常 用 算 法 模 板

Wings

2021 年 8 月 25 日

目录

0 说明

0.1 约定

- **const int** MAXN 为最大数据长度
- **const int** MAXV 为 (抽象) 图最大点数
- **const int** MAXM 为第二维数据最大长度或 (抽象) 图最大边数
- **const int** MAXQ 为最大询问
- *#ifdef GDB* 表示 GDB 调试, 从文件读取数据等
- *#ifdef NGCS* 表示输出中间变量调试 (NGCS = Nothing Gold Can Stay)
- 若无特殊情况, 数组从下标 1 开始存储数据
- 图论带边权一般以邻接表建图; 否则用向量存邻接点
- 网络流邻接表边从下标 0 开始, 因为要建反边; 否则从 1 开始

- 为节省篇幅, 代码进行部分压行

0.2 模板头

```
#include <cstdio>
#include <algorithm>
#include <iostream>
#include <cstring>
#include <cmath>
#include <vector>
#include <queue>
#include <set>
#include <map>
#include <unordered_map>
#include <string>
#define lowbit(x) (x&(-x))
#define LCH(x) (x<<1)
#define RCH(x) (x<<1|1)
using namespace std;

typedef long long LL;
typedef unsigned long long ULL;
typedef long double LD;
typedef pair<int, int> PII;
typedef vector<int> VI;

const int INTINF = 0x3f3f3f3f;
const LL INF = 0x3f3f3f3f3f3f3f3f;

const int P = 998244353;
const int MAXN = 1e5 + 10;

int T, n;

int main() {
#ifdef GDB
    freopen("X.in", "r", stdin);
    freopen("X.out", "w", stdout);
#endif
    scanf("%d", &T);
    while (T--) {
        scanf("%d", &n);
    }
    return 0;
}
```

1 数据结构

1.1 并查集

1.1.1 基本操作

```
// 可以维护其他数据，如集合大小，集合最值. 注意仅代表元的值有效
int n, fa[MAXN];
/* 初始化并查集 */
void init(int n) {
    for (int i = 1; i <= n; i++) fa[i] = i;
}
/* 查询并返回 x 所在集合的代表元; 路径压缩 */
int find(int x) {
    return x == fa[x] ? x : fa[x] = find(fa[x]);
}
/* 合并 x, y 所在集合 (x→y); 返回合并前是否在不同集合中; 无按秩合并 */
bool uni(int x, int y) {
    if (find(x) == find(y))
        return false;
    fa[find(x)] = find(y);
    return true;
}
/* 查询是否 x, y 是否在同一集合中 */
bool query(int x, int y) {
    return find(x) == find(y);
}
```

1.1.2 带边权

```
int n, fa[MAXN], d[MAXN]; // d[x] 表示 x 到 fa[x] 的距离
/* 初始化并查集 */
void init(int n) {
    for (int i = 1; i <= n; i++) fa[i] = i, d[i] = 0;
}
int find(int x) {
    if (x == fa[x])
        return x;
    int fx = fa[x]; //先记录一下 x 当前的 fa
    fa[x] = find(fa[x]);
    d[x] += d[fx]; //x 到现在的 fa (即代表元) 的路径为两段之和
    return fa[x];
}
/* 把 x 所在的集合连接到 y 所在的集合上, 且 x 到 y 的距离为 dis */
/* 返回合并前是否在不同集合中 */
bool uni(int x, int y, int dis) {
    int fx = find(x), fy = find(y);
    if (fx == fy)
        return false;
    d[fx] = dis + d[y] - d[x];
    fa[fx] = fy;
    return true;
}
/* 查询是否 x, y 是否在同一集合中 */
bool query(int x, int y) {
    return find(x) == find(y);
}
```

1.2 树状数组

```
int n, t[MAXN];
/* 单点修改 (加) */
void update(int pos, int x) {
    for (int i = pos; i <= n; i += lowbit(i))
        t[i] += x;
}
/* 查询前缀和 */
int presum(int pos) {
    int res = 0;
    for (int i = pos; i; i -= lowbit(i))
        res += t[i];
    return res;
}
/* 查询区间 [l, r] 的和 */
int query(int l, int r) {
    return presum(r) - presum(l-1);
}
```

1.3 线段树

1.3.1 区间修改 & 区间查询

```
struct Node {             // 线段节点
    int l, r, mid, len;
    LL val, tag;         // val 为线段和的值, tag 为标记值
} t[MAXN << 2]; // 线段节点要开 4 倍数据点的大小
int n;
/* 更新一条完整的线段 */
void updateNode(int u, LL d) {
    t[u].val += d * t[u].len;
    t[u].tag += d;   //打上标记, 不再往下
}
/* 将标记下放 */
void pushDown(int u) {
    updateNode(LCH(u), t[u].tag);
    updateNode(RCH(u), t[u].tag);
    t[u].tag = 0;    //取消标记
}
/* 由左右儿子线段合并更新当前线段 */
void pushUp(int u) {
    t[u].val = t[LCH(u)].val + t[RCH(u)].val;
}
/* 递归建树 */
void build(LL data[], int l, int r, int u) {
    t[u] = Node{l, r, (l+r)>>1, r-l+1};
    if (l == r) t[u].val = data[l]; //该线段只有一个点
    else {                                //分成左右两边递归求和
        build(data, l, t[u].mid, LCH(u));
        build(data, t[u].mid+1, t[u].r, RCH(u));
        pushUp(u);
    }
}
/* 初始化一棵线段树 */
void init(int _n, LL data[]) {
    build(data, 1, _n, 1);
}
/* 区间修改 */
void update(int l, int r, LL d, int u = 1) {
    if (t[u].l == l && t[u].r == r)
```

```
38          updateNode(u, d); // 找到对应线段更新
39      else {
40          pushDown(u);  // 访问 u 的儿子线段, 需要先下放标记更新
41          if (l > t[u].mid)
42              update(l, r, d, RCH(u)); //更新的线段全在该区间右边
43          else if (r <= t[u].mid)
44              update(l, r, d, LCH(u)); // 全在左边
45          else { // 跨越了左右两边
46              update(l, t[u].mid, d, LCH(u));
47              update(t[u].mid+1, r, d, RCH(u));
48          }
49          pushUp(u);  // 由儿子线段的更新后的值计算当前线段值
50      }
51  }
52  /* 区间查询 */
53  LL query(int l, int r, int u = 1) {
54      if (t[u].l == l && t[u].r == r) return t[u].val;
55      pushDown(u);
56      if (l > t[u].mid) return query(l, r, RCH(u));
57      if (r <= t[u].mid) return query(l, r, LCH(u));
58      else return query(l, t[u].mid, LCH(u)) +
59                  query(t[u].mid+1, r, RCH(u));
60  }
```

