts，就需要撤销 q[i].ts+1~ts 时刻的修改对答案的影响，比较难处理。因此，我们可以沿时间戳增量修改的时候，将已经用到的修改操作中的颜色，与被修改位置的颜色交换，那么下一次需要撤销这次修改时，就等价于再对这个位置进行一次修改操作，而由于之前修改和被修改的颜色进行了交换，因此直接执行这次修改操作恰好是撤销的效果。

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

using namespace std;

const int N = 10010, M = 1000010;

int n, m, B;
int color[N], belong[N];
int qcnt, pcnt; // qcnt为询问编号，pcnt为操作的编号
struct Query {
    int id, l, r, ts; // id表示当前询问的编号，ts表示当前询问处于第ts次操作后，第ts+1操作前
    bool operator < (const Query &W) const {
        if (belong[l] != belong[W.l]) return belong[l] < belong[W.l];
        if (belong[r] != belong[W.r]) return belong[r] < belong[W.r];
        return ts < W.ts;
    }
} q[N];
struct Modify {
    int x, c; // 将下标为x的位置的颜色修改成c
} p[N];
int cnt[M], ans[N];

void build() {
```

```
    B = pow(n, 2.0 / 3);
    for (int i = 1; i <= n; i++) belong[i] = (i - 1) / B + 1;
}

void add(int x, int &res) {
    if (++cnt[x] == 1) res++;
}

void del(int x, int &res) {
    if (--cnt[x] == 0) res--;
}

// 将编号为ts的操作的影响作用到编号为i的询问
void modify(int ts, int i, int &res) {
    // 如果第ts次操作的位置在第i次询问的区间内部，就需要删除原来的颜色，再加上新颜色，以对答案造成影响
    if (p[ts].x >= q[i].l && p[ts].x <= q[i].r) {
        del(color[p[ts].x], res);
        add(p[ts].c, res);
    }
    // 上面只是修改cnt和res，实际的颜色修改，技巧，交换原来的颜色，和第ts次操作的颜色，这样下次需要撤销这次操作，相当于执行这次颜色被交换过的新操作。
    swap(color[p[ts].x], p[ts].c);
}

int main() {
    scanf("%d%d", &n, &m);
    build();
    for (int i = 1; i <= n; i++) scanf("%d", &color[i]);
    for (int i = 0; i < m; i++) {
        char op[2];
        int l, r;
        scanf("%s%d%d", op, &l, &r);
        if (*op == 'Q') {
            ++qcnt;
            q[qcnt] = {qcnt, l, r, pcnt}; // 询问
        } else {
            p[++pcnt] = {l, r}; // 修改
        }
    }
    // 对于询问排序
    sort(q + 1, q + qcnt + 1);
    // 枚举每一个询问，初始下标区间为[l,r]为[1, 0]，为空，不同颜色个数res=0，处
```

