算法竞赛个人模板

Cu_OH_2

2024 年 10 月 26 日

目录

1 通用

1.1 基础框架

```cpp
#include<bits/stdc++.h>
using namespace std;
using ll = long long;

void solve()
{
    return;
}

int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);
    int T = 1;
    //cin >> T;
    while (T--) solve();
    return 0;
}
```

1.2 实用代码

```cpp
// debug 常用宏
#define debug(x) cout << #x << " = " << x << endl

// 本地文件读写
freopen("A.in", "r", stdin);
freopen("A.out", "w", stdout);

// builtin 系列位运算
__builtin_ffs(x); // 最低位1是第几位 (从1开始，不存在则0)
__builtin_clz(x)/__builtin_clzll(x); // 前导高0的个数
__builtin_ctz(x)/__builtin_ctzll(x); // 末尾低0的个数
__builtin_popcount(x)/__builtin_popcountll(x); // 1的个数
__builtin_parity(x); // 1的个数的奇偶性

// 最高位 1 的位置 (从0开始，注意x不能为0)
__lg(x);

// long double 用浮点函数后面加l
sqrtl(x)/fabsl(x)/cosl(x);

// 随机数生成器 (C++11，返回unsigned/ull)
mt19937 mt(time(0));
mt19937_64 mt64(time(0));
mt();
mt64();
shuffle(v.begin(), v.end(), mt);

// 读入包含空格的一行字符串
getline(cin, str);

// 优先队列自定义比较函数
priority_queue<T, vector<T>, decltype(cmp)> pq1(cmp); // lambda函数
priority_queue<T, vector<T>, decltype(&cmp)> pq1(cmp); // 普通函数
```

1.3 编译指令

- 启用 C++14 标准：`-std=c++14`
- STL debug：`-D_GLIBCXX_DEBUG`
- 内存错误检查：`-fsanitize=address`
- 未定义行为检查：`-fsanitize=undefined`

1.4 常犯错误

- 爆 `long long`
- 数组首尾边界未初始化
- 组间数据未清空重置
- 交互题没换 `endl`
- `size()` 参与减法导致溢出
- `for(j)` 循环写成 `++i`
- 输入没写全/输入顺序错
- 输入浮点数导致超时
- `n` 和 `m` 混淆

2 动态规划

2.1 单调队列优化多重背包

- $dp_j = \max\limits_k\{dp_{j-kw_i}\}$，对于模 w_i 的每个余数维护一个单调队列
- 时间复杂度：$O(nm)$

```cpp
const int N = 100005;
const int M = 40005;

ll n, m; // 种数、容积
ll v[N], w[N], k[N]; // 价值、体积、数量
ll dp[M]; // 使用i容积的最大价值

struct Node
{
    ll key, id;
};

void solve()
{
    cin >> n >> m;
    for (int i = 1; i <= n; ++i) cin >> v[i] >> w[i] >> k[i];
    for (int i = 1; i <= n; ++i)
    {
        vector<deque<Node>> dq(w[i]);
        auto key = [&](int j) { return dp[j] - j / w[i] * v[i]; }; // dp[j]在比较
            基准下的指标
        auto join = [&](int j) // dp[j]入队
        {
            auto& q = dq[j % w[i]];
            while (q.size() && key(j) >= q.back().key) q.pop_back();
            q.push_back({ key(j),j });
            return;
        };
        for (int j = m; j >= max(0ll, m - k[i] * w[i]); --j) join(j);
        for (int j = m; j >= w[i]; --j)
        {
            auto& q = dq[j % w[i]];
            while (q.size() && q.front().id >= j) q.pop_front();
            if (j - k[i] * w[i] >= 0) join(j - k[i] * w[i]);
            dp[j] = max(dp[j], q.front().key + j / w[i] * v[i]);
        }
    }
    ll ans = 0;
    for (int i = 0; i <= m; ++i) ans = max(ans, dp[i]);
    cout << ans << '\n';
    return;
}
```

2.2 二进制分组优化多重背包

- 可以使用 bitset 继续优化
- 时间复杂度：$O(nm\log k)$

```cpp
const int N = 100005;
const int M = 40005;

struct Item
{
    ll v, w; // 价值、体积
};

ll n, m; // 种数、容积
ll dp[M]; // 使用i容积的最大价值

void solve()
{
    cin >> n >> m;
    vector<Item> items;
    ll x, y, z;
    for (int i = 1; i <= n; ++i)
    {
        ll b = 1;
        cin >> x >> y >> z;
        while (z > b)
        {
            z -= b;
            items.push_back({ x * b, y * b });
            b <<= 1;
        }
        items.push_back({ x * z, y * z });
    }
    for (auto e : items)
    {
        for (int i = m; i >= e.w; --i)
        {
            dp[i] = max(dp[i], dp[i - e.w] + e.v);
        }
    }
    ll ans = 0;
    for (int i = 0; i <= m; ++i) ans = max(ans, dp[i]);
    cout << ans << '\n';
    return;
}
```

2.3 动态 DP

- 如果转移只涉及相邻两个位置，可以尝试将转移方程表示为矩阵乘法；由于矩阵乘法满足结合律，可以用线段树维护，实现动态带修改 DP
- 时间复杂度：$O((q+n)\log n)$

```cpp
const int N = 200005;
const ll INFLL = 0x3f3f3f3f3f3f3f3f;

struct SegTree
{
   struct Node
   {
      int lef, rig;
      array<array<ll, 2>, 2> mat;
   };
   vector<Node> tree;

   void update(int src)
   {
      for (int i = 0; i < 2; ++i)
      {
         for (int j = 0; j < 2; ++j)
         {
            auto v1 = tree[src << 1].mat[i][1] + tree[src << 1 | 1].mat[1][j];
            auto v2 = tree[src << 1].mat[i][0] + tree[src << 1 | 1].mat[1][j];
            auto v3 = tree[src << 1].mat[i][1] + tree[src << 1 | 1].mat[0][j];
            tree[src].mat[i][j] = min({ v1, v2, v3 });
         }
      }
      return;
   }

   void settle(int src, ll val)
   {
      tree[src].mat[1][1] = val;
      tree[src].mat[0][0] = 0;
      tree[src].mat[0][1] = tree[src].mat[1][0] = INFLL;
   }

   SegTree(int x) { tree.resize(x * 4 + 1); }

   void build(int src, int lef, int rig, ll arr[])
   {
      tree[src].lef = lef;
      tree[src].rig = rig;
      if (lef == rig)
      {
         settle(src, arr[lef]);
         return;
      }
      int mid = lef + (rig - lef) / 2;
      build(src << 1, lef, mid, arr);
      build(src << 1 | 1, mid + 1, rig, arr);
      update(src);
      return;
   }

   void modify(int src, int pos, ll val)
   {
      if (tree[src].lef == tree[src].rig)
      {
         settle(src, val);
         return;
      }
      int mid = tree[src].lef + (tree[src].rig - tree[src].lef) / 2;
      if (pos <= mid) modify(src << 1, pos, val);
      else modify(src << 1 | 1, pos, val);
      update(src);
      return;
   }

   ll query() { return tree[1].mat[1][1] * 2; }
};

int n, q, k;
ll a[N], x;

void solve() // CF1814E
{
   cin >> n;
   for (int i = 1; i <= n - 1; ++i) cin >> a[i];
   SegTree sgt(n - 1);
   sgt.build(1, 1, n - 1, a);
   cin >> q;
   for (int i = 1; i <= q; ++i)
   {
      cin >> k >> x;
      sgt.modify(1, k, x);
      cout << sgt.query() << '\n';
   }
   return;
}
```

3 字符串

3.1 KMP 算法

- 字符串下标从 0 开始
- next_i 表示 t_i 失配时下一次匹配的位置，其中 next_n 无作用，仅构成前缀数组
- 前缀数组 $\pi_i = \text{next}_{i+1} + 1$ 代表前缀 $t_{[0,i]}$ 的最长前后缀长度
- 时间复杂度：构建 $O(m)$/匹配 $O(n)$

```cpp
struct KMP
{
   string t;
   vector<int> next;

   KMP() {}
   KMP(const string& str) { init(str); }

   void init(const string& str)
   {
      t = str;
      next.resize(t.size() + 1);
      next[0] = -1;
      for (int i = 1; i <= t.size(); ++i)
      {
         int now = next[i - 1];
         while (now != -1 && t[i - 1] != t[now]) now = next[now];
         next[i] = now + 1;
      }
      return;
   }

   int first(const string& s)
   {
      int ps = 0, pt = 0;
      while (ps < s.size())
      {
         while (pt != -1 && s[ps] != t[pt]) pt = next[pt];
         ps++, pt++;
         if (pt == t.size()) return ps - t.size();
      }
      return -1;
   }

   vector<int> every(const string& s)
   {
      vector<int> v;
      int ps = 0, pt = 0;
      while (ps < s.size())
      {
         while (pt != -1 && s[ps] != t[pt]) pt = next[pt];
         ps++, pt++;
         if (pt == t.size())
         {
            v.push_back(ps - t.size());
            pt = next[pt];
         }
      }
      return v;
   }
};
```

3.2 扩展 KMP 算法

- 字符串下标从 0 开始
- z_i 表示后缀 $t_{[i,n-1]}$ 与母串的最长公共前缀
- 该算法还可以求模式串与文本串每个后缀的 LCP
- 时间复杂度：$O(n)$

```cpp
struct ExKMP
{
   string t;
   vector<int> z;

   ExKMP(const string& str)
   {
      t = str;
      z.resize(t.size());
      z[0] = t.size();
      int l = 0, r = -1;
      for (int i = 1; i < t.size(); ++i)
      {
         if (i <= r && z[i - l] < r - i + 1) z[i] = z[i - l];
         else
         {
            z[i] = max(0, r - i + 1);
            while (i + z[i] < t.size() && t[z[i]] == t[i + z[i]]) z[i]++;
         }
         if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
      }
   }

   vector<int> lcp(const string& s)
   {
      vector<int> res(s.size());
      int l = 0, r = -1;
      for (int i = 0; i < s.size(); ++i)
      {
         if (i <= r && z[i - l] < r - i + 1) res[i] = z[i - l];
         else
         {
            res[i] = max(0, r - i + 1);
            while (i + res[i] < s.size() && res[i] < t.size() && t[res[i]] ==
                  s[i + res[i]]) res[i]++;
         }
         if (i + res[i] - 1 > r) l = i, r = i + res[i] - 1;
      }
      return res;
   }
};
```

3.3 字典树

- 每个结点代表一个前缀
- 字母表变化时需要修改 F 和 ALPSZ
- 若需要搜索整棵树，用一个数组记录出边可以降低常数
- 时间复杂度：$O(n)$

```
struct Trie
{
    static const int ALPSZ = 26;
    vector<vector<int>> trie;
    vector<int> tag;
    // vector<vector<int>> out;

    int F(char c) { return c - 'a'; }

    Trie() { init(); }

    void init()
    {
        create();
        return;
    }
    int create()
    {
        trie.push_back(vector<int>(ALPSZ));
        tag.push_back(0);
        // out.push_back(vector<int>());
        return trie.size() - 1;
    }
    void insert(const string& t)
    {
        int now = 0;
        for (auto e : t)
        {
            if (!trie[now][F(e)])
            {
                int newNode = create();
                // out[now].push_back(F(e));
                trie[now][F(e)] = newNode;
            }
            now = trie[now][F(e)];
            tag[now]++;
        }
        return;
    }
    int count(const string& pre)
    {
        int now = 0;
        for (auto e : pre)
        {
            now = trie[now][F(e)];
            if (now == 0) return 0;
        }
        return tag[now];
    }
};
```

3.4 AC 自动机

- fail 指针指向模式串前缀的最长后缀状态，每转移一次状态都需要上跳 $O(n)$ 次
- 字母表变化时需要修改 F 和 ALPSZ
- trie 图优化：建立 fail 指针时，fail 指针指向的结点可能依然失配，需要多次跳转才能到达匹配结点。可以将所有结点的空指针补全为该结点的跳转终点，此时根据 BFS 序，在计算 fail[tr[x][i]] 时，fail[x] 一定已遍历过，可以将 fail[tr[x][i]] 直接置为 tr[fail[x]][i]
- last 优化：多模式匹配时，对于文本串的每个前缀 s'，沿 fail 指针路径寻找为 s' 后缀的模式串，途中可能经过无贡献的模式串真前缀结点；使用 last 数组来跳转可以跳过真前缀结点直接到达上方第一个模式串结点。last 数组可以完美替代 fail 数组
- 树上差分优化：统计每种模式串出现次数时，每匹配到一个模式串都要向上跳转，这个过程相当于 fail 树上前缀加，可以用差分优化
- 时间复杂度：构建 $O(|\Sigma|\sum m)$/优化匹配 $O(n)$

```
struct ACAM
{
    static const int ALPSZ = 26;

    vector<vector<int>> trie; // trie树指针
    vector<int> tag; // 标记数组
    vector<int> fail; // 失配函数
    vector<int> last; // 跳转路径上一个模式串结点
    vector<int> cnt; // 计数器
    int ord; // 结点个数

    int F(char c) { return c - 'a'; }

    ACAM() { init(); }

    void init()
    {
        ord = -1;
        newNode();
    }
    int newNode()
    {
        trie.push_back(vector<int>(ALPSZ));
        tag.push_back(0);
        return ++ord;
    }
    void addPat(const string& t)
    {
        int now = 0;
        for (auto e : t)
        {
            if (!trie[now][F(e)]) trie[now][F(e)] = newNode();
            now = trie[now][F(e)];
        }
        tag[now]++;
        return;
    }
    void buildAM()
    {
        fail.resize(ord + 1);
        last.resize(ord + 1);
        cnt.resize(ord + 1);
        queue<int> q;
        for (int i = 0; i < ALPSZ; ++i)
        {
            // 第一层结点的fail指针都指向0，不需要处理
            if (trie[0][i]) q.push(trie[0][i]);
        }
        while (q.size())
        {
            int now = q.front();
            q.pop();
            for (int i = 0; i < ALPSZ; ++i)
            {
                int son = trie[now][i];
                if (son)
                {
                    fail[son] = trie[fail[now]][i];
                    if (tag[fail[son]]) last[son] = fail[son];
                    else last[son] = last[fail[son]];
                    q.push(trie[now][i]);
                }
                else trie[now][i] = trie[fail[now]][i];
            }
        }
        return;
    }
    int count(const string& s)
    {
        int now = 0, ans = 0;
        for (auto e : s)
        {
            now = trie[now][F(e)];
            int p = now;
            while (p)
            {
                ans += tag[p];
                p = last[p];
            }
        }
        return ans;
    }
};
```

3.5 后缀自动机

- 每个结点代表一系列长度连续、endpos 集合相同的子串
- 字母表变化时需要修改 F 和 ALPSZ
- 时间复杂度：$O(n)$

```
struct SAM
{
    static const int ALPSZ = 26;
    struct State
    {
        int maxlen; // 结点代表的最长子串长度
        int link; // 后缀链接，连向不在该点中的最长后缀
        vector<int> next;
        State(): maxlen(0), link(-1) { next.resize(ALPSZ); }
    };
    vector<State> node;
    vector<ll> cnt; // 子串出现次数 (endpos集合大小)
    int now; // 接收上一个字符到达的结点
    int size; // 当前结点个数

    inline int F(char c) { return c - 'a'; }

    SAM(int x)
    {
        node.resize(x * 2 + 5);
        cnt.resize(x * 2 + 5);
        now = 0; // 从根节点开始转移
        size = 1; // 建立一个代表空串的根节点
    }

    void extend(char c)
    {
        int nid = size++;
        cnt[nid] = 1;
        node[nid].maxlen = node[now].maxlen + 1;
        int p = now;
        while (p != -1 && node[p].next[F(c)] == 0)
        {
            node[p].next[F(c)] = nid;
            p = node[p].link;
        }
        if (p == -1) node[nid].link = 0; // 连向根结点
        else
```

```cpp
        {
            int ori = node[p].next[F(c)];
            if (node[p].maxlen + 1 == node[ori].maxlen) node[nid].link = ori;
            else
            {
                // 将ori结点的一部分拆出来分成新结点split
                int split = size++;
                node[split].maxlen = node[p].maxlen + 1;
                node[split].link = node[ori].link;
                node[split].next = node[ori].next;
                while (p != -1 && node[p].next[F(c)] == ori)
                {
                    node[p].next[F(c)] = split;
                    p = node[p].link;
                }
                node[ori].link = node[nid].link = split;
            }
        }
        now = nid;
        return;
    }

    void build(const string& s)
    {
        for (auto e : s) extend(e);
        return;
    }

    void DFS(int x, vector<vector<int>>& son)
    {
        for (auto e : son[x])
        {
            DFS(e, son);
            cnt[x] += cnt[e]; // link树上父节点endpos为所有子结点endpos之和
        }
        return;
    }

    void count() // 计算endpos大小
    {
        // 建立link树
        vector<vector<int>> son(size);
        for (int i = 1; i < size; ++i) son[node[i].link].push_back(i);

        // 在link树上dfs
        DFS(0, son);
        return;
    }

    ll substr() // 本质不同子串个数
    {
        ll res = 0;
        for (int i = 1; i < size; ++i)
        {
            res += node[i].maxlen - node[node[i].link].maxlen;
        }
        return res;
    }
};
```

3.6 回文自动机

- 每个结点代表一个本质不同回文串
- link 链：多字串 → 单字符 → 偶根 → 奇根
- 求每个本质不同回文子串次数：最后由母串向子串传递
- 求每个前缀的后缀回文子串个数：新建时由最长回文后缀向新串传递
- 时间复杂度：$O(n)$

```cpp
struct PAM
{
    struct State
    {
        int len; // 长度
        int link; // 最长回文后缀结点
        vector<int> next; // 两边加上某字符时对应的结点
        State() { next.resize(26); }
        State(int x, int y): len(x), link(y) { next.resize(26); }
    };
    vector<State> node;
    vector<ll> cnt; // 本质不同回文串出现次数
    int now; // 接收上一个字符到达的结点
    int size; // 当前结点个数

    inline int F(char c) { return c - 'a'; }

    PAM(int x)
    {
        node.resize(x + 3);
        node[0] = State(-1, 0); // 奇根，link无意义
        node[1] = State(0, 0); // 偶根，link指向奇根
        cnt.resize(x + 3);
        now = 0; // 第一个字符由奇根转移
        size = 2;
    }

    void build(const string& s)
    {
        auto find = [&](int x, int p) // 寻找x后缀中左方为s[p]的最长回文子串
        {
            while (p - node[x].len - 1 < 0 || s[p] != s[p - node[x].len - 1]) x =
                node[x].link;
            return x;
        };
        for (int i = 0; i < s.size(); ++i)
        {
            now = find(now, i);
            if (!node[now].next[F(s[i])]) // 对应结点不存在则需要新建
            {
                int nid = size++;
                node[nid].len = node[now].len + 2; // 新建状态结点
                node[nid].link = 1; // 若now=0，对应结点为单字符，指向偶根
                if (now) node[nid].link = node[find(node[now].link, i)].next[F(s[i
                    ])]; // 否则指向再前一个结点的扩展
                node[now].next[F(s[i])] = nid;
            }
            now = node[now].next[F(s[i])];
            cnt[now]++;
        }
        for (int i = size - 1; i >= 2; --i) cnt[node[i].link] += cnt[i]; // 数量
            由母串向子串传递
        return;
    }
};
```

3.7 Manacher 算法

- 用 n+1 个分隔符将字符串分隔可以将奇偶回文子串过程统一处理
- 时间复杂度：$O(n)$

```cpp
struct Manacher
{
    vector<int> odd, even; // 以[i]或[i,i+1]为中心的最长回文串半径
    void work(const string& s)
    {
        odd.resize(s.size());
        even.resize(s.size() - 1);
        int lef = 0, rig = -1, r;
        for (int i = 0; i < s.size(); ++i)
        {
            if (i > rig) r = 1;
            else r = min(odd[lef + rig - i], rig - i) + 1; // 利用对称位置答案
            while (i - r >= 0 && i + r < s.size() && s[i - r] == s[i + r]) r++; //
                暴力扩展
            odd[i] = --r; // 记录答案
            if (i + r > rig) lef = i - r, rig = i + r; // 扩展lef,rig范围
        }
        lef = 0, rig = -1;
        for (int i = 0; i + 1 < s.size(); ++i)
        {
            if (i + 1 > rig) r = 1;
            else r = min(even[lef + rig - i - 1], rig - i) + 1;
            while (i + 1 - r >= 0 && i + r < s.size() && s[i + 1 - r] == s[i + r])
                r++;
            even[i] = --r;
            if (i + r > rig) lef = i + 1 - r, rig = i + r;
        }
        return;
    }
};
```