1.3.2 动态开点 & 合并

```
1   struct Node {
2       int l, r, mid, len;  // 不再利用完全二叉树的下标性质
3       int lch, rch;        // 而是直接分配下标, 从而动态开点
4       LL val, tag;         // 初始值需要满足可以在开点时赋值, 如全 0
5   } t[MAXM]; // 预先估计一下点的个数
6   int n, idx = 0, root = -1;
7   /* 动态开点, 左右儿子没开, 为 0; 返回新建节点标号 */
8   int newNode(int l, int r) {
9       t[++idx] = Node{l, r, (l+r)>>1, r-l+1};
10      return idx;
11  }
12  /* 初始化线段树, 建立一个 [1, n] 区间的节点 */
13  void init(int _n) {
14      root = newNode(1, _n);
15  }
16  /* 更新一条完整的线段 */
17  void updateNode(int u, LL d) {
18      t[u].val += d * t[u].len;
19      t[u].tag += d;   //打上标记, 不再往下
20  }
21  /* 将标记下放 */
22  void pushDown(int u) {
23      if (!t[u].lch)    // 第一次访问, 开点
24          t[u].lch = newNode(t[u].l, t[u].mid);
25      updateNode(t[u].lch, t[u].tag);
26      if (!t[u].rch)
27          t[u].rch = newNode(t[u].mid+1, t[u].r);
28      updateNode(t[u].rch, t[u].tag);
29      t[u].tag = 0;
30  }
31  /* 由左右儿子线段合并更新当前线段 */
32  void pushUp(int u) {
33      t[u].val = t[t[u].lch].val + t[t[u].rch].val;
34  }
```

```
/* 区间修改 */
void update(int l, int r, LL d, int u = -1) {
    if (u == -1) u = root;
    if (t[u].l == l && t[u].r == r)
        updateNode(u, d); // 找到对应线段更新
    else {
        pushDown(u);  // 访问 u 的儿子线段，需要先下放标记更新
        if (l > t[u].mid)
            update(l, r, d, t[u].rch); //更新的线段全在该区间右边
        else if (r <= t[u].mid)
            update(l, r, d, t[u].lch); // 全在左边
        else { // 跨越了左右两边
            update(l, t[u].mid, d, t[u].lch);
            update(t[u].mid+1, r, d, t[u].rch);
        }
        pushUp(u);  // 由儿子线段的更新后的值计算当前线段值
    }
}
/* 区间查询 */
LL query(int l, int r, int u = -1) {
    if (u == -1) u = root;
    if (t[u].l == l && t[u].r == r) return t[u].val;
    pushDown(u);
    if (l > t[u].mid) return query(l, r, t[u].rch);
    if (r <= t[u].mid) return query(l, r, t[u].lch);
    else return query(l, t[u].mid, t[u].lch) +
                query(t[u].mid+1, r, t[u].rch);
}

/* 新建一棵线段树为 x, y 两棵线段树的合并，返回合并后的节点标号 */
int merge(int x, int y, int l, int r) {
    if (!x || !y) return x | y;
    int rt = newNode(l, r);
    if (l == r) {    // 按需合并
        t[rt].val = t[x].val + t[y].val, t[rt].tag = t[x].tag + t[y].tag;
        return rt;
    }
    t[rt].lch = merge(t[x].lch, t[y].lch, l, t[rt].mid);
    t[rt].rch = merge(t[x].rch, t[y].rch, t[rt].mid+1, r);
    pushUp(rt);
    return rt;
}

/* 将 y 合并到 x 上，返回合并后的节点标号 (即 x) */
int merge(int x, int y, int l, int r) {
    if (!x || !y) return x || y;
    if (l == r) {
        t[x].val += t[y].val, t[x].tag += t[y].tag;
        return x;
    }
    t[x].lch = merge(t[x].lch, t[y].lch, l, t[x].mid);
    t[x].rch = merge(t[x].rch, t[y].rch, t[x].mid+1, r);
    pushUp(x);
    return x;
}
```

1.4 ST 表

```
int n, lg[MAXN], pw[MAXN], st[MAXN][30];  // st[i][j] 表示 [i,i+2^j) 的最值
/* 预处理出 log_2 x 和 2^k */
```

```
void init() {
    lg[0] = -1, pw[0] = 1;
    for (int i = 1; i <= n; i++) lg[i] = lg[i>>1] + 1;
    for (int i = 1; i <= lg[n]; i++) pw[i] = pw[i-1] << 1;
}
/* 建立 ST 表 */
void build(int data[]) {
    //st[i][0] 存数据 st[i][0] = [i, i + 2^0) = data[i]
    for (int i = 1; i <= n; i++) st[i][0] = data[i];
    for (int j = 1; j <= lg[n]; j++)                    // 注意先枚举 j
        for (int i = 1; i + pw[j] <= n + 1; i++) // 左闭右开, 到 n+1
            //把 [i,i+2^j) 分成 [i,i+2^{j-1}) 和 [i+2^{j-1},i+2^j) 两段
            st[i][j] = min(st[i][j-1], st[i+pw[j-1]][j-1]);
}
/* 查询 [l, r] 中的的最值 */
int query(int l, int r) {
    int k = lg[r - l + 1];
    // 由于 2^{⌊log 2x⌋} > x/2, 故前后两个长度为 2^{⌊log 2x⌋} 区间就可以取完所有的元素
    return min(st[l][k], st[r+1-pw[k]][k]); //设 r' = r + 1, 查询 [l,r')
}
```

1.5 主席树 (静态区间 k 小)

```
int nn, b[MAXN];    // nn 为不重复个数, b[] 是离散化用的数组
/* 先将数组离散化, 返回的是不重复的数字个数 */
int discretize(int _n, int data[]) {
    for (int i = 1; i <= _n; i++) b[i] = data[i];
    sort(b + 1, b + 1 + _n);
    return nn = unique(b + 1, b + 1 + _n) - b - 1;
}
/* 返回 x 离散化以后的 id 值 */
int id(int x) { return lower_bound(b + 1, b + 1 + nn, x) - b; }
struct Node {    // val 保存的是前缀和, 区间要差分一下
    int l, r, mid, lch, rch, val;
} t[MAXM];    // 点一般开 4n + mlogn 稍大一点, 如 4n + m(logn + 1)
int n, root[MAXN], idx = 0;
/* 递归建一棵空树, 返回值为节点编号 */
int build(int l, int r) {
    int u = ++idx;
    t[u] = Node{l, r, (l+r)>>1, 0, 0, 0};
    if (l < r) {
        t[u].lch = build(l, t[u].mid);
        t[u].rch = build(t[u].mid + 1, r);
    }
    return u;
}
/* 从上一棵树的对应节点拉边过来建树, 返回值为节点编号 */
int update(int pre, int pos, int x) {
    int u = ++idx;
    t[u] = t[pre];  // 复制上一棵树对应节点的所有信息
    if (t[u].l == t[u].r) t[u].val += x;
    else {
        if (pos <= t[u].mid)
            t[u].lch = update(t[pre].lch, pos, x);
        else t[u].rch = update(t[pre].rch, pos, x);
        t[u].val = t[t[u].lch].val + t[t[u].rch].val;
    }
    return u;
}
/* 递归找 [l, r] 对应区间的值 */
```

```
int qry(int lu, int ru, int k) {
    if (t[lu].l == t[lu].r) return t[lu].l;
    int x = t[t[ru].lch].val - t[t[lu].lch].val; // 差分
    if (k <= x) return qry(t[lu].lch, t[ru].lch, k);
    else return qry(t[lu].rch, t[ru].rch, k - x);
}
/* 查询 [l, r] 中第 k 小, 注意返回值 x 是离散化以后的, 需要 b[x] 得到原数 */
int query(int l, int r, int k) { return qry(root[l-1], root[r], k); }
/* 离散化, 并根据权值建树 */
void init(int _n, int data[]) {
    nn = discretize(_n, data);
    root[0] = build(1, n);
    for (int i = 1; i <= _n; i++) // 根据权值建立主席树
        root[i] = update(root[i-1], id(data[i]), 1);
}
```

