

XCPC算法

XCPC算法

一、基础算法

1. 排序

快速排序

归并排序

2. 二分

整数二分

浮点数二分

3. 高精度

高精度加法

高精度减法

高精度乘低精度

高精度除以低精度

4. 双指针算法

5. 位运算

6. 离散化

7. 区间合并

8. 启发式合并

9. 最小表示法

10. 点分治与点分树

11. CDQ分治

二、数据结构

1. 链表

单链表

双链表

2. 栈与队列

栈

3. 单调栈, 单调队列

4. 哈希表

一般哈希

字符串哈希

5. 前缀和与差分

一维前缀和

二维前缀和

一维差分

二维差分

6. 树状数组

7. 线段树

单点修改

区间修改(lazy tag)

可持久化线段树

8. 堆

9. 平衡树

Treap

Splay

fhq treap

- 树套树
- 动态树
- 左偏树

三、图论

- 1.DFS
- 2.BFS
- 3.拓扑排序
- 4.最短路
 - dijkstra
 - 堆优化dijkstra
 - bellman_ford
 - spfa
 - spfa判断负环
 - floyd
 - 差分约束
- 5.最小生成树
 - Prim
 - Kruskal
 - 朱刘算法
- 6.二分图
 - 染色法判定二分图
 - 匈牙利算法
- 7.最近公共祖先
- 8.有向图的强连通分量
- 9.无向图的双连通分量
- 10.欧拉路径和欧拉回路
- 11.网络流
 - 最大流
 - 最小割
 - 费用流
- 12.2-SAT
- 13.prufer编码

四、字符串

- 1.kmp
- 2.Trie
- 3.AC自动机
- 4.后缀数组
- 5.后缀自动机 (SAM)
6. manacher算法
- 7.回文自动机 (PAM)

五、动态规划 (DP)

- 1.数字三角形模型
- 2.最大上升子序列模型
- 3.背包模型
- 4.状态机模型
- 5.状态压缩dp
- 6.区间dp
- 7.树形dp/记忆化搜索
- 8.数位dp

- 9.基环树dp
- 10.插头dp
- 11.四边形不等式优化
- 12.数据结构优化dp
 - 单调队列优化dp

六、数学

- 1.质数
 - 试除法判定质数
 - 试除法分解质因数
 - 线性筛法求素数
- 2.约数
 - 试除法求所有约数
 - 约数个数与约数之和
- 3.欧拉函数
 - 求欧拉函数
 - 筛法求欧拉函数
- 4.快速幂
- 5.扩展欧几里得算法
 - 欧几里德算法
- 6.中国剩余定理
- 7.高斯消元
- 8.组合计数
 - 递推法求组合数
 - 通过预处理逆元的方式求组合数
 - Lucas定理
 - 分解质因数法求组合数
- 9.卡特兰数
- 10.容斥原理
- 11.博弈论
- 12.同余
- 13.矩阵乘法
- 14.概率与数学期望
- 15.积性函数
- 16.BSGS
- 17.FFT
- 18.生成函数
- 19.Burnside引理和Polya定理
- 20.斯特林数
- 21.线性基

七、计算几何

- 1.二维计算几何基础
- 2.凸包
- 3.半平面交
- 4.最小圆覆盖
- 5.三维计算几何基础
- 6.三维凸包
- 7.旋转卡壳
- 8.三角剖分
- 9.扫描线

一、基础算法

1.排序

快速排序

```
1 void quick_sort(int q[], int l, int r)
2 {
3     if (l >= r) return;
4
5     int i = l - 1, j = r + 1, x = q[l + r >> 1];
6     while (i < j)
7     {
8         do i ++ ; while (q[i] < x);
9         do j -- ; while (q[j] > x);
10        if (i < j) swap(q[i], q[j]);
11    }
12    quick_sort(q, l, j), quick_sort(q, j + 1, r);
13 }
```