```
    在第ts=0个操作之后，第ts+1=1个操作之前
  for (int i = 1, l = 1, r = 0, res = 0, ts = 0; i <= qcnt; i++) {
    // 将[l,r,ts]移动到[q[i].l, q[i].r, q[i].ts]
    while (r < q[i].r) add(color[++r], res);
    while (l > q[i].l) add(color[--l], res);
    while (r > q[i].r) del(color[r--], res);
    while (l < q[i].l) del(color[l++], res);
    while (ts < q[i].ts) modify(++ts, i, res); // 需要将ts+1~q[i].ts
        的操作造成的影响累加到答案上
    while (ts > q[i].ts) modify(ts--, i, res); // 需要消除q[i].ts+1~
        ts的操作对答案的影响
    ans[q[i].id] = res;
  }
  for (int i = 1; i <= qcnt; i++) printf("%d\n", ans[i]);
  return 0;
}
```

1.8.4 AcWing2523（回滚莫队）

回滚莫队，一般用在当区间维护的答案只具有“可加性”或者只具有“可减性”时，这里只讨论，只具有“可加性”的情况。对于此类情况，回滚莫队能将删除操作 del 全部转化为插入操作 add 。

依次处理询问，我们对询问进行分段处理，把左端点处于同一块的询问放在一起处理。

对于这些左端点处于同一块的询问来说，它们的右端点递增，我们再细分为两种情况。

1. 左右端点在同一块内：直接暴力做就行了，l, r 指针移动是 O(n) 的。

2. 左右端点跨块，分为两部分：左端点所属块的部分，和右边的部分，初始化区间 r = R[belong[q[i].l]], l = r + 1，右端点向右一直 add，左端点向左 add，每次做完左端点需要归位并消除影响。

取块大小为 $B = \sqrt{n}$ 的话，总复杂度为 $O(n\sqrt{n} + m\sqrt{n})$ 。

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

using namespace std;

typedef long long LL;

const int N = 100010;

int n, m, B, sz;
int belong[N], L[N], R[N], a[N];
```

```
struct Query {
   int id, l, r;
   bool operator < (const Query &W) const {
      if (belong[l] != belong[W.l]) return belong[l] < belong[W.l];
      return r < W.r;
   }
} q[N];
vector<int> alls;
int cnt[N];
LL ans[N];

void build() {
   B = sqrt(n), sz = (n - 1) / B + 1;
   for (int i = 1; i <= n; i++) belong[i] = (i - 1) / B + 1;
   for (int i = 1; i <= sz; i++) L[i] = (i - 1) * B + 1, R[i] = L[i] +
       B - 1;
   R[sz] = n;
}

int find(int x) {
   return lower_bound(alls.begin(), alls.end(), x) - alls.begin();
}

void add(int x, LL &res) {
   cnt[x]++;
   res = max(res, (LL)cnt[x] * alls[x]);
}

int main() {
   scanf("%d%d", &n, &m); build();
   for (int i = 1; i <= n; i++) {
      scanf("%d", &a[i]);
      alls.push_back(a[i]);
   }
   sort(alls.begin(), alls.end());
   alls.erase(unique(alls.begin(), alls.end()), alls.end());
   for (int i = 1; i <= n; i++) a[i] = find(a[i]);
   for (int i = 0, l, r; i < m; i++) {
      scanf("%d%d", &l, &r);
      q[i] = {i, l, r};
   }
   sort(q, q + m);
   for (int i = 0; i < m; ) {
```

```cpp
        int j = i;
        while (j + 1 < m && belong[q[i].l] == belong[q[j + 1].l]) j++;
        // 此时[i,j]区间内的所有询问的左端点属于同一块，右端点递增
        // 暴力求块内（左右端点在同一块内的询问）
        while (i <= j && belong[q[i].l] == belong[q[i].r]) {
            LL res = 0;
            for (int k = q[i].l; k <= q[i].r; k++) add(a[k], res);
            ans[q[i].id] = res;
            // 清空cnt
            for (int k = q[i].l; k <= q[i].r; k++) cnt[a[k]]--;
            i++;
        }
        // 求跨块，分为两部分：左边第一个块内的部分和它右边块的部分
        LL res = 0;
        int block_id = belong[q[i].l];
        int r = R[block_id], l = r + 1; // 莫队区间初始化
        // 右端点递增，只存在add操作，左端点先初始化到block_id块的右端点，然后向左使用add操作
        while (i <= j) {
            while (r < q[i].r) add(a[++r], res);
            LL tmp = res; // 备份
            while (l > q[i].l) add(a[--l], res);
            ans[q[i].id] = res;
            // 清空左边部分对于cnt[]的影响，且让l回到初始位置
            while (l <= R[belong[q[i].l]]) cnt[a[l++]] -- ;
            res = tmp;
            i++;
        }
        // 清空cnt，对于每一块只会执行一次，复杂度为n根号n
        memset(cnt, 0, sizeof cnt);
    }
    for (int i = 0; i < m; i++) printf("%lld\n", ans[i]);
    return 0;
}
```

1.8.5 树上莫队

通过树的 DFS 序或者欧拉序将树上问题转化为序列的区间询问问题，再用莫队处理。

2 图论

2.1 最短路

2.1.1 朴素 Dijkstra

```
int g[N][N];
int dist[N];
bool st[N];

int dijkstra() {
   memset(dist, 0x3f, sizeof dist);
   dist[1] = 0;
   for (int i = 0; i < n; i++) {
      int t = -1;
      for (int j = 1; j <= n; j++) {
         if (!st[j] && (t == -1 || dist[j] < dist[t]))
            t = j;
      }
      st[t] = true;
      for (int j = 1; j <= n; j++) {
         dist[j] = min(dist[j], dist[t] + g[t][j]);
      }
   }
   if (dist[n] == 0x3f3f3f3f) return -1;
   return dist[n];
}
```

2.1.2 堆优化 Dijkstra

```
typedef pair<int, int> PII;

int n;
int h[N], w[N], e[N], ne[N], idx;
int dist[N];
bool st[N];

int dijkstra() {
   memset(dist, 0x3f, sizeof dist);
   dist[1] = 0;
   priority_queue<PII, vector<PII>, greater<PII>> heap;
   heap.push({0, 1});
   while (heap.size()) {
```

```cpp
        auto t = q.front();
        q.pop();
        int ver = t.second;
        if (st[ver]) continue;
        st[ver] = true;
        for (int i = h[ver]; ~i; i = ne[i]) {
            int j = e[i];
            if (dist[j] > dist[ver] + w[i]) {
                dist[j] = dist[ver] + w[i];
                heap.push({dist[j], j});
            }
        }
    }
    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

2.1.3 Bellman-Ford

```cpp
// 不超过k条边的最短路
int n, m, k;
int dist[N], backup[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 < k; i++) { // 限制边数则k次循环，否则n次，且不需要 backup[]
        memcpy(backup, dist, sizeof dist);
        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], backup[a] + w);
        }
    }
    if (dist[n] > 0x3f3f3f3f / 2) return -1;
    return dist[n];
}
```

2.1.4 SPFA

```
// 最短路
int n, m;
int h[N], e[N], w[N], ne[N], idx;
int dist[N];
bool st[N];

int spfa() {
   memset(dist, 0x3f, sizeof dist);
   dist[1] = 0;
   queue<int> q;
   q.push(1);
   st[1] = true;
   while (q.size()) {
      int t = q.front();
      q.pop();
      st[t] = false;
      for (int i = h[t]; i != -1; 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;
            }
         }

      }
   }
   if (dist[n] == 0x3f3f3f3f) return -1;
   return dist[n];
}

// 最短路判负环 / 最长路判正环（需要改变不等号方向）
int n, m;
int h[N], e[M], w[M], ne[M], idx;
int dist[N], cnt[N];
bool st[N];

void add(int a, int b, int c) {
   e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
}
```

