ACM 模板

数据结构

并查集

```
struct dsu{
    int n;
    vector<int> fa;
    dsu(int _n) :n(_n){
        fa.resize(n + 1);
        iota(fa.begin(), fa.end(), 0);
    }
    int find(int x){
        return x == fa[x] ? x : fa[x] = find(fa[x]);
    }
    int merge(int x, int y){
        int fax = find(x), fay = find(y);
        if(fax == fay)return 0; // 一个集合
        return fa[find(x)] = find(y); // 合并到哪个集合了
    }
};
```

树状数组

```
#define lowbit(x) ((x)&(-(x)))
template<class T>
struct Fenwick_tree{
    Fenwick_tree(int size){
        n = size;
        tree.assign(n + 1, 0);
    }
    T query(int l, int r){
        auto query = [&](int pos){
            T res = 0;
            while(pos){ res += tree[pos]; pos -= lowbit(pos); }
            return res;
        };
        return query(r) - query(l - 1);
    }
    void update(int pos, T num){
        while(pos <= n){ tree[pos] += num; pos += lowbit(pos); }
    }
private:
    int n;
    vector<T> tree;
};
```

线段树

```
template <class Data,class Num>
struct Segment_Tree{
    inline void update(int l,int r,Num x){update(1,l,r,x);}
    inline Data query(int l,int r){return query(1,l,r);}
    Segment_Tree(vector<Data>& a){
        n=a.size();
        tree.assign(n*4+1,{});
        build(a,1,1,n);
    }
    private:
    int n;
    struct Tree{int l,r;Data data;};
    vector<Tree> tree;
    inline void pushup(int pos){
        tree[pos].data=tree[pos<<1].data+tree[pos<<1|1].data;
    }
    inline void pushdown(int pos){
        tree[pos<<1].data=tree[pos<<1].data+tree[pos].data.lazytag;
        tree[pos<<1|1].data=tree[pos<<1|1].data+tree[pos].data.lazytag;
        tree[pos].data.lazytag=Num::zero();
    }
    void build(vector<Data>& a,int pos,int l,int r){
        tree[pos].l=l;tree[pos].r=r;
        if(l==r){tree[pos].data=a[l-1];return;}
        int mid=(tree[pos].l+tree[pos].r)>>1;
        build(a,pos<<1,l,mid);
        build(a,pos<<1|1,mid+1,r);
        pushup(pos);
    }
    void update(int pos,int& l,int& r,Num& x){
        if(l>tree[pos].r||r<tree[pos].l)return;
        if(l<=tree[pos].l&&tree[pos].r<=r)
{tree[pos].data=tree[pos].data+x;return;}
        pushdown(pos);
        update(pos<<1,l,r,x);update(pos<<1|1,l,r,x);
        pushup(pos);
    }
    Data query(int pos,int& l,int& r){
        if(l>tree[pos].r||r<tree[pos].l)return Data::zero();
        if(l<=tree[pos].l&&tree[pos].r<=r)return tree[pos].data;
        pushdown(pos);
        return query(pos<<1,l,r)+query(pos<<1|1,l,r);
    }
};
struct Num{
    ll add;
    inline static Num zero(){return {0};}
    inline Num operator+(Num b){return {add+b.add};}
};
```

```
struct Data{
    ll sum,len;
    Num lazytag;
    inline static Data zero(){return {0,0,Num::zero()};}
    inline Data operator+(Num b){return {sum+len*b.add,len,lazytag+b};}
    inline Data operator+(Data b){return
{sum+b.sum,len+b.len,Num::zero()};}
};
```

图论

存图

```
struct Graph{
    int n;
    struct Edge{ int to, w; };
    vector<vector<Edge>> graph;
    Graph(int _n){ n = _n; graph.assign(n + 1, vector<Edge>()); };
    void add(int u, int v, int w){ graph[u].push_back({ v,w }); }
};
```