归并排序

```
1 void merge_sort(int q[], int l, int r)
2 {
3     if (l >= r) return;
4
5     int mid = l + r >> 1;
6     merge_sort(q, l, mid);
7     merge_sort(q, mid + 1, r);
8
9     int k = 0, i = l, j = mid + 1;
10    while (i <= mid && j <= r)
11        if (q[i] <= q[j]) tmp[k ++ ] = q[i ++ ];
12        else tmp[k ++ ] = q[j ++ ];
13
14    while (i <= mid) tmp[k ++ ] = q[i ++ ];
15    while (j <= r) tmp[k ++ ] = q[j ++ ];
16
17    for (i = l, j = 0; i <= r; i ++, j ++ ) q[i] = tmp[j];
18 }
```

2.二分

整数二分

```
1  bool check(int x) { /* ... */ } // 检查x是否满足某种性质
2
3  // 区间[l, r]被划分成[l, mid]和[mid + 1, r]时使用:
4  int bsearch_1(int l, int r)
5  {
6      while (l < r)
7      {
8          int mid = l + r >> 1;
9          if (check(mid)) r = mid;    // check()判断mid是否满足性质
10         else l = mid + 1;
11     }
12     return l;
13 }
14 // 区间[l, r]被划分成[l, mid - 1]和[mid, r]时使用:
15 int bsearch_2(int l, int r)
16 {
17     while (l < r)
18     {
19         int mid = l + r + 1 >> 1;
20         if (check(mid)) l = mid;
21         else r = mid - 1;
22     }
23     return l;
24 }
25
```

浮点数二分

```
1  bool check(double x) { /* ... */ } // 检查x是否满足某种性质
2
3  double bsearch_3(double l, double r)
4  {
5      const double eps = 1e-6;    // eps 表示精度，取决于题目对精度的要求
6      while (r - l > eps)
7      {
8          double mid = (l + r) / 2;
9          if (check(mid)) r = mid;
10         else l = mid;
11     }
12     return l;
13 }
```

3.高精度

高精度加法

```
1 // C = A + B, A >= 0, B >= 0
2 vector<int> add(vector<int> &A, vector<int> &B)
3 {
4     if (A.size() < B.size()) return add(B, A);
5
6     vector<int> C;
7     int t = 0;
8     for (int i = 0; i < A.size(); i++)
9     {
10         t += A[i];
11         if (i < B.size()) t += B[i];
12         C.push_back(t % 10);
13         t /= 10;
14     }
15
16     if (t) C.push_back(t);
17     return C;
18 }
```

高精度减法

```
1 // C = A - B, 满足A >= B, A >= 0, B >= 0
2 vector<int> sub(vector<int> &A, vector<int> &B)
3 {
4     vector<int> C;
5     for (int i = 0, t = 0; i < A.size(); i++)
6     {
7         t = A[i] - t;
8         if (i < B.size()) t -= B[i];
9         C.push_back((t + 10) % 10);
10        if (t < 0) t = 1;
11        else t = 0;
12    }
13
14    while (C.size() > 1 && C.back() == 0) C.pop_back();
15    return C;
16 }
```

高精度乘低精度

```
1 // C = A * b, A >= 0, b >= 0
2 vector<int> mul(vector<int> &A, int b)
3 {
4     vector<int> C;
```

```

5
6     int t = 0;
7     for (int i = 0; i < A.size() || t; i++)
8     {
9         if (i < A.size()) t += A[i] * b;
10        C.push_back(t % 10);
11        t /= 10;
12    }
13
14    while (C.size() > 1 && C.back() == 0) C.pop_back();
15
16    return C;
17 }

```

高精度除以低精度

```

1 // A / b = C ... r, A >= 0, b > 0
2 vector<int> div(vector<int> &A, int b, int &r)
3 {
4     vector<int> C;
5     r = 0;
6     for (int i = A.size() - 1; i >= 0; i--)
7     {
8         r = r * 10 + A[i];
9         C.push_back(r / b);
10        r %= b;
11    }
12    reverse(C.begin(), C.end());
13    while (C.size() > 1 && C.back() == 0) C.pop_back();
14    return C;
15 }

