算法

倍增

LAC

```cpp
const int MAX = 1e6;

// dep[i]表示节点i的深度
int dep[MAX];
// dp[i][j]表示节点i第2^j的父节点
int dp[MAX][32];
// 邻接表存树
vector<int> s[MAX];

//时间复杂度O(n)
void dfs(int v, int fa) {
    dep[v] = dep[fa] + 1;
    dp[v][0] = fa;
    //dp[i][j]代表第i个节点的第2^i个祖先
    //第i个节点的第2^i个祖先是第i个节点的第2^(i-1)个节点的2^(i-1)个节点
    //递推公式:dp[i][j]=dp[dp[i][j-1]][j-1];
    for (int i = 1; (1 << i) <= dep[v]; ++i)
        dp[v][i] = dp[dp[v][i - 1]][i - 1];
    //遍历子节点
    for (int i = 0; i < s[v].size(); ++i) {
        //不能回到父节点
        if (s[v][i] == fa)continue;
        dfs(s[v][i], v);
    }
}

int lca(int x, int y) {
    //将深度较大的节点换到x上，就不用分类讨论了
    if (dep[x] < dep[y])swap(x, y);
    //计算出两个节点的高度差，将两个节点移到同一层
    int tmp = dep[x] - dep[y];
    //移动的过程和快速幂类似，将高度差按二进制位分解
    for (int i = 0; tmp; ++i) {
        //二进制位为1的，就移动相应步
        if ((tmp & (1 << i))) {
```

```
            x = dp[x][i];
            tmp ^= (1 << i);
        }
    }
    if (y == x)return x;
    //两个节点一起移动
    for (int i = 29; i >= 0; --i) {
        //移动要满足移动后两节点不重合，并且在最大步数范围内
        if ((1 << i) <= dep[x] && dp[x][i] != dp[y][i]) {
            x = dp[x][i];
            y = dp[y][i];
        }
    }
    return dp[x][0];
}
```

ST 表

```cpp
const int MAX = 1e6;

// 元素个数
int n;
int a[MAX];

// st[i][j]表示从i开始2^j的区间内的最大值
int st[MAX][32];


void init() {
    //st表初始化，st[i][0]=a[i]
    for (int i = 1; i <= n; ++i)
        st[i][0] = a[i];
    //利用递推公式求解st表
    for (int j = 1; j < 30; ++j)
        for (int i = 1; i + (1 << j)-1 <= n; ++i)
            st[i][j] = max(st[i][j - 1], st[i + (1 << (j - 1))][j - 1]);
}

int ask(int l, int r) {
    int rg = r - l + 1, res = -1;
    //利用st表每次移动2^j步，找到最大值即可
    for (int i = 0; l + (1 << i) - 1 <= r; ++i)
        if (rg & (1 << i))
            res = max(res, st[l][i]),
            rg ^= (1 << i), l += (1 << i);
    return res;
}
```

分块

```cpp
const int MAX = 5e5 + 100;
typedef long long ll;
ll a[MAX], b[MAX];
int n, m;
// st[i]代表第i个块开始的位置，ed[i]表示第i个块结束的位置
int st[MAX], ed[MAX], pos[MAX];
// add整块的增量，sum维护区间和
ll add[MAX], sum[MAX];
// block 块大小，t块个数
int block, t;
// 初始化分块
void init()
{
    //块大小
    block = sqrt(n);
    //块的个数
    t = (n + block - 1) / block;
    for (int i = 1; i <= n; ++i)
    {
        //第i个元素所在块
        pos[i] = (i + block - 1) / block;
        //sum维护区间和
        sum[pos[i]] += a[i];
    }
    for (int i = 1; i <= t; ++i)
    {
        ed[i] = i * block;
        st[i] = (i - 1) * block + 1;
        add[i] = 0;
    }
    //最后一个块可能不是整块
    ed[t] = n;
}

// 区间修改
void update(int l, int r, ll d)
{
    int p = pos[l], q = pos[r];
    // 修改区间在同一个块内
    if (p == q)
    {
```

```cpp
        sum[p] += (r - l + 1) * d;
        for (int i = l; i <= r; ++i)
            a[i] += d;
    }
    else
    {
        // 修改整块
        for (int i = p + 1; i <= q - 1; ++i)
            add[i] += d;
        // 修改左边多余部分
        for (int i = l; i <= ed[p]; ++i)
            a[i] += d, sum[p] += d;
        // 修改右边多余部分
        for (int i = st[q]; i <= r; ++i)
            a[i] += d, sum[q] += d;
    }
}

// 区间查询
ll ask(int l, int r)
{
    int p = pos[l], q = pos[r];
    ll ans = 0;
    // 查询区间在同一个块内
    if (p == q)
    {
        for (int i = l; i <= r; ++i)
            ans += a[i] + add[p];
    }
    else
    {
        // 查询整块
        for (int i = p + 1; i <= q - 1; ++i)
            ans += sum[i] + add[i] * (ed[i] - st[i] + 1);

        // 查询两边多余部分
        for (int i = l; i <= ed[p]; ++i)
            ans += a[i] + add[p];

        for (int i = st[q]; i <= r; ++i)
            ans += a[i] + add[q];
    }
```

```
    return ans;
}
```