3.8 最小表示法

- 求循环 rotate 得到的 n 种表示中字典序最小的一种
- 时间复杂度：$O(n)$

```cpp
const int N = 300005;

int n, a[N];

void solve()
{
    cin >> n;
    for (int i = 1; i <= n; ++i) cin >> a[i];
    auto nrm = [](int x) { return (x - 1) % n + 1; };
    int p1 = 1, p2 = 2, len = 1;
    while (p1 <= n && p2 <= n & len <= n)
    {
        if (a[nrm(p1 + len - 1)] == a[nrm(p2 + len - 1)]) len++;
        else if (a[nrm(p1 + len - 1)] < a[nrm(p2 + len - 1)]) p2 += len, len = 1;
        else p1 += len, len = 1;
        if (p1 == p2) p1++;
    }
    int ans = min(p1, p2);
    return;
}
```

3.9 字符串哈希

- 字符串下标从 1 开始！
- 应用：$O(\log)$ 比较字典序、$O(n\log^2)$ 求最长公共子串
- 时间复杂度：$O(n)$

```cpp
const int M1 = 998244389;
const int M2 = 998244391;
const int B = 257;
const int N = 1000005;

struct Base
{
```

```
    array<ll, N> pow{};
    Base(int mod)
    {
        pow[0] = 1;
        for (int i = 1; i <= N - 1; ++i) pow[i] = pow[i - 1] * B % mod;
    }
    const ll operator[](int idx) const { return pow[idx]; }
} p1(M1), p2(M2);

struct Hash
{
    vector<ll> hash1, hash2;
    void build(const string& s)
    {
        int n = s.size() - 1;
        hash1.resize(n + 1);
        hash2.resize(n + 1);
        for (int i = 1; i <= n; ++i)
        {
            hash1[i] = (hash1[i - 1] * B % M1 + s[i] - 'a' + 1) % M1;
            hash2[i] = (hash2[i - 1] * B % M2 + s[i] - 'a' + 1) % M2;
        }
        return;
    }
    ll merge(ll x, ll y) { return x << 31 | y; }
    ll calc(int lef, int rig)
    {
        ll res1 = (hash1[rig] - hash1[lef - 1] * p1[rig - lef + 1] % M1 + M1) %
            M1;
        ll res2 = (hash2[rig] - hash2[lef - 1] * p2[rig - lef + 1] % M2 + M2) %
            M2;
        return merge(res1, res2);
    }
};
```

4 数学

4.1 快速模

- BarretReduction：x 不能超过 $O(\text{mod}^2)$，保险起见最好最后再模一次
- 时间复杂度：$O(1)$

```
struct BarretReduction
{
    ll m, p;
    void init(int mod)
    {
        m = ((__int128)1 << 64) / mod;
        p = mod;
    }
    ll operator()(ll x) { return x - ((__int128(x) * m) >> 64) * p; }
} br;

ll qmod(ll a, ll b, ll mod) { return a * b - ll(ld(a) / mod * b + 1e-8) * mod;
    }
```

4.2 快速幂

- 特殊情况下可能还要对 res 和 a 的初值进行取模
- 若 p 较大且 mod 为质数可以将 p 对 $\text{mod} - 1$ 取模
- 利用费马小定理求逆元需要注意仅当 mod 为质数时有效
- 时间复杂度：$O(\log p)$

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

ll inv(ll a, ll mod)
{
    return qpow(a, mod - 2, mod);
}
```

4.3 矩阵快速幂

- 递推式可以表示为矩阵乘法时，快速求数列第 i 项
- 时间复杂度：$O(n^3 \log p)$

```
const int MOD = 1e9 + 7;

struct Square
{
    int n;
    vector<vector<ll>> a;
    Square(int n): n(n) { a.resize(n, vector<ll>(n)); }
    void unit()
    {
        for (int i = 0; i < n; ++i)
        {
            a[i][i] = 1;
        }
        return;
    }
};

Square mult(const Square& lhs, const Square& rhs)
{
    assert(lhs.n == rhs.n);
    int n = lhs.n;
    Square res(n);
    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            for (int k = 0; k < n; ++k)
            {
                res.a[i][j] += lhs.a[i][k] * rhs.a[k][j] % MOD;
                res.a[i][j] %= MOD;
            }
        }
    }
    return res;
}

Square qpow(Square a, ll p)
{
    int n = a.n;
    Square res(n);
    res.unit();
    while (p)
    {
        if (p & 1) res = mult(res, a);
        a = mult(a, a);
        p >>= 1;
    }
    return res;
}
```

4.4 矩阵求逆

- 初等变换消元
- 时间复杂度：$O(n^3)$

```
const int MOD = 1e9 + 7;

ll qpow(ll a, ll p)
{
    ll res = 1;
    while (p)
    {
        if (p & 1) res = res * a % MOD;
        a = a * a % MOD;
        p >>= 1;
    }
    return res;
}

ll inv(ll x) { return qpow(x, MOD - 2); }

struct Square
{
    int n;
    vector<vector<ll>> a;
    Square(int n): n(n) { a.resize(n, vector<ll>(n)); }

    void unit()
    {
        for (int i = 0; i < n; ++i)
        {
            for (int j = 0; j < n; ++j)
            {
                a[i][j] = (i == j);
            }
        }
        return;
    }

    bool inverse()
    {
        Square rig(n);
        rig.unit();
        for (int i = 0; i < n; ++i)
        {
            // 找到第i列最大值所在行
            ll tar = i;
            for (int j = i + 1; j < n; ++j)
            {
                if (abs(a[j][i]) > abs(a[tar][i])) tar = j;
            }
            // 与第i行交换
            if (tar != i)
            {
                for (int j = 0; j < n; ++j)
                {
                    swap(a[i][j], a[tar][j]);
                    swap(rig.a[i][j], rig.a[tar][j]);
                }
```

```
        }
        // 不可逆
        if (a[i][i] == 0) return 0;
        // 消去
        ll iv = inv(a[i][i]);
        for (int j = 0; j < n; ++j)
        {
            if (i == j) continue;
            ll t = a[j][i] * iv % MOD;
            for (int k = i; k < n; ++k)
            {
                a[j][k] += MOD - a[i][k] * t % MOD;
                a[j][k] %= MOD;
            }
            for (int k = 0; k < n; ++k)
            {
                rig.a[j][k] += MOD - rig.a[i][k] * t % MOD;
                rig.a[j][k] %= MOD;
            }
        }
        // 归一
        for (int j = 0; j < n; ++j)
        {
            a[i][j] *= iv;
            a[i][j] %= MOD;
            rig.a[i][j] *= iv;
            rig.a[i][j] %= MOD;
        }
    }
    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            a[i][j] = rig.a[i][j];
        }
    }
    return 1;
  }
};
```

4.5 排列奇偶性

- 交换任意两个数，排列奇偶性改变
- 排列奇偶性等于逆序对数的奇偶性
- 求环的个数可以线性得到排列奇偶性
- 时间复杂度：$O(n)$

```
void solve()
{
    cin >> n;
    for (int i = 1; i <= n; ++i) cin >> a[i];
    int inv = n & 1;
    vector<bool> vis(n + 1);
    for (int i = 1; i <= n; ++i)
    {
        if (vis[i]) continue;
        int cur = i;
        while (!vis[cur])
        {
            vis[cur] = 1;
            cur = a[cur];
        }
        inv ^= 1;
    }
    return;
}
```

4.6 线性基

- 求一组数子集的最大异或和
- 数组中非零元素表示一组线性基
- 线性基大小表征线性空间维数
- 线性基中没有异或和为 0 的子集
- 线性基中各数二进制最高位不同
- 时间复杂度：$O(b)$

```
const int N = 55;
const int B = 50;

template<int bit>
struct LinearBasis
{
    vector<ll> v;
    LinearBasis() { v.resize(bit); }
    void insert(ll x)
    {
        for (int i = bit - 1; i >= 0; --i)
        {
            if (x >> i & 1ll)
            {
                if (v[i]) x ^= v[i];
                else
                {
                    v[i] = x;
                    break;
                }
            }
        }
        return;
    }
    ll qmax()
    {
        ll res = 0;
        for (int i = bit - 1; i >= 0; --i)
        {
            if ((res ^ v[i]) > res) res ^= v[i];
        }
        return res;
    }
    void merge(const LinearBasis<bit>& b)
    {
        for (auto e : b.v) insert(e);
        return;
    }
};
```

4.7 高精度

- 注意时间复杂度
- 时间复杂度：$O(n)/O(n^2)$

```
const int N = 5005;
struct L
{
    array<ll, N> a{};
    int len = 0;
    L() {}
    L(ll x)
    {
        while (x)
        {
            a[len++] = x % 10;
            x /= 10;
        }
    }
    L(const string& s)
    {
        for (int i = 0; i < s.size(); ++i)
        {
            a[i] = s[s.size() - 1 - i] - '0';
            if (a[i]) len = max(len, i + 1);
        }
    }
    L& operator=(const L& rhs)
    {
        a = rhs.a;
        len = rhs.len;
        return *this;
    }
    L& operator+=(const L& rhs)
    {
        for (int i = 0; i < max(len, rhs.len); ++i)
        {
            a[i] += rhs.a[i];
            if (i + 1 < N) a[i + 1] += a[i] / 10;
            a[i] %= 10;
        }
        len = max(len, rhs.len);
        if (len < N && a[len]) len++;
        return *this;
    }
    L operator+(const L& rhs) const
    {
        L res(*this);
        res += rhs;
        return res;
    }
    L& operator-=(const L& rhs)
    {
        for (int i = 0; i < rhs.len; ++i) a[i] -= rhs.a[i];
        for (int i = 0; i < len; ++i)
        {
            if (a[i] < 0)
            {
                a[i] += 10;
                if (i + 1 < N) a[i + 1]--;
            }
        }
        while (len - 1 >= 0 && a[len - 1] == 0) len--;
        return *this;
    }
    L operator-(const L& rhs) const
    {
        L res(*this);
        res -= rhs;
        return res;
    }
    L& operator*=(const ll rhs)
    {
        if (rhs == 0)
        {
            *this = L();
            return *this;
        }
        for (int i = 0; i < len; ++i) a[i] *= rhs;
        for (int i = 0; i < min(len + 20, N); ++i)
        {
            if (i + 1 < N) a[i + 1] += a[i] / 10;
            a[i] %= 10;
            if (a[i]) len = max(len, i + 1);
        }
        return *this;
    }
    L operator*(const ll rhs) const
    {
        L res(*this);
```

```cpp
        res *= rhs;
        return res;
    }
    L operator*(const L& rhs) const
    {
        if (rhs.len == 0) return L();
        L res;
        for (int i = 0; i < len; ++i)
        {
            for (int j = 0; j < rhs.len; ++j) res.a[i + j] += a[i] * rhs.a[j];
        }
        res.len = min(N, len + rhs.len - 1);
        for (int i = 0; i < res.len; ++i)
        {
            if (i + 1 < N) res.a[i + 1] += res.a[i] / 10;
            res.a[i] %= 10;
        }
        if (res.len < N && res.a[res.len]) res.len++;
        return res;
    }
    L& operator*=(const L& rhs)
    {
        *this = *this * rhs;
        return *this;
    }

    L& operator/=(const ll rhs)
    {
        assert(rhs);
        for (int i = len - 1; i >= 0; --i)
        {
            if (i - 1 >= 0) a[i - 1] += a[i] % rhs * 10;
            a[i] /= rhs;
        }
        while (len - 1 >= 0 && a[len - 1] == 0) len--;
        return *this;
    }
    L operator/(const ll rhs) const
    {
        L res(*this);
        res /= rhs;
        return res;
    }
    L operator/(const L& rhs) const
    {
        assert(rhs.len);
        if (*this < rhs) return L();
        L res, rem(*this);
        auto compare = [&](int i)
        {
            if (i + rhs.len < N && rem.a[i + rhs.len]) return true;
            for (int j = rhs.len - 1; j >= 0; --j)
            {
                if (rem.a[i + j] < rhs.a[j]) return false;
                else if (rem.a[i + j] > rhs.a[j]) return true;
            }
            return true;
        };
        for (int i = rem.len - rhs.len; i >= 0; --i)
        {
            while (compare(i))
            {
                res.a[i]++;
                res.len = max(res.len, i + 1);
                for (int j = 0; j < rhs.len; ++j)
                {
                    rem.a[i + j] -= rhs.a[j];
                    if (rem.a[i + j] < 0)
                    {
                        rem.a[i + j] += 10;
                        if (i + j + 1 < N) rem.a[i + j + 1]--;
                    }
                }
            }
        }
        while (rem.len - 1 >= 0 && rem.a[rem.len - 1] == 0) rem.len--;
        return res;
    }
    L& operator/=(const L& rhs)
    {
        *this = *this / rhs;
        return *this;
    }
    L operator%(const L& rhs) const
    {
        assert(rhs.len);
        if (*this < rhs) return *this;
        L res, rem(*this);
        auto compare = [&](int i)
        {
            if (i + rhs.len < N && rem.a[i + rhs.len]) return true;
            for (int j = rhs.len - 1; j >= 0; --j)
            {
                if (rem.a[i + j] < rhs.a[j]) return false;
                else if (rem.a[i + j] > rhs.a[j]) return true;
            }
            return true;
        };
        for (int i = rem.len - rhs.len; i >= 0; --i)
        {
            while (compare(i))
            {
                res.a[i]++;
                res.len = max(res.len, i + 1);
                for (int j = 0; j < rhs.len; ++j)
                {
                    rem.a[i + j] -= rhs.a[j];
                    if (rem.a[i + j] < 0)
                    {
                        rem.a[i + j] += 10;
                        if (i + j + 1 < N) rem.a[i + j + 1]--;
                    }
                }
            }
        }
        while (rem.len - 1 >= 0 && rem.a[rem.len - 1] == 0) rem.len--;
        return rem;
    }
    L& operator%=(const L& rhs)
    {
        *this = *this % rhs;
        return *this;
    }
    ll operator%(const ll rhs) const
    {
        ll res = 0;
        for (int i = N - 1; i >= 0; --i)
        {
            res = res * 10 + a[i];
            res %= rhs;
        }
        return res;
    }
    bool operator<(const L& rhs) const
    {
        if (len < rhs.len) return 1;
        else if (len > rhs.len) return 0;
        for (int i = len - 1; i >= 0; --i)
        {
            if (a[i] < rhs.a[i]) return 1;
            else if (a[i] > rhs.a[i]) return 0;
        }
        return 0;
    }
    bool operator>(const L& rhs) const
    {
        if (len > rhs.len) return 1;
        else if (len < rhs.len) return 0;
        for (int i = len - 1; i >= 0; --i)
        {
            if (a[i] > rhs.a[i]) return 1;
            else if (a[i] < rhs.a[i]) return 0;
        }
        return 0;
    }
    bool operator>=(const L& rhs) const { return !(*this < rhs); }
    bool operator<=(const L& rhs) const { return !(*this > rhs); }
    bool operator==(const L& rhs) const { return a == rhs.a; }
    static L p10(int p) { return L(string("1") + string(p, '0')); }
    L sqrt() const
    {
        L lef(0), rig(p10(len / 2 + 1));
        while (lef < rig - 1)
        {
            L mid = (lef + rig) / 2;
            if (mid * mid <= *this) lef = mid;
            else rig = mid;
        }
        return lef;
    }
};
ostream& operator<<(ostream& out, const L& rhs)
{
    if (rhs.len == 0)
    {
        out << '0';
        return out;
    }
    for (int i = rhs.len - 1; i >= 0; --i) out << rhs.a[i];
    return out;
}
istream& operator>>(istream& in, L& rhs)
{
    string s;
    in >> s;
    rhs = L(s);
    return in;
}
```

4.8 连续乘法逆元

- 注意 mod 必须与 $[1, n]$ 所有数都互质，否则不存在逆元
- 时间复杂度：$O(n)$

```cpp
struct ConInv
{
    vector<ll> inv;
    ConInv(int sz, ll mod)
    {
        inv.resize(sz);
        inv[1] = 1;
        for (int i = 2; i <= sz; ++i)
        {
            inv[i] = (mod - mod / i) * inv[mod % i] % mod;
        }
    }
};
```

4.9 数论分块

- 将区间 $[1, n]$ 根据 $k \bmod i$ 的商分为 $O(\sqrt{n})$ 块
- $\sum_{i=1}^{n} k \bmod i = \sum_{i=1}^{n} k - \lfloor \frac{k}{i} \rfloor \cdot i = kn - \sum_{i=1}^{n} \lfloor \frac{k}{i} \rfloor \cdot i$
- 时间复杂度：$O(\sqrt{n})$

```cpp
void solve()
{
    ll n, k;
    cin >> n >> k;
```

```cpp
    ll ans = 0;
    for (ll lef = 1, rig; lef <= n; lef = rig + 1) // 分块
    {
        if (k >= lef) rig = min(n, k / (k / lef));
        else rig = n; // 该区间大于k（余数都为k）
        ll div = k / lef, len = rig - lef + 1;
        ans += k * len - div * (lef + rig) * len / 2;
    }
    cout << ans << '\n';
    return 0;
}
```

4.10 欧拉函数

- $\phi(x) = x\prod_{p_x}\frac{p_x-1}{p_x}$，其中 p_x 为 x 的质因子
- 若 x 为质数：$\phi(i\cdot x) = \begin{cases} x\phi(i) & i \bmod x = 0 \\ (x-1)\phi(i) & i \bmod x \neq 0 \end{cases}$
- $x = \sum_{d|x}\phi(d)$
- 欧拉定理：若 $\gcd(a,m)=1$ 则 $a^{\phi(m)} \equiv 1 \pmod m$（$m$ 为质数时即费马小定理）
- 若求 $[l,r]$ 内的欧拉函数，可以先筛出 $\sqrt{r}$ 以内的质数，用这些质数贡献范围内的数，再特判所有数 $\sqrt{r}$ 以上的质因子即可，类似素数筛
- 时间复杂度：$O(\sqrt{n})$

```cpp
int phi(int n)
{
    int res = n;
    for (int i = 2; i * i <= n; i++)
    {
        if (n % i == 0) res = res / i * (i - 1);
        while (n % i == 0) n /= i;
    }
    if (n > 1) res = res / n * (n - 1);
    return res;
}
```

4.11 线性素数筛

- 每个数只被其最小的质因数筛一次
- sieve_i 表示 i 是否为合数
- 时间复杂度：$O(n)$

```cpp
struct PrimeSieve
{
    vector<int> sieve;
    vector<ll> prime;

    void build(int x)
    {
        sieve.resize(x + 1);
        sieve[1] = 1;
        for (int i = 2; i <= x; ++i)
        {
            if (sieve[i] == 0) prime.push_back(i);
            for (auto e : prime)
            {
                if (e > x / i) break;
                sieve[i * e] = 1;
                if (i % e == 0) break;
            }
        }
        return;
    }
};
```

4.12 欧几里得算法 + 扩展欧几里得算法

- 扩展欧几里得算法用于求解 $ax+by=\gcd(a,b)$
- 求出一组可行解 (x_0,y_0) 后，可得解集 $\left\{\left(x_0+k\frac{b}{\gcd(a,b)}, y_0-k\frac{a}{\gcd(a,b)}\right)\right\}$
- 求出的可行解不一定是最小正整数，但一定满足 $|x_0|<b, |y_0|<a$
- 求解 $ax+by=c$ 时，可以先求解 $ax+by=\gcd(a,b)$ 得到可行解 (x_0',y_0')，此时原方程的可行解为 $\left(x_0=\frac{c}{\gcd(a,b)}x_0', y_0=\frac{c}{\gcd(a,b)}y_0'\right)$，解集依然为 $\left\{\left(x_0+k\frac{b}{\gcd(a,b)}, y_0-k\frac{a}{\gcd(a,b)}\right)\right\}$
- 扩展欧几里得算法还可以通过解同余方程 $ax\equiv 1 \pmod p$ 求乘法逆元，且只需要满足 a,p 互质，不需要 p 是质数
- 时间复杂度：$O(\log n)$