```

4.双指针算法

```

1 for (int i = 0, j = 0; i < n; i++)
2 {
3     while (j < i && check(i, j)) j++;
4
5     // 具体问题的逻辑
6 }
7 常见问题分类：
8     (1) 对于一个序列，用两个指针维护一段区间
9     (2) 对于两个序列，维护某种次序，比如归并排序中合并两个有序序列的操作

```

5.位运算

```
1 求n的第k位数字: n >> k & 1
2 返回n的最后一位1: lowbit(n) = n & -n
```

6.离散化

```
1  vector<int> alls; // 存储所有待离散化的值
2  sort(alls.begin(), alls.end()); // 将所有值排序
3  alls.erase(unique(alls.begin(), alls.end()), alls.end()); // 去掉重复元素
4
5  // 二分求出x对应的离散化的值
6  int find(int x) // 找到第一个大于等于x的位置
7  {
8      int l = 0, r = alls.size() - 1;
9      while (l < r)
10     {
11         int mid = l + r >> 1;
12         if (alls[mid] >= x) r = mid;
13         else l = mid + 1;
14     }
15     return r + 1; // 映射到1, 2, ...n
16 }
```

7.区间合并

```
1  // 将所有存在交集的区间合并
2  void merge(vector<PII> &segs)
3  {
4      vector<PII> res;
5
6      sort(segs.begin(), segs.end());
7
8      int st = -2e9, ed = -2e9;
9      for (auto seg : segs)
10         if (ed < seg.first)
11         {
12             if (st != -2e9) res.push_back({st, ed});
13             st = seg.first, ed = seg.second;
14         }
15         else ed = max(ed, seg.second);
16
17     if (st != -2e9) res.push_back({st, ed});
18
19     segs = res;
20 }
21
```


8.启发式合并

1 | 待补充

9.最小表示法

1 | 待补充

10.点分治与点分树

1 |

11.CDQ分治

1 |

二、数据结构

1.链表

单链表

```
1 // head存储链表头，e[]存储节点的值，ne[]存储节点的next指针，idx表示当前用到了哪个节点
2 int head, e[N], ne[N], idx;
3
4 // 初始化
5 void init()
6 {
7     head = -1;
8     idx = 0;
9 }
10
11 // 在链表头插入一个数a
12 void insert(int a)
13 {
14     e[idx] = a, ne[idx] = head, head = idx ++ ;
15 }
16
17 // 将头结点删除，需要保证头结点存在
18 void remove()
19 {
20     head = ne[head];
```

```
21 } }
```

双链表

```
1 // e[]表示节点的值, l[]表示节点的左指针, r[]表示节点的右指针, idx表示当前用到了哪个节点
2 int e[N], l[N], r[N], idx;
3
4 // 初始化
5 void init()
6 {
7     // 0是左端点, 1是右端点
8     r[0] = 1, l[1] = 0;
9     idx = 2;
10 }
11
12 // 在节点a的右边插入一个数x
13 void insert(int a, int x)
14 {
15     e[idx] = x;
16     l[idx] = a, r[idx] = r[a];
17     l[r[a]] = idx, r[a] = idx ++ ;
18 }
19
20 // 删除节点a
21 void remove(int a)
22 {
23     l[r[a]] = l[a];
24     r[l[a]] = r[a];
25 }
```

2.栈与队列

栈

```
1 // tt表示栈顶
2 int stk[N], tt = 0;
3
4 // 向栈顶插入一个数
5 stk[ ++ tt] = x;
6
7 // 从栈顶弹出一个数
8 tt -- ;
9
10 // 栈顶的值
11 stk[tt];
12
13 // 判断栈是否为空, 如果 tt > 0, 则表示不为空
14 if (tt > 0)
```

```
15 {
16
17 }
```