莫队

```cpp
const int MAX = 5e4 + 100;
typedef long long ll;
const ll mod = 1e9 + 7;

ll a[MAX];
int n, m, k;
ll ans[MAX];        // 记录对应编号的最终答案
ll cnt[MAX];        // 记录对应数字出现个数
ll res = 0;         // 当前区间的答案
struct S            // 记录询问
{
  int l, r, id;   // l询问左端点，r询问右端点，id询问编号
} s[MAX];
int pos[MAX];       // 记录下标为x所在的块

// 将x位置对答案的影响加到当前区间内
inline void add(int x){res += 2 * cnt[a[x]] + 1, cnt[a[x]]++;}

// 将x位置对答案的影响从当前区间减去
inline void sub(int x){res += 1 - 2 * cnt[a[x]], cnt[a[x]]--;}
void solve()
{
  cin >> n >> m >> k;
    // 对下标分块
  int block = sqrt(n);
  for (int i = 1; i <= n; ++i)
  {
    cin >> a[i];
    pos[i] = (i + block - 1) / block;
  }
    // 记录查询
  for (int i = 1; i <= m; ++i)
  {
    cin >> s[i].l >> s[i].r;
    s[i].id = i;
  }

    // 这是莫队算法与暴力法唯一不同的地方
    // 对查询按莫队算法排序
  sort(s + 1, s + 1 + m, [](S &a, S &b)
    { return pos[a.l] == pos[b.l] ? a.r < b.r : pos[a.l] < pos[b.l]; });
```

```cpp
    int l = 1, r = 0;    // l,r维护当前所在区间
    for (int i = 1; i <= m; ++i)
    {
          // 移动区间完成查询
      while (s[i].l < l)add(--l);
      while (s[i].l > l)sub(l++);
      while (s[i].r > r)add(++r);
      while (s[i].r < r)sub(r--);
      ans[s[i].id] = res; //记录答案
    }
    for (int i = 1; i <= m; ++i)
      cout << ans[i] << '\n';
}
```

图论

拓扑排序

```
void topo()
{
    queue<int> que;
    vector<int> ans;    //记录拓扑序
    // 将一开始入度为0的点加入到队列中
    for(int i = 1;i <= n;++i)
        if(!deg[i])
        {
            que.push(i);
            ans.push_back(i);
        }

    while(!que.empty())
    {
        int u = que.front();
        que.pop();
        for(auto& v:g[u])
        {
            // 终点的入度减1，相当于删掉这条边
            deg[v]--;

            // 删除之后入度为0，说明v已经没有前驱将其加入队列和拓扑序中
            if(!deg[v])
            {
                que.push(v);
                ans.push_back(v);
            }
        }
    }
}
```

最小生成树

kruskal

```cpp
int n, m;
// 并查集
int fa[MAX];
// 边定义，采用直接存边的方式储存图
struct ed
{
    int u, v, w;
    bool operator<(ed &y)
    {
        return w < y.w;
    }
} edge[MAX];
// 并查集实现
void init()
{
    for (int i = 0; i < n; ++i)
        fa[i] = i;
}
int ask(int x)
{
    if (fa[x] == x)
        return x;
    return fa[x] = ask(fa[x]);
}
void merge(int x, int y)
{
    fa[ask(x)] = ask(y);
}

// Kruskal算法
void solve()
{
    cin >> n >> m;
    // 输入边
    for (int i = 0; i < m; ++i)
        cin >> edge[i].u >> edge[i].v >> edge[i].w;
    // 对边权进行排序
    sort(edge, edge + m);
    init();
```

```cpp
    // cnt记录加入边的数量
    // ans记录最后边权和
    int ans = 0, cnt = 0;
    for (int i = 0; i < m && cnt < n - 1; ++i)
    {
        // 不连通的两点才能连边
        if (ask(edge[i].u) != ask(edge[i].v))
        {
            merge(edge[i].u, edge[i].v);
            ans += edge[i].w;
            cnt++;
        }
    }
    // 最后加入的边不足n-1说明不能形成生成树
    if (cnt < n - 1)
        cout << "orz\n";
    else
        cout << ans << '\n';
}
```

Prim

```
typedef long long ll;
typedef pair<int, int> pi;
typedef pair<ll, ll> pl;
typedef vector<int> vi;
typedef vector<char> vc;
typedef vector<pl> vp;
int n, m, u, v, w;

// 链式前向星模板
struct ed
{
    int v, w, next;
} edge[MAX];
int head[MAX], tot = 0;
void add(int uu, int vv, int ww)
{
    edge[tot].w = ww;
    edge[tot].v = vv;
    edge[tot].next = head[uu];
    head[uu] = tot++;
}

// V集合,true代表在V集合中
bool vis[MAX];

// Prim算法
void solve()
{
    memset(head, -1, sizeof head);
    memset(vis, 0, sizeof vis);
    cin >> n >> m;
    // 无向图，插入重边
    for (int i = 0; i < m; ++i)
    {
        cin >> u >> v >> w;
        add(u, v, w), add(v, u, w);
    }
    // 优先队列，STL中默认是大堆，这里改成小堆
    priority_queue<pi, vector<pi>, greater<pi>> que;
    // cnt记录插入节点的数量
    // res记录最小生成树边权和
```

```
    int cnt = 0, res = 0;
    que.push({0, 1});
    while (!que.empty())
    {
        pi now = que.top();
        que.pop();
        // 不在V集合中就加入到最小生成树中
        if (!vis[now.se])
        {
            cnt++;
            res += now.fi;
            for (int j = head[now.se]; j != -1; j = edge[j].next)
                if (!vis[edge[j].v])
                    que.push({edge[j].w, edge[j].v});
        }
        // 将点插入到V集合中
        vis[now.se] = 1;
    }
    // 能将所有顶点插入最小生成树，图才是连通的
    if (cnt < n)
        printf("orz\n");
    else
        printf("%d\n", res);
}
```

Tarjan 算法

求强连通分量:

```cpp
int dfn[N], low[N], dfncnt, s[N], in_stack[N], tp;
int scc[N], sc;  // 结点 i 所在 SCC 的编号
int sz[N];       // 强连通 i 的大小

void tarjan(int u) {
  low[u] = dfn[u] = ++dfncnt, s[++tp] = u, in_stack[u] = 1;
  for (int i = h[u]; i; i = e[i].nex) {
    const int &v = e[i].t;
    if (!dfn[v]) {
      tarjan(v);
      low[u] = min(low[u], low[v]);
    } else if (in_stack[v]) {
      low[u] = min(low[u], dfn[v]);
    }
  }
  if (dfn[u] == low[u]) {
    ++sc;
    while (s[tp] != u) {
      scc[s[tp]] = sc;
      sz[sc]++;
      in_stack[s[tp]] = 0;
      --tp;
    }
    scc[s[tp]] = sc;
    sz[sc]++;
    in_stack[s[tp]] = 0;
    --tp;
  }
}
```