```
bool spfa() {
    queue<int> q;
    for (int i = 1; i <= n; i++) {
        q.push(i);
        st[i] = true;
    }
    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] = cnt[t] + 1;
                if (cnt[j] >= n) return true;
                if (!st[j]) {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }
    return false;
}
```

2.1.5 Floyd

```
int n, m;
int d[N][N];

void init() {
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            if (i != j) d[i][j] = INF;
}

void floyd() {
    for (int k = 1; k <= n ; k++)
        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]);
```

```
}
```

2.2 最小生成树

2.2.1 Prim

```
int n, m;
int g[N][N];
int dist[N]; // dist[i]: i到集合的距离
bool st[N];

int prim() {
    memset(dist, 0x3f, sizeof dist);
    int res = 0; // 最小生成树的边权之和
    for (int i = 0; i < n; i++) {
        int t = -1; // 找到集合外距离集合最近的点
        for (int j = 1; j <= n; j++) {
            if (!st[j] && (t == -1 || dist[j] < dist[t]))
                t = j;
        }
        if (i && dist[t] == INF) return INF; // 如果不是第一个点，并且这个点到集合已经不连通了，就返回
        if (i) res += dist[t]; // 不是第一个数, 把t加到最小生成树
        st[t] = true; // 把t加入集合
        // 用t->j的距离，更新集合外的j到集合的距离
        for (int j = 1; j <= n; j++) {
            dist[j] = min(dist[j], g[t][j]);
        }
    }
    return res;
}
```

2.2.2 Kruskal

```
int n, m;
int p[N]; // 并查集的父节点数组

struct Edge {
    int a, b, w;
    bool operator< (const Edge &W) const {
        return w < W.w;
    }
```

```
} edges[M];

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

int kruskal() {
   sort(edges, edges + m);
   for (int i = 1; i <= n; i++ ) p[i] = i; // 初始化并查集
   int res = 0, cnt = 0; // res:最小生成树边权之和, cnt:当前加入的边数
   for (int i = 0; i < m; i++ ) {
      int a = edges[i].a, b = edges[i].b, w = edges[i].w;
      a = find(a), b = find(b);
      if (a != b) { // 不连通, 就把这条边加到生成树里
         p[a] = b;
         res += w;
         cnt ++ ;
      }
   }
   if (cnt < n - 1) return INF;
   return res;
}
```

2.3 严格次小生成树

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

using namespace std;

typedef long long LL;

const int N = 100010, M = 300010;

int n, m;
int h[N], e[N * 2], w[N * 2], ne[N * 2], idx;
struct Edge {
   int a, b, w;
   bool f;
   bool operator < (const Edge &W) const {
      return w < W.w;
   }
```

```cpp
} edge[M];
int p[N], q[N];
int d1[N][17], d2[N][17];
int fa[N][17], depth[N];
// fa[i][j]:从i开始向上跳2^j步所能到的点
// d1[i][j]:从i开始向上跳2^j步所经过路径上的最大边
// d2[i][j]:从i开始向上跳2^j步所经过路径上的次大边

void add(int a, int b, int c) {
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
}

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

void update(int d1, int d2, int &v1, int &v2) {
    if (d1 > v1) v2 = max(v1, d2), v1 = d1;
    else if (d1 == v1) v2 = max(v2, d2);
    else v2 = max(v2, d1);
}

void bfs(int root) {
    memset(depth, -1, sizeof depth);
    depth[0] = 0, depth[root] = 1;
    int hh = 0, tt = 0;
    q[0] = root;
    while (hh <= tt) {
        int t = q[hh++];
        for (int i = h[t]; ~i; i = ne[i]) {
            int j = e[i];
            if (depth[j] == -1) {
                depth[j] = depth[t] + 1;
                q[++tt ] = j;
                fa[j][0] = t;
                d1[j][0] = w[i], d2[j][0] = INT_MIN;
                for (int k = 1; k <= 16; k++) {
                    fa[j][k] = fa[fa[j][k - 1]][k - 1];
                    int mid = fa[j][k - 1];
                    update(d1[j][k - 1], d2[j][k - 1], d1[j][k], d2[j][k]);
                    update(d1[mid][k - 1], d2[mid][k - 1], d1[j][k], d2[j][
                        k]);
```

```
                }
            }
        }
    }
}

LL kruskal() {
    for (int i = 1; i <= n; i++) p[i] = i;
    sort(edge, edge + m);
    LL res = 0;
    for (int i = 0; i < m; i++) {
        int a = edge[i].a, b = edge[i].b, w = edge[i].w;
        int pa = find(a), pb = find(b);
        if (pa != pb) {
            add(a, b, w), add(b, a, w);
            edge[i].f = true;
            p[pa] = pb;
            res += w;
        }
    }
    return res;
}

void lca(int a, int b, int &v1, int &v2) {
    if (depth[a] < depth[b]) swap(a, b);
    for (int k = 16; k >= 0; k--) {
        if (depth[fa[a][k]] >= depth[b]) {
            update(d1[a][k], d2[a][k], v1, v2);
            a = fa[a][k];
        }
    }
    if (a == b) return;
    for (int k = 16; k >= 0; k--) {
        if (fa[a][k] != fa[b][k]) {
            update(d1[a][k], d2[a][k], v1, v2);
            update(d1[b][k], d2[b][k], v1, v2);
            a = fa[a][k], b = fa[b][k];
        }
    }
    update(d1[a][0], d2[a][0], v1, v2);
    update(d1[b][0], d2[b][0], v1, v2);
}

