数据结构

Kd-Tree

```
//id数组为新建kdtree时所使用点的队列

class poi {
  public:
    int x, y;
    poi() {}
    poi (int x_, int y_) : x (x_), y (y_) {}
} P[200010], p;

//sx, tx, sy, ty表示kdtree元素所覆盖区域的坐标范围
//s为子树内元素个数
//l, r为左右儿子
class kdleaf {
  public:
    int sx, tx, sy, ty, s;
    int l, r;
} kdt[200010];

//x，y坐标排序
bool cmpx (const int &x, const int &y) {
    return P[x].x < P[y].x;
}
bool cmpy (const int &x, const int &y) {
    return P[x].y < P[y].y;
}

//用点y更新点x的区域范围
void renew (int x, int y) {
    if (!y) return;
    kdt[x].sx = min (kdt[x].sx, kdt[y].sx);
    kdt[x].sy = min (kdt[x].sy, kdt[y].sy);
    kdt[x].tx = max (kdt[x].tx, kdt[y].tx);
    kdt[x].ty = max (kdt[x].ty, kdt[y].ty);
}

//利用id[]数组内的[l, r]范围新建kdtree，返回根标号，第一次排序方式以step
为准
int kd_build (int l, int r, int step) {
    if (l > r) return 0;
    int mid = (l + r) >> 1;
    nth_element (id + l, id + mid, id + r + 1, step ? cmpx :
cmpy);
    int x = id[mid];
    kdt[x].sx = kdt[x].tx = P[x].x;
    kdt[x].sy = kdt[x].ty = P[x].y;
    kdt[x].s = 1;
    kdt[x].l = kd_build (l, mid - 1, !step);
    renew (x, kdt[x].l);
    kdt[x].s += kdt[kdt[x].l].s;
    kdt[x].r = kd_build (mid + 1, r, !step);
    renew (x, kdt[x].r);
    kdt[x].s += kdt[kdt[x].r].s;
    return x;
}

//当重量不平衡时使用的重建函数
//取出点head的所有儿子，并重建kdtree
//第一次排序方式按step
//返回重建后的根标号
int rebuild (int head, int step) {
    int l, r, x;
    id[l = r = 1] = head;
    for (; l <= r; l++) {
        x = id[l];
        if (kdt[x].l) id[++r] = kdt[x].l;
        if (kdt[x].r) id[++r] = kdt[x].r;
    }
    return kd_build (1, r, step);
}

//在以root为根的子树中插入元素x，第一次排序方式为step
//考虑了不平衡时的替罪羊重构
void Insert (int &root, int x, int step) {
    if (!root) {
        root = x;
        kdt[root].sx = kdt[root].tx = P[x].x;
        kdt[root].sy = kdt[root].ty = P[x].y;
        kdt[root].s = 1;
        return;
    }
    if (step ? cmpx (x, root) : cmpy (x, root) )
        Insert (kdt[root].l, x, !step);
    else Insert (kdt[root].r, x, !step);
    renew (root, x);
    kdt[root].s++;
    if (kdt[kdt[root].l].s > kdt[kdt[root].r].s * 2 + 1
            || kdt[kdt[root].r].s > kdt[kdt[root].l].s * 2 + 1)
        root = rebuild (root, step);
}

//曼哈顿距离
int dist (poi a, poi b) {
    return abs (a.x - b.x) + abs (a.y - b.y);
}

//计算点p到kdtree元素t所代表区域的最近曼哈顿距离
int Min_dist (poi p, kdleaf t) {
    if (p.x < t.sx) {
        if (p.y < t.sy)
            return abs (p.x - t.sx) + abs (p.y - t.sy);
        else if (p.y > t.ty)
            return abs (p.x - t.sx) + abs (p.y - t.ty);
        else
            return abs (p.x - t.sx);
    } else if (t.sx <= p.x && p.x <= t.tx) {
        if (p.y < t.sy)
            return abs (p.y - t.sy);
        else if (p.y > t.ty)
            return abs (p.y - t.ty);
        else
            return 0;
    } else {
        if (p.y < t.sy)
            return abs (p.x - t.tx) + abs (p.y - t.sy);
        else if (p.y > t.ty)
            return abs (p.x - t.tx) + abs (p.y - t.ty);
        else
            return abs (p.x - t.tx);
    }
}