求割点:

```cpp
/*
洛谷 P3388 【模板】割点（割顶）
*/
#include <iostream>
#include <vector>
using namespace std;
int n, m;  // n: 点数 m: 边数
int dfn[100001], low[100001], inde, res;
// dfn: 记录每个点的时间戳
// low: 能不经过父亲到达最小的编号，inde: 时间戳，res: 答案数量
bool vis[100001], flag[100001];  // flag: 答案 vis: 标记是否重复
vector<int> edge[100001];        // 存图用的

void Tarjan(int u, int father) {  // u 当前点的编号，father 自己爸爸的编号
  vis[u] = true;                  // 标记
  low[u] = dfn[u] = ++inde;  // 打上时间戳
  int child = 0;             // 每一个点儿子数量
  for (auto v : edge[u]) {   // 访问这个点的所有邻居 （C++11）
    if (!vis[v]) {
      child++;                       // 多了一个儿子
      Tarjan(v, u);                  // 继续
      low[u] = min(low[u], low[v]);  // 更新能到的最小节点编号
      if (father != u && low[v] >= dfn[u] && !flag[u]) {  // 主要代码
        // 如果不是自己，且不通过父亲返回的最小点符合割点的要求，并且没有被标记过
        // 要求即为：删了父亲连不上去了，即为最多连到父亲
        flag[u] = true;
        res++;  // 记录答案
      }
    } else if (v != father) {
      // 如果这个点不是自己的父亲，更新能到的最小节点编号
      low[u] = min(low[u], dfn[v]);
    }
  }
  // 主要代码，自己的话需要 2 个儿子才可以
  if (father == u && child >= 2 && !flag[u]) {
    flag[u] = true;
    res++;  // 记录答案
  }
}

int main() {
  cin >> n >> m;                  // 读入数据
  for (int i = 1; i <= m; i++) {  // 注意点是从 1 开始的
```

```cpp
    int x, y;
    cin >> x >> y;
    edge[x].push_back(y);
    edge[y].push_back(x);
  }  // 使用 vector 存图
  for (int i = 1; i <= n; i++)  // 因为 Tarjan 图不一定连通
    if (!vis[i]) {
      inde = 0;      // 时间戳初始为 0
      Tarjan(i, i);  // 从第 i 个点开始，父亲为自己
    }
  cout << res << endl;
  for (int i = 1; i <= n; i++)
    if (flag[i]) cout << i << " ";  // 输出结果
  return 0;
}
```

求割边的话将 `low[v] >= dfn[u]` 改为 `>` 即可。

树链剖分

```
vector<int> sz(n + 1); // 记录子树大小
vevtor<int> fa(n + 1);  // 记录父节点
vector<int> dep(n + 1); // 记录深度
vector<int> hson(n + 1);// 记录重儿子
function<void(int,int)> dfs1 = [&](int u,int ffa)
{
    fa[u] = ffa;
    dep[u] = dep[ffa] + 1;
    sz[u] = 1;
    for(auto& v:g[u])
    {
        if(v == ffa)continue;
        dfs1(v,u);
        sz[u] += sz[v];
        if(sz[v] > sz[hson[u]])
            hson[u] = v;
    }
};

vector<int> a(n + 1);   // 节点权值
vector<int> top(n + 1); // 节点所在链的链头
vector<int> rnk(n + 1); // 将节点原编号转化为我们赋值的新编号
vector<int> id(n + 1);  // 将节点新编号转化为原编号
int t = 0; // 已分配编号的数量
function<void(int,int,int)> dfs2 = [&](int u,int ffa,int tp)
{
    top[u] = tp;
    rnk[u] = ++t;
    id[t] = u;
    if(hson[u])d2(hson[u],u,tp);// 重链
    for(auto&v:g[u])
    {
        if(v == ffa or hson[u] == v)continue;
        dfs2(v,u,v);            // 轻链
    }
};
```

字符串

字典树

```
const int MAX = 1e6 + 100;

// 基础字典树模板
// ===========================================================================
struct Trie
{
        int tr[MAX][26];        // 记录下一个字符所在节点
        int cnt[MAX];           // cnt[p]表示以p节点为结尾的字符串出现的次数
        int tot = 0;            // 新分配的存储位置

        // 初始化，全部置零
        Trie()
        {
                memset(tr,0,sizeof(tr));
                memset(cnt,0,sizeof(cnt));
        }

        // 将字符串插入到字典树中
        void insert(const char* s)
        {
                int p = 0;
                while(*s)
                {
                        int now = *s - 'a';
                        s++;
                        // 如果节点不存在，就创建节点
                        if(!tr[p][now])
                                tr[p][now] = ++tot;

                        //移动到下一个节点
                        p = tr[p][now];
                }
                cnt[p]++;
        }

        // 询问字符串
        int ask(const char* s)
        {
```

```
            int p = 0;
            while(*s)
            {
                int now = *s - 'a';
                s++;

                //如果没有对应节点，说明字典树中没有串s
                if(!tr[p][now])
                    return 0;

                p = tr[p][now];
            }
            // 返回出现的次数
            return cnt[p];
        }
    };
    // ===========================================================================
```