1.6 FHQTreap(带区间翻转标记)

```
struct Node {    //key 为排序用的随机键, size 为子树大小
    int val, key, size, lch, rch, tag;
} t[MAXN];
int idx = 0, root;
/* 初始化, 输入种子 */
void init(int seed) {
    idx = root = 0; // 如果多个树, 则删去 idx = 0
    srand(seed);
}
/* 向上更新子树大小 */
void pushUp(int u) {
    t[u].size = t[t[u].lch].size + 1 + t[t[u].rch].size;
}
/* 标记下放; 这里写的是反转标记, 根据题目修改 */
void pushDown(int u) {
    if (t[u].tag) swap(t[u].lch, t[u].rch);
    t[t[u].lch].tag ^= t[u].tag;
    t[t[u].rch].tag ^= t[u].tag;
    t[u].tag = 0;
}
/* 合并两棵树 l, r */
int merge(int l, int r) {
    if (!l || !r) return l | r; // 有一个为空, 返回另一个
    int u;
    if (t[l].key < t[r].key) {
        pushDown(u = l); t[l].rch = merge(t[l].rch, r);
    }
    else { pushDown(u = r); t[r].lch = merge(l, t[r].lch); }
    pushUp(u);
    return u;
}
/* 根据点的值的排名分割树 u, 前 k 小的在 l 里, 其他在 r 里 */
void splitByRank(int k, int &l, int &r, int u = -1) {
    if (u == -1) u = root;
    if (!u) { l = r = 0; return; }
    pushDown(u);
    if (k <= t[t[u].lch].size) {
        r = u;
        splitByRank(k, l, t[u].lch, t[u].lch);
    }
    else {
        l = u;
```

```
43          splitByRank(k-t[t[u].lch].size-1, t[u].rch, r, t[u].rch);
44      }
45      pushUp(u);
46  }
47  /* 根据点的值分割树 u, 小于 k 的在 l 里, 其他的在 r 里 */
48  void splitByValue(int k, int &l, int &r, int u = -1) {
49      if (u == -1) u = root;
50      if (!u) { l = r = 0; return; }
51      pushDown(u);
52      if (t[u].val <= k) {
53          l = u;
54          splitByValue(k, t[u].rch, r, t[u].rch);
55      }
56      else {
57          r = u;
58          splitByValue(k, l, t[u].lch, t[u].lch);
59      }
60      pushUp(u);
61  }
62  /* 插入一个点 */
63  void insert(int x) {
64      int u, l, r;
65      t[u = ++idx] = Node{x, rand(), 1, 0, 0, 0};
66      splitByValue(x, l, r);
67      root = merge(merge(l, u), r);
68  }
69  /* 删除值为 x 的一个点 */
70  int erase(int x) {
71      int l, r, ll, rr;
72      splitByValue(x - 1, l, r);
73      splitByRank(1, ll, rr, r);
74      t[ll] = Node{0, 0, 0, 0, 0};
75      root = merge(l, rr);
76      return x;
77  }
78  /* 找树 u 中值为 x 的排名 */
79  int getRankByValue(int x, int &u) {
80      int l, r, rk;
81      splitByValue(x-1, l, r, u);
82      rk = t[l].size + 1;
83      u = merge(l, r);
84      return rk;
85  }
86  /* 找树 u 中排名为 k 的值 */
87  int getValueByRank(int k, int &u) {
88      int l, r, v, ll, rr;
89      splitByRank(k-1, l, r, u);
90      splitByRank(1, ll, rr, r);
91      v = t[ll].val;
92      u = merge(l, merge(ll, rr));
93      return v;
94  }
95  /* 找到子树 u 中 x 的前驱 */
96  int getPre(int x, int &u) {
97      int l, r, pre;
98      splitByValue(x-1, l, r, u);
99      pre = getValueByRank(t[l].size, l);
100     u = merge(l, r);
101     return pre;
102 }
```

```
/* 找到子树 u 中 x 的后继 */
int getSuc(int x, int &u) {
    int l, r, suc;
    splitByValue(x, l, r, u);
    suc = getValueByRank(1, r);
    u = merge(l, r);
    return suc;
}
/* 遍历 u */
void iterate(int u) {
    if (u) {
        pushDown(u);
        iterate(t[u].lch);
        printf("%d ", t[u].val);
        iterate(t[u].rch);
    }
}
```

2 动态规划

2.1 背包

2.1.1 01 背包 滚动数组 & 输出方案

```
for (int i = 1; i <= n; i++) for (int j = v; j >= c[i]; j--)
    if (dp[j] < dp[j-c[i]] + w[i]) {
        dp[j] = dp[j-c[i]] + w[i];
        paht[i][j] = true;
    }
int i = n, j = v;
while (i && j) {
    if (path[i][j]) {
        items.push(i);    // 方案
        j -= c[i];
    }
    i--;
}
```

2.1.2 完全背包

```
for (int i = 1; i <= n; i++) for (int j = c[i]; j <= v; j++)
    dp[j] = max(dp[j], dp[j-c[i]] + w[i]);
```

2.1.3 分组背包

```
for (int i = 0; i < n; i++) {
    vector<int> &g = group[i];
    for (int j = v; j > 0; j--) for (int k = 0; k < g.size(); k++)
        if (j-c[g[k]] >= 0) dp[j] = max(dp[j], dp[j-c[g[k]]] + w[g[k]])
}
```

2.2 最长公共子序列

```
for (int i = 1; i <= n; i++) for (int j = 1; j <= m; j++) {
    int ans = 0;
    if (a[i] == b[j]) ans = dp[i-1][j-1] + 1;
    else ans = max(dp[i-1][j], dp[i][j-1]);
    dp[i][j] = ans;
}
```

2.3 优化

2.3.1 决策单调性

```cpp
struct Node { int l, r, p; } que[MAXN];
int dp[MAXN];
int w(int l, int r) {};    //dp[r] = dp[l] + w(l, r);
void DP() {
    int tail = 0;
    que[tail++] = Node{1, n, 0};
    for (int i = 1; i <= n; i++) {
        int cur = -1, L = 0, R = tail-1;
        while (L <= R) {
            int mid = (L+R)>>1;
            if (que[mid].l <= i) cur = mid, L = mid + 1;
            else R = mid - 1;
        }
        int p = que[cur].p, cur = -1;
        dp[i] = dp[p] + w(p, i);
        while (tail && dp[i] + w(i, que[tail-1].l) <= dp[que[tail-1].p] +
        ↪ w(que[tail-1].p, que[tail-1].l))
            cur = que[--tail].l;
        L = que[tail-1].l, R = que[tail-1].r;
        while (L <= R) {
            int mid = (L+R) >> 1;
            if (dp[i] + w(i, mid) <= dp[que[tail-1].p] + w(que[tail-1].p,
            ↪ mid))
                que[tail-1].r = mid - 1, R = mid - 1, cur = mid;
            else L = mid + 1;
        }
        if (cur != -1) que[tail++] = Node{cur, n, i};
    }
}
```