队列

```
1  1. 普通队列:
2
3  // hh 表示队头, tt表示队尾
4  int q[N], hh = 0, tt = -1;
5
6  // 向队尾插入一个数
7  q[ ++ tt] = x;
8
9  // 从队头弹出一个数
10 hh ++ ;
11
12 // 队头的值
13 q[hh];
14
15 // 判断队列是否为空, 如果 hh <= tt, 则表示不为空
16 if (hh <= tt)
17 {
18
19 }
20 2. 循环队列
21
22 // hh 表示队头, tt表示队尾的后一个位置
23 int q[N], hh = 0, tt = 0;
24
25 // 向队尾插入一个数
26 q[tt ++ ] = x;
27 if (tt == N) tt = 0;
28
29 // 从队头弹出一个数
30 hh ++ ;
31 if (hh == N) hh = 0;
32
33 // 队头的值
34 q[hh];
35
36 // 判断队列是否为空, 如果hh != tt, 则表示不为空
37 if (hh != tt)
38 {
39
40 }
```

3.单调栈,单调队列

```
1  常见模型：找出每个数左边离它最近的比它大/小的数
2  int tt = 0;
3  for (int i = 1; i <= n; i ++ )
4  {
5      while (tt && check(stk[tt], i)) tt -- ;
6      stk[ ++ tt] = i;
7  }
8
9  常见模型：找出滑动窗口中的最大值/最小值
10 int hh = 0, tt = -1;
11 for (int i = 0; i < n; i ++ )
12 {
13     while (hh <= tt && check_out(q[hh])) hh ++ ; // 判断队头是否滑出窗口
14     while (hh <= tt && check(q[tt], i)) tt -- ;
15     q[ ++ tt] = i;
16 }
```

4.哈希表

一般哈希

```
1  (1) 拉链法
2      int h[N], e[N], ne[N], idx;
3
4      // 向哈希表中插入一个数
5      void insert(int x)
6      {
7          int k = (x % N + N) % N;
8          e[idx] = x;
9          ne[idx] = h[k];
10         h[k] = idx ++ ;
11     }
12
13     // 在哈希表中查询某个数是否存在
14     bool find(int x)
15     {
16         int k = (x % N + N) % N;
17         for (int i = h[k]; i != -1; i = ne[i])
18             if (e[i] == x)
19                 return true;
20
21         return false;
22     }
23
24 (2) 开放寻址法
25 int h[N];
```

```

26
27 // 如果x在哈希表中，返回x的下标；如果x不在哈希表中，返回x应该插入的位置
28 int find(int x)
29 {
30     int t = (x % N + N) % N;
31     while (h[t] != null && h[t] != x)
32     {
33         t ++ ;
34         if (t == N) t = 0;
35     }
36     return t;
37 }

```

字符串哈希

```

1 typedef unsigned long long ULL;
2 ULL h[N], p[N]; // h[k]存储字符串前k个字母的哈希值, p[k]存储  $P^k \bmod 2^{64}$ 
3
4 // 初始化
5 p[0] = 1;
6 for (int i = 1; i <= n; i ++ )
7 {
8     h[i] = h[i - 1] * P + str[i];
9     p[i] = p[i - 1] * P;
10 }
11
12 // 计算子串 str[l ~ r] 的哈希值
13 ULL get(int l, int r)
14 {
15     return h[r] - h[l - 1] * p[r - l + 1];
16 }

```

5.前缀和与差分

一维前缀和

```

1 S[i] = a[1] + a[2] + ... a[i]
2 a[l] + ... + a[r] = S[r] - S[l - 1]

```

二维前缀和

```

1 S[i, j] = 第i行j列格子左上部分所有元素的和
2 以(x1, y1)为左上角, (x2, y2)为右下角的子矩阵的和为:
3 S[x2, y2] - S[x1 - 1, y2] - S[x2, y1 - 1] + S[x1 - 1, y1 - 1]

```

一维差分

```
1 | 给区间[l, r]中的每个数加上c: B[l] += c, B[r + 1] -= c
```

二维差分

```
1 | 给以(x1, y1)为左上角, (x2, y2)为右下角的子矩阵中的所有元素加上c:  
2 | S[x1, y1] += c, S[x2 + 1, y1] -= c, S[x1, y2 + 1] -= c, S[x2 + 1, y2 + 1] += c
```