01 字典树:

```cpp
#include <bits/stdc++.h>
using namespace std;
const int MAX = 1e6 + 100;

// 带删除的01字典树模板
//=============================================================================================
struct BI_Trie
{
        int tr[MAX][2];         // 记录下一个比特位所在节点
        int cnt[MAX];           // cnt[p] 表示 p 节点为根的子树的节点个数
        int tot = 0;            // 新分配的存储位置

        // 初始化，全部置零
        BI_Trie(){memset(tr,0,sizeof(tr)),memset(cnt,0,sizeof(cnt));}

        // 将数的二进制位插入到字典树中
        void insert(int x)
        {
                int p = 0;
                for(int i = 30;i >= 0;--i)
                {
                        int now = (x >> i) & 1;
                        if(!tr[p][now])
                                tr[p][now] = ++tot;
                        p = tr[p][now];
                        cnt[p]++;
                }
        }

        // 删除数
        void erase(int x)
        {
                int p = 0;
                for(int i = 30;i >= 0;--i)
                {
                        int now = (x >> i) & 1;
                        p = tr[p][now];
                        cnt[p]--;
                }
        }

        // 询问最大异或和
        int max_xor(int x)
```

```
        {
                int res = 0,p = 0;
                for(int i = 30;i >= 0;--i)
                {
                        int now = (x >> i) & 1;

                        // 判断第i位是否能位1
                        // 想要第i位为1，就要异或一个与该位不同的数，1 ^ 0 或 0 ^ 1
                        // 如果另一边存在数，就移动到另一个子树上
                        // 不存在则这一位只能是0，就继续向下
                        if(tr[p][now ^ 1] && cnt[tr[p][now ^ 1]])
                        {
                                p = tr[p][now ^ 1];
                                res |= (1 << i);
                        }
                        else
                                p = tr[p][now];
                }
                return res;
        }
};
//==============================================================================================
```

字符串哈希

```
const int MAX = 1e6 + 100;

// 字符串哈希模板
// ===================================================================================
const ll p[] = { 29, 31 };
const ll M[] = { int(1e9 + 9), 998244353 };
struct SHash
{
    ll pM[2][MAX];  // 记录两组P^i
    ll hsh[2][MAX]; // 记录两组前缀哈希值
    int tot;        // 记录当前维护字符串长度

    // 初始化
    SHash()
    {
        tot = 0;
        for (int i = 0; i < 2; ++i)
        {
            pM[i][0] = 1;
            hsh[i][0] = 0;
        }
    }

    // 添加维护的字符
    void add(char ch)
    {
        ++tot;
        // 双哈希的预处理
        for (int i = 0; i < 2; ++i)
        {
            pM[i][tot] = pM[i][tot - 1] * p[i] % M[i];
            hsh[i][tot] = (hsh[i][tot - 1] * p[i] + ll(ch)) % M[i];
        }
    }

    // 获取子字符串哈希值对
    // hsh[l,r] = hsh[r] - hsh[l - 1] * pM[r - l + 1]
    pair<ll, ll> getHash(int l, int r)
    {
        if (l > r)
            return { 0, 0 };
```

```cpp
        pair<ll, ll> h;

        // 计算第一个哈希值
        h.first = (hsh[0][r] - hsh[0][l - 1] * pM[0][r - l + 1]) % M[0];
        (h.first += M[0]) %= M[0];

        // 计算第二个哈希值
        h.second = (hsh[1][r] - hsh[1][l - 1] * pM[1][r - l + 1]) % M[1];
        (h.second += M[1]) %= M[1];
        return h;
    }
};
// ================================================================================
```

马拉车

```
// 实现时我们用 mid 最右回文子串的中心，r 最右回文子串的右端点，来维护最右回文子串区间
int Manacher(string& s) {
    vector<int> d(s.size() * 2 + 3);
    //初始化字符串
    string str("@");
    for (char ch : s)
        str += "#",str+=ch;
    str += "#$";
    // 我们用 mid,r 来维护最右回文子串
    // len 是最长回文子串的长度
    int r = 0, n = str.size(), len = 0, mid = 0;
    for (int i = 1; i < n - 1; ++i) {

        // 判断 i 是否在最右回文区间内
        if (i <= r)d[i] = min(d[(mid << 1) - i], r - i + 1);
        else d[i] = 1;

        // 中心扩散法求 d[i]
        // 这样写是为了追求代码简洁，因为其他情况并不会进入循环，不影响时间复杂度
        // 就不写额外的判断来区分情况了
        while (str[i + d[i]] == str[i - d[i]])++d[i];

        //更新最右回文子串
        if (i + d[i] - 1 > r)mid = i, r = i + d[i] - 1;

        // 更新最长回文子串的长度
        len = max(len, d[i] - 1);
    }
    return len;
}
```

数据结构

并查集

```
// 并查集模板
// ==================================================================================
int fa[MAX], n, m;
void init(int n)
{
    // 初始时将节点的根节点设为自己
    for (int i = 1; i <= n; ++i)
        fa[i] = i;
}

int ask(int x)
{
    // 如果节点的父节点是其本身，说明已经到根节点了
    if (fa[x] == x)
        return x;
    // 将路径上节点的父节点都改成查询结果
    return fa[x] = ask(fa[x]);
}

void merge(int x, int y)
{
    int i = ask(x), j = ask(y);
    fa[j] = i;
}
// ==================================================================================
```

树状数组

```
const int MAX = 1e6 + 100;

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

// n代表维护的数组长度
// f[]数组代表维护的数组
int n;
int f[MAX];

// =====================================================================
// 基础树状数组
// 支持单点修改，区间查询

// 单点修改
inline void update(int x, int d)
{
    for (; x <= n; x += lowbit(x))
        f[x] += d;
}

// 查询前缀[1,x]的和
inline int ask(int x)
{
    int sum = 0;
    for (; x > 0; x -= lowbit(x))
        sum += f[x];
    return sum;
}

// 求区间[l,r]的和
inline int ask(int l, int r)
{
    return ask(r) - ask(l - 1);
}

// =====================================================================
// 差分+树状数组
// 支持区间修改，单点查询
```

```cpp
// 对差分数组单点修改
inline void update(int x, int d)
{
    for (; x <= n; x += lowbit(x))
        f[x] += d;
}

// 区间修改，即修改差分数组的两个端点
// 在区间[l,r]上加上k
inline void change(int l, int r, int k)
{
    update(l, k);
    update(r + 1, -k);
}

// 单点查询，即求差分数组前缀和
inline int ask(int x)
{
    int sum = 0;
    for (int i = x; i > 0; i -= lowbit(i))
        sum += f[i];
    return sum;
}

// ======================================================================
// 2*差分数组+数组数组
// 支持区间修改，区间查询

// 维护两个差分数组
int f1[MAX], f2[MAX];

// 区间查询
//  对两个差分数组求前缀和
inline int ask1(int x)
{
    int sum = 0;
    for (; x > 0; x -= lowbit(x))
        sum += f1[x];
    return sum;
}
inline int ask2(int x)
{
```

```
    int sum = 0;
    for (; x > 0; x -= lowbit(x))
        sum += f2[x];
    return sum;
}

// 区间查询
//ask(l,r) = sum(r) - sum(l-1)
inline int ask(int l, int r)
{
    return r * ask1(r) - ask2(r) - ((l - 1) * ask1(l - 1) - ask2(l - 1));
}

// 区间修改
// 对两个差分数组做修改
void update1(int x, int d)
{
    for (; x <= n; x += lowbit(x))
        f1[x] += d;
}
void update2(int x, int d)
{
    for (; x <= n; x += lowbit(x))
        f2[x] += d;
}
//修改区间[l,r]
void change(int l, int r, int d)
{
    update1(l, d), update1(r + 1, -d);
    update2(l, (l - 1) * d), update2(r + 1, -r * d);
}
```