3 树相关

3.1 点分治

```cpp
int sz[MAXN], vis[MAXN]; // sz[u] 为每次 dfs 计算以 u 为根的子树的大小
// descendant 保存第一次调用 dfs 的点 u 到包含 u 的树中的所有的点 v 及他们之间的距离
vector<PII> descendant;
/* 遍历, 同时计算 sz 和 descentant
/* 如果第一次调用 dis 参数为 0, dis 表示根 (第一次调用 dfs 的点) 到当前点 u 的距离
/* 不为 0 的话会在所有距离上加上这个第一次调用传入的的参数 dist_0) */
void dfs(int u, int f, int dis) {
    sz[u] = 1;
    descendant.emplace_back(u, dis);
    for (int i = head[u]; i; i = edges[i].next) {
        Edge &e = edges[i];
        if (e.to != f && !vis[e.to]) {
            dfs(e.to, u, dis + e.w);
            sz[u] += sz[e.to];
        }
    }
}
/* 求包含 u 的树的重心 */
int center(int u) {
    // 每一次调用 dfs 前都要清空 descendant
    descendant.clear(); dfs(u, 0, 0);
    int tot_sz = descendant.size();
    for (auto des : descendant) {
```

```
        int is_center = 1;
        int x = des.first;
        for (int i = head[x]; i; i = edges[i].next) {
            Edge &e = edges[i];
            if (vis[e.to]) continue; // e.to 已删除, 不考虑
            // sz[x] > sz[v] ⇒ v 是 x 的儿子, sz[v] > n/2, v 不是重心
            if (sz[x] > sz[e.to] && (sz[e.to] << 1) > tot_sz) {
                is_center = 0; break;
            }
            // sz[x] < sz[v] ⇒ v 是 x 的父亲, tot\_sz - sz[x] 是 v 向上的大小
            if (sz[x] < sz[e.to] && ((tot_sz - sz[x]) << 1) > tot_sz) {
                is_center = 0; break;
            }
        }
        if (is_center) {
            // 找到重心, 需要以重心为根, 求一下树中所有点到重心的距离
            // 这样调用完 center 后就可以保存所有点到重心的距离了
            descendant.clear(); dfs(x, 0, 0); return x;
        }
    }
    return -1;
}

void divide(int u) {
    int c = center(u); vis[c] = 1;    // 标记重心, 删去
    work(); // 已经在 center 函数中处理了 " 经过重心的'半路径'"
            // 在 work 函数中考虑如何把两条半路径组合成一条路径
            // 并考虑如何处理数据, 回答问题
    for (int i = head[c]; i; i = edges[i].next) {
        Edge &e = edges[i];
        if (vis[e.to]) continue; // e.to 已经删除, 不考虑
        iework();    // 考虑把" 假的路径" 容斥掉. 如果是路径, 那么求一次
                     // dfs(e.to, c, e.w) 得到的 descentant 就是所有
                     // " 假的路径", 这时候虽然第一次调用 dfs 的是 e.to,
                     // 但由于加上的 e.w, 也可以认为是点到 c 的距离
        divide(e.to);    // 找到剩余的点 (所在的子树), 继续处理
    }
}
```

4 图论

4.1 匈牙利二分图匹配

```
/* 寻找左边点 u 的匹配, 存图只需要左边点连向右边点的有向边即可 */
bool match(int u) {
    for (int v : G[u]) if (!vis[v]) {
        vis[v] = 1;    // 右边点 v 访问标记
        // 如果右边点没有被左边点匹配到, 让 u 和 v 匹配
        // 或者被匹配的左边点 my[v] 可以找到其他右边点的匹配
        // 那么让 my[v] 和其他右边点匹配, 让 u 和 v 匹配
        if (!my[v] || match(my[v])) {
            mx[u] = v, my[v] = u;    // 记录匹配点
            return true;
        }
    }
    return false;
}
/* 匈牙利算法, 左边点个数 nx, 右边点个数 ny */
int hungarian(int nx, int ny) {
    int cnt = 0;
```

```
    for (int u = 1; u <= nx; u++) {    // 每个左边点都尝试匹配
        for (int v = 1; v <= ny; v++) vis[v] = 0;
        if (match(u)) cnt++;
    }
    return cnt;
}
```

4.2 网络流

4.2.1 Dinic

```
int d[MAXV], cur[MAXV];
//bfs 找可前进边, 即 d[v] = d[u] + 1 且可以流的 (u, v)
bool bfs(int s, int t) {
    memset(d, INTINF, sizeof(d));
    queue<int> Q;
    Q.push(s); d[s] = 0;
    while (!Q.empty()) {
        int u = Q.front(); Q.pop();
        for (int i = head[u]; ~i; i = edges[i].next) {
            Edge &e = edges[i];
            //找还可以流的边
            if (d[e.to] >= INTINF && e.cap > e.flow) {
                d[e.to] = d[u] + 1; Q.push(e.to);
            }
        }
    }
    return d[t] < INTINF;
}
//当前点 u, flow 已经流过的流量, a 还可以继续流的流量
//dfs 找多条增广路
int dfs(int u, int a, int t) {
    //流到终点或者流不下去了
    //流到终点返回可流量, 在下面的 a -= f 时 a=0, 然后结束这个点的搜索
    if (u == t || a == 0)
        return a;
    int f = 0, flow = 0;
    //当前弧优化 &i = cur 搜完了这条路, 就不会再搜这条路了, 所以从后面开始搜
    for (int &i = cur[u]; ~i; i = edges[i].next) {
        Edge &e = edges[i];
        if (d[e.to] == d[u] + 1) {
            f = dfs(e.to, min(a, e.cap - e.flow));
            if (f > 0) {
                e.flow += f; flow += f;
                edges[i^1].flow -= f; a -= f;
                if (!a) break; //没有可以供其他路流的流量 (或者流完了)
            }
        }
    }
    if (a) //GAP 优化 如果后面的都流完了, 这个点还有流量剩余, 那么不用再搜这个点了
        d[u] = -1;
    return flow;
}
int dinic(int s, int t) {
    int flow = 0;
    while (bfs(s, t)) {
        for (int i = 1; i <= v; i++) cur[i] = head[i];
        flow += dfs(s, INTINF);
    }
    return flow;
```

```
}
```

4.2.2 费用流 EK 算法

```
int d[MAXV], a[MAXV], inq[MAXV], p[MAXV];
bool spfa(int s, int t, int &flow, int &cost) {
    memset(d, INTINF, sizeof(d)); memset(a, 0, sizeof(a));
    queue<int> Q;
    Q.push(s), inq[s] = 1;
    d[s] = 0, a[s] = INTINF, p[s] = -1;
    while (!Q.empty()) {
        int u = Q.front();
        Q.pop(), inq[u] = 0;
        for (int i = head[u]; ~i; i = edges[i].next) {
            Edge &e = edges[i];
            if (d[e.to] > d[u] + e.cost && e.cap > e.flow) {
                d[e.to] = d[u] + e.cost;
                a[e.to] = min(a[u], e.cap - e.flow);
                p[e.to] = i;
                if (!inq[e.to]) Q.push(e.to), inq[e.to] = 1;
            }
        }
    }
    if (d[t] >= INTINF) return false;
    flow += a[t];
    cost += d[t] * a[t];
    for (int i = p[t]; ~i; i = p[edges[i^1].to])
        edges[i].flow += a[t], edges[i^1].flow -= a[t];
    return true;
}
void mcmf(int s, int t, int &flow, int &cost) {
    while (spfa(s, t, flow, cost));
}
```