###

6.树状数组

```
1 | int lowbit(int x)  
2 | {  
3 |     return x & (-x);  
4 | }  
5 | void add(int x, int c)  
6 | {  
7 |     for(int i=x; i<=n; i+=lowbit(i)) tr[i] += c;  
8 | }  
9 | int sum(int x)  
10 | {  
11 |     int res=0;  
12 |     for(int i=x; i; i-=lowbit(i)) res += tr[i];  
13 |     return res;  
14 | }
```

7.线段树

单点修改

```
1 | struct Node  
2 | {  
3 |     int l, r;  
4 |     int v; // 区间[l, r]中的最大值  
5 | } tr[N * 4];  
6 |  
7 | void pushup(int u) // 由子节点的信息, 来计算父节点的信息  
8 | {  
9 |     tr[u].v = max(tr[u << 1].v, tr[u << 1 | 1].v);  
10 | }  
11 |
```

```

12 void build(int u, int l, int r)
13 {
14     tr[u] = {l, r};
15     if (l == r) return;
16     int mid = l + r >> 1;
17     build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
18     pushup(u);
19 }
20
21 int query(int u, int l, int r)
22 {
23     if (tr[u].l >= l && tr[u].r <= r) return tr[u].v;    // 树中节点, 已经被完全包含在
    [l, r]中了
24
25     int mid = tr[u].l + tr[u].r >> 1;
26     int v = 0;
27     if (l <= mid) v = query(u << 1, l, r);
28     if (r > mid) v = max(v, query(u << 1 | 1, l, r));
29
30     return v;
31 }
32
33 void modify(int u, int x, int v)
34 {
35     if (tr[u].l == x && tr[u].r == x) tr[u].v = v;
36     else
37     {
38         int mid = tr[u].l + tr[u].r >> 1;
39         if (x <= mid) modify(u << 1, x, v);
40         else modify(u << 1 | 1, x, v);
41         pushup(u);
42     }
43 }

```

区间修改(lazy tag)

```

1  struct Node
2  {
3      int l,r,sum,add;
4  }tr[N*4];
5  void pushup(Node &u,Node &l,Node &r)
6  {
7      u.sum=l.sum+r.sum;
8  }
9  void pushup(int u)
10 {
11     pushup(tr[u],tr[u<<1],tr[u<<1|1]);
12 }
13 void pushdown(int u)

```

```

14 {
15     Node &root=tr[u],&left=tr[u<<1],&right=tr[u<<1|1];
16     if(root.add)
17     {
18         left.add+=root.add,left.sum+=(left.r-left.l+1)*root.add;
19         right.add+=root.add,right.sum+=(right.r-right.l+1)*root.add;
20         root.add=0;
21     }
22 }
23 void build(int u,int l,int r)
24 {
25     if(l==r)
26     {
27         tr[u]={l,r,a[l],0};
28         return;
29     }else
30     {
31         tr[u]={l,r};
32         int mid=(l+r)>>1;
33         build(u<<1,l,mid),build(u<<1|1,mid+1,r);
34         pushup(u);
35     }
36 }
37 void modify(int u,int l,int r,int d)
38 {
39     if(tr[u].l>=l&&tr[u].r<=r)
40     {
41         tr[u].sum+=(tr[u].r-tr[u].l+1)*d;
42         tr[u].add+=d;
43     }else
44     {
45         pushdown(u);
46         int mid=(tr[u].l+tr[u].r)>>1;
47         if(l<=mid)modify(u<<1,l,r,d);
48         if(r>mid)modify(u<<1|1,l,r,d);
49         pushup(u);
50     }
51 }
52 Node query(int u,int l,int r)
53 {
54     if(tr[u].l>=l&&tr[u].r<=r)return tr[u];
55     pushdown(u);
56     int mid=(tr[u].l+tr[u].r)>>1;
57     if(l>mid)return query(u<<1|1,l,r);
58     if(r<=mid)return query(u<<1,l,r);
59     else
60     {
61         auto left=query(u<<1,l,r);
62         auto right=query(u<<1|1,l,r);