树状数组维护 RMQ:

```cpp
#define MAX int(2e5+7)
typedef long long ll;

int f[MAX],a[MAX], n, m;
inline int lowbit(int x) {return x & -x;}

// 单点修改
void update(int x, int d) {
    while (x <= n) {
        f[x] = max(d,a[x]);
        // 用当前节点的子节点更新当前节点
        for (int i = 1; i < lowbit(x); i <<= 1)
            f[x] = max(f[x], f[x - i]);
        x += lowbit(x);
    }
}

// 查询区间最值
int ask(int l,int r) {
    int res = 0;
    while (l <= r) {
        // 如果当前节点维护的区间超过查询区间，就用原数组该位置的值修改答案
        if (lowbit(r) > r - l + 1)
            res = max(res, a[r]),--r;
        // 没超过就直接用当前节点的区间修改答案
        else
            res = max(res, f[r]),r -= lowbit(r);
    }
    return res;
}
```

Splay

```
struct tn
{
    int val = 0, cnt = 0;   // val 记录节点键值，cnt 记录该键值的个数
    int fa = 0, ls = 0, rs = 0; // 记录父节点和左右儿子
    int sz = 0; // 记录以该节点为根的子树中节点的个数
}f[MAX];
int tot = 0, rt = 0;

// 0 是父的左儿子，1 是父的右儿子
int get(int x, int fa)
{ return x == f[fa].ls ? 0 : 1; }

// 更新树大小
void push_up(int x)
{ [x].sz = f[f[x].ls].sz + f[f[x].rs].sz + f[x].cnt; }

// 右旋
void zig(int x)
{
    int ls = f[x].ls;
    if (f[x].fa)
    {
        if (get(x, f[x].fa))f[f[x].fa].rs = ls;
        else f[f[x].fa].ls = ls;
    }
    f[ls].fa = f[x].fa;
    f[x].fa = ls;
    f[x].ls = f[ls].rs;
    if (f[ls].rs)
        f[f[ls].rs].fa = x;
    f[ls].rs = x;
    // 更新树大小
    push_up(x), push_up(ls);
}

// 左旋
void zag(int x)
{
    int rs = f[x].rs;
    if (f[x].fa)
    {
```

```
        if (get(x, f[x].fa))f[f[x].fa].rs = rs;
        else f[f[x].fa].ls = rs;
    }
    f[rs].fa = f[x].fa;
    f[x].fa = rs;
    f[x].rs = f[rs].ls;
    if (f[rs].ls)f[f[rs].ls].fa = x;
    f[rs].ls = x;
    // 更新树大小
    push_up(x), push_up(rs);
}

// 把编号为 x 的节点转到树根
void splay(int x)
{
    for (int fa = f[x].fa; fa; fa = f[x].fa)
    {
        if (f[fa].ls == x) zig(fa);
        else zag(fa);
    }
    rt = x; // x 为新的根
}

// 返回节点编号
void find(int val)
{
    int cur = rt;
    while(cur)
    {
        if(f[cur].val == val)
        {
            splay(cur);
            return;
        }
        if(f[cur].val > val) cur = f[cur].ls;
        else cur = f[cur].rs;
    }
}

// 返回第 k 小值
int k_th(int x, int k)
{
    int ls = f[x].ls;
```

```
    if (f[ls].sz >= k) return k_th(ls, k);
    if (f[x].cnt + f[ls].sz < k) return k_th(f[x].rs, k - f[x].cnt - f[ls].sz);
    splay(x);
    return f[x].val;
}

// 插入 val
void insert(int val)
{
    // 空树就建树
    if (!rt)
    {
        rt = ++tot;
        f[rt].val = val;
        f[rt].sz = f[rt].cnt = 1;
        return;
    }
    int cur = rt;
    while (cur)
    {
        if (f[cur].val == val)
        {
            f[cur].cnt++;
            f[cur].sz++;
            splay(cur);
            return;
        }
        if (f[cur].val > val)
        {
            // 不存在 val 就先创建。
            if (!f[cur].ls)
            {
                f[cur].ls = ++tot;
                f[f[cur].ls].val = val;
                f[f[cur].ls].fa = cur;
            }
            cur = f[cur].ls;
        }
        else
        {
            if (!f[cur].rs)
            {
                f[cur].rs = ++tot;
```

```
                f[f[cur].rs].val = val;
                f[f[cur].rs].fa = cur;
            }
            cur = f[cur].rs;
        }
    }
}

// 删除 val
void remove(int val)
{
    // 先把要删的点摇到树根
    find(val);
    if(--f[rt].cnt)return;

    int ls = f[rt].ls, rs = f[rt].rs;
    f[rt].ls = f[rt].rs = 0;

    // 左树不存在，直接让右树当根。
    if (!ls)
    {
        rt = rs;
        f[rt].fa = 0;
        return;
    }
    f[ls].fa = 0;
    k_th(ls, f[ls].sz);
    f[rt].rs = rs;
    if (rs) f[rs].fa = rt;
    push_up(rt);
}

// 返回 val 的排名
// 不保证 val 在树中，查询前先插入
int rnk(int val)
{
    insert(val);
    int res = f[f[rt].ls].sz + 1;
    remove(val);
    return res;
}

// val 的前驱和后继
```

```cpp
// 不保证 val 存在与树中，使用前先要将 val 插入
int pre(int val)
{
    insert(val);
    int res = k_th(f[rt].ls, f[f[rt].ls].sz);
    remove(val);
    return res;
}
int nxt(int val)
{
    insert(val);
    int res = k_th(f[rt].rs, 1);
    remove(val);
    return res;
}
```

