倍增

定义

倍增法，顾名思义就是翻倍。它能够使线性的处理转化为对数级的处理，大大地优化时间复杂度。

这个方法在很多算法中均有应用，其中最常用的是 RMQ (Ranged Max/Min Query) 问题和求 LCA(Lowest Common Ancestor)。

引例

在你面前的桌子上，摆着无数个重量为任意整数的胡萝卜；
接着告诉你一个数字n，问你要怎么挑选，使得你选出的胡萝卜能够表示出$[1,n]$区间内的所有整数重量?

读完题后我们马上就能想到一种选法，那就是选n个重量为1的胡萝卜，这样就能通过加减表示出$[1,n]$内的所有重量了。

但问题是.....这样挑选的胡萝卜是不是太多了点?

我们很快就能发现，只需要选择重量为$1,2,4,8,16$的胡萝卜，就能表示$[1,31]$内的所有重量

只需要选择重量$2^0,2^1,2^2,\cdots 2^n$的胡萝卜，就能表示$[1,2^{n+1}-1]$内的所有重量。

也就是说， 对于给定的数字n，根本不需要选那么多胡萝卜，只需要$\lfloor log_2 n \rfloor$个胡萝卜就够啦!

由此引例我们得出一个结论：只需要$log_2 n$的预处理，就能表示出$[1,n]$区间内的所有情况。

Vector扩容机制

`std::vector`的`push_back()`操作是**均摊**线性的，其实现原理是每次`size`和`capability`相等并执行`push_back()`操作时，将申请一块**约2倍**大小的新空间，这样总`memcpy`操作的内存大小即是
$O(n)+O(\frac{n}{2})+\cdots+O(1)=O(n)$

[P1226] 【模板】快速幂

题目描述

给你三个整数 a,b,p，求 $a^b \bmod p$。

输入格式

输入只有一行三个整数，分别代表 $a,b,p(0 \le a,b < 2^{31},a+b>0,2 \le p \le 2^{31})$。

输出格式

输出一行一个字符串`a^b mod p=s`，其中 a,b,p 分别为题目给定的值， s 为运算结果。

样例输入

```
2 10 9
```

样例输出

```
2^10 mod 9=7
```

Solution

以样例为例，对于a^b而言，我们先把b写成二进制的形式`b = 1010`

因此，我们可以将a^b分解为$a^2 \times a^8$

对于$a^1, a^2, a^4, a^8, \cdots$我们可以使用$a^i \times a^i = a^{2i}$作为倍增的思路进行递推

至此，我们可以在$O(logb)$的时间内，完成计算a^b的操作

Code

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

[NOIP2013 提高组] 转圈游戏

题目描述

n 个小伙伴（编号从 0 到 $n-1$）围坐一圈玩游戏。按照顺时针方向给 n 个位置编号，从 0 到 $n-1$。最初，第 0 号小伙伴在第 0 号位置，第 1 号小伙伴在第 1 号位置，……，依此类推。游戏规则如下：每一轮第 0 号位置上的小伙伴顺时针走到第 m 号位置，第 1 号位置小伙伴走到第 $m+1$ 号位置，……，依此类推，第 $n-m$ 号位置上的小伙伴走到第 0 号位置，第 $n-m+1$ 号位置上的小伙伴走到第 1 号位置，……，第 $n-1$ 号位置上的小伙伴顺时针走到第 $m-1$ 号位置。

现在，一共进行了 10^k 轮，请问 x 号小伙伴最后走到了第几号位置。

输入格式

共一行，包含四个整数 $n(1 \le n \le 10^6), m(0 < m < n), k(0 < k \le 10^9), x(0 \le x < n)$，每两个整数之间用一个空格隔开。

输出格式

一个整数，表示 10^k 轮后 x 号小伙伴所在的位置编号。

样例输入

```
10 3 4 5
```

样例输出

```
5
```

Solution

小伙伴一共移动了$m \times 10^k$个**单位**位置，因此只需要计算$(x + m \times 10^k)\%n$即可

Code

```
void solve(){
    ll n, m, k, x;
    cin >> n >> m >> k >> x;
    auto qpow = [](ll x, ll a, ll mod){
        ll base = x, rt = 1;
        while(a){
            if(a & 1) rt *= base, rt %= mod;
            base *= base, base %= mod;
            a >>= 1;
        }
        return rt;
    };

    ll ans = qpow(10, k, n);
    ans *= m; ans %= n;
    ans += x; ans %= n;
    cout << ans << endl;
}
```