5 数学

5.1 取模函数

```
int pls(LL a, LL b) {
    a += a < 0 ? P : 0, b += b < 0 ? P : 0;
    return a + b >= P ? a + b - P : a + b;
}
int mult(LL a, LL b) {
    a += a < 0 ? P : 0, b += b < 0 ? P : 0;
    return a * b >= P ? a * b % P : a * b;
}
```

5.2 快速幂

```
/* power(x) 为 x 在模 P 意义下的的逆元, P 必须为质数 */
int power(int a, int b = P-2) {
    int res = 1;
    while (b) {
        if (b&1) res = mult(res, a);
        a = mult(a, a);
        b >>= 1;
    }
    return res;
}
```

5.3 多项式

5.3.1 拉格朗日插值

$$L(x)=\sum_{i=0}^{n-1}y_i\ell_i(x)$$

$$\ell_i(x)=\prod_{j=0,j\neq i}^{n-1}\frac{(x-x_j)}{(x_i-x_j)}=\frac{(x-x_0)\cdots(x-x_{i-1})(x-x_{i+1})\cdots(x-x_{n-1})}{(x_i-x_0)\cdots(x_i-x_{i-1})(x_i-x_{i+1})\cdots(x_i-x_{n-1})}$$

$$L(x)=\sum_{i=0}^{n-1}y_i\frac{\prod_{j\neq i}(x-x_j)}{\prod_{j\neq i}(x_i-x_j)}$$

```
/* 求 n 个点值确定的 n 次数界多项式拉格朗日插值在 m 处的函数值 */
/* 点 (x, y) 从下标 0 开始 */
int Lagrange(int *x, int *y, int n, int m) {
    int L = 0;
    for (int i = 0; i < n; i++) {
        int p = 1, q = 1;
        for (int j = 0; j < n; j++) if (j != i)
            p = mult(p, m - x[j]), q = mult(q, x[i] - x[j]);
        L = pls(L, mult(y[i], mult(p, power(q))));
    }
    return L;
}
```

5.3.2 横坐标连续拉格朗日插值

$$pre_i(\xi)=\prod_{j=0}^{i}(\xi-x_j),suf_i(\xi)=\prod_{j=i}^{n-1}(\xi-x_j)$$

$$\ell_i(x)=\frac{pre_{i-1}(x)\cdot suf_{i+1}(x)}{(-1)^{n-1-i}(x_i-x_0)!(x_{n-1}-x_i)!}$$

```
// finv[x] 表示 (x!)^{-1}, inv_neg_1 表示 (-1)^{-1}
// l[i] 是第 i 项拉格朗日基本多项式
int pre[MAXA], suf[MAXA], finv[MAXN], inv_neg_1, l[MAXN];
/* 预处理阶乘逆元, 注意要处理到最高次数界 */
void init(int n) {
    int facn = 1;
    for (int i = 2; i <= n; i++) facn = mult(facn, i);
    finv[n] = power(facn);
    for (int i = n-1; ~i; i--) finv[i] = mult(i+1, finv[i+1]);
    inv_neg_1 = power(-1);
}
/* 求 n 个点值确定的 n 次数界多项式拉格朗日插值在 x 处的函数值 */
/* 其中点的横坐标为 s,s+1,s+2...,s+n-1, (i, y) 从下标 s 开始 */
int Lagrange(int s, int *y, int n, int x) {
    pre[0] = x - s;
    for (int i = 1; i < n; i++) pre[i] = mult(pre[i-1], x - i - s);
    suf[n] = 1;
    for (int i = n-1; ~i; i--) suf[i] = mult(suf[i+1], x - i - s);
    l[0] = mult(suf[1], finv[n-1]);
    if ((n-1)&1) l[0] = mult(l[0], inv_neg_1);
    for (int i = 1; i < n; i++) {
        l[i] = mult(mult(pre[i-1], suf[i+1]), mult(finv[i], finv[n-1-i]));
        if ((n-1-i)&1) l[i] = mult(l[i], inv_neg_1);
    }
    int L = 0;
```

```
    for (int i = 0; i < n; i++) L = pls(L, mult(y[i], l[i]));
    return L;
}
```

5.3.3 重心拉格朗日插值 (增删点)

$$w_i = \frac{1}{\prod_{j \neq i}(x_i - x_j)}$$

$$L(x) = \frac{\sum_{i=0}^{n-1} \frac{w_i}{x - x_i} y_i}{\sum_{i=0}^{n-1} \frac{w_i}{x - x_i}}$$

```
/* 向点集 S(X, Y) 中加入点 (x, y), W 为重心权 */
void add(int x, int y, VI &X, VI &Y, VI &W) {
    int n = X.size();
    X.push_back(x), Y.push_back(y), W.push_back(1);
    for (int i = 0; i < n; i++), W[i] = mult(W[i], power(X[i] - x));
    for (int i = 0; i < n; i++), W[n] = mult(W[n], x - X[i]);
    W[n] = power(W[n]);         // 维护重心权
}
/* 计算点集 S(X, Y) 表示的多项式在 x 处的函数值, W 为重心权 */
int Lagrange(VI &X, VI &Y, VI &W, int x) {
    int L = 0, g = 0, n = X.size();
    for (int i = 0; i < n; i++) {
        // 如果是已知点, 则直接求, 否则 1/(x-x_i) 没有意义
        if (x == X[i]) return Y[i];
        int p = mult(W[i], power(x - X[i]));
        L = pls(L, mult(p, Y[i])), g = pls(g, p);
    }
    return mult(L, power(g));
}
```

6 计算几何

6.1 误差修正 (eps 和 sgn 函数)

```
const double eps = 1e-8;
int sgn(double x) {
    if (fabs(x) < eps) return 0;
    return x > 0 ? 1 : -1;
}
```

6.2 向量和点

```
struct Vec {
    double x, y;
    Vec operator + (const Vec &B) const { return Vec{x + B.x, y + B.y}; }
    Vec operator - (const Vec &B) const { return Vec{x - B.x, y - B.y}; }
    Vec operator * (const double k) const { return Vec{x * k, y * k}; }
    Vec operator / (const double k) const { return Vec{x / k, y / k}; }
    double operator * (const Vec &B) const { return x * B.x + y * B.y; }
    double operator ^ (const Vec &B) const { return x * B.y - y * B.x; }
    bool operator == (const Vec &B) const {
        return !sgn(x - B.x) && !sgn(y - B.y);
    }
    bool operator < (const Vec &V) const {
        return x == V.x ? y < V.y : x < V.x;
    }
    double length() {
```

```
        Vec A = *this;
        return sqrt(A * A);
    }
    /* 化为长度为 r 的向量 */
    Vec trunc(double r){
        double l = length();
        if(!sgn(l)) return (*this);
        r /= l;
        return Vec{x * r, y * r};
    }
    /* 逆时针旋转 90 度 */
    Vec rotleft(){ return Vec{-y, x}; }
    /* 顺时针旋转 90 度 */
    Vec rotright(){ return Vec{y, -x}; }
    /* 绕着点 P 逆时针旋转 angle */
    Vec rotate(Vec p, double angle) {
        Vec v = (*this) - p;
        double c = cos(angle), s = sin(angle);
        return Vec{p.x + v.x * c - v.y * s, p.y + v.x * s + v.y * c};
    }
};
typedef Vec Point;    // 点的记录方式和向量相同
/* 求两向量的夹角 (无向, 且夹角在 [0, PI] 之间, 弧度制) */
double angle(Vec &A, Vec &B) {
    return acos((A * B) / A.length() / B.length());
}
/* 两点距离 */
double pointDistance(Point A, Point B) { return (A-B).length(); }
/* 三角形面积 */
double triangle_area(Point A, Point B, Point C) {
    Vec v = B - A, u = C - A;
    return fabs(v ^ u) / 2;
}
```