Splay 实现文艺平衡树

```
struct tn
{
    int val = 0;
    int fa = 0, ls = 0, rs = 0;
    int sz = 0;
    int tag = 0;    // 记录区间是否反转
}f[MAX];
int tot = 0, rt = 0, n, m;

// 0 是父的左儿子，1 是父的右儿子
int get(int x, int fa)
{ return x == f[fa].ls ? 0 : 1; }

void push_up(int x)
{ f[x].sz = f[f[x].ls].sz + f[f[x].rs].sz + 1; }

// 给节点打上反转标记
void add_tag(int x)
{
    swap(f[x].ls, f[x].rs);
    f[x].tag ^= 1;
}

// 下传标记
void push_down(int x)
{
    if (f[x].tag)
    {
        add_tag(f[x].ls);
        add_tag(f[x].rs);
        f[x].tag = 0;
    }
}

// 右旋
void zig(int x)
{
    int ls = f[x].ls;
    if (f[x].fa)
    {
        if (get(x, f[x].fa))f[f[x].fa].rs = ls;
```

```
        else f[f[x].fa].ls = ls;
    }
    f[ls].fa = f[x].fa;
    f[x].fa = ls;
    f[x].ls = f[ls].rs;
    if (f[ls].rs)
        f[f[ls].rs].fa = x;
    f[ls].rs = x;

    push_up(x), push_up(ls);
}

// 左旋
void zag(int x)
{
    int rs = f[x].rs;
    if (f[x].fa)
    {
        if (get(x, f[x].fa))f[f[x].fa].rs = rs;
        else f[f[x].fa].ls = rs;
    }
    f[rs].fa = f[x].fa;
    f[x].fa = rs;
    f[x].rs = f[rs].ls;
    if (f[rs].ls)f[f[rs].ls].fa = x;
    f[rs].ls = x;
    push_up(x), push_up(rs);
}

// 把 x 转到树根
void splay(int x)
{
    for (int fa = f[x].fa; fa; fa = f[x].fa)
    {
        if (f[fa].ls == x) zig(fa);
        else zag(fa);
    }
    rt = x;
}

// 找到前面有 x 个数字的节点
// 这里查找不能像一般的平衡树那样比较键值
// 而是要看前面数字的个数
```

```
void find(int x)
{
    int cur = rt;
    while (cur)
    {
        push_down(cur);
        if (f[f[cur].ls].sz == x)
        {
            splay(cur);
            return;
        }
        if (f[f[cur].ls].sz > x) cur = f[cur].ls;
        else  x -= f[f[cur].ls].sz + 1, cur = f[cur].rs;
    }
}

// 插入 val，和普通平衡树相同
void insert(int val)
{
    // 空树就建树
    if (!rt)
    {
        rt = ++tot;
        f[rt].val = val;
        f[rt].sz = 1;
        return;
    }
    int cur = rt;
    while (cur)
    {
        if (f[cur].val == val)
        {
            f[cur].sz++;
            splay(cur);
            return;
        }
        if (f[cur].val > val)
        {
            // 不存在 val 就先创建。
            if (!f[cur].ls)
            {
                f[cur].ls = ++tot;
                f[f[cur].ls].val = val;
```

```
                f[f[cur].ls].fa = cur;
            }
            cur = f[cur].ls;
        }
        else
        {
            if (!f[cur].rs)
            {
                f[cur].rs = ++tot;
                f[f[cur].rs].val = val;
                f[f[cur].rs].fa = cur;
            }
            cur = f[cur].rs;
        }
    }
}


void reverse(int l, int r)
{
    // 先把 r + 1 旋到树根，再把 l - 1 旋到树根，就能得到想要的结果
    find(r + 1); find(l - 1);
    add_tag(f[f[rt].rs].ls);
}

void dfs(int x)
{
    if (!x)return;
    // dfs 的时候也要记得下传标记
    push_down(x);
    dfs(f[x].ls);
    // 我们插入了 0 和 n + 1，输出时不能输出。
    if (f[x].val <= n and f[x].val > 0)cout << f[x].val << ' ';
    dfs(f[x].rs);
}
```

吉如一线段树

有一个长度为 n 的序列 a 。我们使用 a_i 来表示此序列中的 $i-th$ 元素。应对此序列执行以下三种类型的操作。

$0\ \ x\ \ y\ \ t$： 对于每个 $x \le i \le y$ ，我们使用 $min(a_i, t)$ 来替换原来的 a_i 的值。

$1\ \ x\ \ y$： 打印 a_i 的最大值 $x \le i \le y$.

$2\ \ x\ \ y$： 打印 a_i 的总和 $x \le i \le y$.