//计算点p到kdtree元素t所代表区域的最远曼哈顿距离
int Max_dist (poi p, kdleaf t) {
    int res = abs (p.x - t.sx) + abs (p.y - t.sy);
    res = max (res, abs (p.x - t.sx) + abs (p.y - t.ty) );
    res = max (res, abs (p.x - t.tx) + abs (p.y - t.sy) );
    res = max (res, abs (p.x - t.tx) + abs (p.y - t.ty) );
    return res;
}

//计算点 x 到 kdtree 内所有点的最近距离
void ask_min (int root, int x, int step, int &ans) {
    if (!root) return;
    ans = min (ans, dist (P[root], P[x]) );
    if (Min_dist (P[x], kdt[root]) >= ans) return;
    if (step ? cmpx (x, root) : cmpy (x, root) ) {
        ask_min (kdt[root].l, x, !step, ans);
        ask_min (kdt[root].r, x, !step, ans);
    } else {
        ask_min (kdt[root].r, x, !step, ans);
        ask_min (kdt[root].l, x, !step, ans);
    }
}

//计算点 x 到 kdtree 内所有点的最远距离
void ask_max (int root, int x, int step, int &ans) {
    if (!root) return;
    ans = max (ans, dist (P[root], P[x]) );
    if (Max_dist (P[x], kdt[root]) <= ans) return;
    if (step ? cmpx (x, root) : cmpy (x, root) ) {
        ask_max (kdt[root].r, x, !step, ans);
        ask_max (kdt[root].l, x, !step, ans);
    } else {
        ask_max (kdt[root].l, x, !step, ans);
        ask_max (kdt[root].r, x, !step, ans);
    }
}
```

LCA-Tarjan

```
#include<iostream>
#include<cstdio>
#include<vector>
using namespace std;

const int MaxN = 1007;

int n, m, rt;
vector<int> iv[MaxN], qry[MaxN];
```

```
int cnt[MaxN], fa[MaxN];
bool vis[MaxN];

void Init() {
    int i, id, t, chd, a, b;
    char st1[MaxN], st2[MaxN];
    for (i = 1; i <= n; i++) {
        cnt[i] = fa[i] = 0;
        vis[i] = false;
        iv[i].clear();
        qry[i].clear();
    }
    for (i = 1; i <= n; i++) {
        scanf ("%d:(%d)", &id, &t);
        while (t--) {
            scanf ("%d", &chd);
            //cout<<id<<' '<<chd<<endl;
            iv[id].push_back (chd);
            fa[chd] = id;
        }
    }
    for (i = 1; i <= n; i++)
        if (!fa[i]) rt = i;
        else fa[i] = 0;
    scanf ("%d", &m);
    for (i = 1; i <= m; i++) {
        scanf ("%1s%d%d%s", st1, &a, &b, st2);
        //cout<<a<<' '<<b<<endl;
        qry[a].push_back (b);
        qry[b].push_back (a);
    }
    return;
}

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

void Tarjan (int u) {
    int i, j;
    fa[u] = u;
    for (i = 0; i < iv[u].size(); i++) {
        j = iv[u][i];
        if (!fa[j]) {
            Tarjan (j);
            fa[j] = u;
        }
    }
    vis[u] = true;
    for (i = 0; i < qry[u].size(); i++) {
        j = qry[u][i];
        if (vis[j]) cnt[FindF (j)]++;
    }
    return;
}

void Solve() {
    int i;
    Tarjan (rt);
    for (i = 1; i <= n; i++)
        if (cnt[i]) printf ("%d:%d\n", i, cnt[i]);
    return;
}

int main() {
    freopen ("cca.in", "r", stdin);
    freopen ("cca.out", "w", stdout);
    while (scanf ("%d", &n) != EOF) {
        Init();
        Solve();
    }
    return 0;
}
```

Link-Cut Tree

```
//以下为splay元素：
//l, r为左右儿子, p为父亲
//mark为size的增减标记
//change为左右翻转标记