树上问题

树链剖分

```
vector<int> fa, siz, dep, son, dfn, rnk, top;
fa.assign(n + 1, 0);
siz.assign(n + 1, 0);
dep.assign(n + 1, 0);
son.assign(n + 1, 0);
dfn.assign(n + 1, 0);
rnk.assign(n + 1, 0);
top.assign(n + 1, 0);
void hld(int root){
    function<void(int)> dfs1 = [&](int t){
        dep[t] = dep[fa[t]] + 1;
        siz[t] = 1;
        for(auto& [to, w] : graph[t]){
            if(to == fa[t])continue;
            fa[to] = t;
            dfs1(to);
            if(siz[son[t]] < siz[to])son[t] = to;
            siz[t] += siz[to];
        }
    }; dfs1(root);
    int dfn_tail = 0;
    for(int i = 1; i <= n; i++)top[i] = i;
    function<void(int)> dfs2 = [&](int t){
        dfn[t] = ++dfn_tail;
```

```
        rnk[dfn_tail] = t;
        if(!son[t])return;
        top[son[t]] = top[t];
        dfs2(son[t]);
        for(auto& [to, w] : graph[t]){
            if(to == fa[t] || to == son[t])continue;
            dfs2(to);
        }
    }; dfs2(root);
}

int lca(int x, int y){
    while(top[x] != top[y]){
        if(dep[top[x]] < dep[top[y]])swap(x, y);
        x = fa[top[x]];
    }
    if(dep[x] < dep[y])swap(x, y);
    return y;
};
```

强连通分量

```
void tarjan(Graph& g1, Graph& g2){
    int dfn_tail = 0, cnt = 0;
    vector<int> dfn(g1.n + 1, 0), low(g1.n + 1, 0), exist(g1.n + 1, 0),
belong(g1.n + 1, 0);
    stack<int> sta;
    function<void(int)> dfs = [&](int t){
        dfn[t] = low[t] = ++dfn_tail;
        sta.push(t); exist[t] = 1;
        for(auto& [to] : g1.graph[t]){
            if(!dfn[to]){
                dfs(to);
                low[t] = min(low[t], low[to]);
            }
            else if(exist[to])low[t] = min(low[t], dfn[to]);
        }
        if(dfn[t] == low[t]){
            cnt++;
            while(int temp = sta.top()){
                belong[temp] = cnt;
                exist[temp] = 0;
                sta.pop();
                if(temp == t)break;
            }
        }
    };
    for(int i = 1; i <= g1.n; i++)if(!dfn[i])dfs(i);
    g2 = Graph(cnt);
    for(int i = 1; i <= g1.n; i++)g2.w[belong[i]] += g1.w[i];
    for(int i = 1; i <= g1.n; i++)
```

```
        for(auto& [to] : g1.graph[i])
            if(belong[i] != belong[to])g2.add(belong[i], belong[to]);
}
```

拓扑排序

```
void toposort(Graph& g, vector<int>& dis){
    vector<int> in(g.n + 1, 0);
    for(int i = 1; i <= g.n; i++)
        for(auto& [to] : g.graph[i])in[to]++;
    queue<int> que;
    for(int i = 1; i <= g.n; i++)
        if(!in[i]){
            que.push(i);
            dis[i] = g.w[i]; // dp
        }
    while(!que.empty()){
        int u = que.front(); que.pop();
        for(auto& [to] : g.graph[u]){
            in[to]--;
            dis[to] = max(dis[to], dis[u] + g.w[to]); // dp
            if(!in[to])que.push(to);
        }
    }
}
```

字符串

哈希

```
constexpr int N = 2e6;
constexpr ll mod[2] = { 2000000011, 2000000033 }, base[2] = { 20011,20033
};
vector<array<ll, 2>> pow_base(N);

pow_base[0][0] = pow_base[0][1] = 1;
for(int i = 1; i < N; i++){
    pow_base[i][0] = pow_base[i - 1][0] * base[0] % mod[0];
    pow_base[i][1] = pow_base[i - 1][1] * base[1] % mod[1];
}

struct Hash{
    int size;
    vector<array<ll, 2>> hash;
    Hash(){}
    Hash(const string& s){
        size = s.size();
        hash.resize(size);
        hash[0][0] = hash[0][1] = s[0];
```

```
        for(int i = 1; i < size; i++){
            hash[i][0] = (hash[i - 1][0] * base[0] + s[i]) % mod[0];
            hash[i][1] = (hash[i - 1][1] * base[1] + s[i]) % mod[1];
        }
    }
    array<ll, 2> operator[](const array<int, 2>& range)const{
        int l = range[0], r = range[1];
        if(l == 0)return hash[r];
        auto single_hash = [&](bool flag){
            return (hash[r][flag] - hash[l - 1][flag] * pow_base[r - l +
1][flag] % mod[flag] + mod[flag]) % mod[flag];
        };
        return { single_hash(0),single_hash(1) };
    }
};
```