```

```
int main() {
   cin >> n >> m;
   memset(h, -1, sizeof h);
   for (int i = 0; i < m; i++) {
      int a, b, w;
      scanf("%d%d%d", &a, &b, &w);
      edge[i] = {a, b, w};
   }
   LL mins = kruskal();
   bfs(1);
   LL res = 1e18;
   for (int i = 0; i < m; i++) {
      if (!edge[i].f) {
         int a = edge[i].a, b = edge[i].b, w = edge[i].w;
         int v1 = INT_MIN, v2 = INT_MIN;
         lca(a, b, v1, v2);
         if (v1 < w) res = min(res, mins + w - v1);
         else if (v2 < w) res = min(res, mins + w - v2);
      }
   }
   printf("%lld\n", res);
   return 0;
}
```

2.4 有向图的强连通分量

2.5 无向图的双连通分量

2.6 最近公共祖先

2.6.1 倍增

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

using namespace std;

const int N = 20010, M = N * 2;

int n, m;
int h[N], e[M], w[M], ne[M], idx;
int fa[N][16], depth[N];
int dist[N], q[N]; // dist[]维护到根距离

```

```
void add(int a, int b, int c) {
   e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
}

void bfs(int root) {
   int hh = 0, tt = 0;
   memset(depth, -1, sizeof depth);
   memset(dist, 0x3f, sizeof dist);
   q[0] = root, dist[root] = 0;
   depth[0] = 0, depth[root] = 1;
   while (hh <= tt) {
      int t = q[hh++];
      for (int i = h[t]; ~i; i = ne[i]) {
         int j = e[i];
         if (depth[j] == -1) {
            depth[j] = depth[t] + 1;
            dist[j] = dist[t] + w[i];
            q[++tt] = j;
            fa[j][0] = t;
            for (int k = 1; k <= 15; k++) {
               fa[j][k] = fa[fa[j][k - 1]][k - 1];
            }
         }
      }
   }
}

int lca(int a, int b) {
   if (depth[a] < depth[b]) swap(a, b);
   for (int k = 15; k >= 0; k--) {
      if (depth[fa[a][k]] >= depth[b])
         a = fa[a][k];
   }
   if (a == b) return a;
   for (int k = 15; k >= 0; k--) {
      if (fa[a][k] != fa[b][k])
         a = fa[a][k], b = fa[b][k];
   }
   return fa[a][0];
}

int main() {
   cin >> n >> m;
```

```
    memset(h, -1, sizeof h);
    for (int i = 0; i < n - 1; i++) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(a, b, c), add(b, a, c);
    }
    bfs(1);
    while (m--) {
        int x, y;
        scanf("%d%d", &x, &y);
        int p = lca(x, y);
        printf("%d\n", dist[x] + dist[y] - 2 * dist[p]);
    }
    return 0;
}
```

2.6.2 Tarjan

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

using namespace std;

typedef pair<int, int> PII;

const int N = 10010, M = 2 * N;

int n, m;
int h[N], e[M], ne[M], w[M], idx;
int p[N], dist[N], st[N], res[M];
vector<PII> query[N];

void add(int a, int b, int c) {
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
}

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

void tarjan(int u, int fa) {
    st[u] = 1; // 标记为正在搜的点
    for (int i = h[u]; ~i; i = ne[i]) {
```

```
            int j = e[i];
            if (j == fa) continue;
            if (!st[j]) { // 还没搜过的点
                dist[j] = dist[u] + w[i];
                tarjan(j, u);
                p[j] = u;
            }
        }
    // 遍历u的所有查询
    for (auto item : query[u]) {
        int v = item.first, id = item.second;
        if (st[v] == 2) { // 已经搜完回溯的点, find(v)为u, v的LCA
            res[id] = dist[u] + dist[v] - 2 * dist[find(v)];
        }
    }
    st[u] = 2; // 标记为已经搜完回溯的点
}

int main() {
    cin >> n >> m;
    memset(h, -1, sizeof h);
    for (int i = 1; i <= n; i++) p[i] = i;
    for (int i = 0; i < n - 1; i++) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(a, b, c), add(b, a, c);
    }
    for (int i = 0; i < m; i++) {
        int a, b;
        scanf("%d%d", &a, &b);
        query[a].push_back({b, i}); // 需要加双向询问
        query[b].push_back({a, i});
    }
    tarjan(1, -1);
    for (int i = 0; i < m; i++) printf("%d\n", res[i]);
    return 0;
}
```

2.6.3 树剖

```
int lca(int u, int v) {
    while (top[u] != top[v]) {
        if (dep[top[u]] < dep[top[v]]) swap(u, v);
```

```cpp
        u = fa[top[u]];
    }
    if (dep[u] < dep[v]) swap(u, v);
    return v;
}
```