[P3865] 【模板】ST 表

题目背景

这是一道 ST 表经典题——静态区间最大值

请注意最大数据时限只有 0.8s，数据强度不低，请务必保证你的每次查询复杂度为 $O(1)$。若使用更高时间复杂度算法不保证能通过。

如果您认为您的代码时间复杂度正确但是 TLE，可以尝试使用快速读入：

```
inline char get(void) {
    static char buf[1 << 19], *p1 = buf, *p2 = buf;
    if (p1 == p2) {
        p2 = (p1 = buf) + fread(buf, 1, 1 << 19, stdin);
        if (p1 == p2) return EOF;
    }
```

```
    return *p1++;
}

template<typename T>
inline void read(T &x) {
    x = 0; static char c; bool minus = false;
    for (; !(c >= '0' && c <= '9'); c = get()) if (c == '-') minus = true;
    for (; c >= '0' && c <= '9'; x = x * 10 + c - '0', c = get()); if (minus) x =
-x;
}
```

`read` 函数传参即为需要读入的数据存放单元

快速读入作用仅为加快读入，并非强制使用。

题目描述

给定一个长度为 N 的数列，和 M 次询问，求出每一次询问的区间内数字的最大值。

输入格式

第一行包含两个整数 $N, M(1 \leq N \leq 10^5, 1 \leq M \leq 2 \times 10^6)$，分别表示数列的长度和询问的个数。

第二行包含 N 个整数（记为 a_i），依次表示数列的第 i 项。

接下来 M 行，每行包含两个整数 $l_i, r_i(1 \leq l_i \leq r_i \leq N)$，表示查询的区间为 $[l_i, r_i]$。

输出格式

输出包含 M 行，每行一个整数，依次表示每一次询问的结果。

样例输入

```
8 8
9 3 1 7 5 6 0 8
1 6
1 5
2 7
2 6
1 8
4 8
3 7
1 8
```

样例输出

```
9
9
7
7
9
8
7
9
```

提示

对于 30% 的数据，满足 $1 \leq N, M \leq 10$。

对于 70% 的数据，满足 $1 \leq N, M \leq 10^5$。

对于 100% 的数据，满足 $1 \leq N \leq 10^5$，$1 \leq M \leq 2 \times 10^6$，$a_i \in [0, 10^9]$，$1 \leq l_i \leq r_i \leq N$。

Solution

考虑暴力做法。每次都对区间$[l_i, r_i]$扫描一遍，求出最大值。

显然，这个算法会超时。

ST 表基于倍增思想，可以做到$O(nlogn)$预处理，$O(1)$回答每个询问。但是不支持修改操作。

基于倍增思想，我们考虑如何求出区间最大值。可以发现，如果按照一般的倍增流程，每次跳2^i步的话，询问时的复杂度仍旧是$O(logn)$。

我们发现$max(x, x)$，也就是说，区间最大值是一个具有**可重复贡献**性质的问题。即使用来求解的预处理区间有重叠部分，只要这些区间的并是所求的区间，最终计算出的答案就是正确的。

如果手动模拟一下，可以发现我们能使用至多两个预处理过的区间来覆盖询问区间，也就是说询问时的时间复杂度可以被降至$O(1)$，在处理有大量询问的题目时十分有效。

具体实现如下：

令$f(i, j)$表示区间$[i, i + 2^j - 1]$的最大值。

显然$f(i, 0) = a_i$。

根据定义式，第二维就相当于倍增的时候**跳了**$2^j - 1$**步**，依据倍增的思路，写出状态转移方程：$f(i, j) = max(f(i, j - 1), f(i + 2^{j-1}, j - 1))$。

$$\max\left(\boxed{\max(\{a_i, \ldots, a_{i+2^{j-1}-1}\})}\ \boxed{\max(\{a_{i+2^{j-1}}, \ldots, a_{i+2^j-1}\})}\right)$$

$$\Vert$$

$$\boxed{\max(\{a_i, \ldots, a_{i+2^j-1}\})}$$

以上就是预处理部分。而对于查询，可以简单实现如下：

对于每个询问$[l, r]$，我们把它分成两部分：$[l, l + 2^s - 1]$与$[r - 2^s + 1, r]$，其中$s = \lfloor log_2(r - l + 1) \rfloor$。两部分的结果的最大值就是答案。

$$\overbrace{\max = \max(\max_l, \max_r) = 14}$$

$$\overbrace{\max_r = f(4, 2) = 13}$$

$$\overbrace{\max_l = f(2, 2) = 14}$$

0	13	14	4	13	1	5	7

根据上面对于**可重复贡献问题**的论证，由于最大值是**可重复贡献问题**，重叠并不会对区间最大值产生影响。又因为这两个区间完全覆盖了 $[l, r]$，可以保证答案的正确性。

Code

```
template<typename T>
class st{
public:
    int n;
    T f[__lg(maxn) + 1][maxn];
    // 0 - index
    void init(const vector<T>& a){
        n = a.size();
        int t = __lg(n) + 1;
        for(int i = 0; i < n; i++) f[0][i] = a[i];
        for(int i = 1; i < t; i++) for(int j = 0; j < n; j++){
            if(j + (1 << i) > n) break;
            // 此处函数换为自己的
            f[i][j] = max(f[i - 1][j], f[i - 1][j + (1 << (i - 1))]);
        }
    }

    T query(int l, int r){
        int pp = __lg(r - l + 1);
        // 此处函数换为自己的
        return max(f[pp][l], f[pp][r - (1 << pp) + 1]);
    }
};
```