manacher

```
void manacher(const string& _s, vector<int>& r){
    string s(_s.size() * 2 + 1, '$');
    for(int i = 0; i < _s.size(); i++)s[2 * i + 1] = _s[i];
    r.resize(_s.size() * 2 + 1);
    for(int i = 0, maxr = 0, mid = 0; i < s.size(); i++){
        if(i < maxr)r[i] = min(r[mid * 2 - i], maxr - i);
        while(i - r[i] - 1 >= 0 && i + r[i] + 1 <s.size() && s[i - r[i] -
1] == s[i + r[i] + 1]) ++r[i];
        if(i + r[i] > maxr) maxr = i + r[i], mid = i;
    }
}
```

计算几何

```
constexpr double PI = 3.141592653589793116;
constexpr double eps = 1e-8;
using T = double;

// 两浮点数是否相等
bool equal(const T& a, const T& b){
    return abs(a - b) < eps;
}

// 向量
struct vec{
    T x, y;
    vec() :x(0), y(0){}
    vec(const T& _x, const T& _y) :x(_x), y(_y){}

    // 模
    double length()const{
```

```
        return sqrt(x * x + y * y);
    }

    // 与x轴正方向的夹角
    double angle()const{

        double angle = atan2(y, x);
        if(angle < 0)angle += 2 * PI;
        return angle;
    }

    // 逆时针旋转
    void rotate(const double& theta){
        double temp = x;
        x = x * cos(theta) - y * sin(theta);
        y = y * cos(theta) + temp * sin(theta);
    }

    bool operator==(const vec& other)const{ return equal(x, other.x) &&
equal(y, other.y); }
    bool operator<(const vec& other)const{ return angle() == other.angle()
? x < other.x : angle() < other.angle(); }

    vec operator+(const vec& other)const{ return { x + other.x,y + other.y
}; }
    vec operator-()const{ return { -x,-y }; }
    vec operator-(const vec& other)const{ return -other + (*this); }
    vec operator*(const T& other)const{ return { other * x,other * y }; }
    T operator*(const vec& other)const{ return x * other.x + y * other.y;
}

    // 叉积 结果大于0，a在b的顺时针，小于0，a在b的逆时针，等于0共线，可能同向或反向，结
果绝对值表示 a b形成的平行四边行的面积
    T operator^(const vec& other)const{ return x * other.y - y * other.x;
}

    friend istream& operator>>(istream& input, vec& data){
        input >> data.x >> data.y;
        return input;
    }
    friend ostream& operator<<(ostream& output, const vec& data){
        output << fixed << setprecision(6);
        output << data.x << " " << data.y;
        return output;
    }
};

// 求两点间的距离
double distance(const vec& a, const vec& b){
    return (a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y);
}

// 求两向量夹角
```

```
double angle(const vec& a, const vec& b){
    double theta = abs(a.angle() - b.angle());
    if(theta > PI)theta = 2 * PI - theta;
    return theta;
}

// 计算多边形的面积，polygon里必须是存的相邻的点
T polygon_area(const vector<vec>& polygon){
    T ans = 0;
    for(int i = 1; i < polygon.size(); i++)ans += polygon[i - 1] ^
polygon[i];
    ans += polygon[polygon.size() - 1] ^ polygon[0];
    return abs(ans / 2);
}

// 直线
struct Line{
    vec point, direction;

    Line(){}
    Line(const vec& _point, const vec& _direction) :point(_point),
direction(_direction){}
};

// 两直线是否垂直
bool perpendicular(const Line& a, const Line& b){
    return a.direction * b.direction == 0 ? true : false;
}

// 两直线是否平行
bool parallel(const Line& a, const Line& b){
    return (a.direction ^ b.direction) == 0 ? true : false;
}

// 两直线交点
vec intersection(const T& A, const  T& B, const T& C, const T& D, const T&
E, const T& F){
    return { (B * F - C * E) / (A * E - B * D),(C * D - A * F) / (A * E -
B * D) };
}

// 两直线交点
vec intersection(const Line& a, const Line& b){
    return intersection(a.direction.y, -a.direction.x, a.direction.x *
a.point.y - a.direction.y * a.point.x,
        b.direction.y, -b.direction.x, b.direction.x * b.point.y -
b.direction.y * b.point.x);
}
```