2.7 网络流

2.7.1 Dinic

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

using namespace std;

const int N = 100010, M = 200010, INF = 1e9;

int n, m, S, T;
int h[N], e[M], w[M], ne[M], idx;
int q[N];
int d[N], cur[N]; // d[i]表示点i的层次，cur[i]表示i的当前弧

void add(int a, int b, int c) {
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
    e[idx] = a, w[idx] = 0, ne[idx] = h[b], h[b] = idx++;
}

bool bfs() { // 判断残留网络是否存在增广路（即从S到T存在边全大于0的路径）
    int hh = 0, tt = -1;
    memset(d, -1, sizeof d);
    q[++tt] = S, d[S] = 0, cur[S] = h[S];
    while (hh <= tt) {
        int t = q[hh++];
        for (int i = h[t]; ~i; i = ne[i]) {
            int j = e[i];
            if (d[j] == -1 && w[i]) {
                d[j] = d[t] + 1;
                cur[j] = h[j];
                q[++tt] = j;
                if (j == T) return true; // 找到了增广路，此时增广路的流量即f[T]
            }
        }
```

```
    }
    return false; // 残留网络不存在增广路，那么此时原图的可行流流量就是最大流
}

// 从起点到u，流量最大值为limit
int find(int u, int limit) {
    if (u == T) return limit;
    int flow = 0; // u->T的流量
    for (int i = cur[u]; ~i && flow < limit; i = ne[i]) {
        int j = e[i];
        cur[u] = i; // 更新当前弧
        if (d[j] == d[u] + 1 && w[i]) {
            int t = find(j, min(w[i], limit - flow));
            if (!t) d[j] = -1; // 删点
            w[i] -= t, w[i ^ 1] += t, flow += t;
        }
    }
    return flow;
}

int dinic() {
    int max_flow = 0;
    while (bfs()) while (int flow = find(S, INF)) max_flow += flow;
    return max_flow;
}

int main() {
    cin >> n >> m >> S >> T;
    memset(h, -1, sizeof h);
    while (m--) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(a, b, c); // 初始流量为0，对于原图的残留网络，正向边容量为c-0=c，反
            向边容量为0+0=0
    }
    printf("%d\n", dinic());
    return 0;
}
```

2.7.2 EK

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

```

```
using namespace std;

const int N = 1010, M = 20010, INF = 1e9;

int n, m, S, T;
int h[N], e[M], w[M], ne[M], idx;
bool st[N];
int q[N], f[N]; // f[i]表示以i结尾的增广路径的流量（即路径上容量的最小值）
int pre[N]; // pre[i]表示i的前驱边的编号（即指向i的边）

void add(int a, int b, int c) {
   e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
   e[idx] = a, w[idx] = 0, ne[idx] = h[b], h[b] = idx++;
}

bool bfs() { // 判断残留网络是否存在增广路（即从S到T存在边全大于0的路径）
   int hh = 0, tt = -1;
   memset(st, false, sizeof st);
   q[++tt] = S, st[S] = true, f[S] = INF;
   while (hh <= tt) {
      int t = q[hh++];
      for (int i = h[t]; ~i; i = ne[i]) {
         int j = e[i];
         if (!st[j] && w[i]) {
            q[++tt] = j;
            st[j] = true;
            pre[j] = i; // 记录j的前驱边
            f[j] = min(f[t], w[i]);
            if (j == T) return true; // j是终点，即找到了增广路，此时增广路
               的流量即f[T]
         }
      }
   }
   return false; // 残留网络不存在增广路，那么此时原图的可行流流量就是最大流
}

int EK() {
   int flow = 0;
   while (bfs()) { // 如果残留网络存在增广路f'，就将他的流量加到原网络的流量上
      flow += f[T];
      for (int i = T; i != S; i = e[pre[i] ^ 1]) { // 更新残留网络
         w[pre[i]] -= f[T], w[pre[i] ^ 1] += f[T];
      }
```

```
    }
    return flow; // 最大流
}

int main() {
    cin >> n >> m >> S >> T;
    memset(h, -1, sizeof h);
    while (m--) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(a, b, c); // 初始流量为0，对于原图的残留网络，正向边容量为c-0=c，反
            向边容量为0+0=0
    }
    printf("%d\n", EK());
    return 0;
}
```

2.8 二分图

2.8.1 染色法

```
/* O(n+m) */
int n, m;
int h[N], e[M], ne[M] ,idx;
int color[N];

// 返回false染色失败，不是二分图
bool dfs(int u, int c) {
    color[u] = c; // 将u染成c
    for (int i = h[u]; ~i; i = ne[i]) {
        int j = e[i];
        if (!color[j]) { // 如果j未染色
            if (!dfs(j, 3 - c)) return false; // 染色失败
        } else if (color[j] == c) {
            return false; // 与现有颜色矛盾
        }
    }
    return true;
}

bool check() {
    for (int i = 1; i <= n; i++) {
        if (!color[i])
```

```
        if (!dfs(i, 1))
            return false;
    }
    return true;
}
```