[P1816] 忠诚

题目描述

老管家是一个聪明能干的人。他为财主工作了整整 10 年。财主为了让自己账目更加清楚，要求管家每天记 k 次账。由于管家聪明能干，因而管家总是让财主十分满意。但是由于一些人的挑拨，财主还是对管家产生了怀疑。于是他决定用一种特别的方法来判断管家的忠诚，他把每次的账目按 $1, 2, 3 \ldots$ 编号，然后不定时的问管家问题，问题是这样的：在 a 到 b 号账中最少的一笔是多少？为了让管家没时间作假，他总是一次问多个问题。

输入格式

输入中第一行有两个数 $m, n (1 \leq n, m \leq 10^5)$，表示有 m 笔账 n 表示有 n 个问题。

第二行为 m 个数，分别是账目的钱数。

后面 n 行分别是 n 个问题，每行有2 个数字说明开始结束的账目编号。

输出格式

在一行中输出每个问题的答案，以一个空格分割。

样例输入

```
10 3
1 2 3 4 5 6 7 8 9 10
2 7
3 9
1 10
```

样例输出

```
2 3 1
```

Solution

显然，这是一个**不带修**的RMQ问题，由于是讨论**最小值**，$min(x,x)=x$，因此可以使用st表

Code

```
template<typename T>
class st{
public:
    int n;
    T f[__lg(maxn) + 1][maxn];
    // 0 - index
    void init(const vector<T>& a){
        n = a.size();
        int t = __lg(n) + 1;
        for(int i = 0; i < n; i++) f[0][i] = a[i];
        for(int i = 1; i < t; i++) for(int j = 0; j < n; j++){
            if(j + (1 << i) > n) break;
            // 此处函数换为自己的
            f[i][j] = min(f[i - 1][j], f[i - 1][j + (1 << (i - 1))]);
        }
    }

    T query(int l, int r){
        if (r < l) return 0;
        int pp = __lg(r - l + 1);
        // 此处函数换为自己的
        return min(f[pp][l], f[pp][r - (1 << pp) + 1]);
    }
};

st<int> s;

void solve(){
    int n, m; cin >> n >> m;
    vector<int> a(n);
    for (auto& v: a) cin >> v;
    s.init(a);

    while (m--) {
        int l, r; cin >> l >> r;
```

```
        l--, r--;
        cout << s.query(l, r) << ' ';
    }
}
```

[USACO07JAN] Balanced Lineup G

题目描述

每天,农夫 John 的 $n(1 \leq n \leq 5 \times 10^4)$ 头牛总是按同一序列排队。

有一天, John 决定让一些牛们玩一场飞盘比赛。他准备找一群在队列中位置连续的牛来进行比赛。但是为了避免水平悬殊，牛的身高不应该相差太大。John 准备了 $q(1 \leq q \leq 1.8 \times 10^5)$ 个可能的牛的选择和所有牛的身高 $h_i(1 \leq h_i \leq 10^6, 1 \leq i \leq n)$。他想知道每一组里面最高和最低的牛的身高差。

输入格式

第一行两个数 n, q。

接下来 n 行，每行一个数 h_i。

再接下来 q 行，每行两个整数 a 和 b，表示询问第 a 头牛到第 b 头牛里的最高和最低的牛的身高差。

输出格式

输出共 q 行，对于每一组询问，输出每一组中最高和最低的牛的身高差。

样例输入

```
6 3
1
7
3
4
2
5
1 5
4 6
2 2
```

样例输出

```
6
3
0
```

Solution

只需要开两个st表，分别维护最大 / 最小值即可

Code

```
template<typename T>
class st_max{
public:
    int n;
    T f[__lg(maxn) + 1][maxn];
    // 0 - index
    void init(const vector<T>& a){
        n = a.size();
        int t = __lg(n) + 1;
        for(int i = 0; i < n; i++) f[0][i] = a[i];
        for(int i = 1; i < t; i++) for(int j = 0; j < n; j++){
            if(j + (1 << i) > n) break;
            // 此处函数换为自己的
            f[i][j] = max(f[i - 1][j], f[i - 1][j + (1 << (i - 1))]);
        }
    }

    T query(int l, int r){
        int pp = __lg(r - l + 1);
        // 此处函数换为自己的
        return max(f[pp][l], f[pp][r - (1 << pp) + 1]);
    }
};

template<typename T>
class st_min{
public:
    int n;
    T f[__lg(maxn) + 1][maxn];
    // 0 - index
    void init(const vector<T>& a){
        n = a.size();
        int t = __lg(n) + 1;
        for(int i = 0; i < n; i++) f[0][i] = a[i];
        for(int i = 1; i < t; i++) for(int j = 0; j < n; j++){
            if(j + (1 << i) > n) break;
            // 此处函数换为自己的
            f[i][j] = min(f[i - 1][j], f[i - 1][j + (1 << (i - 1))]);
        }
    }

    T query(int l, int r){
        int pp = __lg(r - l + 1);
        // 此处函数换为自己的
        return min(f[pp][l], f[pp][r - (1 << pp) + 1]);
    }
};

st_max<int> smax;
st_min<int> smin;

void solve(){
    int n, q;
```

```
    cin >> n >> q;
    vector<int> a(n);
    for (auto& v: a) cin >> v;

    smax.init(a); smin.init(a);
    while(q--) {
        int l, r; cin >> l >> r;
        l--; r--;
        cout << smax.query(l, r) - smin.query(l, r) << '\n';
    }
}
```