```cpp
ll gcd(ll a, ll b)
{
    return b == 0 ? a : gcd(b, a % b);
}

ll exgcd(ll a, ll b, ll& x, ll& y)
{
    if (b == 0) { x = 1, y = 0; return a; }
    ll d = exgcd(b, a % b, x, y);
    ll newx = y, newy = x - a / b * y;
    x = newx, y = newy;
    return d;
}

ll inv(ll a, ll mod)
{
    ll x, y;
    exgcd(a, mod, x, y);
    return x;
}

ll a, b, x, y, g;

void solve()
{
    cin >> a >> b;
    g = exgcd(a, b, x, y);
    auto M = [](ll x, ll m) {return (x % m + m) % m; };
    cout << M(x, b / g) << '\n';
    return;
}
```

4.13 中国剩余定理

- 用于求解模数两两互质的线性同余方程组 $\begin{cases} x\equiv a_1 \pmod{n_1} \\ x\equiv a_2 \pmod{n_2} \\ \cdots \\ x\equiv a_k \pmod{n_k} \end{cases}$，一定有解
- 两数相乘爆 `long long` 时可能需要快速乘
- 时间复杂度：$O(n\log)$

```cpp
struct CRT
{
    vector<pair<ll, ll>> f;
    inline ll nrm(ll x, ll mod) { return (x % mod + mod) % mod; }
    ll exgcd(ll a, ll b, ll& x, ll& y)
    {
        if (b == 0)
        {
            x = 1, y = 0;
            return a;
        }
        ll d = exgcd(b, a % b, x, y);
        ll newx = y, newy = x - a / b * y;
        x = newx, y = newy;
        return d;
    }
    ll inv(ll a, ll mod)
    {
        ll x, y;
        exgcd(a, mod, x, y);
        return nrm(x, mod);
    }
    void insert(ll r, ll m)
    {
        f.push_back({ r, m });
        return;
    }
    ll work()
    {
        ll mul = 1, ans = 0;
        for (auto e : f) mul *= e.second;
        for (auto e : f)
        {
            ll m = mul / e.second;
            ll c = m * inv(m, e.second);
            ans += c * e.first;
        }
        return nrm(ans, mul);
    }
};
```

4.14 扩展中国剩余定理

- 用于求解模数不互质的线性同余方程组 $\begin{cases} x\equiv a_1 \pmod{n_1} \\ x\equiv a_2 \pmod{n_2} \\ \cdots \\ x\equiv a_k \pmod{n_k} \end{cases}$，可能无解
- 对于两个方程，有 $x=n_1x+a_1=n_2y+a_2$，即 $n_1x-n_2y=a_2-a_1$，用扩展欧几里得算法合并为一个方程，两两合并直到只剩一个方程

- 两数相乘爆 long long 时可能需要快速乘
- 时间复杂度：$O(n \log)$

```
struct ExCRT
{
    vector<pair<ll, ll>> f;
    inline ll nrm(ll x, ll mod) { return (x % mod + mod) % mod; }
    ll qmul(ll a, ll b, ll mod)
    {
        a = nrm(a, mod);
        b = nrm(b, mod);
        ll res = 0;
        while (b)
        {
            if (b & 1) res = (res + a) % mod;
            a = (a + a) % mod;
            b >>= 1;
        }
        return res;
    }
    ll exgcd(ll a, ll b, ll& x, ll& y)
    {
        if (b == 0)
        {
            x = 1, y = 0;
            return a;
        }
        ll d = exgcd(b, a % b, x, y);
        ll newx = y, newy = x - a / b * y;
        x = newx, y = newy;
        return d;
    }
    void insert(ll r, ll m)
    {
        f.push_back({ r, m });
        return;
    }
    pair<ll, ll> work()
    {
        ll x, y;
        while (f.size() >= 2)
        {
            pair<ll, ll> f1 = f.back();
            f.pop_back();
            pair<ll, ll> f2 = f.back();
            f.pop_back();

            ll g = exgcd(f1.second, f2.second, x, y);
            ll c = f2.first - f1.first;
            if (c % g) return { -1, -1 }; // 无解
            x = qmul(x, c / g, f2.second / g); // 输入可能为负，输出非负
            ll m = f1.second / g * f2.second; // m = lcm(m1, m2)
            ll r = (x * f1.second + f1.first) % m; // r = nrm(x) * m1 + r1
            f.push_back({ r, m });
        }
        return f.front();
    }
};
```

4.15 多项式

- 模数 998244353 的原根选用 3
- 时间复杂度：$O(n \log n)$

```
constexpr int MOD = 998244353;

int nrm(int x)
{
    // assume -MOD <= x < 2MOD
    if (x < 0) x += MOD;
    if (x >= MOD) x -= MOD;
    return x;
}

template<class T> T power(T a, ll b)
{
    T res = 1;
    for (; b; b /= 2, a *= a)
    {
        if (b % 2) res *= a;
    }
    return res;
}

struct Z
{
    int x;
    Z(int x = 0): x(nrm(x)) {}
    Z(ll x): x(nrm(x % MOD)) {}
    int val() const { return x; }
    Z operator-() const { return Z(nrm(MOD - x)); }
    Z inv() const
    {
        assert(x != 0);
        return power(*this, MOD - 2);
    }
    Z& operator*=(const Z& rhs)
    {
        x = ll(x) * rhs.x % MOD;
        return *this;
    }
    Z& operator+=(const Z& rhs)
    {
        x = nrm(x + rhs.x);
        return *this;
    }
    Z& operator-=(const Z& rhs)
    {
        x = nrm(x - rhs.x);
        return *this;
    }
    Z& operator/=(const Z& rhs) { return *this *= rhs.inv(); }
    friend Z operator*(const Z& lhs, const Z& rhs)
    {
        Z res = lhs;
        res *= rhs;
        return res;
    }
    friend Z operator+(const Z& lhs, const Z& rhs)
    {
        Z res = lhs;
        res += rhs;
        return res;
    }
    friend Z operator-(const Z& lhs, const Z& rhs)
    {
        Z res = lhs;
        res -= rhs;
        return res;
    }
    friend Z operator/(const Z& lhs, const Z& rhs)
    {
        Z res = lhs;
        res /= rhs;
        return res;
    }
    friend istream& operator>>(istream& is, Z& a)
    {
        ll v;
        is >> v;
        a = Z(v);
        return is;
    }
    friend ostream& operator<<(ostream& os, const Z& a) { return os << a.val();
        }
};

vector<int> rev;
vector<Z> roots{ 0, 1 };

void dft(vector<Z>& a)
{
    int n = a.size();

    if (rev.size() != n)
    {
        int k = __builtin_ctz(n) - 1;
        rev.resize(n);
        for (int i = 0; i < n; i++) rev[i] = rev[i >> 1] >> 1 | (i & 1) << k;
    }

    for (int i = 0; i < n; i++)
    {
        if (rev[i] < i) swap(a[i], a[rev[i]]);
    }
    if (roots.size() < n)
    {
        int k = __builtin_ctz(roots.size());
        roots.resize(n);
        while ((1 << k) < n)
        {
            Z e = power(Z(3), (MOD - 1) >> (k + 1));
            for (int i = 1 << (k - 1); i < (1 << k); i++)
            {
                roots[2 * i] = roots[i];
                roots[2 * i + 1] = roots[i] * e;
            }
            k++;
        }
    }
    for (int k = 1; k < n; k *= 2)
    {
        for (int i = 0; i < n; i += 2 * k)
        {
            for (int j = 0; j < k; j++)
            {
                Z u = a[i + j];
                Z v = a[i + j + k] * roots[k + j];
                a[i + j] = u + v;
                a[i + j + k] = u - v;
            }
        }
    }
    return;
}

void idft(vector<Z>& a)
{
    int n = a.size();
    reverse(a.begin() + 1, a.end());
    dft(a);
    Z inv = (1 - MOD) / n;
    for (int i = 0; i < n; i++) a[i] *= inv;
    return;
}

struct Poly
{
    vector<Z> a;
    Poly() {}
    explicit Poly(int size): a(size) {}
    Poly(const vector<Z>& a): a(a) {}
    Poly(const initializer_list<Z>& a): a(a) {}
    int size() const { return a.size(); }
    void resize(int n) { a.resize(n); }
    Z operator[](int idx) const
    {
        if (idx < size()) return a[idx];
        else return 0;
    }
    Z& operator[](int idx) { return a[idx]; }
    Poly mulxk(int k) const
    {
        auto b = a;
        b.insert(b.begin(), k, 0);
```

```cpp
        return Poly(b);
    }
    Poly modxk(int k) const
    {
        k = min(k, size());
        return Poly(vector<Z>(a.begin(), a.begin() + k));
    }
    Poly divxk(int k) const
    {
        if (size() <= k) return Poly();
        return Poly(vector<Z>(a.begin() + k, a.end()));
    }
    friend Poly operator+(const Poly& a, const Poly& b)
    {
        vector<Z> res(max(a.size(), b.size()));
        for (int i = 0; i < res.size(); i++) res[i] = a[i] + b[i];
        return Poly(res);
    }
    friend Poly operator-(const Poly& a, const Poly& b)
    {
        vector<Z> res(max(a.size(), b.size()));
        for (int i = 0; i < res.size(); i++) res[i] = a[i] - b[i];
        return Poly(res);
    }
    friend Poly operator-(const Poly& a)
    {
        vector<Z> res(a.size());
        for (int i = 0; i < res.size(); i++) res[i] = -a[i];
        return Poly(res);
    }
    friend Poly operator*(Poly a, Poly b)
    {
        if (a.size() == 0 || b.size() == 0) return Poly();
        if (a.size() < b.size()) swap(a, b);
        if (b.size() < 128)
        {
            Poly c(a.size() + b.size() - 1);
            for (int i = 0; i < a.size(); i++)
            {
                for (int j = 0; j < b.size(); j++) c[i + j] += a[i] * b[j];
            }
            return c;
        }
        int sz = 1, tot = a.size() + b.size() - 1;
        while (sz < tot) sz *= 2;
        a.a.resize(sz);
        b.a.resize(sz);
        dft(a.a);
        dft(b.a);
        for (int i = 0; i < sz; ++i) a.a[i] = a[i] * b[i];
        idft(a.a);
        a.resize(tot);
        return a;
    }
    friend Poly operator*(Z a, Poly b)
    {
        for (int i = 0; i < b.size(); i++) b[i] *= a;
        return b;
    }
    friend Poly operator*(Poly a, Z b)
    {
        for (int i = 0; i < a.size(); i++) a[i] *= b;
        return a;
    }
    Poly& operator+=(Poly b) { return (*this) = (*this) + b; }
    Poly& operator-=(Poly b) { return (*this) = (*this) - b; }
    Poly& operator*=(Poly b) { return (*this) = (*this) * b; }
    Poly& operator*=(Z b) { return (*this) = (*this) * b; }
    Poly deriv() const
    {
        if (a.empty()) return Poly();
        vector<Z> res(size() - 1);
        for (int i = 0; i < size() - 1; ++i) res[i] = (i + 1) * a[i + 1];
        return Poly(res);
    }
    Poly integr() const
    {
        vector<Z> res(size() + 1);
        for (int i = 0; i < size(); ++i) res[i + 1] = a[i] / (i + 1);
        return Poly(res);
    }
    Poly inv(int m) const
    {
        Poly x{ a[0].inv() };
        int k = 1;
        while (k < m)
        {
            k *= 2;
            x = (x * (Poly{ 2 } - modxk(k) * x)).modxk(k);
        }
        return x.modxk(m);
    }
    Poly log(int m) const { return (deriv() * inv(m)).integr().modxk(m); }
    Poly exp(int m) const
    {
        Poly x{ 1 };
        int k = 1;
        while (k < m)
        {
            k *= 2;
            x = (x * (Poly{ 1 } - x.log(k) + modxk(k))).modxk(k);
        }
        return x.modxk(m);
    }
    Poly pow(int k, int m) const
    {
        int i = 0;
        while (i < size() && a[i].val() == 0) i++;
        if (i == size() || 1LL * i * k >= m) return Poly(vector<Z>(m));
        Z v = a[i];
        auto f = divxk(i) * v.inv();
        return (f.log(m - i * k) * k).exp(m - i * k).mulxk(i * k) * power(v, k);
    }
    Poly sqrt(int m) const
    {
        Poly x{ 1 };
        int k = 1;
        while (k < m)
        {
            k *= 2;
            x = (x + (modxk(k) * x.inv(k)).modxk(k)) * ((MOD + 1) / 2);
        }
        return x.modxk(m);
    }
    Poly mulT(Poly b) const
    {
        if (b.size() == 0) return Poly();
        int n = b.size();
        reverse(b.a.begin(), b.a.end());
        return ((*this) * b).divxk(n - 1);
    }
    vector<Z> eval(vector<Z> x) const
    {
        if (size() == 0) return vector<Z>(x.size(), 0);
        const int n = max(int(x.size()), size());
        vector<Poly> q(4 * n);
        vector<Z> ans(x.size());
        x.resize(n);
        function<void(int, int, int)> build = [&](int p, int l, int r)
        {
            if (r - l == 1) q[p] = Poly{ 1, -x[l] };
            else
            {
                int m = (l + r) / 2;
                build(2 * p, l, m);
                build(2 * p + 1, m, r);
                q[p] = q[2 * p] * q[2 * p + 1];
            }
        };
        build(1, 0, n);
        function<void(int, int, int, const Poly&)> work = [&](int p, int l, int r
            , const Poly& num)
        {
            if (r - l == 1)
            {
                if (l < ans.size()) ans[l] = num[0];
            }
            else
            {
                int m = (l + r) / 2;
                work(2 * p, l, m, num.mulT(q[2 * p + 1]).modxk(m - l));
                work(2 * p + 1, m, r, num.mulT(q[2 * p]).modxk(r - m));
            }
        };
        work(1, 0, n, mulT(q[1].inv(n)));
        return ans;
    }
};
```

4.16 哥德巴赫猜想

1. 大于等于 6 的整数可以写成三个质数之和
2. 大于等于 4 的偶数可以写成两个质数之和
3. 大于等于 7 的奇数可以写成三个奇质数之和

4.17 组合数学公式

1. $C_n^m = C_{n-1}^m + C_{n-1}^{m-1}$
2. $H_n = \frac{1}{n+1} C_{2n}^n$
3. $S(n,m) = S(n-1,m-1) + mS(n-1,m)$
4. $s(n,m) = s(n-1,m-1) + (n-1)s(n-1,m)$

5 数据结构

5.1 单调栈

1. 求每个数左边或右边第一个大于它的数的位置
2. 时间复杂度：$O(n)$

```cpp
vector<int> stk;
for (int i = 1; i <= n; ++i)
{
    while (stk.size() && a[stk.back()] < a[i])
    {
        rig[stk.back()] = i;
        stk.pop_back();
    }
    if (stk.size()) lef[i] = stk.back();
    else lef[i] = 0;
    stk.push_back(i);
}
while (stk.size())
{
    rig[stk.back()] = n + 1;
    stk.pop_back();
}
```

5.2 哈希表

1. 自定义随机化哈希函数，降低碰撞概率
2. unordered_map 采用开链法，gp_hash_table 采用查探法
3. 时间复杂度：$O(1)$

```
#include<bits/stdc++.h>
#include<unordered_map>

#include<ext/pb_ds/assoc_container.hpp>
#include<ext/pb_ds/hash_policy.hpp>

using namespace std;
using ll = long long;

using namespace __gnu_pbds;

struct CustomHash
{
    static uint64_t splitmix64(uint64_t x)
    {
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    size_t operator()(uint64_t x) const
    {
        static const uint64_t FIXED_RANDOM = chrono::steady_clock::now().
            time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
    }
};

unordered_map<ll, ll, CustomHash> mp;
gp_hash_table<ll, ll, CustomHash> ht;
```

5.3 并查集

1. 使用路径压缩 + 启发式合并保证时间复杂度
2. 时间复杂度：查找 $O(1)$/合并 $O(1)$

```
struct DSU
{
    vector<int> f;
    vector<int> v; // 集合大小
    DSU(int x)
    {
        f.resize(x + 1);
        v.resize(x + 1);
        for (int i = 1; i <= x; ++i) f[i] = i;
        for (int i = 1; i <= x; ++i) v[i] = 1;
    }
    int find(int id) { return f[id] == id ? id : f[id] = find(f[id]); }
    void merge(int x, int y)
    {
        int fx = find(x), fy = find(y);
        if (fx == fy) return;
        if (v[fx] > v[fy]) swap(fx, fy);
        f[fx] = fy;
        v[fy] += v[fx];
        return;
    }
};
```

5.4 ST 表

1. 可重复贡献问题的静态区间查询，需要满足 $f(r,r) = r$，一般是最值/GCD
2. 必要时可以预处理 $\log i$ 加快查询
3. 时间复杂度：建表 $O(n\log n)$/查询 $O(1)$

```
struct ST
{
    int sz;
    vector<vector<ll>> st;

    ST(int x) { init(x); }
    void init(int x)
    {
        sz = x;
        st.resize(32, vector<ll>(sz + 1));
    }
    void build(ll arr[])
    {
        for (int i = 1; i <= sz; ++i) st[0][i] = arr[i];
        int lg = __lg(sz);
        for (int i = 1; i <= lg; ++i)
        {
            for (int j = 1; j <= sz; ++j)
            {
                st[i][j] = st[i - 1][j];
                if (j + (1 << (i - 1)) <= sz)
                {
                    st[i][j] = max(st[i][j], st[i - 1][j + (1 << (i - 1))]);
                }
            }
        }
    }
    ll query(int lef, int rig)
    {
        int len = __lg(rig - lef + 1);
        return max(st[len][lef], st[len][rig - (1 << len) + 1]);
    }
};
```

5.5 笛卡尔树

1. 第一关键字满足二叉搜索树性质，第二关键字满足小根堆性质
2. 按照第一关键字顺序传入，按照第二关键字大小构建
3. 第一关键字通常为下标，此时建得的堆每个子树都拥有一段连续下标
4. 时间复杂度：$O(n)$

```
const ll INFLL = 0x3f3f3f3f3f3f3f3f;

struct CarTree
{
    vector<pair<ll, ll>> v;
    vector<int> ls, rs;
    int sz;
    CarTree(): v(1, { -INFLL, -INFLL }), sz(0) {}
    void insert(ll a, ll b)
    {
        v.push_back({ a, b });
        sz++;
        return;
    }
    void build()
    {
        ls.resize(v.size());
        rs.resize(v.size());
        stack<int> stk;
        stk.push(0);
        for (int i = 1; i <= sz; ++i)
        {
            while (v[stk.top()].second > v[i].second) stk.pop();
            ls[i] = rs[stk.top()];
            rs[stk.top()] = i;
            stk.push(i);
        }
        return;
    }
};
```

5.6 树状数组

1. 动态维护满足区间减法的性质
2. 时间复杂度：建立 $O(n)$/修改 $O(\log n)$/查询 $O(\log n)$

```
struct Fenwick // 普通树状数组
{
    int sz;
    vector<ll> tree;

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

    Fenwick() {}
    Fenwick(int x) { init(x); }
    void init(int x)
    {
        sz = x;
        tree.resize(sz + 1);
    }
    void add(int dst, ll v)
    {
        while (dst <= sz)
        {
            tree[dst] += v;
            dst += lowbit(dst);
        }
        return;
    }
    ll pre(int dst)
    {
        ll res = 0;
        while (dst)
        {
            res += tree[dst];
            dst -= lowbit(dst);
        }
        return res;
    }
    ll rsum(int lef, int rig) { return pre(rig) - pre(lef - 1); }
    void build(ll arr[])
    {
        for (int i = 1; i <= sz; ++i)
        {
            tree[i] += arr[i];
            int j = i + lowbit(i);
            if (j <= sz) tree[j] += tree[i];
        }
        return;
    }
};


struct Fenwick // 时间戳优化，可O(1)清空
```

```cpp
{
    int sz;
    vector<ll> tree;
    vector<int> tag;
    int now;