2.8.2 匈牙利算法

```
int n1, n2, m;
int h[N], e[M], ne[M], idx;
int match[N]; // match[j]:j号女生匹配的男生
/* 二分图最大匹配 O(nm)实际比较快 */
bool st[N];

void add(int a, int b) {
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

bool find(int x) {
    // 遍历x号男生看上的所有女生
    for (int i = h[x]; i != -1; i = ne[i]) {
        int j = e[i];
        if (!st[j]) { // 防止重复匹配
            st[j] = true; // 标即为匹配过了
            if (match[j] == 0 || find(match[j])) { // 如果j号女生还没有匹配
                到男生或者她匹配到的男生可以找到下家
                match[j] = x; // 就把j匹配给x
                return true;
            }
        }
    }
    return false;
}

int calc() {
    int res = 0;
    for (int i = 1; i <= n1; i++) {
        memset(st, false, sizeof st);
        if (find(i)) res++;
    }
    return res;
}
```

2.8.3 最大流之二分图最大匹配

```
/* AcWing2175 飞行员配对方案问题 */
#include <bits/stdc++.h>

using namespace std;

const int N = 210, M = 30010, INF = 1e9;

int m, n, S, T;
int h[N], e[M], w[M], ne[M], idx;
int q[N], d[N], cur[N];

void add(int a, int b, int c) {
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
    e[idx] = a, w[idx] = 0, ne[idx] = h[b], h[b] = idx++;
}

bool bfs() {
    int hh = 0, tt = -1;
    memset(d, -1, sizeof d);
    q[++tt] = S, d[S] = 0, cur[S] = h[S];
    while (hh <= tt) {
        int t = q[hh++];
        for (int i = h[t]; ~i; i = ne[i]) {
            int j = e[i];
            if (d[j] == -1 && w[i]) {
                d[j] = d[t] + 1;
                cur[j] = h[j];
                q[++tt] = 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 && w[i]) {
        int t = find(j, min(w[i], limit - flow));
        if (!t) d[j] = -1;
        w[i] -= t, w[i ^ 1] += t, flow += t;
      }
    }
    return flow;
}

int dinic() {
    int max_flow = 0;
    while (bfs()) while (int flow = find(S, INF)) max_flow += flow;
    return max_flow;
}

int main() {
    cin >> m >> n;
    memset(h, -1, sizeof h);
    S = 0, T = n + 1;
    int a, b, cnt = 0;
    while (cin >> a >> b, a != -1) add(a, b, 1), cnt += 2;
    for (int i = 1; i <= m; i++) add(S, i, 1);
    for (int i = m + 1; i <= n; i++) add(i, T, 1);
    printf("%d\n", dinic());
    for (int i = 0; i < cnt; i += 2) {
      if (!w[i]) printf("%d %d\n", e[i ^ 1], e[i]);
    }
    return 0;
}
```

2.8.4 最大流之二分图多重匹配

```
/* AcWing2179 圆桌问题 */
#include <bits/stdc++.h>

using namespace std;

const int N = 500, M = 90010, INF = 1e9;

int m, n, S, T, tot;
int h[N], e[M], ne[M], w[M], idx;
int q[N], d[N], cur[N];

```

```cpp
void add(int a, int b, int c) {
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
    e[idx] = a, w[idx] = 0, ne[idx] = h[b], h[b] = idx++;
}

bool bfs() {
    int hh = 0, tt = -1;
    memset(d, -1, sizeof d);
    q[++tt] = S, d[S] = 0, cur[S] = h[S];
    while (hh <= tt) {
        int t = q[hh++];
        for (int i = h[t]; ~i; i = ne[i]) {
            int j = e[i];
            if (d[j] == -1 && w[i]) {
                d[j] = d[t] + 1;
                cur[j] = h[j];
                q[++tt] = 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 && w[i]) {
            int t = find(j, min(w[i], limit - flow));
            if (!t) d[j] = -1;
            w[i] -= t, w[i ^ 1] += t, flow += t;
        }
    }
    return flow;
}

int dinic() {
    int max_flow = 0;
    while (bfs()) while (int flow = find(S, INF)) max_flow += flow;
    return max_flow;
```

```
}

int main() {
   cin >> m >> n;
   S = 0, T = m + n + 1;
   memset(h, -1, sizeof h);
   for (int i = 1; i <= m; i++) {
      int x; scanf("%d", &x);
      tot += x, add(S, i, x);
   }
   for (int i = 1; i <= n; i++) {
      int x; scanf("%d", &x);
      add(m + i, T, x);
   }
   for (int i = 1; i <= m; i++) {
      for (int j = 1; j <= n; j++) {
         add(i, m + j, 1);
      }
   }
   if (tot != dinic()) return 0 * puts("0");
   puts("1");
   for (int i = 1; i <= m; i++) {
      for (int j = h[i]; ~j; j = ne[j]) {
         int k = e[j];
         if (k > m && !w[j]) printf("%d ", k - m);
      }
      puts("");
   }
   return 0;
}
```

3 字符串

3.1 Manacher

```
/* O(n)求字符串s的最大回文长度 */
#include <iostream>
#include <cstring>
#include <cstdio>
#include <algorithm>

using namespace std;
```

```cpp

const int N = 2000010;

int n, m, Case;
char s[N], str[N]; // s为原串,str为插入分隔符后的串
int p[N]; // p[i]为str中以下标i为中心的最大回文半径
// p[i]-1为s中以i为回文中心的最大回文长度

void manacher() {
   int rt = 0, mid = 0;
   int res = 0;
   for (int i = 1; i <= m; i++) {
      p[i] = i < rt ? min(p[2 * mid - i], rt - i) : 1;
      while (str[i + p[i]] == str[i - p[i]]) p[i]++;
      if (i + p[i] > rt) {
         rt = i + p[i];
         mid = i;
      }
      res = max(res, p[i] - 1);
   }
   printf("Case %d: %d\n", ++Case, res);
}

int main() {
   str[0] = '!', str[1] = '#'; /* str[0]为哨兵 */
   while (scanf("%s", s), s[0] != 'E') {
      n = strlen(s);
      for (int i = 0; i < n; i++) {
         str[i * 2 + 2] = s[i];
         str[i * 2 + 3] = '#';
      }
      m = n * 2 + 1;
      str[m + 1] = '@'; /* 哨兵 */
      manacher();
   }
   return 0;
}
```