6.3 直线和线段

```
struct Line {
    Point P; Vec v; // 点向式, 注意两点需要转化再存储
    /* 获取直线上某一点 (线段起点) */
    Point point(double t = 0) { return P + (v * t); }
    /* 获取线段终点 */
    Point endpoint() { return point(1); }
    /* 点相对直线的位置, 1 为右边, -1 为左边, 0 在直线上 */
    int pointRelative(Point Q) { return sgn((Q - P) ^ v); }
    /* 点到直线的有向距离, 正为右边, 负为左边, 0 在直线上 */
    double pointDistance(Point Q) { return ((Q - P) ^ v) / v.length(); }
    /* 点在直线上的投影 */
    Point pointProjection(Point Q) {
        return point(((Q - P) * v) / v.length() / v.length());
    }
    /* 点是否在线段上 (不包括端点) 如需包括端点, 改 < 为 <= */
    bool isContainPoint(Point Q) {
        return !pointRelative(Q) && sgn((Q-point()) * (Q-endpoint())) < 0;
    }
    /* 线段长度 */
    double length() { return v.length(); }
    /* 求点 P 关于直线的对称点 */
    Point pointSymmetry(Point P){
        Point Q = pointProjection(P);
        return Point{2 * Q.x - P.x, 2 * Q.y - P.y};
```

```
    }
};
typedef Line Segment; // 线段也用点向式存，注意两点转化
/* 直线是否相交 */
bool isLineIntersection(Line l1, Line l2) { return sgn(l1.v ^ l2.v); }
/* 直线是否平行 */
bool isLineParallel(Line l1, Line l2) {
    return !isLineIntersection(l1, l2);
}
/* 直线是否重合 */
bool isLineCoincident(Line l1, Line l2) {
    return isLineParallel(l1, l2) && !l1.pointRelative(l2.point()) &&
    ↪  !l1.pointRelative(l2.endpoint());
}
/* 求两直线的交点，调用前需保证相交 */
Point getLineIntersection(Line l1, Line l2) {
    Vec u = l1.point() - l2.point();
    double t = (l2.v ^ u) / (l1.v ^ l2.v);
    return l1.point(t);
}
/* 线段是否与直线相交（不包括端点）包括端点该 < 为 <= */
bool isSegmentIntersectLine(Segment s, Line l) {
    return l.pointRelative(s.point()) * l.pointRelative(s.endpoint()) < 0;
}
/* 线段是否相交（是否包括端点取决于上一个函数）*/
bool isSegmentIntersecting(Segment s1, Segment s2) {
    return isSegmentIntersectLine(s1, s2) && isSegmentIntersectLine(s2,
    ↪  s1);
}
```

6.4 多边形

```
struct Polygon {
    int n;        // 数据大可能需要不使用结构体
    Point points[MAXN];  // 逆时针排序
    Line lines[MAXN];    // 边
    void init(int _n, Point _ps[]) {
        n = _n;
        for (int i = 1; i <= n; i++)
            points[i] = _ps[i];
        points[0] = _ps[n];
    }
    /* 获取边 */
    void getLines() {
        for (int i = 0; i < n; i++)
            lines[i+1] = Line{points[i], points[i+1] - points[i]};
    }
    /* 以点 p0 为基点，进行极角排序的比较函数 */
    struct cmp{
        Point p;
        cmp(const Point &p0) { p = p0; }
        bool operator() (const Point &aa, const Point &bb) {
            Point a = aa, b = bb;
            int d = sgn((a - p) ^ ( b - p ));
            if (d == 0)
                return sgn(pointDistance(p, a) - pointDistance(p, b)) < 0;
            return d > 0;
        }
    };
    /* 极角排序 */
```

```
29      void norm() {
30          Point mi = points[1];
31          for (int i = 2; i <= n; i++) mi = min(mi, points[i]);
32          sort(points + 1, points + 1 + n, cmp(mi));
33      }
34      /* 求多边形有向面积 */
35      double area() {
36          double res = 0;
37          for (int i = 0; i < n; i++)
38              res += points[i] ^ points[i + 1];
39          return res / 2.0;
40      }
41      /* 求多边形周长 */
42      double circumFerence() {
43          double sum = 0;
44          for(int i = 0; i < n; i++)
45              sum += pointDistance(points[i], points[i + 1]);
46          return sum;
47      }
48      /* Graham 求凸包; 注意如果有影响, 要特判下所有点共点, 或者共线的特殊情况 */
49      void graham(Polygon &convex, bool is_norm) {
50          if (!is_norm) norm();
51          int &top = convex.n;
52          top = 0;
53          if (n == 1) {
54              top = 1, convex.points[1] = points[1];
55          }
56          else if (n == 2) {
57              top = 2, convex.points[1] = points[1], convex.points[2] =
                ↪  points[2];
58              if (convex.points[1] == convex.points[2]) top--;
59          }
60          else {
61              top = 2, convex.points[1] = points[1], convex.points[2] =
                ↪  points[2];
62              for (int i = 3; i <= n; i++){
63              while (top > 1 && sgn((convex.points[top] - convex.points[top
                ↪  - 1]) ^ (points[i] - convex.points[top - 1])) <= 0)
64                  top--;
65                  convex.points[++top] = points[i];
66              }
67              if(convex.n == 2 && (convex.points[1] == convex.points[2]))
                ↪  convex.n--;//特判
68          }
69          convex.points[0] = convex.points[convex.n];
70      }
71      /* 判断多边形是不是凸的 */
72      bool isConvex(){
73          bool s[2] = {0, 0};
74          for (int i = 0; i < n; i++) {
75              int j = (i + 1) % n;
76              int k = (j + 1) % n;
77              s[sgn((points[j] - points[i]) ^ (points[k] - points[i])) + 1]
                ↪  = true;
78              if(s[0] && s[2]) return false;
79          }
80          return true;
81      }
82      /* 点与多边形位置; 3 点上, 2 边上, 1 内部, 0 外部 */
83      int pointRelation(Point q){
```

```
        for(int i = 1; i <= n; i++) if(points[i] == q) return 3;
        getLines();
        for(int i = 1; i <= n; i++)
            if(lines[i].isContainPoint(q)) return 2;
        int cnt = 0;
        for(int i = 0; i < n; i++){
            int j = i + 1;
            int k = sgn((q - points[j]) ^ (points[i] - points[j]));
            int u = sgn(points[i].y - q.y), v = sgn(points[j].y - q.y);
            if (k > 0 && u < 0 && v >= 0) cnt++;
            if (k < 0 && v < 0 && u >= 0) cnt--;
        }
        return cnt != 0;
    }
};
```