[P1890] gcd区间

题目描述

给定 n 个正整数 $a_1, a_2, \ldots, a_n$。

m 次询问，每次询问给定一个区间 $[l, r]$，输出 $a_l, a_{l+1}, \ldots, a_r$ 的最大公因数。

输入格式

第一行两个整数 $n, m (1 \leq n \leq 10^3, 1 \leq m \leq 10^6)$。
第二行 n 个整数表示 $a_1, a_2, \cdots, a_n (1 \leq a_i \leq 10^9)$。
以下 m 行，每行两个整数 $l, r (1 \leq l \leq r \leq n)$ 表示询问区间的左右端点。

输出格式

共 m 行，每行表示一个询问的答案。

样例输入

```
5 3
4 12 3 6 7
1 3
2 3
5 5
```

样例输出

```
1
3
7
```

Solution

$gcd(a, a) = a$，因此此题仍然可以用st表解决

注意此题构造st表的时间复杂度是$O(n(logn + logw))$，查询的复杂度是$O(logw)$，其中w是a_i的值域

Code

```
template<typename T>
class st{
public:
    int n;
    T f[__lg(maxn) + 1][maxn];
    // 0 - index
    void init(const vector<T>& a){
        n = a.size();
        int t = __lg(n) + 1;
        for(int i = 0; i < n; i++) f[0][i] = a[i];
        for(int i = 1; i < t; i++) for(int j = 0; j < n; j++){
            if(j + (1 << i) > n) break;
            // 此处函数换为自己的
            f[i][j] = gcd(f[i - 1][j], f[i - 1][j + (1 << (i - 1))]);
        }
    }

    T query(int l, int r){
        if (r < l) return 0;
        int pp = __lg(r - l + 1);
        // 此处函数换为自己的
        return gcd(f[pp][l], f[pp][r - (1 << pp) + 1]);
    }
};

st<int> s;

void solve(){
    int n, m; cin >> n >> m;
    vector<int> a(n);
    for (auto& v: a) cin >> v;
    s.init(a);

    while (m--) {
        int l, r; cin >> l >> r;
        l--, r--;
        cout << s.query(l, r) << '\n';
    }
}
```

[SCOI2007] 降雨量

题目描述

我们常常会说这样的话：“X 年是自 Y 年以来降雨量最多的”。它的含义是 X 年的降雨量不超过 Y 年，且对于任意 $Y < Z < X$，Z 年的降雨量严格小于 X 年。例如 2002、2003、2004 和 2005 年的降雨量分别为 4920、5901、2832 和 3890，则可以说“2005 年是自 2003 年以来最多的”，但不能说“2005 年是自 2002 年以来最多的”由于有些年份的降雨量未知，有的说法是可能正确也可以不正确的。

输入格式

输入第一行包含一个正整数 $n(1 \le n \le 5 \times 10^4)$，为已知的数据。以下 n 行每行两个整数 $y_i(-10^9 \le y_i \le 10^9)$ 和 $r_i(1 \le r_i \le 10^9)$，为年份和降雨量，按照年份从小到大排列，即 $y_i < y_{i+1}$。下一行包含一个正整数 $m(1 \le m \le 10^4)$，为询问的次数。以下 m 行每行包含两个数 Y 和 X，即询问“X 年是自 Y 年以来降雨量最多的$-10^9 \le X < Y \le 10^9$。”这句话是“必真”、“必假”还是“有可能”。

输出格式

对于每一个询问，输出 `true`、`false` 或者 `maybe`。

样例输入

```
6
2002 4920
2003 5901
2004 2832
2005 3890
2007 5609
2008 3024
5
2002 2005
2003 2005
2002 2007
2003 2007
2005 2008
```

样例输出

```
false
true
false
maybe
false
```

Solution

对于每次查询X, Y我们分为以下几种情况讨论

如果Y不确定：

1. 如果X也不确定，那么一定是**可能有解**的
2. 在X确定的前提下，如果区间(X, Y)中有**大于等于**X的点，则**无解**，否则**可能有解**

如果X不确定：

1. 在Y确定的前提下，如果区间(X, Y)中有**大于等于**Y的点，则**无解**，否则**可能有解**

如果X, Y都确定

1. 如果$X > Y$，或者区间(X, Y)中有**大于等于**Y的点，则**无解**
2. 如果$X \le Y$，且区间(X, Y)中没有**大于等于**Y的点，并且区间中所有时间都有记录，则**有解**
3. 否则**可能有解**