//以下为LCT独有元素
//pre为轻链父亲
//size为树上的子树大小
class lctleaf {
  public:
    int l, r, p, pre, size, mark;
    bool change;
};

class LinkCutTree {
  public:
    lctleaf a[100010];
    void make (int x) {
      //更新splay内点x的信息
    }
    void makemark (int x) {
      int lson = a[x].l, rson = a[x].r;
      if (a[x].change) {
        if (lson) {
          a[lson].change = !a[lson].change;
          swap (a[lson].l, a[lson].r);
        }
        if (rson) {
          a[rson].change = !a[rson].change;
          swap (a[rson].l, a[rson].r);
        }
        a[x].change = 0;
      }
      if (a[x].mark) {
        if (lson) {
          a[lson].mark += a[x].mark;
          a[lson].size += a[x].mark;
        }
        if (rson) {
          a[rson].mark += a[x].mark;
          a[rson].size += a[x].mark;
        }
        a[x].mark = 0;
      }
    }
    void clrmark (int x) {
      int ll = 0;
      for (; x; x = a[x].p) qq[++ll] = x;
      for (int i = ll; i; i--) makemark (qq[i]);
    }
    void left (int x) {
      int y = a[x].p, z = a[y].p;
      if (a[z].l == y) a[z].l = x;
      else a[z].r = x;
      a[y].r = a[x].l;
```

```
        a[a[x].l].p = y;
        a[x].l = y;
        a[x].p = z;
        a[y].p = x;
        swap (a[y].pre, a[x].pre);
//      make(y);make(x);
    }

    void right (int x) {
        int y = a[x].p, z = a[y].p;
        if (a[z].l == y) a[z].l = x;
        else a[z].r = x;
        a[y].l = a[x].r;
        a[a[x].r].p = y;
        a[x].r = y;
        a[x].p = z;
        a[y].p = x;
        swap (a[y].pre, a[x].pre);
//      make(y);make(x);
    }
    void splay (int x) {
        int y, z;
        clrmark (x);
        while (a[x].p) {
            y = a[x].p;
            z = a[y].p;
            if (z) {
                if (a[z].l == y) {
                    if (a[y].l == x) right (y), right (x);
                    else left (x), right (x);
                } else {
                    if (a[y].l == x) right (x), left (x);
                    else left (y), left (x);
                }
            } else if (a[y].l == x) right (x);
            else left (x);
        }
    }

    //查找splay中点x的next
    int next (int x) {
        splay (x);
        makemark (x);
        x = a[x].r;
        do {
            makemark (x);
            if (a[x].l) x = a[x].l;
            else break;
        } while (1);
        return x;
    }

    //变更重链操作
    void access (int x) {
        int p = 0;
        while (x) {
            splay (x);
            a[a[x].r].pre = x;
            a[a[x].r].p = 0;
            a[x].r = p;
            a[p].p = x;
            a[p].pre = 0;
            p = x;
//          make(x);
            x = a[x].pre;
        }
    }

    //将点x置为根，之前的根为root
    //同时更新了size
    void move_root (int root, int x) {
        access (x);
        splay (x);
        a[a[x].l].change = !a[a[x].l].change;
        swap (a[a[x].l].l, a[a[x].l].r);
        a[a[x].l].pre = a[a[x].l].p;
        a[a[x].l].p = 0;
        a[x].l = 0;