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

    Fenwick(int x)
    {
        sz = x;
        tree.resize(sz + 1);
        tag.resize(sz + 1);
        now = 0;
    }
    void clear()
    {
        now++;
        return;
    }
    void add(int dst, ll v)
    {
        while (dst <= sz)
        {
            if (tag[dst] != now) tree[dst] = 0;
            tree[dst] += v;
            tag[dst] = now;
            dst += lowbit(dst);
        }
        return;
    }
    ll pre(int dst)
    {
        ll res = 0;
        while (dst)
        {
            if (tag[dst] == now) res += tree[dst];
            dst -= lowbit(dst);
        }
        return res;
    }
    ll rsum(int lef, int rig) { return pre(rig) - pre(lef - 1); }
    void build(ll arr[])
    {
        for (int i = 1; i <= sz; ++i)
        {
            tree[i] += arr[i];
            int j = i + lowbit(i);
            if (j <= sz) tree[j] += tree[i];
        }
        return;
    }
};
```

5.7 二维树状数组

1. 时间复杂度：修改 $O(\log^2 n)$/查询 $O(\log^2 n)$

```cpp
struct Fenwick2
{
    int sz;
    vector<vector<ll>> tree;

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

    Fenwick2(int x)
    {
        sz = x;
        tree.resize(sz + 1, vector<ll>(sz + 1));
    }

    void add(int x, int y, ll val)
    {
        for (int i = x; i <= sz; i += lowbit(i))
        {
            for (int j = y; j <= sz; j += lowbit(j))
            {
                tree[i][j] += val;
            }
        }
        return;
    }

    ll pre(int x, int y)
    {
        ll res = 0;
        for (int i = x; i >= 1; i -= lowbit(i))
        {
            for (int j = y; j >= 1; j -= lowbit(j))
            {
                res += tree[i][j];
            }
        }
        return res;
    }

    ll sum(int x1, int y1, int x2, int y2)
    {
        return pre(x2, y2) - pre(x1 - 1, y2) - pre(x2, y1 - 1) + pre(x1 - 1, y1 -
            1);
    }
};
```

5.8 线段树

1. 时间复杂度：建立 $O(n)$/询问 $O(\log n)$/修改 $O(\log n)$

```cpp
struct SegTree // 维护区间和，支持区间加减
{
    struct Node
    {
        int lef, rig;
        ll val, tag;
    };
    vector<Node> tree;

    SegTree(int x) { tree.resize(x * 4 + 1); }

    // 由子节点及其标记更新父节点
    void update(int src)
    {
        ll lw = tree[src << 1].rig - tree[src << 1].lef + 1;
        ll rw = tree[src << 1 | 1].rig - tree[src << 1 | 1].lef + 1;
        ll lv = tree[src << 1].val + tree[src << 1].tag * lw;
        ll rv = tree[src << 1 | 1].val + tree[src << 1 | 1].tag * rw;
        tree[src].val = lv + rv;
        return;
    }

    // 下传标记并消耗
    void pushdown(int src)
    {
        if (tree[src].lef < tree[src].rig)
        {
            tree[src << 1].tag += tree[src].tag;
            tree[src << 1 | 1].tag += tree[src].tag;
        }
        ll wid = tree[src].rig - tree[src].lef + 1;
        tree[src].val += tree[src].tag * wid;
        tree[src].tag = 0;
        return;
    }

    void build(int src, int lef, int rig)
    {
        tree[src] = { lef, rig, 0, 0 };
        if (lef == rig) return;
        int mid = lef + (rig - lef) / 2;
        build(src << 1, lef, mid);
        build(src << 1 | 1, mid + 1, rig);
        update(src);
        return;
    }

    void modify(int src, int lef, int rig, ll val)
    {
        if (lef <= tree[src].lef && tree[src].rig <= rig)
        {
            tree[src].tag += val;
            return;
        }
        pushdown(src);
        if (lef <= tree[src << 1].rig) modify(src << 1, lef, rig, val);
        if (rig >= tree[src << 1 | 1].lef) modify(src << 1 | 1, lef, rig, val);
        update(src);
        return;
    }

    ll query(int src, int lef, int rig)
    {
        pushdown(src);
        if (lef <= tree[src].lef && tree[src].rig <= rig) return tree[src].val;
        ll res = 0;
        if (lef <= tree[src << 1].rig) res += query(src << 1, lef, rig);
        if (rig >= tree[src << 1 | 1].lef) res += query(src << 1 | 1, lef, rig);
        return res;
    }
};
struct SegTree // 维护区间和，支持单点修改（无标记）/二分查找第一个区间和大于等于x的
    右端点
{
    struct Node
    {
        int lef, rig;
        ll val;
    };
    vector<Node> tree;

    SegTree(int x) { tree.resize(x * 4 + 1); }

    // 由子节点及其标记更新父节点
    void update(int src)
    {
        tree[src].val = tree[src << 1].val + tree[src << 1 | 1].val;
        return;
    }

    void build(int src, int lef, int rig)
    {
        tree[src] = { lef, rig, 0 };
        if (lef == rig) return;
        int mid = lef + (rig - lef) / 2;
        build(src << 1, lef, mid);
        build(src << 1 | 1, mid + 1, rig);
        update(src);
        return;
    }

    void assign(int src, int pos, ll val)
    {
        if (tree[src].lef == tree[src].rig)
        {
            tree[src].val = val;
            return;
        }
        if (pos <= tree[src << 1].rig) assign(src << 1, pos, val);
        else assign(src << 1 | 1, pos, val);
        update(src);
        return;
    }

    ll query(int src, int lef, int rig)
```

```
    {
        if (lef <= tree[src].lef && tree[src].rig <= rig) return tree[src].val;
        ll res = 0;
        if (lef <= tree[src << 1].rig) res += query(src << 1, lef, rig);
        if (rig >= tree[src << 1 | 1].lef) res += query(src << 1 | 1, lef, rig);
        return res;
    }

    int bis(int src, int lef, ll& tar)
    {
        if (tree[src].lef == lef)
        {
            if (tree[src].val < tar)
            {
                tar -= tree[src].val;
                return 0;
            }
            if (tree[src].rig == lef) return lef;
            if (tree[src << 1].val >= tar) return bis(src << 1, lef, tar);
            tar -= tree[src << 1].val;
            lef = tree[src << 1 | 1].lef;
            return bis(src << 1 | 1, lef, tar);
        }
        if (lef <= tree[src << 1].rig)
        {
            int res = bis(src << 1, lef, tar);
            if (res) return res;
            lef = tree[src << 1 | 1].lef;
        }
        return bis(src << 1 | 1, lef, tar);
    }
};


struct SegTree // 维护最大值，支持区间加减/二分查询第一个大于等于x的数
{
    struct Node
    {
        int lef, rig;
        ll val, tag;
    };
    vector<Node> tree;

    SegTree(int x) { tree.resize(x * 4 + 1); }

    // 由子节点及其标记更新父节点
    void update(int src)
    {
        ll lv = tree[src << 1].val + tree[src << 1].tag;
        ll rv = tree[src << 1 | 1].val + tree[src << 1 | 1].tag;
        tree[src].val = max(lv, rv);
        return;
    }

    // 下传标记并消耗
    void pushdown(int src)
    {
        if (tree[src].lef < tree[src].rig)
        {
            tree[src << 1].tag += tree[src].tag;
            tree[src << 1 | 1].tag += tree[src].tag;
        }
        tree[src].val += tree[src].tag;
        tree[src].tag = 0;
        return;
    }

    void build(int src, int lef, int rig)
    {
        tree[src] = { lef, rig, 0, 0 };
        if (lef == rig) return;
        int mid = lef + (rig - lef) / 2;
        build(src << 1, lef, mid);
        build(src << 1 | 1, mid + 1, rig);
        update(src);
        return;
    }

    void modify(int src, int lef, int rig, ll val)
    {
        if (lef <= tree[src].lef && tree[src].rig <= rig)
        {
            tree[src].tag += val;
            return;
        }
        pushdown(src);
        if (lef <= tree[src << 1].rig) modify(src << 1, lef, rig, val);
        if (rig >= tree[src << 1 | 1].lef) modify(src << 1 | 1, lef, rig, val);
        update(src);
        return;
    }

    ll query(int src, int lef, int rig)
    {
        pushdown(src);
        if (lef <= tree[src].lef && tree[src].rig <= rig) return tree[src].val;
        ll res = 0;
        if (lef <= tree[src << 1].rig) res = max(res, query(src << 1, lef, rig));
        if (rig >= tree[src << 1 | 1].lef) res = max(res, query(src << 1 | 1, lef
            , rig));
        return res;
    }

    int bis(int src, int lef, ll tar)
    {
        pushdown(src);
        if (tree[src].lef == lef)
        {
            if (tree[src].val < tar) return 0;
            if (tree[src].rig == lef) return lef;
            if ((tree[src << 1].val + tree[src << 1].tag) >= tar) return bis(src
                << 1, lef, tar);
            else return query(src << 1 | 1, tree[src << 1 | 1].lef, tar);
        }
        if (lef <= tree[src << 1].rig)
        {
            int res = bis(src << 1, lef, tar);
            if (res) return res;
            lef = tree[src << 1 | 1].lef;
        }
        return bis(src << 1 | 1, lef, tar);
    }
};

struct SegTree // 维护最大值，支持区间取最大值/二分查询第一个大于等于x的数
{
    struct Node
    {
        int lef, rig;
        ll val, tag;
    };
    vector<Node> tree;

    SegTree(int x) { tree.resize(x * 4 + 1); }

    // 由子节点及其标记更新父节点
    void update(int src)
    {
        ll lv = max(tree[src << 1].val, tree[src << 1].tag);
        ll rv = max(tree[src << 1 | 1].val, tree[src << 1 | 1].tag);
        tree[src].val = max(lv, rv);
        return;
    }

    // 下传标记并消耗
    void pushdown(int src)
    {
        if (tree[src].lef < tree[src].rig)
        {
            tree[src << 1].tag = max(tree[src << 1].tag, tree[src].tag);
            tree[src << 1 | 1].tag = max(tree[src << 1 | 1].tag, tree[src].tag);
        }
        tree[src].val = max(tree[src].val, tree[src].tag);
        tree[src].tag = 0;
        return;
    }

    void build(int src, int lef, int rig)
    {
        tree[src] = { lef, rig, 0, 0 };
        if (lef == rig) return;
        int mid = lef + (rig - lef) / 2;
        build(src << 1, lef, mid);
        build(src << 1 | 1, mid + 1, rig);
        update(src);
        return;
    }

    void modify(int src, int lef, int rig, ll val)
    {
        if (lef <= tree[src].lef && tree[src].rig <= rig)
        {
            tree[src].tag = max(tree[src].tag, val);
            return;
        }
        pushdown(src);
        if (lef <= tree[src << 1].rig) modify(src << 1, lef, rig, val);
        if (rig >= tree[src << 1 | 1].lef) modify(src << 1 | 1, lef, rig, val);
        update(src);
        return;
    }

    ll query(int src, int lef, int rig)
    {
        pushdown(src);
        if (lef <= tree[src].lef && tree[src].rig <= rig) return tree[src].val;
        ll res = 0;
        if (lef <= tree[src << 1].rig) res = max(res, query(src << 1, lef, rig));
        if (rig >= tree[src << 1 | 1].lef) res = max(res, query(src << 1 | 1, lef
            , rig));
        return res;
    }

    int bis(int src, int lef, ll tar)
    {
        pushdown(src);
        if (tree[src].lef == lef)
        {
            if (tree[src].val < tar) return 0;
            if (tree[src].rig == lef) return lef;
            if (max(tree[src << 1].val, tree[src << 1].tag) >= tar) return bis(src
                << 1, lef, tar);
            else return query(src << 1 | 1, tree[src << 1 | 1].lef, tar);
        }
        if (lef <= tree[src << 1].rig)
        {
            int res = bis(src << 1, lef, tar);
            if (res) return res;
            lef = tree[src << 1 | 1].lef;
        }
        return bis(src << 1 | 1, lef, tar);
    }
};
```

5.9 历史最值线段树

1. 维护区间历史最值，支持区间加减
2. 上方标记一定新于下方标记，因此下传可以整体施加
3. 时间复杂度：建立 $O(n)$/询问 $O(\log n)$/修改 $O(\log n)$

```
struct SegTree
{
    struct Node
```

```
    {
        int lef, rig;
        ll mval; // 历史最值
        ll tag, mtag; // 当前修改标签、tag生命周期内最值
    };
    vector<Node> tree;
    ll merge(ll x, ll y) { return min(x, y); } // 最大还是最小
    void affect(ll& x, ll y) { x = merge(x, y); } // 取最值
    void update(int src) // 由子节点及其标记更新父节点
    {
        ll lv = tree[src << 1].mval + merge(tree[src << 1].mtag, 0);
        ll rv = tree[src << 1 | 1].mval + merge(tree[src << 1 | 1].mtag, 0);
        tree[src].mval = merge(lv, rv);
        return;
    }
    void pushdown(int src) // 下传标记并消耗
    {
        if (tree[src].lef < tree[src].rig)
        {
            affect(tree[src << 1].mtag, tree[src << 1].tag + tree[src].mtag);
            affect(tree[src << 1 | 1].mtag, tree[src << 1 | 1].tag + tree[src].
                mtag);
            tree[src << 1].tag += tree[src].tag;
            tree[src << 1 | 1].tag += tree[src].tag;
        }
        tree[src].mval += merge(tree[src].mtag, 0);
        tree[src].mtag = tree[src].tag = 0;
        return;
    }
    void mark(int src, ll val) // 更新标记
    {
        tree[src].tag += val;
        affect(tree[src].mtag, tree[src].tag);
        return;
    }

    SegTree(int x) { tree.resize(x * 4 + 1); }

    void build(int src, int lef, int rig)
    {
        tree[src] = { lef, rig, 0, 0, 0 };
        if (lef == rig) return;
        int mid = lef + (rig - lef) / 2;
        build(src << 1, lef, mid);
        build(src << 1 | 1, mid + 1, rig);
        update(src);
        return;
    }
    void modify(int src, int lef, int rig, ll val)
    {
        if (lef <= tree[src].lef && tree[src].rig <= rig)
        {
            mark(src, val);
            return;
        }
        pushdown(src);
        if (lef <= tree[src << 1].rig) modify(src << 1, lef, rig, val);
        if (rig >= tree[src << 1 | 1].lef) modify(src << 1 | 1, lef, rig, val);
        update(src);
        return;
    }
    ll query(int src, int lef, int rig)
    {
        pushdown(src);
        if (lef <= tree[src].lef && tree[src].rig <= rig) return tree[src].mval;
        ll res = 0;
        if (lef <= tree[src << 1].rig) res = merge(res, query(src << 1, lef, rig)
            );
        if (rig >= tree[src << 1 | 1].lef) res = merge(res, query(src << 1 | 1,
            lef, rig));
        return res;
    }
};
```

5.10 动态开点线段树

1. 需要特别注意空间大小
2. 时间复杂度：询问 $O(\log m)$/修改 $O(\log m)$

```
struct SegTree
{
    struct Node
    {
        int ls = 0, rs = 0;
        ll val = 0, tag = 0;
    };
    vector<Node> tree;
    int ord;
    SegTree(int x)
    {
        tree.resize(x * 64 + 1);
        ord = 1;
    }
    void push(int src, int lef, int rig)
    {
        if (lef < rig)
        {
            if (!tree[src].ls) tree[src].ls = ++ord;
            if (!tree[src].rs) tree[src].rs = ++ord;
            tree[tree[src].ls].tag += tree[src].tag;
            tree[tree[src].rs].tag += tree[src].tag;
        }
        tree[src].val += tree[src].tag * (rig - lef + 1);
        tree[src].tag = 0;
        return;
    }
    void modify(int src, int lef, int rig, int l, int r, ll val)
    {
        if (lef >= l && rig <= r)
        {
            tree[src].tag += val;
            return;
        }
        int mid = lef + (rig - lef) / 2;
        if (l <= mid)
        {
            if (!tree[src].ls) tree[src].ls = ++ord;
            modify(tree[src].ls, lef, mid, l, r, val);
        }
        if (r >= mid + 1)
        {
            if (!tree[src].rs) tree[src].rs = ++ord;
            modify(tree[src].rs, mid + 1, rig, l, r, val);
        }
        tree[src].val += (min(rig, r) - max(lef, l) + 1) * val;
        return;
    }
    ll query(int src, int lef, int rig, int l, int r)
    {
        push(src, lef, rig);
        if (lef >= l && rig <= r) return tree[src].val;
        ll res = 0;
        int mid = lef + (rig - lef) / 2;
        if (l <= mid)
        {
            if (!tree[src].ls) tree[src].ls = ++ord;
            res += query(tree[src].ls, lef, mid, l, r);
        }
        if (r >= mid + 1)
        {
            if (!tree[src].rs) tree[src].rs = ++ord;
            res += query(tree[src].rs, mid + 1, rig, l, r);
        }
        return res;
    }
};
```

5.11 可持久化线段树

1. 需要特别注意空间大小，若维护区间超过较大记得把 32 换成 64
2. 建空根：可以不靠离散化维护大区间，但要谨慎考虑空间复杂度
3. 维护值域：将序列元素逐个插入，由前缀和性质，区间值域上性质蕴含在新树和旧树的差之中
4. 标记永久化：为了防止新操作影响旧结点，路过结点时标记不下放，也不通过子结点更新父结点，而是直接改变每个结点的值，并在向下搜索时记录累积标记值；此时不支持单点赋值
5. 区间第 k 大也可以用整体二分/划分树求解
6. 时间复杂度：所有操作 $O(\log m)$

```
struct PerSegTree // 维护区间和，支持区间加减
{
    struct Node
    {
        int ls, rs;
        ll val, tag;
        Node(): ls(0), rs(0), val(0), tag(0) {}
    };
    vector<Node> tree;
    vector<int> root;
    int size;
    ll L, R;

    int _build(ll l, ll r, ll a[])
    {
        int now = size++;
        if (l == r) tree[now].val = a[l];
        else
        {
            ll m = l + (r - l) / 2;
            tree[now].ls = _build(l, m, a);
            tree[now].rs = _build(m + 1, r, a);
            tree[now].val = tree[tree[now].ls].val + tree[tree[now].rs].val;
        }
        return now;
    }
    void init(ll l, ll r, int cnt, ll a[]) // 建初始树
    {
        size = 0;
        L = l, R = r;
        tree.resize(cnt * 32 + 5);
        root.push_back(_build(L, R, a));
        return;
    }
    void init(ll l, ll r, int cnt) // 建一个空根
    {
        size = 1;
        L = l, R = r;
        tree.resize(cnt * 32 + 5);
        root.push_back(0);
        return;
    }
    void modify(int ver, ll lef, ll rig, ll val) { root.push_back(_modify(root[
        ver], L, R, lef, rig, val)); }
    int _modify(int src, ll l, ll r, ll lef, ll rig, ll val)
    {
        int now = size++;
        tree[now] = tree[src];
        if (lef <= l && r <= rig) tree[now].tag += val;
        else if (l <= rig && r >= lef)
        {
            tree[now].val += val * (min(rig, r) - max(lef, l) + 1);
            ll m = l + (r - l) / 2;
```

```cpp
        if (lef <= m) tree[now].ls = _modify(tree[now].ls, l, m, lef, rig, val
            );
        if (rig > m) tree[now].rs = _modify(tree[now].rs, m + 1, r, lef, rig,
            val);
    }
    return now;
  }
  ll query(int ver, ll lef, ll rig) { return _query(root[ver], L, R, lef, rig,
      0); }
  ll _query(int src, ll l, ll r, ll lef, ll rig, ll tag)
  {
    tag += tree[src].tag;
    if (lef <= l && r <= rig) return tree[src].val + (r - l + 1) * tag;
    else if (l <= rig && r >= lef)
    {
        int m = l + (r - l) / 2;
        ll res = 0;
        if (lef <= m) res += _query(tree[src].ls, l, m, lef, rig, tag);
        if (rig > m) res += _query(tree[src].rs, m + 1, r, lef, rig, tag);
        return res;
    }
    else return 0;
  }
  ll kth(ll lef, ll rig, int k) { return _kth(root[lef - 1], root[rig], L, R,
      k); }
  ll _kth(int osrc, int nsrc, ll l, ll r, int k)
  {
    if (l == r) return l;
    int nsum = tree[tree[nsrc].ls].val + tree[tree[nsrc].ls].tag;
    int osum = tree[tree[osrc].ls].val + tree[tree[osrc].ls].tag;
    int dif = nsum - osum;
    int m = l + (r - l) / 2;
    if (dif >= k) return _kth(tree[osrc].ls, tree[nsrc].ls, l, m, k);
    else return _kth(tree[osrc].rs, tree[nsrc].rs, m + 1, r, k - dif);
  }
};
```

5.12 李超线段树

1. 谨慎使用，注意浮点数精度和结点初始化问题
2. 标记永久化，整条链每一层的值都可能是答案
3. 时间复杂度：建立 $O(n)$/修改 $O(\log^2 n)$/查询 $O(\log n)$