Code

```
template<typename T>
class st{
public:
    int n;
    T f[__lg(maxn) + 1][maxn];
    // 0 - index
    void init(const vector<T>& a){
        n = a.size();
        int t = __lg(n) + 1;
        for(int i = 0; i < n; i++) f[0][i] = a[i];
        for(int i = 1; i < t; i++) for(int j = 0; j < n; j++){
            if(j + (1 << i) > n) break;
            // 此处函数换为自己的
            f[i][j] = max(f[i - 1][j], f[i - 1][j + (1 << (i - 1))]);
        }
    }

    T query(int l, int r){
        if (r < l) return 0;
        int pp = __lg(r - l + 1);
        // 此处函数换为自己的
        return max(f[pp][l], f[pp][r - (1 << pp) + 1]);
    }
};

st<int> st1;

void solve(){
    int n; cin >> n;
    vector<int> a, pool;

    for (int i = 0; i < n; i++) {
        int u, v; cin >> u >> v;
        pool.push_back(u); a.push_back(v);
    }
    a.push_back(0); pool.push_back(inf);

    st1.init(a);

    int m; cin >> m;
    while (m--) {
        int u, v; cin >> u >> v;
        int pu = lower_bound(all(pool), u) - pool.begin();
        int pv = lower_bound(all(pool), v) - pool.begin();

        if (pool[pv] != v) {
            if (pool[pu] != u) {cout << "maybe\n";}
            else if (st1.query(pu + 1, pv - 1) >= a[pu]) {cout << "false\n";}
            else {cout << "maybe\n";}
        }
        else if (pool[pu] != u) {
            if (st1.query(pu, pv - 1) >= a[pv]) {cout << "false\n";}
            else {cout << "maybe\n";}
```

```
        }
        else if (a[pu] < a[pv] || st1.query(pu + 1, pv - 1) >= a[pv]) {cout <<
"false\n";}
        else if (pv - pu == v - u && a[pu] > a[pv] && st1.query(pu + 1, pv - 1) <
a[pv]) {cout << "true\n";}
        else {cout << "maybe\n";}
    }
}
```

2771 天才ACM

题目描述

给定一个整数 M，对于任意一个整数集合 S，定义“校验值”如下:

从集合 S 中取出 M 对数(即 $2M$ 个数，不能重复使用集合中的数，如果 S 中的整数不够 M 对，则取到不能取为止)，使得**每对数的差的平方**之和最大，这个最大值就称为集合 S 的**校验值**。

现在给定一个长度为 N 的数列 A 以及一个整数 T。

我们要把 A 分成若干段，使得每一段的**校验值**都不超过 T。

求最少需要分成几段。

输入格式

第一行输入整数 $K(1 \leq K \leq 12)$，代表有 K 组测试数据。

对于每组测试数据，第一行包含三个整数 $N, M, T(1 \leq N, M \leq 5 \times 10^5, 0 \leq T \leq 10^{18})$ 。

第二行包含 N 个整数，表示数列$A_1, A_2, \cdots A_N(0 \leq A_i \leq 2^{20})$。

输出格式

对于每组测试数据，输出其答案，每个答案占一行。

样例输入

```
2
5 1 49
8 2 1 7 9
5 1 64
8 2 1 7 9
```

样例输出

```
2
1
```

Solution

首先，我们考虑如何得到一个集合的校验值

我们假设集合内只有四个元素$0 \le a \le b \le c \le d$，很明显$(b-a)^2+(d-c)^2 \le (c-a)^2+(d-b)^2$，因此，我们只需要讨论$(c-a)^2+(d-b)^2$和$(d-a)^2+(c-b)^2$之间的大小关系

设左式小于等于右式，展开完全平方式并消去平方项后的式子如下所示：

$$\begin{aligned} -2ac-2db &\le -2ad-2cb \\ 2ad-2ac &\le 2bd-2bc \\ a(d-c) &\le b(d-c) \\ a &\le b \end{aligned}$$

因此假设的不等式成立

该结论也可以推广到集合元素更多的场景

接下来，我们考虑如何解决分段问题

首先考虑暴力做法：

每次从最左边的节点L开始，不断尝试区间$[L,L],[L,L+1],\cdots[L,n]$是否满足要求，并且以此贪心地选择区间

由于每个区间需要$O(n\log n)$排序后再$O(n)$计算校验值，同时**至多**有n个这样的区间，因此，时间复杂度是$O(n^2\log n)$

一个显而易见的优化是，我们在寻找右端点的时候，可以利用**二分**，但如果考虑极端情况（即最后分段数量等于n）的话，我们仍然要对$O(n)$个起点进行二分，而单次二分的时间复杂度为$O(\frac{n}{2}\log(\frac{n}{2})+\frac{n}{4}\log(\frac{n}{4})+\cdots)=O(n\log n)$，因此，时间复杂度仍为$O(n^2\log n)$