线段树解题，解题要用到 $Lazy - Tag$ 标记来实现高效的修改。如何设计标记就是解题的关键。吉如一在其论文中提到的解决方法定义的四个标记，巧妙的将区间最值和区间和结合起来。

对于线段树的每个节点，我们定义四个标记；区间和 sum、区间最大值 ma，区间严格次大值 se，最大值个数 cnt 。

当我们要用 $min(a_i, x)$对区间$[l, r]$进行修改时，在线段树上定位到对应区间后，有以下三种情况：

- 当 $ma \le x$ 时，这次修改不影响节点，不进行修改。
- 当 $se < x < ma$ 时，这次修改值影响最大值，更新 $sum = sum - cnt \times (ma - x)$，并且修改最大值 $ma = x$。
- 当 $se \ge x$ 时，无法直接修改这个节点，递归它的左右儿子。

上述算法的关键是严格次大值 se，他起到剪枝的作用。这样看似很暴力的操作，实际复杂度并不是很高，在吉如一的国家队论文中进行详细的证明，其时间复杂度是 $O(m \log n)$ 。

```cpp
// 定义线段树
struct
{
    ll l, r, sum, cnt, ma, se;
} f[MAX << 2];
// 求左右子节点的函数
inline int ls(int k) { return k << 1; }
inline int rs(int k) { return k << 1 | 1; }
inline int md(int l, int r) { return (l + r) >> 1; }
int n, m;
// 合并区间，维护四个标记
inline void push_up(int k)
{
    f[k].sum = f[ls(k)].sum + f[rs(k)].sum;  // 维护区间和
    f[k].ma = max(f[ls(k)].ma, f[rs(k)].ma); // 维护区间最大值
    if (f[ls(k)].ma == f[rs(k)].ma)
    { // 维护区间次大值和最大值个数
        f[k].se = max(f[ls(k)].se, f[rs(k)].se);
        f[k].cnt = f[ls(k)].cnt + f[rs(k)].cnt;
    }
    else
    {
        f[k].se = max(f[ls(k)].se, f[rs(k)].se);
        f[k].se = max(f[k].se, min(f[ls(k)].ma, f[rs(k)].ma));
        f[k].cnt = f[ls(k)].ma > f[rs(k)].ma ? f[ls(k)].cnt : f[rs(k)].cnt;
    }
}
// 建树
void build(int k, int l, int r)
{
    f[k].l = l, f[k].r = r;
    if (l == r)
    {
        f[k].cnt = 1;
        f[k].se = -1;
        f[k].ma = f[k].sum = input();
        return;
    }
    int m = md(l, r);
    build(ls(k), l, m);
    build(rs(k), m + 1, r);
    push_up(k);
}
```

```cpp
// 给节点打上标记
void add_tag(int k, int x)
{
    if (x >= f[k].ma) return;
    f[k].sum -= f[k].cnt * (f[k].ma - x);
    f[k].ma = x;
}

// 下传标记
void push_down(int k)
{
    add_tag(ls(k), f[k].ma);
    add_tag(rs(k), f[k].ma);
}
// 区间最值修改
void change(int k, int l, int r, ll x)
{
    if (x >= f[k].ma)
        return; // 大于区间最大值，直接退出递归
    if (l <= f[k].l && r >= f[k].r && x > f[k].se)
    { // 大于严格次大值，可以对区间进行修改
        add_tag(k, x);
        return;
    }
    // 不满足上面两种情况，递归左右子节点
    push_down(k);
    int m = md(f[k].l, f[k].r);
    if (l <= m) change(ls(k), l, r, x);
    if (r > m) change(rs(k), l, r, x);
    push_up(k);
}
// 查询区间和
ll ask1(int k, int l, int r)
{
    if (l <= f[k].l && r >= f[k].r)
        return f[k].sum;
    push_down(k);
    int m = md(f[k].l, f[k].r);
    ll res = 0;
    if (l <= m) res += ask1(ls(k), l, r);
    if (r > m) res += ask1(rs(k), l, r);
    return res;
```

```
}
// 查询区间最值
ll ask2(int k, int l, int r)
{
    if (l <= f[k].l && r >= f[k].r)
        return f[k].ma;
    push_down(k);
    int m = md(f[k].l, f[k].r);
    ll res = 0;
    if (l <= m) res = ask2(ls(k), l, r);
    if (r > m) res = max(res, ask2(rs(k), l, r));
    return res;
}
```

计算几何

基础模板

```
constexpr double eps = 1e-15;
struct Point;
double abs(const Point &x);

struct Point
{
    double x, y; // 二维向量，表示一个点
    Point(double _x = 0, double _y = 0) : x(_x), y(_y) {}

    // 向量基本运算的实现+ - · *
    Point operator+(const Point &a) const { return Point{x + a.x, y + a.y}; }
    Point operator-(const Point &a) const { return Point{x - a.x, y - a.y}; }
    Point operator-() const { return Point{-x, -y}; }

    double operator|(const Point &a) const { return x * a.x + y * a.y; }  // 点乘
    double operator*(const Point &a) const { return x * a.y - y * a.x; }  // 叉乘
    Point operator*(const double a) const { return Point{x * a, y * a}; } // 向量数乘
    double pow() const { return x * x + y * y; }                          // 向量取平方

    // 求距离
    // 点到点的距离
    double disPoint(const Point &a) const
    {
        return sqrt((x - a.x) * (x - a.x) + (y - a.y) * (y - a.y));
    }

    // 点到直线的距离
    // a,b直线上两点
    double disline(const Point &a, const Point &b) const
    {
        Point ap = (*this) - a, ab = b - a;
        return abs(ap * ab) / abs(ab);
    }

    // 点到线段的距离
    // a,b线段两端点
    double disSeg(const Point &a, const Point &b) const
    {
```

```
        // 判断点和线段的位置关系
        if ((((*this) - a) | (b - a)) <= -eps || (((*this) - b) | (a - b)) <= -eps)
            return min(this->disPoint(a), this->disPoint(b));
        return disline(a, b);
    }

    // 重载 < 方便找线段交点
    bool operator<(const Point &b) const
    {
        if (y == b.y)
            return x < b.x;
        return y < b.y;
    }
};
double triangle(Point &a, Point &b, Point &c) { return (b - a) * (b - c) / 2.0; } // 求三角形面积
double abs(const Point &x) { return sqrt(x.pow()); }                                // 向量取模


/*
line类用来实现直线，线段
记录直线上两点或线段两个端点，和直线的方向向量
*/
struct line
{
    Point x1, x2;
    Point dVec;

    line(const Point &_x1 = {0, 0}, const Point &_x2 = {0, 0}) : x1(_x1),
                                                               x2(_x2)
    {
        dVec = x2 - x1;
    }

    // 判断直线是否平行
    bool is_parallel(const line &b) const
    {
        return abs(dVec * b.dVec) <= eps;
    }

    // 求两直线交点
    Point line_intersection(const line &b) const
    {
        // 直线平行返回无穷大
```

```cpp
        if (is_parallel(b))
            return {INT_MAX, INT_MAX};

        // 带入公式
        Point c = x2 - b.x2;
        double K = (b.dVec * c) / (dVec * b.dVec);
        Point res = x2 + dVec * K;
        return res;
    }

    // 求两线段交点
    // 交点不存在返回 INT_MAX
    Point seg_intersection(const line & b) const
    {
        // 判断平行
        if (is_parallel(b))
        {
            // 判断是否共线
            if ((b.x2 - x1) * (b.x1 - x1) == 0)
            {
                Point mi = max(min(x1, x2), min(b.x1, b.x2));
                Point ma = min(max(x1, x2), max(b.x1, b.x2));
                // 判断两线段是否重合
                if (ma.x >= mi.x && ma.y >= mi.y)
                    return mi;
            }
            return {INT_MAX, INT_MAX};
        }

        // 判断是否分别在线段的两端
        if (((b.x2 - x1) * dVec) * ((b.x1 - x1) * dVec) > 0)
            return {INT_MAX, INT_MAX};
        if (((x1 - b.x1) * b.dVec) * ((x2 - b.x1) * b.dVec) > 0)
            return {INT_MAX, INT_MAX};

        return line_intersection(b);
    }
};
```