6.5 圆

```
const double PI = 3.14159265358;
struct Circle {
    Point O; double r;  //圆心和半径
    /* 面积 */
    double area() { return PI * r * r; }
    /* 周长 */
    double circumFerence() { return 2 * PI * r; }
    /* 点和圆的关系; 0 圆上, 1 圆内, -1 圆外 */
    int pointRelation(Point P){
        double dst = pointDistance(O, P);
        if (sgn(dst - r) < 0) return 1;
        else if (!sgn(dst - r)) return 0;
        return -1;
    }
    /* 直线和圆的关系; 返回交点个数 */
    int lineRelation(Line l){
        double dst = fabs(l.pointDistance(O));
        if (sgn(dst - r) < 0) return 2;
        else if (!sgn(dst - r)) return 1;
        return 0;
    }
    /* 两圆的关系; 5 相离, 4 外切, 3 相交, 2 内切, 1 内含 */
    int circleRelation(Circle v) {
        double d = pointDistance(O, v.O);
        if (sgn(d - r - v.r) > 0) return 5;
        if (sgn(d - r - v.r) == 0) return 4;
        double l = fabs(r - v.r);
        if (sgn(d - r - v.r) < 0 && sgn(d - l) > 0) return 3;
        if (sgn(d - l) == 0) return 2;
        return 1;
    }
    /* 求两个圆的交点; 返回交点个数 */
    int getCircleIntersection(Circle v, Point &p1, Point &p2) {
        int rel = circleRelation(v);
        if (rel == 1 || rel == 5) return 0;
        double d = pointDistance(O, v.O);
        double l = (d * d + r * r - v.r * v.r) / (2 * d);
        double h = sqrt(r * r - l * l);
        Point tmp = O + (v.O - O).trunc(l);
        p1 = tmp + ((v.O - O).rotleft().trunc(h));
        p2 = tmp + ((v.O - O).rotright().trunc(h));
        if(rel == 2 || rel == 4) return 1;
```

```
43          return 2;
44      }
45      /* 求直线和圆的交点; 返回交点个数 */
46      int getLineIntersection(Line l, Point &p1, Point &p2){
47          if(!(*this).lineRelation(l)) return 0;
48          Point A = l.pointProjection(O);
49          double d = l.pointDistance(O);
50          d = sqrt(r * r - d * d);
51          if (!sgn(d)) {
52              p1 = p2 = A;
53              return 1;
54          }
55          p1 = A + l.v.trunc(d), p2 = A - l.v.trunc(d);
56          return 2;
57      }
58      /* 求过某点的切线; 返回切线条数 */
59      int getTangentLine(Point q, Line &u, Line &v) {
60          int x = pointRelation(q);
61          if(x == 2) return 0;
62          if(x == 1) {
63              v = u = Line{q, q - O.rotleft()};
64              return 1;
65          }
66          double d = O.distance(q), l = r * r / d, h = sqrt(r * r - l * l);
67          u = Line{q, O + ((q-O).trunc(l) + (q-O).rotleft().trunc(h)) - q};
68          v = Line{q, O + ((q-O).trunc(l) + (q-O).rotright().trunc(h)) - q};
69          return 2;
70      }
71      /* 求三角形 OAB 与圆相交的面积 */
72      double triangIntersectingArea(Point A, Point B) {
73          if(!sgn((O - A) ^ (O - B))) return 0.0;
74          Point q[5];
75          int len = 0;
76          q[len++] = A;
77          Line l(A, B - A);
78          Point p1,p2;
79          if (getLineIntersection(l, q[1], q[2]) == 2) {
80              if(sgn((A - q[1]) * (B - q[1])) < 0) q[len++] = q[1];
81              if(sgn((A - q[2]) * (B - q[2])) < 0) q[len++] = q[2];
82          }
83          q[len++] = B;
84          if (len == 4 && sgn((q[0] - q[1]) * (q[2] - q[1])) > 0)
85              swap(q[1], q[2]);
86          double res = 0;
87          for (int i = 0; i < len - 1; i++) {
88              if (!pointRelation(q[i]) || !pointRelation(q[i + 1])) {
89                  double arg = angle(q[i] - O, q[i + 1] - O);
90                  res += r * r * arg / 2.0;
91              }
92              else res += fabs((q[i] - O) ^ (q[i + 1] - O)) / 2.0;
93          }
94          return res;
95      }
96  };
97  /* 三角形外接圆 */
98  Circle getOuterCircle(Point A, Point B, Point C) {
99      Line u = Line((A + B) / 2, ((A + B) / 2) + ((B - A).rotleft()));
100     Line v = Line((B + C) / 2, ((B + C) / 2) + ((C - B).rotleft()));
101     Point O = get_line_intersect(u, v);
102     double r = pointDistance(A, O);
```

```
    return Circle{O, r};
}
/* 三角形内切圆 */
Circle getInnerCircle(Point A, Point B, Point C) {
    double m = atan2(B.y - A.y, B.x - A.x)
    double n = atan2(C.y - A.y, C.x - A.x);
    Line u = Line{A, Point{cos((n + m) / 2), sin((n + m) / 2)}};
    m = atan2(A.y - B.y, A.x - B.x), n = atan2(C.y - B.y, c.x - B.x);
    Line v = Line{B, Point{cos((n+m)/2),sin((n+m)/2)}}
    Point O = get_line_intersect(u, v);
    Line AB = Line{A, B - A};
    r = fabs(AB.pointDistance(O));
    return Circle{C, r};
}
/* 两圆相交面积 */
double circleIntersectingArea(Circle c1, Circle c2){
    int rel = c1.circleRelation(c2);
    if(rel >= 4) return 0.0;
    if(rel <= 2) return min(c1.area(),c2.area());
    double d = pointDistance(c1.O, c2.O);
    double hf = (c1.r + c2.r + d) / 2.0;
    double ss = 2 * sqrt(hf * (hf - c1.r) * (hf - c2.r) * (hf - d));
    double a1 = acos((c1.r*c1.r + d*d - c2.r*c2.r) / (2.0 * c1.r * d));
    a1 *= c1.r * c1.r;
    double a2 = acos((c2.r*c2.r + d*d - c1.r*c1.r) / (2.0 * c2.r * d));
    a2 *= c2.r * c2.r;
    return a1 + a2 - ss;
}
```

7 杂项

7.1 最长单调子序列

```
int len = 1;
b[0] = a[0];
for (int i = 1; i < n; i++) {
    if (a[i] >= b[len-1]) b[len++] = a[i];
    else {
        int p = lower_bound(b, b + len, a[i]);
        b[p] = a[i];
    }
}
```

7.2 莫队

7.2.1 普通莫队

```
int n, q, id[MAXN], block_size, a[MAXN];
struct Query {
    int idx, l, r;
    bool operator < (const Query &Q) const {
        // 排序规则: l 在同一块, 按 r 递增排序; 不在同一块按块编号递增排序
        return id[l] == id[Q.l] ? r < Q.r : l < Q.l;
    }
} query[MAXN];
/* 增加值, 更新维护信息 */
void add(int c) {}
/* 删除值, 更新维护信息 */
void del(int c) {}
/* 通过当前维护的信息计算当前询问的答案 */
```

```
void getAns(int idx) {}
/* 初始化，计算块大小和块编号，并离线询问排序 */
void init() {
    block_size = sqrt(n);
    for (int i = 1; i <= n; i++)
        id[i] = (i-1) / block_size + 1;
    sort(query + 1, query + q + 1);
}
void solve() {
    init();
    int l = query[1].l, r = l-1;
    for (int i = 1; i <= q; i++) {
        while (l > query[i].l) add(a[--l]);
        while (r < query[i].r) add(a[++r]);
        while (l < query[i].l) del(a[l++]);
        while (r > query[i].r) del(a[r--]);
        getAns(query[i].idx);
    }
}
```