对于朴素的倍增，即每次**新增**区间大小分别为$2^{\log(n-L+1)},\cdots 2^1,2^0$，这样在上述极端情况下，仍然和二分没有差距

考虑对倍增进行优化，每次**新增区间**大小从1开始，如果新增这个区间后仍然能够满足要求，则**下一次新增区间大小翻倍**，如果区间大小过大（超过数组大小）或者区间不满足要求，则将**下一次新增区间大小减半**，如果某时刻新增区间大小为0，将区间终点的下一个位置作为下一次区间起点，继续计算

单次优化后的倍增时间复杂度为$O(1\log 1)+2\log 2+\cdots\lceil len\rceil\log(\lceil len\rceil))$，其中$len$为该区间最后的大小，若$len$为$O(n)$级别，则至多会有$O(\log n)$个区间，此时时间复杂度为$O(n\log^2 n)$

此时的时间复杂度瓶颈主要在排序，对于倍增长度p和区间$[L,R]$而言，我们的新区间$[L,R+p]$中，$[L,R]$的元素实际上是有序的，我们只需要对区间$[R+1,R+p]$进行排序即可，这样，对于$[L,R],[R+1,R+p]$两段**有序**区间我们就可以$O(n)$地进行归并操作

这样，我们进行**快速排序**的总长度是$O(len)$级别的，其中len为该区间最后的大小，因此，时间复杂度将为了$O(n\log n)$，可以通过本题

Code

```
void solve() {
    int n, m; ll t;
    cin >> n >> m >> t;
    vector<int> a(n), b(n), c(n);
    for (auto& v: a) cin >> v;

    auto merge = [&](int l, int mid, int r){
```

```
        for (int i = mid; i < r; i++) c[i] = a[i];
        sort(c.begin() + mid, c.begin() + r);
        int i = l, j = mid, k = l;
        while (i < mid && j < r) {
            if (c[i] < c[j]) b[k++] = c[i++];
            else b[k++] = c[j++];
        }
        while (i < mid) b[k++] = c[i++];
        while (j < r) b[k++] = c[j++];
    };

    auto check = [&](int l, int mid, int r){
        merge(l, mid, r);
        ll res = 0;
        for (int i = 0; i < m; i++) {
            if (l + i >= r - i - 1) break;
            res += 1ll * (b[r - i - 1] - b[l + i]) * (b[r - i - 1] - b[l + i]);
        }
        return res <= t;
    };

    int l = 0, ans = 0;
    while (l < n) {
        int mid = l, p = 1;
        c[l] = a[l];

        while (p) {
            if (mid + 1 + p <= n && check(l, mid + 1, mid + 1 + p)) {
                mid += p; p *= 2;
                for (int i = l; i <= mid; i++) c[i] = b[i];
            }
            else p /= 2;
        }
        l = mid + 1; ans++;
    }

    cout << ans << endl;
}
```

[# 49522] CSES1867 Company Queries

题目描述

一家公司有n名员工，他们形成一个树形层次结构，其中每个员工都有一个直接上司，除了总经理以外。

你的任务是处理以下形式的q次查询：谁是员工 x 的层次结构中第 k 级的上司?

输入格式

第一行有两个整数 n 和 $q(1 \leq n, q \leq 2 \times 10^5)$：员工和查询的数量。 员工编号为$1, 2, \cdots, n$，员工1为总经理。

下一行有$n-1$个整数$e_2, e_3, \cdots, e_n(1 \leq e_i \leq i-1)$：对应每个员工$2, 3, \cdots n$他们的直接上司。

最后，有q行描述查询。 每行有两个整数x和$k(1 \leq x, k \leq n)$：员工x的第k级上司是谁?