        a[root].size -= a[x].size;
        a[x].size += a[root].size;
    }

    //查找点x的根
    int find_root (int x) {
        access (x);
        splay (x);
        while (a[x].l) x = a[x].l;
        splay (x);
        return x;
    }

    //查询lca
    int lca (int x, int y) {
        int root = find_root (x);
        access (x);
        access (y);
        splay (x);
        int xx = x;
        while (a[xx].l) xx = a[xx].l;
        splay (xx);
        if (xx == root) return x;
        else return a[xx].pre;
    }
} lct;
```

Quick Sort

```
void work (int l, int r) {
    int i = l, j = r, m = rand() % (r - l + 1) + l;
    swap (a[m], a[i]);
    m = a[i];
    while (i < j) {
        while (i < j && a[j] >= m) j--;
        if (i < j) a[i] = a[j], i++;
        while (i < j && a[i] <= m) i++;
        if (i < j) a[j] = a[i], j--;
    }
    a[i] = m;
    if (l < i - 1) work (l, i - 1);
    if (i + 1 < r) work (i + 1, r);
}
```

Splay - GY

```
//s为子树节点个数
//data为节点权值
//Min为子树权值最小值

class splay {
  public:
    int lson, rson, p, data, Min, s;
} spl[10010];
```

```
void make (int x) {
    spl[x].s = 1;
    spl[x].Min = spl[x].data;
    int lson = spl[x].lson, rson = spl[x].rson;
    if (lson){
        spl[x].s += spl[lson].s;
        spl[x].Min = min (spl[x].Min, spl[lson].Min);
    }
    if (rson){
        spl[x].s += spl[rson].s;
        spl[x].Min = min (spl[x].Min, spl[rson].Min);
    }
}

void Right (int x) {
    int y = spl[x].p, z = spl[y].p;
    if (spl[z].lson == y) spl[z].lson = x;
    else spl[z].rson = x;
    spl[x].p = z;
    spl[y].lson = spl[x].rson;
    spl[spl[y].lson].p = y;
    spl[x].rson = y;
    spl[y].p = x;
    make (y);
    make (x);
}

void Left (int x) {
    int y = spl[x].p, z = spl[y].p;
    if (spl[z].lson == y) spl[z].lson = x;
    else spl[z].rson = x;
    spl[x].p = z;
    spl[y].rson = spl[x].lson;
    spl[spl[y].rson].p = y;
    spl[x].lson = y;
    spl[y].p = x;
    make (y);
    make (x);
}

void splay (int x) {
    int y, z;
    while (spl[x].p) {
        y = spl[x].p;
        z = spl[y].p;
        if (z) {
            if (spl[z].lson == y) {
                if (spl[y].lson == x) Right (y), Right (x);
                else Left (x), Right (x);
            } else {
                if (spl[y].lson == x) Right (x), Left (x);
                else Left (y), Left (x);
            }
        } else if (spl[y].lson == x) Right (x);
        else Left (x);
    }
    root = x;
}

//寻找第 num 个元素
int finds (int num) {
    if (spl[root].s < num) return 0;
    int x = root;
    int lson;
    for (; x;) {
        lson = spl[x].lson;
        if (num == spl[lson].s + 1) return x;
        if (num <= spl[lson].s) x = lson;
        else num -= spl[lson].s + 1, x = spl[x].rson;
    }
    return 0;
}

//寻找最小值的位置
int findminpos() {
    int x = root;
    int Min = spl[root].Min;
    for (; x;) {
        if (spl[x].data == Min) return x;
        if (spl[spl[x].lson].Min == Min) x = spl[x].lson;
        else x = spl[x].rson;