3.2 KMP

```cpp
/* 求出模板串P在模式串S中所有出现的位置的起始下标 */
#include <iostream>
```

```cpp
#include <algorithm>

using namespace std;

const int N = 10010, M = 100010;

int n, m; // n,m分别为p,s的长度
char p[N], s[M];
int ne[N]; // ne[i]表示以i结尾的真后缀能够匹配前缀的最大长度

int main() {
   cin >> n >> p + 1 >> m >> s + 1;
   for (int i = 2, j = 0; i <= n; i++) {
      while (j && p[i] != p[j + 1]) j = ne[j];
      if (p[i] == p[j + 1]) j++;
      ne[i] = j;
   }
   for (int i = 1, j = 0; i <= m; i++) {
      while (j && (s[i] != p[j + 1])) j = ne[j];
      if (s[i] == p[j + 1]) j++;
      if (j == n) { // 匹配完成
         printf("%d ", i - 1 - n + 1);
         j = ne[j];
      }
   }
   return 0;
}
```

3.3 AC 自动机

```cpp
/*
 * AcWing1282 搜索关键词
 * 输入：T组数据，给定n，然后n个单词，最后给一个字符串表示文章
 * 求有多少个单词（可以是相同的）在文章中出现
*/
#include <iostream>
#include <cstring>
#include <algorithm>

using namespace std;

const int N = 10010 * 50, M = 1000010;
```

```
int T, n;
int son[N][26], cnt[N], idx;
int ne[N];
char str[M];
int ans;
int q[N];

void insert(char str[]) {
   int p = 0;
   for (int i = 0; str[i]; i++) {
      int u = str[i] - 'a';
      if (!son[p][u]) son[p][u] = ++idx;
      p = son[p][u];
   }
   cnt[p]++;
}

void build() {
   int hh = 0, tt = -1;
   for (int i = 0; i < 26; i++) {
      if (son[0][i]) q[++tt] = son[0][i];
   }
   while (hh <= tt) {
      int t = q[hh++];
      for (int i = 0; i < 26; i++) {
         if (!son[t][i]) son[t][i] = son[ne[t]][i];
         else {
            ne[son[t][i]] = son[ne[t]][i];
            q[++tt] = son[t][i];
         }
      }
   }
}

void init() {
   memset(ne, 0, sizeof ne);
   idx = 0;
   memset(cnt, 0, sizeof cnt);
   memset(son, 0, sizeof son);
}

void solve() {
```

```
    init();
    cin >> n;
    for (int i = 0; i < n; i++) {
        scanf("%s", str);
        insert(str);
    }
    build();
    scanf("%s", str);
    ans = 0;
    for (int i = 0, j = 0; str[i]; i++) {
        int u = str[i] - 'a';
        j = son[j][u];
        int k = j;
        while (k) {
            ans += cnt[k];
            cnt[k] = 0;
            k = ne[k];
        }
    }
    cout << ans << endl;
}

int main() {
    cin >> T;
    while (T--) solve();
    return 0;
}
```

4 其他

4.1 离散化

```
vector<int> alls;
sort(alls.begin(), alls.end());
alls.erase(unique(alls.begin(), alls.end()), alls.end());
int find(int x) {
    return lower_bound(alls.begin(), alls.end(), x) - alls.begin();
}
```