输出格式

每个查询在一行上输出一个整数，表示答案。如果不存在第k级上司，则输出-1。

样例输入

```
5 3
1 1 3 3
4 1
4 2
4 3
```

样例输出

```
3
1
-1
```

Solution

首先考虑暴力做法：

每次往上跳**一级**，连续k次，这样做法的时间复杂度是$O(nq)$的

如果使用类似**快速幂**的做法，我们就可以在$O(n\log n)$的时间复杂度内得到某个节点的k级祖先

我们假设$f[j][i]$为节点i的2^j级祖先，可以由以下式子得到：

$$f[i][j] = f[i-1][f[i-1][j]]$$

其现实意义是，i的2^j级祖先是i的2^{j-1}级祖先的2^{j-1}级祖先

Code

```
namespace LCA{
vector<int> e[maxn];
int f[__lg(maxn) + 1][maxn], dep[maxn], n, lim;

void dfs(int u, int fa) {
    f[0][u] = fa; dep[u] = dep[fa] + 1;
    for (auto v: e[u])
        if (v != fa) dfs(v, u);
}

// init前需要输入树边
void init(int sz, int rt) {
    n = sz; lim = __lg(n);
    dep[rt] = 1; dfs(rt, 0);
    for (int i = 1; i <= lim; i++)
        for (int u = 1; u <= n; u++) f[i][u] = f[i - 1][f[i - 1][u]];
}

void clear() {
    for (int i = 1; i <= n; i++) e[i].clear();
    for (int i = 1; i <= lim; i++) for (int u = 1; u <= n; u++) f[i][u] = 0;
```

```
}
}

void solve() {
    int n, q; cin >> n >> q;
    for (int i = 2; i <= n; i++) {
        int u; cin >> u;
        LCA::e[u].push_back(i);
    }

    LCA::init(n, 1);

    while (q--){
        int u, k; cin >> u >> k;
        if (LCA::dep[u] <= k) {cout << "-1\n"; continue;}
        for (int i = __lg(k); i >= 0; i--) u = (k >> i) & 1 ? LCA::f[i][u] : u;
        cout << u << '\n';
    }
}
```

最近公共祖先

最近公共祖先简称 LCA（Lowest Common Ancestor）。两个节点的最近公共祖先，就是这两个点的公共祖先里面，离根**最远**的那个。 为了方便，我们记某点集$s = \{v_1, v_2, \cdots, v_n\}$的最近公共祖先为$\mathrm{LCA}(v_1, v_2, \cdots, v_n)$或$\mathrm{LCA}(s)$。

性质

1. $\mathrm{LCA}(u) = u$
2. u是v的祖先，当且仅当$\mathrm{LCA}(u, v) = u$
3. 如果u不为v的祖先并且v不为u的祖先，那么u, v分别处于$\mathrm{LCA}(u, v)$的两棵不同子树中
4. 前序遍历中，$\mathrm{LCA}(s)$出现在所有s中元素之前，后序遍历中$\mathrm{LCA}(s)$则出现在所有s中元素之后；
5. 两点集并的最近公共祖先为两点集分别的最近公共祖先的最近公共祖先，即 $\mathrm{LCA}(A \cup B) = \mathrm{LCA}(\mathrm{LCA}(A), \mathrm{LCA}(B))$;
6. 两点的最近公共祖先必定处在树上两点间的最短路上；
7. $d(u, v) = h(u) + h(v) - 2h(\mathrm{LCA}(u, v))$，其中$d(u, v)$是树上两点间的距离，$h(u)$代表某点到树根的距离

[P3379] 【模板】最近公共祖先（LCA）

题目描述

如题，给定一棵有根多叉树，请求出指定两个点直接最近的公共祖先。

输入格式

第一行包含三个正整数 $N, M, S(1 \leq N, M, \leq 5 \times 10^5, 1 \leq S \leq N)$，分别表示树的结点个数、询问的个数和树根结点的序号。

接下来 $N-1$ 行每行包含两个正整数 $x, y(1 \leq x, y \leq N)$，表示 x 结点和 y 结点之间有一条直接连接的边（数据保证可以构成树）。

接下来 M 行每行包含两个正整数 $a, b(1 \leq a, b \leq N)$，不保证$a \neq b$，表示询问 a 结点和 b 结点的最近公共祖先。

输出格式

输出包含 M 行，每行包含一个正整数，依次为每一个询问的结果。

样例输入

```
5 5 4
3 1
2 4
5 1
1 4
2 4
3 2
3 5
1 2
4 5
```

样例输出

```
4
4
1
4
4
```

Solution

首先我们考虑朴素算法：

可以每次找深度比较大的那个点，让它向上跳。显然在树上，这两个点最后一定会相遇，相遇的位置就是想要求的LCA。 或者先向上调整深度较大的点，令他们深度相同，然后再共同向上跳转，最后也一定会相遇。

朴素算法预处理时需要 dfs 整棵树，时间复杂度为$O(n)$，单次查询时间复杂度为$O(n)$。如果树满足随机性质，则时间复杂度与这种随机树的期望高度有关。

接下来，我们来考虑倍增算法：

倍增算法是最经典的 LCA 求法，他是朴素算法的改进算法。通过预处理$f[i][x]$数组，我们可以得到x的2^i级祖先，并且能够在$O(\log n)$的时间内得到x的任意级祖先，$f[0][x]$可以快速地预处理出来，而$f[i][x]$则可以通过$f[i-1][f[i-1][x]]$快速得到

现在我们看看如何优化朴素算法，假设我们要求$\mathrm{LCA}(x, y)$

首先，将两者调整到同一高度，在**深度**已知的情况下，相当于将较深的节点上跳深度之差，这可以用$O(\log n)$实现

如果此时x, y相等，说明其中一个为另一个的**祖先**，直接返回即可

如果此时不等，我们就两者分别上跳$2^k, 2^{k-1}, \cdots, 2^1, 2^0$层，如果跳某一层的时候，两者上跳后的值**相等**了（此处注意不要上跳超过深度），那我们就**不进行**该次跳跃