```



```

63         Node res;
64         pushup(res, left, right);
65         return res;
66     }
67 }

```

可持久化线段树

```

1  #include <cstdio>
2  #include <cstring>
3  #include <iostream>
4  #include <algorithm>
5  #include <vector>
6
7  using namespace std;
8
9  const int N = 100010, M = 10010;
10
11  int n, m;
12  int a[N];
13  vector<int> nums;
14
15  struct Node
16  {
17      int l, r;
18      int cnt;
19  }tr[N * 4 + N * 17];
20
21  int root[N], idx;
22
23  int find(int x)
24  {
25      return lower_bound(nums.begin(), nums.end(), x) - nums.begin();
26  }
27
28  int build(int l, int r)
29  {
30      int p = ++ idx;
31      if (l == r) return p;
32      int mid = l + r >> 1;
33      tr[p].l = build(l, mid), tr[p].r = build(mid + 1, r);
34      return p;
35  }
36
37  int insert(int p, int l, int r, int x)
38  {
39      int q = ++ idx;
40      tr[q] = tr[p];
41      if (l == r)

```

```

42     {
43         tr[q].cnt ++ ;
44         return q;
45     }
46     int mid = l + r >> 1;
47     if (x <= mid) tr[q].l = insert(tr[p].l, l, mid, x);
48     else tr[q].r = insert(tr[p].r, mid + 1, r, x);
49     tr[q].cnt = tr[tr[q].l].cnt + tr[tr[q].r].cnt;
50     return q;
51 }
52
53 int query(int q, int p, int l, int r, int k)
54 {
55     if (l == r) return r;
56     int cnt = tr[tr[q].l].cnt - tr[tr[p].l].cnt;
57     int mid = l + r >> 1;
58     if (k <= cnt) return query(tr[q].l, tr[p].l, l, mid, k);
59     else return query(tr[q].r, tr[p].r, mid + 1, r, k - cnt);
60 }
61
62 int main()
63 {
64     scanf("%d%d", &n, &m);
65     for (int i = 1; i <= n; i ++ )
66     {
67         scanf("%d", &a[i]);
68         nums.push_back(a[i]);
69     }
70
71     sort(nums.begin(), nums.end());
72     nums.erase(unique(nums.begin(), nums.end()), nums.end());
73
74     root[0] = build(0, nums.size() - 1);
75
76     for (int i = 1; i <= n; i ++ )
77         root[i] = insert(root[i - 1], 0, nums.size() - 1, find(a[i]));
78
79     while (m -- )
80     {
81         int l, r, k;
82         scanf("%d%d%d", &l, &r, &k);
83         printf("%d\n", nums[query(root[r], root[l - 1], 0, nums.size() - 1, k)]);
84     }
85
86     return 0;
87 }

```

8.堆

```
1 priority_queue<int> heap;//大根堆
2 priority_queue<int,vector<int>,greater<int> >//小根堆
```