    }
    return 0;
}
```

Splay - LQY

```
void Zig (int x) {
    int p = spl[x].fa;
    if (spl[spl[p].fa].lc == p) spl[spl[p].fa].lc = x;
    else spl[spl[p].fa].rc = x;
    spl[x].fa = spl[p].fa;
    spl[p].fa = x;
    spl[p].lc = spl[x].rc;
    spl[spl[x].rc].fa = p;
    spl[x].rc = p;
    return;
}

void Zag (int x) {
    int p = spl[x].fa;
    if (spl[spl[p].fa].lc == p) spl[spl[p].fa].lc = x;
    else spl[spl[p].fa].rc = x;
    spl[x].fa = spl[p].fa;
    spl[p].fa = x;
    spl[p].rc = spl[x].lc;
    spl[spl[x].lc].fa = p;
    spl[x].lc = p;
    return;
}

void Splay (int x) {
    int i, j, k;
    k = x;
    while (a[k].f) {
        i = a[k].f;
        if (!a[i].f) {
            if (a[i].lc == k) Zig (k);
            else Zag (k);
        } else {
            if (a[a[i].f].lc == i) {
                if (a[i].lc == k) Zig (i), Zig (k);
                else Zag (k), Zig (k);
            } else {
                if (a[i].rc == k) Zag (i), Zag (k);
                else Zig (k), Zag (k);
            }
        }
    }
    root = x;
    return;
```

```
}

int Find (int key) {
    int x = root;
    while (x) {
        if (spl[x].key == key) return x;
        if (key < spl[x].key) x = spl[x].lc;
        else x = spl[x].rc;
    }
    return -1;
}

void Insert (int key) {
    int x = root, p = 0;
    while (x) {
        if (spl[x].key == key) return;
        if (key < spl[x].key) x = spl[x].lc;
        else x = spl[x].rc;
        p = x;
    }
    tot++;
    spl[tot].key = key;
    spl[tot].lc = spl[tot].rc = 0;
    spl[tot].fa = p;
    if (!p)
        if (key < spl[p].key) spl[p].lc = tot;
        else spl[p].rc = tot;
    Splay (tot);
    return;
}

void Delete (int key) {
    int x = Find (key);
    if (x == -1) return;
    spl[x].key = 0;
    if (spl[x].lc == 0 && spl[x].rc == 0) {
        if (spl[spl[x].fa].lc == x) spl[spl[x].fa].lc = 0;
        else spl[spl[x].fa].rc = 0;
    } else if (spl[x].lc && !spl[x].rc) {
        if (spl[spl[x].fa].lc == x) spl[spl[x].fa].lc = spl[x].lc;
        else spl[spl[x].fa].rc = spl[x].lc;
        spl[x].lc = spl[x].fa;
    } else if (spl[x].rc && !spl[x].lc) {
        if (spl[spl[x].fa].lc == x) spl[spl[x].fa].lc = spl[x].rc;
        else spl[spl[x].fa].rc = spl[x].rc;
        spl[x].rc = spl[x].fa
    } else {
        int k = spl[x].rc;
        while (spl[k].lc) k = spl[k].lc;
        spl[x].key = spl[k].key;
        spl[spl[k].fa].lc = spl[k].rc;
        spl[spl[k].rc].fa = spl[k].fa;
        spl[k].key = 0;
    }
    return;
}
```

Treap Merge-Split - WP

```
#define Pair pair<int,int>

int build (int L, int R) {
    if (L == R) {
        f[L] = 0;
        g[L] = 0;
        size[L] = 1;
        return L;
    }
    int u = 0;
    for (int i = L; i <= R; i++)
        if (dot[i].key > dot[u].key) u = i;
    if (u > L) ls[u] = build (L, u - 1);
    if (u < R) rs[u] = build (u + 1, R);
    update (u);
    return u;
}

int merge (int a, int b) {
    if (!a || !b) return a | b;