7.2.2 带修莫队

```
// q 是询问个数，T 是修改总时间（个数），qc 是询问和修改的总数（读入）
int T = 0, n, q = 0, qc, block_size, id[MAXN], a[MAXN];
struct Query {
    int idx, t, l, r;
    bool operator < (const Query &Q) const {
        // 排序规则：l 在同一块，r 在同一块，按 t 递增排序；
        // r 不在同一块按 r 所在块编号递增排序；
        // l 不在同一块按 l 所在块编号递增排序
        return id[l] == id[Q.l] ? id[r] == id[Q.r] ? t < Q.t :
                id[r] < id[Q.r] : id[l] < id[Q.l];
    }
} query[MAXQ];
struct Modify { int pos, a; } modify[MAXQ];
/* 增加值，更新维护信息 */
void add(int c) {}
/* 删除值，更新维护信息 */
void del(int c) {}
/* 通过当前维护的信息计算当前询问的答案 */
void getAns(int idx) {}
/* 应用时间为 t 的修改，如果修改的点在当前维护的区间，则需要统计维护 */
void change(int l, int r, int t) {
    int &pos = modify[t].pos, &c = modify[t].a;
    if (l <= pos && pos <= r)
        del(a[pos]), add(c);
    swap(c, a[pos]);
}
/* 初始化，计算块大小和块编号，并离线询问排序 */
void init() {
    block_size = pow(n, 2./3.);
    for (int i = 1; i <= n; i++)
        id[i] = (i-1) / block_size + 1;
    sort(query+1, query+q+1);
}
void solve() {
    init();
    int l = query[1].l, r = l-1, t = 0;
    for (int i = 1; i <= q; i++) {
        while (l > query[i].l) add(a[--l]);
```

```
        while (r < query[i].r) add(a[++r]);
        while (l < query[i].l) del(a[l++]);
        while (r > query[i].r) del(a[r--]);
        while (t < query[i].t) change(l, r, ++t);
        while (t > query[i].t) change(l, r, t--);
        getAns(query[i].idx);
    }
}
```

7.2.3 回滚莫队

```
// 求区间重要度, 重要度: 某数出现的次数乘以他本身, 这个值的最大值
int n, q, block_size, id[MAXN], ID = 0, a[MAXN], cnt[MAXN], tmpcnt[MAXN];
LL num[MAXN], ans[MAXN];
unordered_map<int, int> mp;         // 原题数据需要离散化
/* 获得块左端点 */
int getBlockR(int block_id) { return min(block_id * block_size, n); }
/* 获得块右端点 */
int getBlockL(int block_id) { return getBlockR(block_id-1) + 1; }
struct Query {
    int idx, l, r;
    bool operator < (const Query &Q) const {
        // 排序规则: l 在同一块, 按 r 递增排序; 不在同一块按块编号递增排序
        return id[l] == id[Q.l] ? r < Q.r : id[l] < id[Q.l];
    }
} query[MAXN];
/* 初始化, 计算块大小和块编号, 并离线询问排序 */
void init() {
    block_size = sqrt(n);
    for (int i = 1; i <= n; i++)
        id[i] = (i-1) / block_size + 1;
    sort(query+1, query+1+q);
}
void solve() {
    init();
    int l = 1, r = l-1; LL mx = -1;
    for (int i = 1; i <= q; i++) {
        // l 和上一个不在同一块中, 回滚永久信息
        if (id[query[i].l] != id[query[i-1].l]) {
            mx = -1;
            while (r >= l) --cnt[a[r--]];
            l = getBlockL(id[query[i].l]+1);
            r = l-1;
        }
        // 维护永久信息
        while (r < query[i].r) {
            cnt[a[++r]]++;
            mx = max(mx, cnt[a[r]] * num[a[r]]);
        }
        // 暴力维护临时信息
        LL tmpmx = -1;
        for (int j = query[i].l; j <= min(query[i].r,
        ↪  getBlockR(id[query[i].l])); j++)
            // 注意统计临时信息要合并永久信息的部分
            tmpmx = max(tmpmx, (++tmpcnt[a[j]] + cnt[a[j]]) * num[a[j]]);
        // 回滚临时信息
        for (int j = query[i].l; j <= min(query[i].r,
        ↪  getBlockR(id[query[i].l])); j++)
            tmpcnt[a[j]]--;
        ans[query[i].idx] = max(tmpmx, mx);    // 更新答案
```

```
48          }
49  }
```

8 附录

8.1 Vim 配置

新建文件 ~/.vimrc

```
set nu                  " 显示行号
set ai                  " 自动缩进
set cindent             "C 语言自动缩进
set ts=4                "tab 宽度为 4
set sw=4                " 缩进宽度为 4

imap {}<CR> {<CR><SPACE><CR>}<UP><END><BACKSPACE>    " 映射, 方便写 (个人习惯)

" 设置编译运行的命令快捷键
map <F5> :call Compile()<CR>
map <F6> :call Run()<CR>
map <F7> :call CompileRun()<CR>

func! Compile()
    silent w
    silent !clear
    exe "!g++ % -o %< -std=c++11 -Wall -DNGCS -g"
endfunc

func! Run()
    silent !clear
    exe "!time ./%<"
endfunc

func! CompileRun()
    exe Compile()
    exe Run()
endfunc
```

8.2 Vim 录制宏

录制: 在 normal 模式下按 q, 然后再按 a-z 中的某个字符 (小写, 代表覆盖. 大写代表追加). 在 normal 模式下按 q 结束录制.

输入: 在 normal 模式下按下 @, 再按下 a-z 中的某个字符, 输入对应字符保存的宏.

8.3 GDB 命令

功能	命令
设置断点	b 行号/函数名
当条件满足时断点	b 行号/函数名 if 条件
打印变量/表达式的值	p 变量名/表达式
持续显示变量/表达式值	disp 变量名/表达式
单步运行	n
单步进入	s
继续 (下一个断点)	c
退出	q
(重新) 运行	r

8.4 对拍

8.4.1 Windows

```
@echo off
:loop
    X-data.exe > X.in
    X.exe < X.in > X.out
```

```
    X-bf.exe < X.in > X-bf.out
    fc X.out X-bf.out
    if not errorlevel 1 goto loop
    pause
goto loop
```

8.4.2 Linux

```
while true; do
    ./X-data > X.in
    ./X < X.in > X.out
    ./X-bf < X.in > X-bf.out
    if diff X.out X-bf.out; then
        printf "AC\n"
    else
        printf "WA\n"
        cat d.in
        exit 0
    fi
done
```

8.5 数学公式

8.5.1 几何

缺球体积	$V = \pi H^2 \left(R - \frac{H}{3}\right)$
两圆相交弓型高度/	$h_1 = r_1 - \frac{r_1^2 - r_2^2 + d^2}{2d}$
两球相交缺球高度	$h_2 = r_2 - \frac{r_2^2 - r_1^2 + d^2}{2d}$