```cpp
const int N = 100005;
const double EPS = 1e-9;

struct Seg
{
  double k, b;
  int lef, rig;
  void init(int x0, int y0, int x1, int y1)
  {
    lef = x0, rig = x1;
    if (x0 == x1)
    {
        k = 0, b = max(y0, y1);
    }
    else
    {
        k = double(y1 - y0) / (x1 - x0);
        b = y0 - x0 * k;
    }
  }
  double at(int x) { return k * x + b; }
} seg[N];

struct LCSegTree
{
  struct Node
  {
    int lef, rig, id;
  };
  vector<Node> tree;

  LCSegTree(int x) { tree.resize(x * 4 + 1); }

  void build(int src, int lef, int rig)
  {
    tree[src] = { lef, rig, 0 };
    if (lef == rig) return;
    int mid = (lef + rig) / 2;
    build(src << 1, lef, mid);
    build(src << 1 | 1, mid + 1, rig);
    return;
  }

  void add(int src, int id)
  {
    if (seg[id].lef <= tree[src].lef && seg[id].rig >= tree[src].rig)
    {
        update(src, id);
        return;
    }
    if (seg[id].lef <= tree[src << 1].rig) add(src << 1, id);
    if (seg[id].rig >= tree[src << 1 | 1].lef) add(src << 1 | 1, id);
    return;
  }

  bool compare(int id1, int id2, int x)
  {
    if (id1 == 0) return 1;
    if (id2 == 0) return 0;
    double r1 = seg[id1].at(x);
    double r2 = seg[id2].at(x);
    if (fabs(r1 - r2) < EPS) return id2 < id1;
    else return r2 > r1 + EPS;
  }

  void update(int src, int id)
  {
    int mid = (tree[src].lef + tree[src].rig) / 2;
    if (compare(tree[src].id, id, mid)) swap(tree[src].id, id);
    if (tree[src].lef == tree[src].rig) return;
    if (compare(tree[src].id, id, tree[src].lef)) update(src << 1, id);
    if (compare(tree[src].id, id, tree[src].rig)) update(src << 1 | 1, id);
    return;
  }

  int query(int src, int x)
  {
    if (tree[src].lef == tree[src].rig) return tree[src].id;
    int r = query(src << 1 | (x >= tree[src << 1 | 1].lef), x);
    return compare(r, tree[src].id, x) ? tree[src].id : r;
  }
};
```

6 树论

6.1 LCA

1. 倍增做法
2. 时间复杂度：$O(\log n)$

```cpp
const int N = 500005;

vector<int> node[N];

struct LCA
{
  vector<int> d; // 到根距离
  vector<vector<int>> st;

  void dfs(int x)
  {
    for (auto e : node[x])
    {
        if (e == st[x][0]) continue;
        d[e] = d[x] + 1;
        st[e][0] = x;
        dfs(e);
    }
    return;
  }

  void build(int sz)
  {
    int lg = __lg(sz);
    for (int i = 1; i <= lg; ++i)
    {
        for (int j = 1; j <= sz; ++j)
        {
            if (d[j] >= (1 << i))
            {
                st[j][i] = st[st[j][i - 1]][i - 1];
            }
        }
    }
    return;
  }

  LCA(int x, int root)
  {
    d.resize(x + 1);
    st.resize(x + 1, vector<int>(32));
    dfs(root);
    build(x);
  }

  int query(int a, int b)
  {
    if (d[a] < d[b]) swap(a, b);
    int dif = d[a] - d[b];
    for (int i = 0; dif >> i; ++i)
    {
        if (dif >> i & 1) a = st[a][i];
    }
    if (a == b) return a;
    else
    {
        for (int i = 31; i >= 0; --i)
        {
            while (st[a][i] != st[b][i])
            {
                a = st[a][i];
                b = st[b][i];
            }
        }
        return st[a][0];
    }
  }
};
```

6.2 树的直径

1. 距离任一点最远的点一定是直径的一端
2. 时间复杂度：$O(n)$

```cpp
const int N = 200005;

struct Edge { int to; ll v; };
```

```cpp
vector<Edge> node[N];

pair<int, ll> farthest(int id, ll d, int pa)
{
    pair<int, ll> ret = { id,d };
    for (auto e : node[id])
    {
        pair<int, ll> res;
        if (e.to != pa) res = farthest(e.to, d + e.v, id);
        if (res.second > ret.second) ret = res;
    }
    return ret;
}

int n, m;

void solve()
{
    cin >> n >> m;
    int u, v;
    ll w;
    for (int i = 1; i <= m; ++i)
    {
        cin >> u >> v >> w;
        node[u].push_back({ v,w });
        node[v].push_back({ u,w });
    }
    int s = farthest(1, 0, -1).first;
    auto res = farthest(s, 0, -1);
    int t = res.first;
    ll d = res.second;
    return;
}
```

6.3 树哈希

1. 用于判断有根树同构
2. 无根树可通过找重心转换为有根树，若有两个重心需要同时考虑
3. 不同的树需要共用同一套 map
4. 时间复杂度：$O(\log n)$

```cpp
struct TreeHash
{
    int n, root;
    vector<vector<int>> node;
    vector<int> hav;
    map<vector<int>, int> mp;
    int ord = 0;

    void getTree(vector<int>& p)
    {
        n = p.size() - 1;
        node.clear();
        node.resize(n + 1);
        hav.clear();
        hav.resize(n + 1);
        root = -1;
        for (int i = 1; i <= n; ++i)
        {
            if (p[i])
            {
                node[p[i]].push_back(i);
                node[i].push_back(p[i]);
            }
            else root = i;
        }
        return;
    }

    void getD(int id, int pa, vector<int>& sz, vector<int>& d)
    {
        sz[id] = 1;
        int res = 0;
        for (auto e : node[id])
        {
            if (e != pa)
            {
                getD(e, id, sz, d);
                sz[id] += sz[e];
                res = max(res, sz[e]);
            }
        }
        if (id == root) d[id] = res;
        else d[id] = max(res, n - sz[id]);
        return;
    }

    vector<int> center()
    {
        vector<int> res;
        vector<int> sz(n + 1), d(n + 1);
        int mnn = n;
        getD(root, -1, sz, d);
        for (int i = 1; i <= n; ++i) mnn = min(mnn, d[i]);
        for (int i = 1; i <= n; ++i)
        {
            if (d[i] == mnn) res.push_back(i);
        }
        return res;
    }

    vector<int> hash(vector<int>& p)
    {
        vector<int> res;
        getTree(p);
        auto v = center();
        for (auto e : v) dfs(e, -1), res.push_back(hav[e]);
        sort(res.begin(), res.end());
        return res;
    }

    int hash(vector<int>& p, int root)
    {
        getTree(p);
        dfs(root, -1);
        return hav[root];
    }

    void dfs(int id, int pa)
    {
        vector<int> v;
        for (auto e : node[id])
        {
            if (e != pa)
            {
                dfs(e, id);
                v.push_back(hav[e]);
            }
        }
        sort(v.begin(), v.end());
        if (mp.count(v) == 0) mp[v] = ++ord;
        hav[id] = mp[v];
        return;
    }
};
```

6.4 树链剖分

1. 每个结点最多向上跳 $O(\log n)$ 次，但总链数为 $O(n)$
2. 重链结点的 DFS 序连续，通常由此配合线段树，维护树上两点间路径相关性质
3. 时间复杂度：$O(\log n)$

```cpp
const int N = 100005;

vector<int> node[N];

struct HLD
{
    vector<int> pa, dep, sz, hson;
    vector<int> top, dfn, rnk;
    int ord = 0;

    HLD(int x, int root)
    {
        pa.resize(x + 1);
        dep.resize(x + 1);
        sz.resize(x + 1);
        hson.resize(x + 1);
        top.resize(x + 1);
        dfn.resize(x + 1);
        rnk.resize(x + 1);
        build(root);
        decom(root);
    }

    void build(int x)
    {
        sz[x] = 1;
        int mxsz = 0;
        for (auto e : node[x])
        {
            if (e != pa[x])
            {
                pa[e] = x;
                dep[e] = dep[x] + 1;
                build(e);
                sz[x] += sz[e];
                if (sz[e] > mxsz)
                {
                    mxsz = sz[e];
                    hson[x] = e;
                }
            }
        }
        return;
    }

    void decom(int x)
    {
        top[x] = x;
        dfn[x] = ++ord;
        rnk[ord] = x;
        if (hson[pa[x]] == x) top[x] = top[pa[x]];
        for (auto e : node[x]) if (e == hson[x]) decom(e);
        for (auto e : node[x]) if (e != pa[x] && e != hson[x]) decom(e);
        return;
    }

    int lcm(int u, int v)
    {
        while (top[u] != top[v])
        {
            if (dep[u] < dep[v]) v = pa[top[v]];
            else u = pa[top[u]];
        }
        if (dep[u] < dep[v]) return u;
        else return v;
    }
};
```

6.5 树上启发式合并

1. 维护一个用于得出答案的状态，离线预处理每个子树的答案
2. 可以用遍历 DFS 序代替递归的贡献计算以优化常数
3. 时间复杂度：状态更新次数 $O(n \log n)$

```
const int N = 100005;

vector<int> node[N];

int n;
ll a[N];

struct DsuOnTree
{
    struct State
    {
        vector<int> cnt;
        map<int, ll> mp;
        State() { init(); }
        void init() { cnt.resize(1e5 + 1); }
        void add(ll val)
        {
            if (cnt[val]) mp[cnt[val]] -= val;
            if (mp[cnt[val]] == 0) mp.erase(cnt[val]);
            cnt[val]++;
            mp[cnt[val]] += val;
            return;
        }
        void del(ll val)
        {
            mp[cnt[val]] -= val;
            if (mp[cnt[val]] == 0) mp.erase(cnt[val]);
            cnt[val]--;
            if (cnt[val]) mp[cnt[val]] += val;
            return;
        }
        ll ans() { return mp.rbegin()->second; }
    } state;
    vector<int> big; // 每个结点的重子
    vector<int> sz; // 每个子树的大小
    vector<ll> ans; // 每个子树的答案
    const int root = 1;

    DsuOnTree()
    {
        big.resize(n + 1);
        sz.resize(n + 1);
        ans.resize(n + 1);
    }
    void dfs0(int x, int p)
    {
        sz[x] = 1;
        for (auto e : node[x])
        {
            if (e == p) continue;
            dfs0(e, x);
            sz[x] += sz[e];
            if (sz[big[x]] < sz[e]) big[x] = e;
        }
        return;
    }
    void del(int x, int p) // 删除子树贡献
    {
        state.del(a[x]);
        for (auto e : node[x])
        {
            if (e == p) continue;
            del(e, x);
        }
        return;
    }
    void add(int x, int p) // 计算子树贡献
    {
        state.add(a[x]);
        for (auto e : node[x])
        {
            if (e == p) continue;
            add(e, x);
        }
        return;
    }
    void dfs(int x, int p, bool keep)
    {
        for (auto e : node[x]) // 计算轻子子树答案
        {
            if (e == big[x] || e == p) continue;
            dfs(e, x, 0);
        }
        if (big[x]) dfs(big[x], x, 1); // 计算重子子树答案和贡献
        for (auto e : node[x]) // 计算轻子子树贡献
        {
            if (e == big[x] || e == p) continue;
            add(e, x);
        }
        state.add(a[x]); // 计算自己贡献
        ans[x] = state.ans(); // 计算答案
        if (keep == 0) del(x, p); // 删除子树贡献
        return;
    }
    void work()
    {
        dfs0(root, 0);
        dfs(root, 0, 0);
        return;
    }
};

void solve()
{
    cin >> n;
    for (int i = 1; i <= n; ++i) cin >> a[i];
    int u, v;
    for (int i = 1; i <= n - 1; ++i)
    {
        cin >> u >> v;
        node[u].push_back(v);
        node[v].push_back(u);
    }
    DsuOnTree dot;
    dot.work();
    for (int i = 1; i <= n; ++i) cout << dot.ans[i] << ' ';
    cout << endl;
    return;
}
```

6.6 点分治

1. 以重心为根分治子树，再考虑所有经过重心的路径
2. 通常用于树上路径计数问题
3. 时间复杂度：处理结点次数 $O(n \log n)$

```
const int N = 100005;
const int D[3][2] = { -1, 0, 1, -1, 0, 1 };

int n, sz[N], maxd[N];
string s;
vector<int> node[N];
bool vis[N];
multiset<pair<int, int>> st;

void getRoot(int x, int fa, int sum, int& root)
{
    sz[x] = 1, maxd[x] = 0;
    for (auto e : node[x])
    {
        if (vis[e] || e == fa) continue;
        getRoot(e, x, sum, root);
        sz[x] += sz[e];
        maxd[x] = max(maxd[x], sz[e]);
    }
    maxd[x] = max(maxd[x], sum - sz[x]);
    if (maxd[x] < maxd[root]) root = x;
    return;
}

void dfs(int x, int fa, pair<int, int> p)
{
    p.first += D[s[x] - 'a'][0];
    p.second += D[s[x] - 'a'][1];
    st.insert(p);
    for (auto e : node[x])
    {
        if (vis[e] || e == fa) continue;
        dfs(e, x, p);
    }
    return;
}

ll work(int x)
{
    ll res = 0;
    multiset<pair<int, int>> ns;
    for (auto e : node[x])
    {
        if (vis[e]) continue;
        dfs(e, x, make_pair(0, 0));
        for (auto p : st)
        {
            pair<int, int> inv;
            inv.first = -(p.first + D[s[x] - 'a'][0]);
            inv.second = -(p.second + D[s[x] - 'a'][1]);
            if (inv == make_pair(0, 0)) res++;
            res += ns.count(inv);
        }
        for (auto p : st) ns.insert(p);
        st.clear();
    }
    return res;
}

ll divide(int x)
{
    ll res = 0;
    vis[x] = 1;
    res += work(x);
    for (auto e : node[x])
    {
        if (vis[e]) continue;
        int root = 0;
        getRoot(e, x, sz[e], root);
        res += divide(root);
    }
    return res;
}

void solve()
{
    cin >> n >> s;
    s = ' ' + s;
    for (int i = 1; i <= n - 1; ++i)
    {
        int u, v;
        cin >> u >> v;
        node[u].push_back(v);
        node[v].push_back(u);
    }
    maxd[0] = n + 1;
    int root = 0;
    getRoot(1, 0, n, root);
    cout << divide(root) << '\n';
```

```
    return;
}
```

7 图论

7.1 2-SAT

1. 按照推导关系建有向图，判断是否有两个矛盾点在同一强连通分量中
2. 需要以结点 $[1, 2n]$ 建图，最后可以得到一组合法构造
3. 时间复杂度：$O(n+m)$

```cpp
const int N = 2000005;

vector<int> node[N];

struct Tarjan
{
    int sz, cnt, ord;
    stack<int> stk;
    vector<vector<int>> g; // 新图
    vector<int> dfn, low, id, val;
    Tarjan(int x)
    {
        sz = x; // 点数
        cnt = 0; // 强连通分量个数
        ord = 0; // 时间戳
        dfn.resize(sz + 1); // dfs序
        low.resize(sz + 1); // 能到达的最小dfn
        id.resize(sz + 1); // 对应的强连通分量编号
        val.resize(sz + 1); // 新图点权
    }
    void dfs(int x)
    {
        stk.push(x);
        dfn[x] = low[x] = ++ord;
        for (auto e : node[x])
        {
            if (dfn[e] == 0)
            {
                dfs(e);
                low[x] = min(low[x], low[e]);
            }
            else if (id[e] == 0)
            {
                low[x] = min(low[x], low[e]);
            }
        }
        if (dfn[x] == low[x]) // x为强连通分量的根
        {
            cnt++;
            while (dfn[stk.top()] != low[stk.top()])
            {
                id[stk.top()] = cnt;
                stk.pop();
            }
            id[stk.top()] = cnt;
            stk.pop();
        }
        return;
    }
    void shrink()
    {
        for (int i = 1; i <= sz; ++i)
        {
            if (id[i] == 0) dfs(i);
        }
        return;
    }
    void rebuild()
    {
        for (int i = 1; i <= sz; ++i)
        {
            for (auto e : node[i])
            {
                if (id[i] != id[e]) g[id[i]].push_back(id[e]);
            }
        }
        return;
    }
};

struct TwoSat
{
    int sz;
    vector<int> res;
    inline int negate(int x)
    {
        if (x > sz) return x - sz;
        else return x + sz;
    }
    TwoSat(int x)
    {
        sz = x;
        res.resize(sz + 1);
    }
    bool work()
    {
        Tarjan tj(sz * 2);
        tj.shrink();
        for (int i = 1; i <= sz; ++i)
        {
            if (tj.id[i] == tj.id[negate(i)]) return 0;
        }
        for (int i = 1; i <= sz; ++i)
        {
            res[i] = tj.id[i] < tj.id[negate(i)];
        }
        return 1;
    }
};

void solve() // P4782
{
    ll n, m;
    cin >> n >> m;
    for (int i = 1; i <= m; ++i)
    {
        bool a, b;
        ll x, y;
        cin >> x >> a >> y >> b;
        node[x + a * n].push_back(y + (!b) * n);
        node[y + b * n].push_back(x + (!a) * n);
    }
    TwoSat ts(n);
    if (!ts.work()) cout << "IMPOSSIBLE\n";
    else
    {
        cout << "POSSIBLE\n";
        for (int i = 1; i <= n; ++i) cout << ts.res[i] << ' ';
    }
    return;
}
```

7.2 Bellman-Ford 算法

1. 适用于任何边权的单源最短路问题
2. 求出最短路后可判断负环
3. 时间复杂度：$O(nm)$

```cpp
const int N = 1505;
const ll INFLL = 0x3f3f3f3f3f3f3f3f;

struct Edge {ll to, v;};

vector<Edge> node[N];

struct BellmanFord
{
    int sz;
    vector<ll> dis;

    BellmanFord(int x)
    {
        sz = x;
        dis.resize(sz + 1, INFLL);
    }

    void work(int s)
    {
        dis[s] = 0;
        for (int i = 1; i <= sz - 1; ++i)
        {
            for (int j = 1; j <= sz; ++j)
            {
                for (auto e : node[j])
                {
                    dis[e.to] = min(dis[e.to], dis[j] + e.v);
                }
            }
        }
        return;
    }

    bool negCir()
    {
        for (int i = 1; i <= sz; ++i)
        {
            for (auto e : node[i])
            {
                if (dis[e.to] > dis[i] + e.v) return 1;
            }
        }
        return 0;
    }
};
```

7.3 Dijkstra 算法

1. 只适用于边权非负的图
2. 注意特判图不连通的情况
3. 时间复杂度：朴素 $O(n^2)$/堆优化 $O(m \log m)$

```cpp
const int N = 100005;
const ll INFLL = 0x3f3f3f3f3f3f3f3f;

struct Edge { int to, v; };

vector<Edge> node[N];

struct Dijkstra
{
    struct Node
    {
        int id;
        ll d;
        bool operator < (const Node& p1) const
        {
```

```
        return d > p1.d;
    }
};

int sz;
vector<int> vis;
vector<ll> dis;

Dijkstra(int x)
{
    sz = x;
    vis.resize(sz + 1);
    dis.resize(sz + 1, INFLL);
}

void workO(int s) // 堆优化
{
    priority_queue<Node> pq;
    dis[s] = 0;
    pq.push({ s,0 });
    while (pq.size())
    {
        int now = pq.top().id;
        pq.pop();
        if (vis[now] == 0)
        {
            vis[now] = 1; // 被取出一定是最短路
            for (auto e : node[now])
            {
                if (vis[e.to] == 0 && dis[e.to] > dis[now] + e.v)
                {
                    dis[e.to] = dis[now] + e.v;
                    pq.push({ e.to,dis[e.to] });
                }
            }
        }
    }
    return;
}

void workS(int s) // 朴素
{
    auto take = [&](int x)
    {
        vis[x] = 1;
        for (auto e : node[x])
        {
            dis[e.to] = min(dis[e.to], dis[x] + e.v);
        }
        return;
    };
    dis[s] = 0;
    take(s);
    for (int i = 1; i <= sz - 1; ++i)
    {
        ll mnn = INFLL;
        int id = 0;
        for (int j = 1; j <= sz; ++j)
        {
            if (vis[j] == 0 && dis[j] < mnn)
            {
                mnn = dis[j];
                id = j;
            }
        }
        if (mnn == INFLL) return;
        take(id);
    }
    return;
}
};
```

7.4 Floyd 算法

1. 多源最短路、最短路计数、最小环计数
2. 时间复杂度：$O(n^3)$

