Caderno

pedroteosousa

Contents

1 Geometry

1.1 Point Struct

```
typedef long long type;

double EPS = 1e-12;

struct point {
    type x, y;
    point(type xp = 0.0, type yp = 0.0) {
        x = xp;
        y = yp;
    }
    point(const fpoint &p) {
        x = p.x;
        y = p.y;
    }
    point operator+ (const point &p) const {return point(x+p.x, y+p.y);}
    point operator- (const point &p) const {return point(x-p.x, y-p.y);}
    point operator* (type c) {return point(c*x, c*y);}
    point operator/ (type c) {return point(x/c, y/c);}

    bool operator<(const point &p) {return x < p.x || x == p.x && y < p.y;}
};

type dot(point p, point q) {return p.x*q.x+p.y*q.y;}
type dist(point p, point q) {return sqrt(dot(p-q,p-q));}
type cross(point p, point q) {return p.x*q.y-p.y*q.x;}

point projectInLine(point c, point a, point b) {
    return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);
}
point projectInSegment(point c, point a, point b) {
    point lineP = projectInLine(c, a, b);
    type maxDist = max(dist(a, lineP), dist(b, lineP));
    if (maxDist > dist(a, b)) {
        if (dist(a, c) > dist(b, c)) return b;
        else return a;
    }
    else return lineP;
}
```

1.2 Convex Hull

```
double side(point a, point b, point c) {
    return cross(a, b) + cross(b, c) + cross(c, a);
}

vector<point> convex_hull(vector<point> p) {
    int n = p.size(), k = 0;
    if (n == 1) return p;
    vector<point> hull(2*n);

    sort(p.begin(), p.end());

    for(int i=0; i<n; i++) {
        while(k>=2 && (side(hull[k-2], hull[k-1], p[i]) <= 0)) k--;
        hull[k++] = p[i];
    }

    for(int i=n-2,t=k+1; i>=0; i--) {
        while(k>=t && (side(hull[k-2], hull[k-1], p[i]) <= 0)) k--;
        hull[k++] = p[i];
    }

    hull.resize(k-1);
    return hull;
}
```

2 Data Structures

2.1 Trie

```
const int A = 26;

typedef struct trie {
    struct node {
        int to[A], freq, end;
    };
    struct node t[N];
    int sz = 0;
    int offset = 'a';

    // init trie
    void init() {
        memset(t, 0, sizeof(struct node));
    }
    // insert string
    void insert(char *s, int p = 0) {
        t[p].freq++;
        if (*s == 0) {
            t[p].end++;
            return;
        }
        if (t[p].to[*s - offset] == 0)
            t[p].to[*s - offset] = ++sz;
        insert(s+1, t[p].to[*s - offset]);
    }

    // check if string is on trie
    int find(char *s, int p = 0) {
        if (*s == 0)
            return t[p].end;
        if (t[p].to[*s - offset] == 0)
            return false;
        return find(s+1, t[p].to[*s - offset]);
    }

    // count the number of strings that have this prefix
    int count(char *s, int p = 0) {
        if (*s == 0)
            return t[p].freq;
        if (t[p].to[*s - offset] == 0)
            return 0;
        return count(s+1, t[p].to[*s - offset]);
    }

    // erase a string
    int erase(char *s, int p = 0) {
        if (*s == 0 && t[p].end) {
            --t[p].end;
            return --t[p].freq;
        }
        if ((*s == 0 && t[p].end == 0) || t[p].to[*s - offset] == 0)
            return -1;
        int count = erase(s+1, t[p].to[*s - offset]);
        if (count == 0)
            t[p].to[*s - offset] = 0;
        if (count == -1)
            return -1;
        return --t[p].freq;
    }
} trie;
```

2.2 BIT

```
int b[N];

int update(int p, int val, int n) {
    for(;p < n; p += p & -p) b[p] += val;
}

int getsum(int p) {
    int sum = 0;
    for(; p != 0; p -= p & -p) {
        sum += b[p];
    }
    return sum;
}
```

2.3 Recursive Segment Tree

```
int t[N<<1];

void build(int n) {
    for(int i = n-1; i > 0; i--) t[i] = min(t[i<<1], t[i<<1|1]);
}

void modify(int pos, int val, int n) {
    for(t[pos += n] = val; pos != 1; pos>>=1)
        t[pos>>1] = min(t[pos], t[pos^1]);
}

int query(int l, int r, int n) { // [l, r)
    int resp = 1000000007;
    for(l += n, r += n; l < r; l >>= 1, r >>= 1) {
        if (l&1) resp = min(resp, t[l++]);
        if (r&1) resp = min(resp, t[--r]);
    }
    return resp;
}
```

2.4 Lazy Segment Tree

```
int seg[4*N];
int lazy[4*N];

void do_lazy(int root, int ll, int rl) {
    seg[root] += lazy[root];
    if (ll != rl) {
        lazy[2*root+1] += lazy[root];
        lazy[2*root+2] += lazy[root];
    }
    lazy[root] = 0;
}

int update(int root, int ll, int rl, int l, int r, int val) {
    do_lazy(root, ll, rl);
    if (r < ll || l > rl) return seg[root];
    if (ll >= l && rl <= r) {
        lazy[root] += val;
        do_lazy(root, ll, rl);
        return seg[root];
    }
    int update_left = update(2*root+1, ll, (ll+rl)/2, l, r, val);
    int update_right = update(2*root+2, (ll+rl)/2+1, rl, l, r, val);
    return seg[root] = min(update_left, update_right);
}

int query(int root, int ll, int rl, int l, int r) {
    do_lazy(root, ll, rl);
    if (r < ll || l > rl) return inf;
    if (ll >= l && rl <= r) return seg[root];
    int query_left = query(2*root+1, ll, (ll+rl)/2, l, r);
    int query_right = query(2*root+2, (ll+rl)/2+1, rl, l, r);
    return min(query_left, query_right);
}
```