    if (dot[a].kay > dot[b].key) {
        rs[a] = merge (rs[a], b);
        update (a);
        return a;
    } else {
        ls[b] = merge (a, ls[b]);
        update (b);
        return b;
    }
}

Pair split (int a, int k) {
   if (!k) return make_pair (0, a);
   if (k == size[a]) return make_pair (a, 0);
   if (k == size[ls[a]]) {
      int b = ls[a];
      ls[a] = 0;
      update (a);
      update (b);
      return make_pair (b, a);
   } else if (k < size[ls[a]]) {
      Pair p = split (ls[a], k);
      ls[a] = p.second;
      update (a);
      p.second = a;
      return p;
   } else if (k == size[ls[a]] + 1) {
      int b = rs[a];
      rs[a] = 0;
      update (a);
      return make_pair (a, b);
   } else {
      Pair p = split (rs[a], k - size[ls[a]] - 1);
      rs[a] = p.first;
      update (a);
      p.first = a;
      return p;
   }
}
```

Treap - LQY

```
#include<iostream>
#include<cstdio>
#include<ctime>
using namespace std;

const int MaxN = 100007;

typedef struct treapnode {
   treapnode *lc, *rc;
   int key, pro;
} treapnode, *treap;
```

```
typedef struct {
    int a, b;
} node;

treap null, root;
int n, m, delt;
long long s;
node sh[MaxN], gr[MaxN];

int Cmp (const void *a, const void *b) {
    return (* (node*) b).b - (* (node*) a).b;
}

void Init() {
    scanf ("%d%d", &n, &m);
    for (int i = 1; i <= n; i++)
        scanf ("%d%d", &sh[i].a, &sh[i].b);
    qsort (sh + 1, n, sizeof (node), Cmp);
    for (int i = 1; i <= m; i++)
        scanf ("%d%d", &gr[i].a, &gr[i].b);
    qsort (gr + 1, m, sizeof (node), Cmp);

    srand (time (0) );
    null = new treapnode;
    null->lc = null;
    null->rc = null;
    root = null;
    return;
}

void Right_Rotate (treap &p) {
    treap q;
    q = p->lc;
    p->lc = q->rc;
    q->rc = p;
    p = q;
    return;
}

void Left_Rotate (treap &p) {
    treap q;
    q = p->rc;
    p->rc = q->lc;
    q->lc = p;
    p = q;
    return;
}

void Insert (treap &x, int key) {
    if (x == null) {
        x = new treapnode;
        x->lc = null;
        x->rc = null;
        x->key = key;
        x->pro = rand();
        return;
    }
    if (key <= x->key) {
        Insert (x->lc, key);
        if (x->lc->pro < x->pro)
            Right_Rotate (x);
    } else {
        Insert (x->rc, key);
        if (x->rc->pro < x->pro)
            Left_Rotate (x);
    }
    return;
}

void Search (treap &x, int key) {
    if (x == null) return;
    if (key <= x->key) {
        delt = x->key;
        Search (x->lc, key);
    } else Search (x->rc, key);
    return;
}

void Delete (treap &x, int key) {
    if (x == null) return;
    if (key < x->key) Delete (x->lc, key);
    else if (key > x->key) Delete (x->rc, key);
    else {
        if (x->rc == null && x->lc == null) {
            delete x;
            x = null;
        } else if (x->rc == null) {
            treap tt;
            tt = x;
            x = x->lc;
            delete tt;
        } else if (x->lc == null) {
            treap tt;
            tt = x;