9.平衡树

Treap

```
1 #include<bits/stdc++.h>
2 using namespace std;
3 const int N=1e5+5,INF=0x3f3f3f3f;
4 int n;
5 struct Node
6 {
7     int l,r;
8     int key,val;
9     int cnt,size;
10 }tr[N];
11 int root,idx;
12 void pushup(int p)
13 {
14     tr[p].size=tr[tr[p].l].size+tr[tr[p].r].size+tr[p].cnt;
15 }
16 int get_node(int key)
17 {
18     tr[++idx].key=key;
19     tr[idx].val=rand();
20     tr[idx].cnt=tr[idx].size=1;
21     return idx;
22 }
23 void zig(int &p)
24 {
25     int q=tr[p].l;
26     tr[p].l=tr[q].r,tr[q].r=p,p=q;
27     pushup(tr[p].r),pushup(p);
28 }
29 void zag(int &p)
30 {
31     int q=tr[p].r;
32     tr[p].r=tr[q].l,tr[q].l=p,p=q;
33     pushup(tr[p].l),pushup(p);
34 }
35 void build()
36 {
37     get_node(-INF),get_node(INF);
38     root=1,tr[1].r=2;
39     pushup(root);
```

```

40 }
41 void insert(int &p,int key)
42 {
43     if(!p)p=get_node(key);
44     else if(tr[p].key==key)tr[p].cnt++;
45     else if(tr[p].key>key)
46     {
47         insert(tr[p].l,key);
48         if(tr[tr[p].l].val>tr[p].val)zig(p);
49     }
50     else
51     {
52         insert(tr[p].r,key);
53         if(tr[tr[p].r].val>tr[p].val)zag(p);
54     }
55     pushup(p);
56 }
57 void remove(int &p,int key)
58 {
59     if(!p)return;
60     if(tr[p].key==key)
61     {
62         if(tr[p].cnt>1)tr[p].cnt--;
63         else if(tr[p].l||tr[p].r)
64         {
65             if(!tr[p].r||tr[tr[p].l].val>tr[tr[p].r].val)
66             {
67                 zig(p);
68                 remove(tr[p].r,key);
69             }else
70             {
71                 zag(p);
72                 remove(tr[p].l,key);
73             }
74         }
75         else p=0;
76     }
77     else if(tr[p].key>key)remove(tr[p].l,key);
78     else remove(tr[p].r,key);
79     pushup(p);
80 }
81 int get_rank_by_key(int p,int key)
82 {
83     if(!p)return 0;
84     if(tr[p].key==key)return tr[tr[p].l].size;
85     if(tr[p].key>key)return get_rank_by_key(tr[p].l,key);
86     return tr[tr[p].l].size+tr[p].cnt+get_rank_by_key(tr[p].r,key);
87 }
88 int get_key_by_rank(int p,int rank)

```

```

89  {
90      if(!p)return INF;
91      if(tr[tr[p].l].size>=rank)return get_key_by_rank(tr[p].l,rank);
92      if(tr[tr[p].l].size+tr[p].cnt>=rank)return tr[p].key;
93      return get_key_by_rank(tr[p].r,rank-tr[tr[p].l].size-tr[p].cnt);
94  }
95  int get_prev(int p,int key)
96  {
97      if(!p)return -INF;
98      if(tr[p].key>=key)return get_prev(tr[p].l,key);
99      return max(tr[p].key,get_prev(tr[p].r,key));
100 }
101 int get_next(int p,int key)
102 {
103     if(!p)return INF;
104     if(tr[p].key<=key)return get_next(tr[p].r,key);
105     return min(tr[p].key,get_next(tr[p].l,key));
106 }
107 int main()
108 {
109     build();
110     int T;
111     cin>>T;
112     while(T-->0)
113     {
114         int c,x;
115         cin>>c>>x;
116         if(c==1)insert(root,x);
117         else if(c==2)remove(root,x);
118         else if(c==3)cout<<get_rank_by_key(root,x)<<endl;
119         else if(c==4)cout<<get_key_by_rank(root,x+1)<<endl;
120         else if(c==5)cout<<get_prev(root,x)<<endl;
121         else cout<<get_next(root,x)<<endl;
122     }
123     return 0;
124 }

```

Splay

```

1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  const int N = 100010;
6
7  int n, m;
8  struct Node

```

```

9  {
10     int s[2], p, v;
11     int size, flag;
12
13     void init(int _v, int _p)
14     {
15         v = _v, p = _p;
16         size = 1;
17     }
18 }tr[N];
19 int root, idx;
20
21 void pushup(int x)
22 {
23     tr[x].size = tr[tr[x].s[0]].size + tr[tr[x].s[1]].size + 1;
24 }
25
26 void pushdown(int x)
27 {
28     if (tr[x].flag)
29     {
30         swap(tr[x].s[0], tr[x].s[1]);
31         tr[tr[x].s[0]].flag ^= 1;
32         tr[tr[x].s[1]].flag ^= 1;
33         tr[x].flag = 0;
34     }
35 }
36
37 void rotate(int x)
38 {
39     int y = tr[x].p, z = tr[y].p;
40     int k = tr[y].s[1] == x; // k=0表示x是y的左儿子; k=1表示x是y的右儿子
41     tr[z].s[tr[z].s[1] == y] = x, tr[x].p = z;
42     tr[y].s[k] = tr[x].s[k ^ 1], tr[tr[x].s[k ^ 1]].p = y;
43     tr[x].s[k ^ 1] = y, tr[y].p = x;
44     pushup(y), pushup(x);
45 }
46
47 void splay(int x, int k)
48 {
49     while (tr[x].p != k)
50     {
51         int y = tr[x].p, z = tr[y].p;
52         if (z != k)
53             if ((tr[y].s[1] == x) ^ (tr[z].s[1] == y)) rotate(x);
54             else rotate(y);
55         rotate(x);
56     }
57     if (!k) root = x;