```
const int N = 505;
const int MOD = 998244353;
const ll INFLL = 0x3f3f3f3f3f3f3f3f;

int n, m;
ll cnt[N][N]; // 最短条数
ll dis[N][N]; // 最短路长度
ll edg[N][N]; // 边长

void solve()
{
    cin >> n >> m;
    for (int i = 1; i <= n; ++i)
    {
        for (int j = 1; j <= n; ++j)
        {
            if (i == j) dis[i][j] = 0;
            else dis[i][j] = INFLL;
            cnt[i][j] = 0;
            edg[i][j] = 0;
        }
    }
    for (int i = 1; i <= m; ++i)
    {
        int u, v, w;
        cin >> u >> v >> w;
        dis[u][v] = edg[u][v] = w;
        cnt[u][v] = 1;
    }
    map<ll, ll> ans;
    for (int k = 1; k <= n; ++k)
    {
        // 用指向最大编号点的边作为一个环的代表
        for (int i = 1; i < k; ++i)
        {
            if (edg[i][k] && cnt[k][i])
            {
                ans[edg[i][k] + dis[k][i]] += cnt[k][i];
                ans[edg[i][k] + dis[k][i]] %= MOD;
            }
        }
        // 最短路计数
        for (int i = 1; i <= n; ++i)
        {
            for (int j = 1; j <= n; ++j)
            {
                if (dis[i][k] + dis[k][j] < dis[i][j])
                {
                    dis[i][j] = dis[i][k] + dis[k][j];
                    cnt[i][j] = cnt[i][k] * cnt[k][j] % MOD;
                }
                else if (dis[i][j] == dis[i][k] + dis[k][j])
                {
                    cnt[i][j] += cnt[i][k] * cnt[k][j] % MOD;
                    cnt[i][j] %= MOD;
                }
            }
        }
    }
    if (ans.empty()) cout << "-1 -1\n";
    else cout << ans.begin()->first << ' ' << ans.begin()->second << '\n';
    return;
}
```

7.5 Kosaraju 算法

1. 求有向图强连通分量
2. 时间复杂度：$O(n+m)$

```
const int N = 10005;

vector<int> node[N];

struct Kosaraju
{
    int sz, index = 0;
    vector<int> vis, ord;
    vector<vector<int>> rev;
    vector<int> id; // 强连通分量编号
    Kosaraju(int x)
    {
        sz = x;
        vis.resize(sz + 1);
        id.resize(sz + 1);
        rev.resize(sz + 1);
        ord.resize(1);
        for (int i = 1; i <= sz; ++i)
        {
            for (auto e : node[i])
            {
                rev[e].push_back(i);
            }
        }
        for (int i = 1; i <= sz; ++i) if (vis[i] == 0) dfs1(i);
        for (int i = sz; i >= 1; --i) if (id[ord[i]] == 0) index++, dfs2(ord[i]);
    }

    void dfs1(int x)
    {
        vis[x] = 1;
        for (auto e : node[x])
        {
            if (vis[e] == 0) dfs1(e);
        }
        ord.push_back(x);
        return;
    }

    void dfs2(int x)
    {
        id[x] = index;
        for (auto e : rev[x])
        {
            if (id[e] == 0) dfs2(e);
        }
        return;
    }
};
```

7.6 Hierholzer 算法

1. 求欧拉通路，支持重边、有向边
2. 使用前需要保证欧拉通路存在，且从其端点开始 DFS
3. 欧拉通路存在当且仅当奇数度的结点有 0 个或 2 个
4. DFS 后栈内为欧拉通路的倒序，需要进行翻转
5. 时间复杂度：$O(n+m)$

```
int vis[M];
vector<int> node[N];
vector<int> stk;

void dfs(int x)
{
```

```
    for (auto e : node[x])
    {
        if (vis[e.second]) continue;
        vis[e.second] = 1;
        dfs(e.first);
    }
    stk.push_back(x);
}
```

7.7 Tarjan 算法

1. 时间复杂度：$O(n+m)$

```
struct SCC // 有向图强连通分量+缩点
{
    int sz, cnt, ord;
    stack<int> stk;
    vector<int> dfn, low, id;
    vector<vector<int>> g; // 新图
    SCC(int x)
    {
        sz = x; // 点数
        cnt = 0; // 连通分量个数
        ord = 0; // 时间戳
        dfn.resize(sz + 1); // dfs序
        low.resize(sz + 1); // 能到达的最小dfn
        id.resize(sz + 1); // 连通分量编号
    }
    void dfs(int x)
    {
        stk.push(x);
        dfn[x] = low[x] = ++ord;
        for (auto e : node[x])
        {
            if (dfn[e] == 0) // 未访问过
            {
                dfs(e);
                low[x] = min(low[x], low[e]);
            }
            else if (id[e] == 0) // 在栈中
            {
                low[x] = min(low[x], dfn[e]);
            }
        }
        if (dfn[x] == low[x]) // x为强连通分量的根
        {
            cnt++;
            while (stk.top() != x)
            {
                id[stk.top()] = cnt;
                stk.pop();
            }
            id[stk.top()] = cnt;
            stk.pop();
        }
        return;
    }
    void shrink()
    {
        for (int i = 1; i <= sz; ++i)
        {
            if (id[i] == 0) dfs(i);
        }
        return;
    }
    void rebuild()
    {
        g.resize(cnt + 1);
        for (int i = 1; i <= sz; ++i)
        {
            for (auto e : node[i])
            {
                if (id[i] != id[e]) g[id[i]].push_back(id[e]);
            }
        }
        return;
    }
};

struct VBCC // 无向图点双连通分量和割点
{
    int sz, ord;
    stack<int> stk;
    vector<int> dfn, low, tag;
    vector<vector<int>> bcc;
    VBCC(int x)
    {
        sz = x; // 点数
        ord = 0; // 时间戳
        dfn.resize(sz + 1); // dfs序
        low.resize(sz + 1); // 能到达的最小dfn
        tag.resize(sz + 1); // 是否割点
    }
    void dfs(int x, int fa)
    {
        stk.push(x);
        dfn[x] = low[x] = ++ord;
        int son = 0;
        for (auto e : node[x])
        {
            if (dfn[e] == 0) // 未访问过
            {
                son++;
                dfs(e, x);
                low[x] = min(low[x], low[e]);
                if (low[e] >= dfn[x]) // x可能是割点
                {
                    if (fa) tag[x] = 1; // 不是dfs的根，则为割点
                    bcc.emplace_back();
                    while (stk.top() != e)
                    {
                        bcc.back().push_back(stk.top());
                        stk.pop();
                    }
                    bcc.back().push_back(stk.top());
                    stk.pop();
                    bcc.back().push_back(x);
                }
            }
            else if (e != fa) // 祖先
            {
                low[x] = min(low[x], dfn[e]);
            }
        }
        if (fa == 0 && son >= 2) tag[x] = 1; // 特判dfs根是否为割点
        if (fa == 0 && son == 0) bcc.emplace_back(1, x); // 特判dfs根是否单独为一个分量
        return;
    }
    void work()
    {
        for (int i = 1; i <= sz; ++i)
        {
            if (dfn[i]) continue;
            while (stk.size()) stk.pop();
            dfs(i, 0);
        }
        return;
    }
};

struct EBCC // 无向图边双连通分量和割边
{
    int sz, ord;
    vector<int> dfn, low, tag, vis;
    vector<vector<int>> bcc;
    EBCC(int x, int y)
    {
        sz = x; // 点数
        ord = 0; // 时间戳
        dfn.resize(sz + 1); // dfs序
        low.resize(sz + 1); // 能到达的最小dfn
        vis.resize(sz + 1); // 是否已加入连通分量
        tag.resize(y + 1); // 是否割边
    }
    void dfs0(int x, int fa)
    {
        dfn[x] = low[x] = ++ord;
        for (auto e : node[x])
        {
            if (dfn[e.to] == 0) // 未访问过
            {
                dfs0(e.to, x);
                low[x] = min(low[x], low[e.to]);
                if (low[e.to] > dfn[x]) tag[e.id] = 1; // 是割边
            }
            else if (e.to != fa) // 祖先
            {
                low[x] = min(low[x], dfn[e.to]);
            }
        }
        return;
    }
    void dfs(int x)
    {
        bcc.back().push_back(x);
        vis[x] = 1;
        for (auto e : node[x])
        {
            if (vis[e.to]) continue;
            if (tag[e.id]) continue;
            dfs(e.to);
        }
        return;
    }
    void work()
    {
        for (int i = 1; i <= sz; ++i)
        {
            if (dfn[i]) continue;
            dfs0(i, 0);
        }
        for (int i = 1; i <= sz; ++i)
        {
            if (vis[i]) continue;
            bcc.emplace_back();
            dfs(i);
        }
        return;
    }
};
```

7.8 圆方树

1. 对点双中的任意三点 a,b,c，一定存在 $a\to b\to c$ 的简单路径
2. 时间复杂度：$O(n+m)$

```
int n, m;
vector<int> node[N];

struct RSTree
{
    int sz, ord, cnt;
    stack<int> stk;
    vector<int> dfn, low, tag;
    vector<vector<int>> g;
    RSTree(int x)
    {
        cnt = x; // 方点编号
```

```
        sz = x; // 点数
        ord = 0; // 时间戳
        dfn.resize(sz + 1); // dfs序
        low.resize(sz + 1); // 能到达的最小dfn
        g.resize(sz * 2 + 1); // 圆方树
    }
    void dfs(int x, int fa)
    {
        stk.push(x);
        dfn[x] = low[x] = ++ord;
        for (auto e : node[x])
        {
            if (dfn[e] == 0) // 未访问过
            {
                dfs(e, x);
                low[x] = min(low[x], low[e]);
                if (low[e] >= dfn[x])
                {
                    cnt++;
                    while (stk.top() != e)
                    {
                        g[cnt].push_back(stk.top());
                        g[stk.top()].push_back(cnt);
                        stk.pop();
                    }
                    g[cnt].push_back(stk.top());
                    g[stk.top()].push_back(cnt);
                    stk.pop();
                    g[cnt].push_back(x);
                    g[x].push_back(cnt);
                }
            }
            else if (e != fa) // 祖先
            {
                low[x] = min(low[x], dfn[e]);
            }
        }
        return;
    }
    void work()
    {
        for (int i = 1; i <= sz; ++i)
        {
            if (dfn[i]) continue;
            while (stk.size()) stk.pop();
            dfs(i, 0);
        }
        return;
    }
};
```

7.9 K 短路

1. 利用 A* 算法，以估价函数值优先搜索，第 k 次访问某结点的路径即 k 短路
2. 时间复杂度：$O(nk\log n)$

```
const int N = 1005;
const ll INFLL = 0x3f3f3f3f3f3f3f3f;

struct E
{
    ll to, v;
};

struct V
{
    ll id, d;
    bool operator<(const V& v) const { return d > v.d; }
};

int n, m, k;
vector<E> node[N];

struct Dijkstra
{
    int sz;
    vector<ll> d;
    vector<int> vis;
    priority_queue<V> pq;
    vector<vector<E>> rev;

    void rebuild()
    {
        for (int i = 1; i <= sz; ++i)
        {
            for (auto e : node[i])
            {
                rev[e.to].push_back({ i,e.v });
            }
        }
        return;
    }
    Dijkstra(int x, int s)
    {
        sz = x;
        d.resize(sz + 1, INFLL);
        vis.resize(sz + 1);
        rev.resize(sz + 1);
        rebuild();
        d[1] = 0;
        pq.push({ 1,0 });
        while (pq.size())
        {
            auto now = pq.top();
            pq.pop();
            if (vis[now.id]) continue;
            vis[now.id] = 1;
            for (auto e : rev[now.id])
            {
                if (vis[e.to] == 0 && d[e.to] > d[now.id] + e.v)
                {
                    d[e.to] = d[now.id] + e.v;
                    pq.push({ e.to, d[e.to] });
                }
            }
        }
    }
};

void solve()
{
    cin >> n >> m >> k;
    int u, v, w;
    for (int i = 1; i <= m; ++i)
    {
        cin >> u >> v >> w;
        node[u].push_back({ v,w });
    }
    Dijkstra dj(n, n);
    priority_queue<V> pq;
    vector<int> vis(n + 1);
    pq.push({ n,dj.d[n] });
    vector<ll> ans(k, -1);
    while (pq.size())
    {
        auto now = pq.top();
        pq.pop();
        if (now.id == 1 && vis[now.id] < k) ans[vis[now.id]] = now.d;
        vis[now.id]++;
        for (auto e : node[now.id])
        {
            if (vis[e.to] >= k) continue;
            pq.push({ e.to,now.d - dj.d[now.id] + e.v + dj.d[e.to] });
        }
    }
    for (int i = 0; i < k; ++i) cout << ans[i] << '\n';
    return;
}
```

7.10 Dinic 算法

1. 求有向网络最大流/最小割，可应用于二分图最大匹配
2. `cap` 表示残量，`cap` 为 0 的边满流
3. 时间复杂度：最差 $O(n^2m)$/二分图匹配 $O(m\sqrt{n})$

```
const ll INFLL = 0x3f3f3f3f3f3f3f3f;
const int N = 3005;

struct Edge
{
    int to; // 终点
    int rev; // 反向边对其起点的编号
    ll cap; // 残量
    Edge() {}
    Edge(int to, int rev, ll cap) :to(to), rev(rev), cap(cap) {}
};

vector<Edge> node[N];

void AddEdge(int from, int to, ll cap)
{
    int x = node[to].size();
    int y = node[from].size();
    node[from].push_back(Edge(to, x, cap));
    node[to].push_back(Edge(from, y, 0));
    return;
}

struct Dinic
{
    int sz;
    vector<int> dep; // 每个点所属层深度
    vector<int> done; // 每个点下一个要处理的邻接边
    queue<int> q;

    Dinic(int x)
    {
        sz = x;
        dep.resize(sz + 1);
        done.resize(sz + 1);
    }

    bool bfs(int s, int t) // 建立分层图
    {
        for (int i = 1; i <= sz; ++i) dep[i] = 0;
        q.push(s);
        dep[s] = 1;
        done[s] = 0;
        bool f = 0;
        while (q.size())
        {
            int now = q.front();
            q.pop();
            if (now == t) f = 1; // 到达终点说明存在增广路
            for (auto e : node[now])
            {
                if (e.cap && dep[e.to] == 0) // 还有残量且未访问过
                {
                    q.push(e.to);
                    done[e.to] = 0; // 有增广路，需要重新处理
                    dep[e.to] = dep[now] + 1;
                }
            }
        }
        return f;
    }
```

```cpp
    ll dfs(int x, int t, ll flow) // 统计增广路总流量
    {
        if (x == t || flow == 0) return flow; // 找到汇点或断流
        ll rem = flow; // 结点x当前剩余流量
        for (int i = done[x]; i < node[x].size() && rem; ++i)
        {
            done[x] = i; // 前i-1条边已经搞定，不会再有增广路
            auto& e = node[x][i];
            if (e.cap && dep[e.to] == dep[x] + 1)// 还有残量且为下一层
            {
                ll inflow = dfs(e.to, t, min(rem, e.cap)); // 计算流向e.to的最大流量
                if (inflow == 0) dep[e.to] = 0; // e.to无法流入，本次增广不再考虑
                e.cap -= inflow; // 更新残量
                node[e.to][e.rev].cap += inflow; // 更新反向边
                rem -= inflow; // 消耗流量
            }
        }
        return flow - rem;
    }

    ll work(int s, int t)
    {
        ll aug = 0, ans = 0;
        while (bfs(s, t))
        {
            while (aug = dfs(s, t, INFLL)) ans += aug;
        }
        return ans;
    }
};
```

7.11 SSP 算法

1. 求最小费用最大流
2. 无法处理负环，需要用强制满流法预处理：先将负权边手动置为满流（反向建边即可）并计入答案，再引入虚拟源点和虚拟汇点，使虚拟源点连向终点，起点连向虚拟汇点，跑一遍最大流（注意清空流量）
3. 时间复杂度：$O(nmF)$（伪多项式，与最大流有关）

```cpp
const int N = 5005;
const ll INFLL = 0x3f3f3f3f3f3f3f3f;

struct Edge
{
    int to; // 终点
    int rev; // 反向边对其起点的编号
    ll cap; // 残量
    ll cost; // 单位流量费用
    Edge() {}
    Edge(int to, int rev, ll cap, ll cost) :to(to), rev(rev), cap(cap), cost(
        cost) {}
};

vector<Edge> node[N];

void addEdge(int from, int to, ll cap, ll cost)
{
    int x = node[to].size();
    int y = node[from].size();
    node[from].push_back(Edge(to, x, cap, cost));
    node[to].push_back(Edge(from, y, 0, -cost));
    return;
}

struct SSP
{
    int sz;
    vector<ll> dis; // 源点到i的最小单位流量费用
    vector<int> vis;
    vector<int> done; // 每个点下一个要处理的邻接边
    queue<int> q;
    ll minc, maxf;

    SSP(int x)
    {
        sz = x;
        dis.resize(sz + 1);
        vis.resize(sz + 1);
        done.resize(sz + 1);
        minc = maxf = 0;
    }

    bool spfa(int s, int t) // 寻找单位流量费用最小的增广路
    {
        vis.assign(sz + 1, 0);
        done.assign(sz + 1, 0);
        dis.assign(sz + 1, INFLL);
        dis[s] = 0;
        q.push(s);
        vis[s] = 1;
        while (q.size())
        {
            int now = q.front();
            q.pop();
            vis[now] = 0;
            for (auto e : node[now])
            {
                if (e.cap && dis[e.to] > dis[now] + e.cost) // 还有残量且可松弛
                {
                    dis[e.to] = dis[now] + e.cost;
                    if (vis[e.to] == 0) q.push(e.to), vis[e.to] = 1;
                }
            }
        }
        return dis[t] != INFLL;
    }

    ll dfs(int x, int p, int t, ll flow) // 沿增广路计算流量和费用
    {
        if (x == t || flow == 0) return flow; // 找到汇点或断流
        vis[x] = 1; // 防止零权环死循环
        ll rem = flow; // 结点x当前剩余流量
        for (int i = done[x]; i < node[x].size() && rem; ++i)
        {
            done[x] = i; // 前i-1条边已经搞定，不会再有增广路
            auto& e = node[x][i];
            if (e.to != p && vis[e.to] == 0 && e.cap && dis[e.to] == dis[x] + e.
                cost)
            {
                ll inflow = dfs(e.to, x, t, min(rem, e.cap)); // 计算流向e.to的最大
                    流量
                e.cap -= inflow; // 更新残量
                node[e.to][e.rev].cap += inflow; // 更新反向边
                rem -= inflow; // 消耗流量
            }
        }
        vis[x] = 0; // 出递归栈后可重新访问
        return flow - rem;
    }

    void work(int s, int t)
    {
        ll aug = 0;
        while (spfa(s, t))
        {
            while (aug = dfs(s, 0, t, INFLL))
            {
                maxf += aug;
                minc += dis[t] * aug;
            }
        }
        return;
    }
};
```

7.12 原始对偶算法

1. 求最小费用最大流
2. 对负环的处理同 SSP 算法
3. 时间复杂度：$O(mF\log m)$（伪多项式，与最大流有关）