            x = x->rc;
            delete tt;
        } else if (x->lc->pro < x->rc->pro) {
            Right_Rotate (x);
            Delete (x->rc, key);
        } else {
            Left_Rotate (x);
            Delete (x->lc, key);
        }
    }
    return;
}

void Solve() {
    int i, j;
    //for(i=1;i<=n;i++)
    //    cout<<sh[i].a<<' '<<sh[i].b<<endl;
    //cout<<endl;
    //for(i=1;i<=m;i++)
    //    cout<<gr[i].a<<' '<<gr[i].b<<endl;
    //cout<<endl;
    i = 1;
    j = 1;
    while (i <= n) {
        while (j <= m && gr[j].b >= sh[i].b)
            Insert (root, gr[j].a), j++;
        delt = -1;
        Search (root, sh[i].a);
        if (delt == -1) {
            printf ("-1");
            return;
        }
        s += delt;
        Delete (root, delt);
        i++;
    }
    cout << s;
    return;
}

int main() {
    freopen ("gourmet.in", "r", stdin);
    freopen ("gourmet.out", "w", stdout);
    Init();
    Solve();
    return 0;
}
```

可并堆_斜堆 - GY

```
class rec {
  public:
    int lson, rson, num;
} heap[1000010];

int merge (int a, int b) {
    if (!a) return b;
    if (!b) return a;
    if (heap[a].num < heap[b].num) swap (a, b);
    heap[a].rson = merge (heap[a].rson, b);
    swap (heap[a].lson, heap[a].rson);
    return a;
}

int Del (int x) {
    return merge (heap[x].lson, heap[x].rson);
}
```

树分块 - GY

```
//BZOJ 1086
//读入一棵树，对其分块，每块大小在[B,3B]内
//树上莫队维护链的信息时，可以不保存 lca 的信息，来简化实现过程
//带修改树上莫队 O(N ^ (5/3))，排序时按 l 所在块、r 所在块、操作时间 t 三关键字排序
//执行莫队算法时将时间也纳入考虑，不断执行、撤销修改操作
//块大小为 O(N^(2/3))复杂度最优
void dfs(int x)
{
    for (int i=st[x];i;i=ne[i])
    if (go[i]!=fa[x])
    {
        dfs(go[i]);
        for (int
j=1;j<=res[go[i]][0];j++)res[x][++res[x][0]]=res[go[i]][j];
        if (res[x][0]>B)
        {
            s++;
            for (int j=1;j<=res[x][0];j++)num[res[x][j]]=s;
            wei[s]=x;
            res[x][0]=0;
        }
    }
    res[x][++res[x][0]]=x;res[x][0]=res[x][0];
    if (res[x][0]>B)
    {
        s++;
        for (int j=1;j<=res[x][0];j++)num[res[x][j]]=s;
        wei[s]=x;
        res[x][0]=0;
    }
}
void Add(int x,int y){ne[++pt]=st[x];st[x]=pt;go[pt]=y;}
int main()
{
    freopen("royal.in","r",stdin);
    freopen("royal.out","w",stdout);
    scanf("%d%d",&N,&B);
    for (int i=1;i<N;i++)
    {
        scanf("%d%d",&x,&y);
        Add(x,y);Add(y,x);
    }
    for (q[l=r=0]=1;l<=r;l++)
    {
        int x=q[l];
        for (int i=st[x];i;i=ne[i])
        if (go[i]!=fa[x])
        q[++r]=go[i],fa[go[i]]=x;
    }
    dfs(1);
    if (!s)
    {
        printf("0");
        return 0;
    }
    for (int i=1;i<=res[1][0];i++)num[res[1][i]]=s;
    printf("%d\n",s);
    for (int i=1;i<=N;i++)printf("%d ",num[i]);
    printf("\n");
    for (int i=1;i<=s;i++)printf("%d ",wei[i]);
    return 0;
}
```

树链剖分 - GY

```
//TreeSize[N] 点x的子树大小
//Depth[N] 点x的深度
//HeavyChild[N] 点x的重儿子
//Block[N] 点x所在的链标号，同时也是该链的最高点的序号
//NodeID[N] 点x在序列中的位置
//IndexToNode[N] 序列中第x个点的序号
//BlockLeft[N] 编号为x的链在序列中的左端点的位置（链的最高点）