最后，x, y的值会停留在他们LCA的儿子处，此时对任意一点取父亲节点，我们就得到了$\mathrm{LCA}(x, y)$

预处理的时间复杂度为$O(n\log n)$，单次查询的时间复杂度为$O(\log n)$，此时算法的性能已经足够优秀，能够通过本题

还有一种求LCA的方法，是基于用欧拉序列，将树上问题转化为 RMQ 问题

欧拉序列：对一棵树进行 DFS，无论是**第一次访问**还是**回溯**，每次到达一个结点时都将编号记录下来，可以得到一个长度为$2n-1$的序列，这个序列被称作这棵树的欧拉序列。

在下文中，把结点u在欧拉序列中第一次出现的位置编号记为$idx(u)$（也称作节点u的欧拉序），把欧拉序列本身记作$E[1, \cdots, 2n-1]$。

有了欧拉序列，LCA 问题可以在线性时间内转化为 RMQ 问题，即

$$idx(\mathrm{LCA}(u, v)) = min(idx(k)|k \in E[idx(u), \cdots idx(v)])$$

这个等式不难理解：从u走到v的过程中一定会经过$\mathrm{LCA}(u, v)$，且不会经过$\mathrm{LCA}(u, v)$的祖先。因此，从u走到v的过程中经过的欧拉序最小的结点就是$\mathrm{LCA}(u, v)$。

用 DFS 计算欧拉序列的时间复杂度是$O(n)$，且欧拉序列的长度也是$O(n)$，所以 LCA 问题可以在$O(n)$的时间内转化成等规模的 RMQ 问题。

而对于 RMQ 问题，我们可以选用**st表**进行求解，此时，预处理的时间复杂度为$O(n\log n)$，单次查询的时间复杂度为$O(1)$

注意：使用st表查询时，需要注意$idx(u), idx(v)$之间的大小关系

Code

```
// 倍增LCA
namespace LCA{
vector<int> e[maxn];
int f[__lg(maxn) + 1][maxn], dep[maxn], n, lim;

void dfs(int u, int fa) {
    f[0][u] = fa; dep[u] = dep[fa] + 1;
    for (auto v: e[u])
        if (v != fa) dfs(v, u);
}

// init前需要输入树边
void init(int sz, int rt) {
    n = sz; lim = __lg(n);
    dep[rt] = 1; dfs(rt, 0);
    for (int i = 1; i <= lim; i++) for (int u = 1; u <= n; u++) f[i][u] = f[i -
1][f[i - 1][u]];
}
```

```
int lca(int u, int v) {
    if (dep[u] < dep[v]) swap(u, v);
    for (int i = lim; i >= 0; i--) if (dep[u] - (1 << i) >= dep[v]) u = f[i][u];
    if (u == v) return u;

    for (int i = lim; i >= 0; i--) {
        if (!f[i][u]) continue;
        if (f[i][u] != f[i][v]) u = f[i][u], v = f[i][v];
    }
    return f[0][u];
};

void clear() {
    for (int i = 1; i <= n; i++) e[i].clear();
    for (int i = 1; i <= lim; i++) for (int u = 1; u <= n; u++) f[i][u] = 0;
}
}
```

```
// 欧拉序转RMQ求LCA
namespace LCA{
vector<int> e[maxn], euler;
int idx[maxn], n;

template<typename T>
class st{
public:
    int n;
    T f[__lg(maxn) + 1][maxn];

    T cmp(int u, int v) {
        return idx[u] < idx[v] ? u : v;
    }

    // 0 - index
    void init(const vector<T>& a){
        n = a.size();
        int t = __lg(n) + 1;
        for(int i = 0; i < n; i++) f[0][i] = a[i];
        for(int i = 1; i < t; i++) for(int j = 0; j < n; j++){
            if(j + (1 << i) > n) break;
            // 此处函数换为自己的
            f[i][j] = cmp(f[i - 1][j], f[i - 1][j + (1 << (i - 1))]);
        }
    }

    T query(int l, int r){
        int pp = __lg(r - l + 1);
        // 此处函数换为自己的
        return cmp(f[pp][l], f[pp][r - (1 << pp) + 1]);
    }
};

st<int> s;
```

```
void dfs(int u, int fa) {
    idx[u] = euler.size(); euler.push_back(u);
    for (auto v: e[u]) {
        if (v == fa) continue;
        dfs(v, u);
        euler.push_back(u);
    }
}

// init前需要输入树边
void init(int sz, int rt) {
    n = sz; dfs(rt, 0);
    s.init(euler);
}

int lca(int u, int v) {
    if (idx[u] > idx[v]) swap(u, v);
    return s.query(idx[u], idx[v]);
};

void clear() {
    for (int i = 1; i <= n; i++) e[i].clear();
    euler.clear();
}
}
```

######