```cpp
const int N = 5005;
const ll INFLL = 0x3f3f3f3f3f3f3f3f;

struct Edge
{
    int to; // 终点
    int rev; // 反向边对其起点的编号
    ll cap; // 残量
    ll cost; // 单位流量费用
    Edge() {}
    Edge(int to, int rev, ll cap, ll cost) :to(to), rev(rev), cap(cap), cost(
        cost) {}
};

vector<Edge> node[N];

void addEdge(int from, int to, ll cap, ll cost)
{
    int x = node[to].size();
    int y = node[from].size();
    node[from].push_back(Edge(to, x, cap, cost));
    node[to].push_back(Edge(from, y, 0, -cost));
    return;
}

struct PrimalDual
{
    struct NodeInfo
    {
        int id;
        ll d;
        bool operator < (const NodeInfo& p1) const
        {
            return d > p1.d;
        }
    };

    int sz;
    vector<ll> h; // 势能
    vector<int> vis;
    vector<int> done; // 每个点下一个要处理的邻接边
    vector<ll> dis;
    queue<int> q;
    priority_queue<NodeInfo> pq;
    ll minc, maxf;

    PrimalDual(int x)
    {
        sz = x;
        h.resize(sz + 1, INFLL);
        vis.resize(sz + 1);
        done.resize(sz + 1);
        dis.resize(sz + 1);
        minc = maxf = 0;
    }

    void spfa(int s) // 求初始势能
    {
        h[s] = 0;
        q.push(s);
        vis[s] = 1;
        while (q.size())
```

```cpp
        {
            auto now = q.front();
            q.pop();
            vis[now] = 0;
            for (auto e : node[now])
            {
                if (e.cap && h[e.to] > h[now] + e.cost)
                {
                    h[e.to] = h[now] + e.cost;
                    if (vis[e.to] == 0) q.push(e.to), vis[e.to] = 1;
                }
            }
        }
        return;
    }

    bool dijkstra(int s, int t)
    {
        dis.assign(sz + 1, INFLL);
        vis.assign(sz + 1, 0);
        done.assign(sz + 1, 0);
        dis[s] = 0;
        pq.push({ s,0 });
        while (pq.size())
        {
            int now = pq.top().id;
            pq.pop();
            if (vis[now] == 0)
            {
                vis[now] = 1; // 被取出一定是最短路
                for (auto e : node[now])
                {
                    ll cost = e.cost + h[now] - h[e.to];
                    if (vis[e.to] == 0 && e.cap && dis[e.to] > dis[now] + cost)
                    {
                        dis[e.to] = dis[now] + cost;
                        pq.push({ e.to,dis[e.to] });
                    }
                }
            }
        }
        vis.assign(sz + 1, 0); // 还原vis
        return dis[t] != INFLL;
    }

    ll dfs(int x, int t, ll flow) // 沿增广路计算流量和费用
    {
        if (x == t || flow == 0) return flow; // 找到汇点或断流
        vis[x] = 1; // 防止零权环死循环
        ll rem = flow; // 结点x当前剩余流量
        for (int i = done[x]; i < node[x].size() && rem; ++i)
        {
            done[x] = i; // 前i-1条边已经搞定，不会再有增广路
            auto& e = node[x][i];
            if (vis[e.to] == 0 && e.cap && e.cost == h[e.to] - h[x]) // 势能差等于
                费用表明是最短路
            {
                ll inflow = dfs(e.to, t, min(rem, e.cap)); // 计算流向e.to的最大流量
                e.cap -= inflow; // 更新残量
                node[e.to][e.rev].cap += inflow; // 更新反向边
                rem -= inflow; // 消耗流量
            }
        }
        vis[x] = 0; // 出递归栈后可重新访问
        return flow - rem;
    }

    void work(int s, int t)
    {
        spfa(s);
        ll aug = 0;
        while (dijkstra(s, t))
        {
            for (int i = 1; i <= sz; ++i) h[i] += dis[i]; // 更新势能
            while (aug = dfs(s, t, INFLL))
            {
                maxf += aug;
                minc += aug * h[t];
            }
        }
        return;
    }
};
```

7.13 Prim 算法

1. 选点法最小生成树，适用于稠密图
2. 注意特判图不连通的情况
3. 时间复杂度：$O(n^2)$

```cpp
const int N = 5005;
const int M = 200005;

const ll INFLL = 0x3f3f3f3f3f3f3f3f;

struct Edge {ll to, v;};

vector<Edge> node[N];
int n, m;

struct Prim
{
    int sz;
    vector<int> vis;
    vector<ll> dis;

    Prim(int x)
    {
        sz = x;
        vis.resize(sz + 1);
        dis.resize(sz + 1, INFLL);
    }

    ll work()
    {
        int now = 1;
        ll ans = 0;
        for (int i = 1; i <= sz - 1; ++i)
        {
            vis[now] = 1;
            for (auto e : node[now])
            {
                dis[e.to] = min(dis[e.to], e.v);
            }
            ll mnn = INFLL;
            for (int j = 1; j <= sz; ++j)
            {
                if (vis[j] == 0 && dis[j] < mnn)
                {
                    mnn = dis[j];
                    now = j;
                }
            }
            if (mnn == INFLL) return 0; // 不连通
            ans += mnn;
        }
        return ans;
    }
};
```

7.14 Kruskal 算法

1. 选边法最小生成树，适用于稀疏图
2. 注意特判图不连通的情况
3. 时间复杂度：$O(m\log m)$

```cpp
const int N = 5005;
const int M = 200005;

struct Edge
{
    ll x, y, v;
    bool operator <(const Edge& e)
    {
        return v < e.v;
    }
};

Edge e[M];
int n, m;

ll kruskal()
{
    DSU dsu(n);
    ll ans = 0;
    sort(e + 1, e + 1 + m);
    for (int i = 1; i <= m; ++i)
    {
        if (dsu.find(e[i].x) != dsu.find(e[i].y))
        {
            ans += e[i].v;
            dsu.merge(e[i].x, e[i].y);
        }
    }
    return ans;
}
```

7.15 Kruskal 重构树

1. 用于解决最小瓶颈路问题
2. 时间复杂度：建立 $O(n)$/查询 $O(\log n)$

```cpp
const int N = 100005;

struct DSU
{
    vector<int> f;
    void init(int x)
    {
        f.resize(x + 1);
        for (int i = 1; i <= x; ++i) f[i] = i;
        return;
    }
    int find(int id) { return f[id] == id ? id : f[id] = find(f[id]); }
    void attach(int x, int y) // 将fx连向fy，不按秩合并
    {
        int fx = find(x), fy = find(y);
        f[fx] = fy;
        return;
    }
};

struct LCA
{
    vector<int> d;
    vector<vector<int>> st;
    void dfs(int x, vector<vector<int>>& son)
    {
        for (auto e : son[x])
        {
```

```
        d[e] = d[x] + 1;
        st[e][0] = x;
        dfs(e, son);
      }
      return;
    }
    void build(int x)
    {
      int lg = int(log2(x));
      for (int i = 1; i <= lg; ++i)
      {
        for (int j = 1; j <= x; ++j)
        {
          if (d[j] >= (1 << i))
          {
            st[j][i] = st[st[j][i - 1]][i - 1];
          }
        }
      }
      return;
    }
    void init(int x)
    {
      d.resize(x + 1);
      st.resize(x + 1, vector<int>(32));
      return;
    }
    int query(int x, int y)
    {
      if (d[x] < d[y]) swap(x, y);
      int dif = d[x] - d[y];
      for (int i = 0; dif >> i; ++i)
      {
        if (dif >> i & 1) x = st[x][i];
      }
      if (x == y) return x;
      for (int i = 31; i >= 0; --i)
      {
        while (st[x][i] != st[y][i])
        {
          x = st[x][i];
          y = st[y][i];
        }
      }
      return st[x][0];
    }
};

struct Edge
{
    ll x, y, v;
    bool operator<(const Edge& rhs) const { return v < rhs.v; }
} edg[N];

struct KrsRebTree
{
    int size; // 当前结点数, 最多为n*2-1
    vector<vector<int>> son; // 子结点
    vector<ll> val; // 点权
    LCA lca;
    DSU dsu;

    void build(int n, int m)
    {
      son.resize(n * 2);
      val.resize(n * 2);
      dsu.init(n * 2 - 1);
      size = n;
      sort(edg + 1, edg + 1 + m);
      for (int i = 1; i <= m && size < n * 2 - 1; ++i)
      {
        int fx = dsu.find(edg[i].x);
        int fy = dsu.find(edg[i].y);
        if (fx == fy) continue;
        size++;
        dsu.attach(fx, size);
        dsu.attach(fy, size);
        son[size].push_back(fx);
        son[size].push_back(fy);
        val[size] = edg[i].v;
      }
      lca.init(size);
      for (int i = n + 1; i <= size; ++i)
      {
        if (dsu.find(i) == i) lca.dfs(i, son); // 对所有树的根dfs
      }
      lca.build(size);
      return;
    }
    ll query(int x, int y)
    {
      if (dsu.find(x) == dsu.find(y)) return val[lca.query(x, y)];
      else return -1;
    }
};
```

8 计算几何

8.1 二维整数坐标相关

```
const ll INF = 1e18;

struct P
{
    ll x, y;

    P(): x(0), y(0) {}
    P(ll x, ll y): x(x), y(y) {}
    P operator-(const P& rhs) const { return P(x - rhs.x, y - rhs.y); }
    P operator+(const P& rhs) const { return P(x + rhs.x, y + rhs.y); }
    ll operator*(const P& rhs) const { return x * rhs.x + y * rhs.y; }
    ll len2() { return *this * *this; }
};

ll sqr(ll x) { return x * x; }
ll dis2(const P& p1, const P& p2) { return (p1 - p2).len2(); }
ll cross(const P& p1, const P& p2) { return p1.x * p2.y - p2.x * p1.y; }

ll closest(vector<P>& p) // 最近点对, P7883
{
    sort(p.begin(), p.end(), [](auto x, auto y) { return x.x < y.x; });
    function<ll(int, int)> work = [&](int lef, int rig)
    {
      if (lef == rig - 1) return INF;
      int mid = lef + (rig - lef) / 2;
      ll midx = p[mid].x;
      ll low = min(work(lef, mid), work(mid, rig));
      int lp = lef, rp = mid;
      vector<P> v;
      while (lp < mid || rp < rig)
      {
        if (lp < mid && (rp == rig || p[rp].y > p[lp].y)) v.push_back(p[lp++])
          ;
        else v.push_back(p[rp++]);
      }
      for (int i = lef; i < rig; ++i) p[i] = v[i - lef];
      v.clear();
      for (int i = lef; i < rig; ++i)
      {
        if (sqr(abs(p[i].x - midx)) < low) v.push_back(p[i]);
      }
      for (int i = 1; i < v.size(); ++i)
      {
        for (int j = i - 1; j >= 0; --j)
        {
          if (sqr(v[i].y - v[j].y) >= low) break;
          low = min(low, dis2(v[i], v[j]));
        }
      }
      return low;
    };
    return work(0, p.size());
}

ll diameter(vector<P>& p) // 凸包直径
{
    // m >= 3 & counterclockwise
    int m = p.size(), k = 1;
    ll res = 0;
    for (int i = 0; i < m; ++i)
    {
      while (cross(p[(i + 1) % m] - p[i], p[k] - p[i]) <= cross(p[(i + 1) % m]
          - p[i], p[(k + 1) % m] - p[i]))
        k = (k + 1) % m;
      res = max(res, dis2(p[i], p[k]));
      res = max(res, dis2(p[(i + 1) % m], p[k]));
    }
    return res;
}

vector<P> convex(vector<P>& p) // 求凸包
{
    // m >= 2, least points & counterclockwise
    int m = p.size();
    sort(p.begin(), p.end(),
        [](auto x, auto y)
        {
          if (x.x == y.x) return x.y < y.y;
          return x.x < y.x;
        });
    vector<P> res;
    vector<int> stk;
    auto top = [&](int x) { return stk[stk.size() - x]; };
    for (int i = 0; i < m; ++i)
    {
      while (stk.size() >= 2 && cross(p[top(1)] - p[top(2)], p[i] - p[top(1)])
          <= 0) stk.pop_back();
      stk.push_back(i);
    }
    for (auto e : stk) res.push_back(p[e]);
    stk.clear();
    for (int i = m - 1; i >= 0; --i)
    {
      while (stk.size() >= 2 && cross(p[top(1)] - p[top(2)], p[i] - p[top(1)])
          <= 0) stk.pop_back();
      stk.push_back(i);
    }
    for (int i = 1; i + 1 < stk.size(); ++i) res.push_back(p[stk[i]]);
    return res;
}

ld area(vector<P>& p) // 多边形面积
{
    // counterclockwise
    int m = p.size();
    ll res = 0;
    for (int i = 1; i < m - 1; ++i) res += cross(p[i] - p[0], p[(i + 1) % m] - p
        [0]);
    return res / 2;
}
```

8.2 二维浮点数坐标相关

```
using ld = long double;

constexpr ld INF = 1e100;
constexpr ld PI = acosl(-1);
constexpr ld EPS = 1e-9;
```

```cpp
struct P
{
    ld x, y;

    P(): x(0), y(0) {}
    P(ld x, ld y): x(x), y(y) {}

    P operator-(const P& rhs) const { return P(x - rhs.x, y - rhs.y); }
    P operator+(const P& rhs) const { return P(x + rhs.x, y + rhs.y); }
    ld operator*(const P& rhs) const { return x * rhs.x + y * rhs.y; }
    ld len() { return sqrtl(*this * *this); }
    void rotate(ld rad, const P& p = P(0, 0)) // counterclockwise
    {
        P rel(*this - p);
        *this = P(rel.x * cos(rad) - rel.y * sin(rad), rel.x * sin(rad) + rel.y *
            cos(rad)) + p;
        return;
    }
};

ld deg_to_rad(int x) { return x * PI / 180; }
ld sqr(ld x) { return x * x; }
ld dis(const P& p1, const P& p2) { return (p1 - p2).len(); }
ld cross(const P& p1, const P& p2) { return p1.x * p2.y - p2.x * p1.y; }
ld area(const P& p1, const P& p2, const P& p3) { return fabsl(cross(p2 - p1, p3
     - p1)) / 2; }

P intersect(const P& p1, const P& p2, const P& p3, const P& p4) // 直线p1p2和
    p3p4的交点，需确保交点唯一存在
{
    ld s1 = cross(p2 - p1, p3 - p1);
    ld s2 = cross(p2 - p1, p4 - p1);
    return P((p3.x * s2 - p4.x * s1) / (s2 - s1), (p3.y * s2 - p4.y * s1) / (s2
         - s1));
}

ld closest(vector<P>& p) // 最近点对，P1429
{
    sort(p.begin(), p.end(), [](auto x, auto y) { return x.x < y.x; });
    function<ld(int, int)> work = [&](int lef, int rig)
    {
        if (lef == rig - 1) return INF;
        int mid = lef + (rig - lef) / 2;
        ld midx = p[mid].x;
        ld low = min(work(lef, mid), work(mid, rig));
        int lp = lef, rp = mid;
        vector<P> v;
        while (lp < mid || rp < rig)
        {
            if (lp < mid && (rp == rig || p[rp].y > p[lp].y)) v.push_back(p[lp++])
                ;
            else v.push_back(p[rp++]);
        }
        for (int i = lef; i < rig; ++i) p[i] = v[i - lef];
        v.clear();
        for (int i = lef; i < rig; ++i)
        {
            if (fabsl(p[i].x - midx) < low) v.push_back(p[i]);
        }
        for (int i = 1; i < v.size(); ++i)
        {
            for (int j = i - 1; j >= 0; --j)
            {
                if (v[i].y - v[j].y >= low) break;
                low = min(low, dis(v[i], v[j]));
            }
        }
        return low;
    };
    return work(0, p.size());
}

array<ld, 3> circle(const P& p1, const P& p2, const P& p3) // 三点定圆
{
    P a(2 * (p1.x - p2.x), 2 * (p1.x - p3.x));
    P b(2 * (p1.y - p2.y), 2 * (p1.y - p3.y));
    P c(p1 * p1 - p2 * p2, p1 * p1 - p3 * p3);
    P o(cross(c, b) / cross(a, b), cross(c, a) / cross(b, a));
    return { o.x, o.y, dis(o, p1) };
}

array<ld, 3> circle(vector<P>& p) // 最小圆覆盖
{
    shuffle(p.begin(), p.end(), mt19937(time(0)));
    int m = p.size();
    P c;
    ld r = 0;
    for (int i = 0; i < m; ++i)
    {
        if (dis(p[i], c) <= r + EPS) continue;
        c = p[i], r = 0;
        for (int j = 0; j < i; ++j)
        {
            if (dis(p[j], c) <= r + EPS) continue;
            c.x = (p[i].x + p[j].x) / 2;
            c.y = (p[i].y + p[j].y) / 2;
            r = dis(p[i], p[j]) / 2;
            for (int k = 0; k < j; ++k)
            {
                if (dis(p[k], c) < r + EPS) continue;
                auto cir = circle(p[i], p[j], p[k]);
                c.x = cir[0], c.y = cir[1], r = cir[2];
            }
        }
    }
    return { c.x, c.y, r };
}

array<P, 4> rectangle(vector<P>& p) // 最小矩形覆盖
{
    // convex & counterclockwise
    array<P, 4> res{};
    ld ans = INF;
    int m = p.size();
    int top = 1, lef = -1, rig = 1;
    for (int i = 0; i < m; ++i)
    {
        P bot = p[(i + 1) % m] - p[i];
        while (bot * (p[(rig + 1) % m] - p[i]) >= bot * (p[rig] - p[i])) rig = (
            rig + 1) % m;
        while (cross(p[(top + 1) % m] - p[i], bot) <= cross(p[top] - p[i], bot))
            top = (top + 1) % m;
        if (lef == -1) lef = top;
        while (bot * (p[(lef + 1) % m] - p[i]) <= bot * (p[lef] - p[i])) lef = (
            lef + 1) % m;
        ld lb = (p[i] - p[lef]) * bot / bot.len(), rb = (p[rig] - p[i]) * bot /
            bot.len();
        ld base = lb + rb;
        ld high = cross(bot, p[top] - p[i]) / bot.len();
        ld area = base * high;
        if (area < ans)
        {
            ans = area;
            ld o = bot.len();
            bot.x /= o, bot.y /= o;
            res[0] = p[i] + P(-bot.x * lb, -bot.y * lb);
            res[1] = p[i] + P(bot.x * rb, bot.y * rb);
            P h(-bot.y * high, bot.x * high);
            res[2] = res[1] + h, res[3] = res[0] + h;
        }
    }
    return res;
}
```

9 杂项算法

9.1 普通莫队算法

1. 时间复杂度：$O((n+m)\sqrt{n})$

```cpp
const int N = 50005;
const int M = 50005;

ll n, m, k, a[N], BLOCK;
ll ans[M];

struct Q
{
    ll l, r, id;
    bool operator<(const Q& rhs) const
    {
        // 奇偶化排序优化常数
        int lb = l / BLOCK, rb = rhs.l / BLOCK;
        if (lb == rb)
        {
            if (r == rhs.r) return 0;
            else return (r < rhs.r) ^ (lb & 1);
        }
        else return lb < rb;
    }
} q[M];

void solve()
{
    cin >> n >> m >> k;
    BLOCK = sqrt(m); // 块大小
    for (int i = 1; i <= n; ++i) cin >> a[i];

    // 离线处理询问
    for (int i = 1; i <= m; ++i) q[i].id = i, cin >> q[i].l >> q[i].r;
    sort(q + 1, q + 1 + m);

    // 计算首个询问答案
    vector<int> cnt(k + 1);
    for (int i = q[1].l; i <= q[1].r; ++i) cnt[a[i]]++;
    ll res = 0;
    for (int i = 1; i <= k; ++i) res += cnt[i] * cnt[i];
    ans[q[1].id] = res;

    // 开始转移
    ll l = q[1].l, r = q[1].r;
    auto del = [&](int p)
    {
        res -= cnt[a[p]] * cnt[a[p]];
        cnt[a[p]]--;
        res += cnt[a[p]] * cnt[a[p]];
        return;
    };
    auto add = [&](int p)
    {
        res -= cnt[a[p]] * cnt[a[p]];
        cnt[a[p]]++;
        res += cnt[a[p]] * cnt[a[p]];
        return;
    };
    for (int i = 2; i <= m; ++i)
    {
        while (r < q[i].r) add(++r);
        while (r > q[i].r) del(r--);
        while (l < q[i].l) del(l++);
        while (l > q[i].l) add(--l);
        ans[q[i].id] = res;
    }
    for (int i = 1; i <= m; ++i) cout << ans[i] << '\n';
    return;
}
```

9.2 带修改莫队算法

1. 时间复杂度：n, m, t 同级时 $O(n^{\frac{5}{3}})$