4.2 高精度

```cpp
struct hll {
   int num[4010], len, sign;
   hll() { len = 0, sign = 1; }
   hll(int x) { *this = x; }
   hll(long long x) { *this = x; }
   hll(char *ss) { *this = ss; }
   hll(string ss) { *this = ss; }
   hll& operator = (const int &x) {
      int val = x;
      if (val >= 0) sign = 1, len = 0;
      else if (val < 0) sign = -1, val = -val, len = 0;
      do {
         num[len++] = val % 10, val /= 10;
      } while (val);
      return *this;
   }
   hll& operator = (const long long &x) {
      long long val = x;
      if (val >= 0) sign = 1, len = 0;
      else if (val < 0) sign = -1, val = -val, len = 0;
      do {
         num[len++] = val % 10, val /= 10;
      } while (val);
      return *this;
   }
   hll& operator = (const string &ss) {
      len = ss.size();
      int start;
      if (ss[0] == '-') sign = -1, start = 1;
      else sign = 1, start = 0;
      for (int i = len - 1; i >= start; i--) num[len - i - 1] = ss[i]
         - '0';
      if (sign == -1) len--;
      return *this;
   }
   hll& operator = (const char *ss) {
      len = strlen(ss);
      int start;
      if (ss[0] == '-') sign = -1, start = 1;
      else sign = 1, start = 0;
      for (int i = len - 1; i >= start; i--) num[len - i - 1] = ss[i]
         - '0';
```

```
        if (sign == -1) len--;
        return *this;
    }
    hll& operator = (const hll &t) {
        len = t.len, sign = t.sign;
        for (int i = 0; i < len; i++) num[i] = t.num[i];
        return *this;
    }
    int abs_cmp(const hll &a, const hll &b) const { // |a|>|b|时返回1，
        相等返回0，小于返回-1
        if (a.len > b.len) return 1;
        else if (a.len < b.len) return -1;
        else {
            for (int i = a.len - 1; i >= 0; i--) {
                if (a.num[i] < b.num[i]) return -1;
                if (a.num[i] > b.num[i]) return 1;
            }
            return 0;
        }
    }
    int cmp(const hll &t) const { // *this与t比较，小于返回-1，等于返回0，大
        于返回1
        if (sign != t.sign) {
            if (sign == 1) return 1;
            else return -1;
        } else {
            if (abs_cmp(*this, t) == 1) return sign;
            else if (abs_cmp(*this, t) == 0) return 0;
            else return -sign;
        }
    }
    hll abs_plus(const hll &a, const hll &b) { // |a|+|b|, ans的符号与a
        和b原来的符号相同
        hll ans;
        ans.sign = a.sign;
        for (int i = 0, carry = 0; i < a.len || i < b.len || carry; i++)
            {
            if (i < a.len) carry += a.num[i];
            if (i < b.len) carry += b.num[i];
            ans.num[ans.len++] = carry % 10;
            carry /= 10;
        }
        return ans;
```

```
    }
    hll abs_minus(const hll &a, const hll &b) { // ||a|-|b||, ans的符号为|a|-|b|的符号
        hll ans, c, d;
        if (abs_cmp(a, b) >= 0) ans.sign = 1, c = a, d = b;
        else ans.sign = -1, c = b, d = a;
        for (int i = 0, borrow = 0; i < c.len; i++) {
            borrow = c.num[i] - borrow;
            if (i < d.len) borrow -= d.num[i];
            ans.num[ans.len++] = (borrow + 10) % 10;
            if (borrow >= 0) borrow = 0;
            else borrow = 1;
        }
        while (ans.len > 1 && ans.num[ans.len - 1] == 0) ans.len--; // 去除前导0
        return ans;
    }
    bool operator == (const hll &t) const { return cmp(t) == 0; }
    bool operator != (const hll &t) const { return !(cmp(t) == 0); }
    bool operator < (const hll &t) const { return cmp(t) == -1; }
    bool operator > (const hll &t) const { return cmp(t) == 1; }
    bool operator <= (const hll &t) const { return !(cmp(t) == 1); }
    bool operator >= (const hll &t) const { return !(cmp(t) == -1); }
    hll operator + (const hll &t) {
        hll ans;
        if (sign == t.sign) { // 同号 直接相加，符号不变
            ans = abs_plus(*this, t);
        } else { // 异号
            if (sign == 1) { // 前正 + 后负 == 前绝对值 - 后绝对值
                ans = abs_minus(*this, t);
            } else { // 前负 + 后正 == 后绝对值 - 前绝对值
                ans = abs_minus(t, *this);
            }
        }
        return ans;
    }
    hll operator - (const hll &t) {
        hll ans;
        if (sign == t.sign) { // 同号
            ans = abs_minus(*this, t);
            if (sign == 1) { // 前正 - 后正
                ; // 不用做了
            } else { // 前负 - 后负
```

```
            ans.sign *= -1;
         }
      } else { // 异号
         if (sign == 1) { // 前正 - 后负 == 前绝对值 + 后绝对值
            ans = abs_plus(*this, t);
         } else { // 前负 - 后正 == -(前绝对值 + 前绝对值)
            ans = abs_plus(t, *this);
            ans.sign = -1;
         }
      }
      return ans;
   }
   hll operator * (const hll &t) { // 高精度*高精度
      hll ans;
      memset(ans.num, 0, len + t.len << 2);
      ans.sign = sign * t.sign;
      ans.len = len + t.len - 1; // a位数乘以b位数，得到的结果是a+b-1位数，或a+b位数
      for (int i = 0; i < len; i++) {
         for (int j = 0; j < t.len; j++)
            ans.num[i + j] += num[i] * t.num[j];
      }
      for (int i = 0; i < ans.len - 1; i++) {
         if (ans.num[i] >= 10) {
            ans.num[i + 1] += ans.num[i] / 10;
            ans.num[i] %= 10;
         }
      }
      // 看最高位是否需要进位，如果有进位，答案最终是a+b位数，否则是a+b-1位数
      if (ans.num[ans.len - 1] >= 10) {
         ans.num[ans.len] = ans.num[ans.len - 1] / 10;
         ans.num[ans.len - 1] %= 10;
         ans.len++;
      }
      while (ans.len > 1 && ans.num[ans.len - 1] == 0) ans.len--; // 去除前导0
      return ans;
   }
   hll operator += (const hll &t) {
      return *this + t;
   }
   hll operator *= (const hll &t) {
      return *this * t;
```

```
    }
    void print() {
        if (sign == -1) putchar('-');
        for (int i = len - 1; i >= 0; i--) putchar(num[i] + '0');
    }
};
```