凸包

```
void solve()
{
	int n;
	cin >> n;
	vector<Point> p(n);
	for(int i = 0;i < n;++i)
		cin >> p[i].x >> p[i].y;
	sort(p.begin(),p.end());
	vector<int> st(n);
	vector<bool> f(n);
	int tp = -1;
	for(int i = 0;i < n;++i)
	{
		while(tp > 0 and
			(p[st[tp]] - p[st[tp - 1]]) * (p[i] - p[st[tp - 1]]) <= 0)
			f[st[tp--]] = 0;
		f[i] = 1;
		st[++tp] = i;
	}
	int cnt = tp;	// 记录下凸壳点数量
	for(int i = n - 1;i >= 0;--i)
	{
		while(tp > cnt and
			(p[st[tp]] - p[st[tp - 1]]) * (p[i] - p[st[tp - 1]]) <= 0)
			f[st[tp--]] = 0;
		f[i] = 1;
		st[++tp] = i;
	}
	double s = p[st[0]].disPoint(p[st[tp - 1]]);
	for(int i = 0;i < tp - 1;++i)
		s += p[st[i]].disPoint(p[st[i + 1]]);
	printf("%.2lf\n",s);
}
```

旋转卡壳

```cpp
int sta[N], top;  // 将凸包上的节点编号存在栈里，第一个和最后一个节点编号相同

ll pf(ll x) { return x * x; }

ll dis(int p, int q) { return pf(a[p].x - a[q].x) + pf(a[p].y - a[q].y); }

ll sqr(int p, int q, int y) { return abs((a[q] - a[p]) * (a[y] - a[q])); }

ll mx;

void get_longest() {  // 求凸包直径
  int j = 3;
  if (top < 4) {
    mx = dis(sta[1], sta[2]);
    return;
  }
  for (int i = 1; i < top; ++i) {
    while (sqr(sta[i], sta[i + 1], sta[j]) <=
           sqr(sta[i], sta[i + 1], sta[j % top + 1]))
      j = j % top + 1;
    mx = max(mx, max(dis(sta[i + 1], sta[j]), dis(sta[i], sta[j])));
  }
}
```

数论

快速幂

```
typedef long long ll;
ll a, b, p;
ll qpow(ll a, ll b) {
    ll res = 1;
    while (b > 0) {
        if (b & 1) res = res * a % p;
        a = a * a % p;
        b >>= 1;
    }
    return res;
}
```

求逆元

```
const int MAX = 1e5 + 100;
int n;
// 扩展欧几里得 求逆元
// 适用于任何模数，但前提式求得数要和模数互质
// 时间复杂度为O(log n)
ll exgcd(ll a,ll b,ll& x,ll& y){
    if(!b){
        x=1,y=0;
        return a;
    }
    int d=exgcd(b,a%b,x,y),x0=x,y0=y;
    x=y0;
    y=x0-a/b*y0;
    return d;
}
ll inv1(ll a){
    ll x,y;
    exgcd(a,p,x,y);
    return (x%p+p)%p;
}

// 费马小定理求逆元
// 只适用于模数是质数的情况
// 时间复杂度为O(log n)
ll qpow(ll a,ll b){
    ll res=1;
    while(b>0){
        if(b&1)res=res*a%p;
        a=a*a%p;
        b>>=1;
    }
    return res;
}
ll inv2(ll a,ll p){
    return qpow(a,p-2);
}

// 递推大表求逆元
// 要求模数为质数
// 一般适用于在求解题目过程中要运算很多逆元，但数据的值域比较小的时候
// O(n)初始化，O(1)查询
```

```
ll inv[MAX];//记录逆元
void init(){
    inv[1]=1;
    for(int i=2;i<=n;++i)
        inv[i]=(ll)(p-p/i)*inv[p%i]%p;
}
```

筛法

欧拉筛求因数个数:

```
vector<bool> is_p(MAX,true);
vector<int> pri;
vector<int> d(MAX),cnt(MAX);
void init() {
    is_p[0] = is_p[1] = 0;
    for(int i = 2;i < MAX;++i) {
        if(is_p[i]) {
            pri.push_back(i);
            d[i] = 2;
            cnt[i] = 1;
        }
        for(auto& j : pri) {
            if(i * j > MAX) break;
            is_p[i * j] = 0;
            if(i % j == 0) {
                cnt[i * j] = cnt[i] + 1;
                d[i * j] = d[i] / cnt[i] * (cnt[i * j] + 1);
                break;
            }
            // 最小质因数为 j，且幂次为 1
            cnt[i * j] = 1;
            // 积性函数定义
            d[i * j] = d[i] * d[j];
        }
    }
}
```