```cpp
const int N = 150005;
const int M = 150005;

ll BLOCK;

struct Q
{
    ll l, r, id, t;
    bool operator<(const Q& rhs) const
    {
        // 左右端点都分块
        if (l / BLOCK == rhs.l / BLOCK)
        {
            if (r / BLOCK == rhs.r / BLOCK) return t < rhs.t;
            else return r / BLOCK < rhs.r / BLOCK;
        }
        else return l / BLOCK < rhs.l / BLOCK;
    }
} q[M];

struct C
{
    ll p, o, v;
} c[M];

ll n, m, a[N], ans[N];

void solve()
{
    cin >> n >> m;
    BLOCK = pow(n, 2.0 / 3);
    for (int i = 1; i <= n; ++i) cin >> a[i];
    ll mxx = *max_element(a + 1, a + 1 + n);

    // 离线处理询问
    char op;
    ll t = 0, ord = 0, u, v;
    for (int i = 1; i <= m; ++i)
    {
        cin >> op >> u >> v;
        if (op == 'R') c[++t] = { u, a[u], v }, a[u] = v;
        else ord++, q[ord] = { u, v, ord, t };
    }
    sort(q + 1, q + 1 + ord);

    // 计算首个询问答案
    vector<ll> cnt(mxx + 1);
    ll res = 0, l = q[1].l, r = q[1].r, nowt = t;
    auto del = [&](int p)
    {
        cnt[a[p]]--;
        if (cnt[a[p]] == 0) res--;
        return;
    };
    auto add = [&](int p)
    {
        cnt[a[p]]++;
        if (cnt[a[p]] == 1) res++;
        return;
    };
    auto chg = [&](int p, ll v)
    {
        if (p >= l && p <= r) del(p);
        a[p] = v;
        if (p >= l && p <= r) add(p);
        return;
    };
    while (nowt > q[1].t) a[c[nowt].p] = c[nowt].o, nowt--;
    for (int i = l; i <= r; ++i) add(i);
    ans[q[1].id] = res;

    // 开始转移
    for (int i = 2; i <= ord; ++i)
    {
        for (int j = q[i - 1].t + 1; j <= q[i].t; ++j) chg(c[j].p, c[j].v);
        for (int j = q[i - 1].t; j > q[i].t; --j) chg(c[j].p, c[j].o);
        while (r < q[i].r) add(++r);
        while (r > q[i].r) del(r--);
        while (l < q[i].l) del(l++);
        while (l > q[i].l) add(--l);
        ans[q[i].id] = res;
    }
    for (int i = 1; i <= ord; ++i) cout << ans[i] << '\n';
    return;
}

int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);
    int T = 1;
    // cin >> T;
    while (T--) solve();
    return 0;
}
```

9.3 莫队二次离线

1. 莫队转移超过 $O(1)$ 时，将所有转移离线并利用贡献可拆分性快速预处理
2. 时间复杂度：$O(n\sqrt{n})$

```cpp
const int B = 14;
const int N = 100005;

ll n, m, k;
ll a[N], BLOCK;
```

```cpp
struct Q
{
    ll l, r, id, ans;
    bool operator<(const Q& rhs) const
    {
        int lb = l / BLOCK, rb = rhs.l / BLOCK;
        if (lb == rb)
        {
            if (r == rhs.r) return 0;
            else return (r < rhs.r) ^ (lb & 1);
        }
        else return lb < rb;
    }
} q[N];

void solve()
{
    cin >> n >> m >> k;
    BLOCK = sqrt(n);
    for (int i = 1; i <= n; ++i) cin >> a[i];
    for (int i = 1; i <= m; ++i)
    {
        cin >> q[i].l >> q[i].r;
        q[i].id = i;
        q[i].ans = 0;
    }
    sort(q + 1, q + 1 + m);
    q[0].l = 1, q[0].r = 0, q[0].ans = 0;
    int lef = 1, rig = 0;
    array<vector<vector<int>>, 2> req{ vector<vector<int>>(n + 1), vector<vector
        <int>>(n + 1) };
    for (int i = 1; i <= m; ++i)
    {
        if (rig < q[i].r) req[0][lef].push_back(i), rig = q[i].r;
        if (lef > q[i].l) req[1][rig].push_back(i), lef = q[i].l;
        if (rig > q[i].r) req[0][lef].push_back(i), rig = q[i].r;
        if (lef < q[i].l) req[1][rig].push_back(i), lef = q[i].l;
    }
    vector<ll> tar;
    for (int i = 0; i < (1 << B); ++i)
    {
        if (__builtin_popcount(i) == k) tar.push_back(i);
    }
    vector<ll> cnt(1 << B), pre(n + 2), suf(n + 2);
    for (int i = 1; i <= n; ++i)
    {
        pre[i] = cnt[a[i]];
        for (auto e : req[0][i])
        {
            if (q[e - 1].r < q[e].r)
            {
                for (int j = q[e - 1].r + 1; j <= q[e].r; ++j) q[e].ans -= cnt[a[j
                    ]];
            }
            else
            {
                for (int j = q[e].r + 1; j <= q[e - 1].r; ++j) q[e].ans += cnt[a[j
                    ]];
            }
        }
        for (auto e : tar) cnt[a[i] ^ e]++;
    }
    fill(cnt.begin(), cnt.end(), 0ll);
    for (int i = n; i >= 1; --i)
    {
        suf[i] = cnt[a[i]];
        for (auto e : req[1][i])
        {
            if (q[e - 1].l > q[e].l)
            {
                for (int j = q[e - 1].l - 1; j >= q[e].l; --j) q[e].ans -= cnt[a[j
                    ]];
            }
            else
            {
                for (int j = q[e].l - 1; j >= q[e - 1].l; --j) q[e].ans += cnt[a[j
                    ]];
            }
        }
        for (auto e : tar) cnt[a[i] ^ e]++;
    }
    lef = 1, rig = 0;
    for (int i = 1; i <= m; ++i)
    {
        q[i].ans += q[i - 1].ans;
        while (rig < q[i].r) q[i].ans += pre[++rig];
        while (lef > q[i].l) q[i].ans += suf[--lef];
        while (rig > q[i].r) q[i].ans -= pre[rig--];
        while (lef < q[i].l) q[i].ans -= suf[lef++];
    }
    vector<ll> ans(m + 1);
    for (int i = 1; i <= m; ++i) ans[q[i].id] = q[i].ans;
    for (int i = 1; i <= m; ++i) cout << ans[i] << '\n';
    return;
}
```

9.4 整体二分

1. 在答案值域上将多个需要二分解决的询问划分到两个区间中
2. 注意分到右半区间的询问目标值要削减
3. 时间复杂度：$O(q \log m)$

```cpp
const int N = 300005;

struct Fenwick { /*带时间戳树状数组*/ }fen;
struct Discret { /*离散化*/ }D;

struct Q
```

```
{
    int l, r, k, id;
}q[N];

int n, m;
pair<int, int> a[N];
int ans[N];

void bis(int lef, int rig, int ql, int qr)
{
    if (lef == rig - 1)
    {
        for (int i = ql; i < qr; ++i) ans[q[i].id] = lef;
        return;
    }
    int mid = lef + rig >> 1;
    for (int i = lef; i < mid; ++i) fen.add(a[i].second, 1);
    queue<Q> q1, q2;
    for (int i = ql; i < qr; ++i)
    {
        int cnt = fen.rsum(q[i].l, q[i].r);
        if (cnt < q[i].k) q2.push({ q[i].l,q[i].r,q[i].k - cnt,q[i].id });
        else q1.push(q[i]);
    }
    int qm = ql + q1.size();
    for (int i = ql; i < qr; ++i)
    {
        if (q1.size()) q[i] = q1.front(), q1.pop();
        else q[i] = q2.front(), q2.pop();
    }
    fen.clear();
    bis(lef, mid, ql, qm);
    bis(mid, rig, qm, qr);
    return;
}

void solve()
{
    cin >> n >> m;
    fen.init(n);
    for (int i = 1; i <= n; ++i)
    {
        cin >> a[i].first;
        a[i].second = i;
        D.insert(a[i].first);
    }
    D.work();
    for (int i = 1; i <= n; ++i) a[i].first = D[a[i].first];
    sort(a + 1, a + 1 + n);
    for (int i = 1; i <= m; ++i)
    {
        cin >> q[i].l >> q[i].r >> q[i].k;
        q[i].id = i;
    }
    bis(1, n + 1, 1, m + 1);
    for (int i = 1; i <= m; ++i) cout << D.v[ans[i] - 1] << '\n';
    return;
}
```

9.5 三分

1. 函数必须严格凸/严格凹
2. 时间复杂度：$O(\log n)$

```
// 浮点数三分
ld tes(ld lef, ld rig)
{
    if (fabs(lef - rig) < 1e-7) return lef;
    ld midl = lef + (rig - lef) / 3;
    ld midr = rig - (rig - lef) / 3;
    ld resl = check(midl), resr = check(midr);
    if (resl > resr) return tes(lef, midr);
    else return tes(midl, rig);
}

// 整数三分 [l,r]
ll tes(ll lef, ll rig)
{
    if (lef == rig) return lef;
    ll midl = lef + (rig - lef) / 3;
    ll midr = rig - (rig - lef) / 3;
    ll resl = check(midl), resr = check(midr);
    if (resl >= resr) return tes(lef, midr - 1);
    else return tes(midl + 1, rig);
}
```

9.6 离散化

1. 注意下标从 0 开始还是 1 开始
2. 时间复杂度：$O(\log n)$

```
struct Discret
{
    vector<ll> v;
    void insert(ll val)
    {
        v.push_back(val);
        return;
    }
    void work()
    {
        sort(v.begin(), v.end());
        v.erase(unique(v.begin(), v.end()), v.end());
        return;
    }
    void clear()
    {
        v.clear();
        return;
    }
    ll operator[](ll val)
    {
        return lower_bound(v.begin(), v.end(), val) - v.begin();
    }
};
```

9.7 快速排序

1. 两倍常数，但跳过所有与基准相等的值
2. 时间复杂度：$O(n \log n)$

```
const int N = 100005;

int n;
ll a[N];

int median(int x, int y, int z)
{
    if (a[x] > a[y] && a[z] > a[y]) return a[x] > a[z] ? z : x;
    else if (a[x] < a[y] && a[z] < a[y]) return a[x] < a[z] ? z : x;
    else return y;
}

void QuickSort(int lef, int rig) // [lef, rig]
{
    if (rig <= lef) return;
    int mid = lef + (rig - lef) / 2;
    int pivot = median(lef, mid, rig);
    swap(a[pivot], a[lef]);
    int lp = lef; // 第一个等于基准的值
    for (int i = lef + 1; i <= rig; ++i)
    {
        if (a[i] < a[lef]) swap(a[i], a[++lp]);
    }
    swap(a[lef], a[lp]);
    int rp = lp; // 最后一个等于基准的值
    for (int i = lp + 1; i <= rig; ++i)
    {
        if (a[i] == a[lp]) swap(a[i], a[++rp]);
    }
    QuickSort(lef, lp - 1);
    QuickSort(rp + 1, rig);
    return;
}
```

9.8 枚举集合

1. 时间复杂度：跳转 $O(1)$

```
struct EnumSet
{
    vector<int> subset(int x) // 枚举x的子集
    {
        vector<int> res;
        for (int i = x; i >= 1; i = (i - 1) & x) res.push_back(i);
        res.push_back(0);
        return res;
    }

    vector<int> kset(int b, int k) // 枚举b位大小为k的集合
    {
        vector<int> res;
        int now = (1 << k) - 1;
        while (now < (1 << b))
        {
            res.push_back(now);
            int lowbit = now & -now;
            int x = now + lowbit;
            int y = ((now & ~x) / lowbit) >> 1;
            now = x | y;
        }
        return res;
    }

    vector<int> superset(int x, int b) // 枚举x的b位超集
    {
        vector<int> res;
        for (int i = x; i < (1 << b); i = (i + 1) | x) res.push_back(i);
        return res;
    }
};
```

9.9 CDQ 分治 +CDQ 分治 = 多维偏序

1. n 维偏序需要 n 层 CDQ 分治
2. 第 i 层 CDQ 将第 i 维降为二进制，同时将整个区间按第 $i+1$ 维归并排序，然后调用第 $i+1$ 层 CDQ，第 $n-1$ 层 CDQ 递归将左右分别按第 n 维排序，再用双指针按照第 n 维大小归并，同时计算左部前 $n-2$ 维全 0 元素对右部前 $n-2$ 维全 1 元素的贡献

3. 其余注意事项见 “CDQ 分治 + 数据结构 = 多维偏序”
4. 时间复杂度：$O(nd\log^{d-1} n)$

```cpp
const int N = 100005;

struct Elem
{
    ll a, b, c;
    ll cnt, id;
    bool xtag;
    bool operator!=(const Elem& e) const
    {
        return a != e.a || b != e.b || c != e.c;
    }
}e[N], ee[N], eee[N];

int n, k, ans[N], res[N];

bool bya(const Elem& e1, const Elem& e2)
{
    if (e1.a == e2.a && e1.b == e2.b) return e1.c < e2.c;
    else if (e1.a == e2.a) return e1.b < e2.b;
    else return e1.a < e2.a;
}

void cdq2(int lef, int rig)
{
    if (lef == rig - 1) return;
    int mid = lef + rig >> 1;
    cdq2(lef, mid);
    cdq2(mid, rig);
    int p1 = lef, p2 = mid, now = lef;
    int sum = 0;
    while (now < rig)
    {
        // 左半部分xtag为0的可以贡献右半部分xtag为1的
        if (p2 == rig || p1 < mid && ee[p1].c <= ee[p2].c)
        {
            eee[now] = ee[p1++];
            sum += eee[now].cnt * (eee[now].xtag == 0);
        }
        else
        {
            eee[now] = ee[p2++];
            res[eee[now].id] += sum * (eee[now].xtag == 1);
        }
        now++;
    }
    for (int i = lef; i < rig; ++i) ee[i] = eee[i];
    return;
}

void cdq1(int lef, int rig)
{
    if (lef == rig - 1) return;
    int mid = lef + rig >> 1;
    cdq1(lef, mid);
    cdq1(mid, rig);
    int p1 = lef, p2 = mid, now = lef;
    while (now < rig)
    {
        if (p2 == rig || p1 < mid && e[p1].b <= e[p2].b)
        {
            ee[now] = e[p1++];
            ee[now].xtag = 0;
        }
        else
        {
            ee[now] = e[p2++];
            ee[now].xtag = 1;
        }
        now++;
    }
    for (int i = lef; i < rig; ++i) e[i] = ee[i];
    cdq2(lef, rig);
    return;
}

void solve()
{
    cin >> n >> k;
    vector<Elem> ori(n);
    for (int i = 0; i < n; ++i)
    {
        cin >> ori[i].a >> ori[i].b >> ori[i].c;
        ori[i].cnt = 1;
    }
    sort(ori.begin(), ori.end(), bya);
    int cnt = 0;
    for (auto& x : ori)
    {
        if (cnt == 0 || e[cnt] != x) cnt++, e[cnt] = x, e[cnt].id = cnt;
        else e[cnt].cnt++;
    }
    cdq1(1, cnt + 1);
    for (int i = 1; i <= cnt; ++i)
    {
        res[e[i].id] += e[i].cnt - 1;
        ans[res[e[i].id]] += e[i].cnt;
    }
    for (int i = 0; i < n; ++i) cout << ans[i] << '\n';
    return;
}
```

9.10 CDQ 分治 + 数据结构 = 多维偏序

1. DP 时贡献有顺序要求，分治的顺序为：解决左半、合并、解决右半
2. 注意小于等于和小于的情况做法细节不同
3. 根据需要进行离散化和去重
4. 时间复杂度：$O(n\log^{d-1} n)$

```cpp
const int N = 100005;

struct Fenwick { /*带时间戳最大值树状数组*/ } fen;
struct Discret { /*离散化*/ } D;

struct Elem
{
    ll a, b, c;
    ll w, dp;
    bool operator!=(const Elem& e) const { return a != e.a || b != e.b || c != e
        .c; }
} e[N];

int n;

bool bya(const Elem& e1, const Elem& e2)
{
    if (e1.a == e2.a && e1.b == e2.b) return e1.c < e2.c;
    else if (e1.a == e2.a) return e1.b < e2.b;
    else return e1.a < e2.a;
}

bool byb(const Elem& e1, const Elem& e2)
{
    if (e1.b == e2.b) return e1.c < e2.c;
    else return e1.b < e2.b;
}

void cdq(int lef, int rig)
{
    if (e[lef].a == e[rig - 1].a) return;
    int mid = lef + (rig - lef) / 2;

    // 需要保证e[mid-1].a和e[mid].a不同
    if (e[lef].a == e[mid].a)
    {
        while (e[lef].a == e[mid].a) mid++;
    }
    else
    {
        while (e[mid - 1].a == e[mid].a) mid--;
    }

    // 解决左半
    cdq(lef, mid);

    // 解决合并
    sort(e + lef, e + mid, byb);
    sort(e + mid, e + rig, byb);
    int p1 = lef, p2 = mid;
    while (p2 < rig)
    {
        while (p1 < mid && e[p1].b < e[p2].b)
        {
            fen.add(D[e[p1].c], e[p1].dp);
            p1++;
        }
        e[p2].dp = max(e[p2].dp, e[p2].w + fen.pres(D[e[p2].c] - 1));
        p2++;
    }
    fen.clear();

    // 解决右半
    sort(e + mid, e + rig, bya); // 复原排序
    cdq(mid, rig);
    return;
}

void solve()
{
    cin >> n;
    vector<Elem> ori(n);
    for (int i = 0; i < n; ++i)
    {
        cin >> ori[i].a >> ori[i].b >> ori[i].c >> ori[i].w;
        ori[i].dp = ori[i].w;
        D.insert(ori[i].c);
    }
    D.work();
    fen.init(D.v.size());
    sort(ori.begin(), ori.end(), bya);
    int cnt = 0;
    for (auto& x : ori)
    {
        if (cnt == 0 || e[cnt] != x) e[++cnt] = x;
        else e[cnt].dp = e[cnt].w = max(e[cnt].w, x.w);
    }
    cdq(1, cnt + 1);
    ll ans = 0;
    for (int i = 1; i <= cnt; ++i) ans = max(ans, e[i].dp);
    cout << ans << '\n';
    return;
}
```

10 博弈论

10.1 Fibonacci 博弈

1. 有一堆石子，两人轮流取。先手第一次不能直接取完。每次至少取一个，但最多取上一个人的两倍。取走最后一个石子的人获胜
2. 结论：是斐波那契数则先手必败，否则先手必胜
3. 时间复杂度：$O(\log n)$

```cpp
bool Fibonacci(ll x) // 返回先手是否必胜
{
    ll a = 1, b = 1;
    while (max(a, b) <= x)
    {
        if (a < b) a += b;
        else b += a;
        if (max(a, b) == x) return 0;
    }
    return 1;
}
```

10.2 Wythoff 博弈

1. 有两堆石子，两人轮流取。每次可以在一堆中取任意个石子或在两堆中取同样多的任意个石子，取走最后一个石子的人获胜
2. 结论：是黄金分割数则先手必败，否则先手必胜
3. x 和 y 极大时需要注意精度问题
4. 时间复杂度：$O(1)$

```cpp
bool Wythoff(ll x, ll y) // 返回先手是否必胜
{
    const double K = ((1.0 + sqrt(5.0)) / 2.0);
    ll res = abs(x - y) * K;
    return res != min(x, y);
}
```

10.3 Green Hackenbush 博弈

1. 版本 1：有一棵有根树，两人轮流选择一个子树删除，删除根结点的人失败
2. 结论 1：叶结点 SG 值为 0，其他结点 SG 值为所有邻接点 SG 值 +1 的异或和
3. 版本 2：有一颗有根树，两人轮流删除一条边以及不与根相连的部分，无边可删的人失败
4. 结论 2：叶结点父边 SG 值为 1，中间结点父边 SG 值为所有邻接边 SG 值异或和 +1
5. 时间复杂度：$O(n)$

```cpp
void dfs(int x, int fa)
{
    sg[x] = 0;
    for (auto e : node[x])
    {
        if (e == fa) continue;
        dfs(e, x);
        sg[x] ^= sg[e] + 1;
    }
    return;
}
```