//BlockRight[N] 编号为x的链在序列中的右端点的位置（链的最低点）
//可以将所有点建立一棵线段树
//得出的序列保证了每条轻重链连续
//并保证了每个点的子树内除其所在重链外所有点都连续

//预处理每个点的基本信息
void dfs_size (int x) {
    TreeSize[x] = 1;
    for (int i = st[x]; i; i = ne[i]) {
        int y = go[i];
        if (y == fa[x]) continue;
        fa[y] = x;
        Depth[y] = Depth[x] + 1;
        dfs_size (y);
        TreeSize[x] += TreeSize[y];
        if (TreeSize[HeavyChild[x]] < TreeSize[y])
            HeavyChild[x] = y;
    }
}

//轻重链剖分
void dfs_lh (int x, int block) {
    Block[x] = block;
    NodeID[x] = ++idx;
    IndexToNode[idx] = x;
    if (!BlockLeft[block])
        BlockLeft[block] = idx;
    BlockRight[block] = idx;
    if (HeavyChild[x])
        dfs_lh (HeavyChild[x], block);
    for (int i = st[x]; i; i = ne[i]) {
        int y = go[i];
        if (y == fa[x] || y == HeavyChild[x])
            continue;
        dfs_lh (y, y);
    }
}

//主进程
void Decomposition (int s, int N) {
    idx = 0;
    fa[s] = 0;
    memset (HeavyChild, 0, sizeof (HeavyChild) );
    dfs_size (s);
    dfs_lh (s, s);
}

//（x，y）的lca
int lca (int x, int y) {
    while (Block[x] != Block[y]) {
        if (Depth[Block[x]] < Depth[Block[y]])
            swap (x, y);
        x = fa[Block[x]];
    }
    if (Depth[x] < Depth[y]) return x;
    else   return y;
}
```

极角序凸包

```
#include<iostream>
#include<cstdio>
#include<cmath>
using namespace std;

const int MaxN = 1007;

typedef struct {
    int x, y;
```

```
} node;

int n, m, l, mq[MaxN];
double dis;
node zr[MaxN];

inline int Chaji (node a, node b, node c) {
    return (b.x - a.x) * (c.y - a.y) - (c.x - a.x) * (b.y - a.y);
}

int Cmp (const void *a, const void *b) {
    node m = * (node*) a;
    node n = * (node*) b;
    int tmp = Chaji (zr[0], m, n);
    if (tmp == 0) return m.x - n.x;
    return tmp;
}

void Init() {
    int i;
    node t;
    scanf ("%d%d", &n, &l);
    for (i = 0; i < n; i++) {
        scanf ("%d%d", &zr[i].x, &zr[i].y);
        if (zr[i].x < zr[0].x) t = zr[0], zr[0] = zr[i], zr[i] = t;
        if (zr[i].x == zr[0].x && zr[i].y < zr[0].y) t = zr[0],
zr[0] = zr[i], zr[i] = t;
    }
    n--;
    qsort (zr + 1, n, sizeof (node), Cmp);
    return;
}

inline double Dis (node a, node b) {
    return sqrt (1.0 * (a.x - b.x) * (a.x - b.x) + 1.0 * (a.y -
b.y) * (a.y - b.y) );
}

void Solve() {
    int i;
    ++m;
    mq[m] = 0;
    ++m;
    mq[m] = 1;
    i = 2;
    while (i <= n) {
        while (m > 2 && Chaji (zr[mq[m - 1]], zr[mq[m]], zr[i]) >=
0) m--;
        mq[++m] = i;
        i++;
    }
    for (i = 1; i < m; i++)
        dis += Dis (zr[mq[i]], zr[mq[i + 1]]);
    dis += Dis (zr[mq[m]], zr[mq[1]]);
    dis += 2 * 3.1415926535 * l;
    printf ("%.0lf", dis);
    return;
}

int main() {
    freopen ("wall.in", "r", stdin);
    freopen ("wall.out", "w", stdout);
    Init();
    Solve();
    return 0;
}
```

线段树合并

```
merge(a,b):
    如果 a,b 中有一个不含任何元素，就返回另一个
    如果 a,b 都是叶子，返回 merge_leaf(a,b)
    返回 merge(a->l,b->l)与 merge(a->r,b->r)连接成的树
```