```

```

58 }
59
60 void insert(int v)
61 {
62     int u = root, p = 0;
63     while (u) p = u, u = tr[u].s[v > tr[u].v];
64     u = ++ idx;
65     if (p) tr[p].s[v > tr[p].v] = u;
66     tr[u].init(v, p);
67     splay(u, 0);
68 }
69
70 int get_k(int k)
71 {
72     int u = root;
73     while (true)
74     {
75         pushdown(u);
76         if (tr[tr[u].s[0]].size >= k) u = tr[u].s[0];
77         else if (tr[tr[u].s[0]].size + 1 == k) return u;
78         else k -= tr[tr[u].s[0]].size + 1, u = tr[u].s[1];
79     }
80     return -1;
81 }
82
83 void output(int u)
84 {
85     pushdown(u);
86     if (tr[u].s[0]) output(tr[u].s[0]);
87     if (tr[u].v >= 1 && tr[u].v <= n) printf("%d ", tr[u].v);
88     if (tr[u].s[1]) output(tr[u].s[1]);
89 }
90
91 int main()
92 {
93     scanf("%d%d", &n, &m);
94     for (int i = 0; i <= n + 1; i ++ ) insert(i);
95     while (m -- )
96     {
97         int l, r;
98         scanf("%d%d", &l, &r);
99         l = get_k(l), r = get_k(r + 2);
100         splay(l, 0), splay(r, l);
101         tr[tr[r].s[0]].flag ^= 1;
102     }
103     output(root);
104     return 0;
105 }

```

fhq treap

1 | 待补充

树套树

1 | 待补充

动态树

1 | 待补充

左偏树

三、图论

1.DFS

```
1  深度优先遍历
2
3  int dfs(int u)
4  {
5      st[u] = true; // st[u] 表示点u已经被遍历过
6
7      for (int i = h[u]; i != -1; i = ne[i])
8      {
9          int j = e[i];
10         if (!st[j]) dfs(j);
11     }
12 }
```

2.BFS

```
1  宽度优先遍历
2
3  queue<int> q;
4  st[1] = true; // 表示1号点已经被遍历过
5  q.push(1);
6
7  while (q.size())
8  {
9      int t = q.front();
```



```

10     q.pop();
11
12     for (int i = h[t]; i != -1; i = ne[i])
13     {
14         int j = e[i];
15         if (!st[j])
16         {
17             st[j] = true; // 表示点j已经被遍历过
18             q.push(j);
19         }
20     }
21 }

```

3.拓扑排序

```

1  bool topsort()
2  {
3      int hh = 0, tt = -1;
4
5      // d[i] 存储点i的入度
6      for (int i = 1; i <= n; i ++ )
7          if (!d[i])
8              q[ ++ tt] = i;
9
10     while (hh <= tt)
11     {
12         int t = q[hh ++ ];
13
14         for (int i = h[t]; i != -1; i = ne[i])
15         {
16             int j = e[i];
17             if (-- d[j] == 0)
18                 q[ ++ tt] = j;
19         }
20     }
21
22     // 如果所有点都入队了，说明存在拓扑序列；否则不存在拓扑序列。
23     return tt == n - 1;
24 }

```

4.最短路

dijkstra

```
1  时间复杂度是  $O(n^2+m)$ 
2  , n
3  表示点数, m
4  表示边数
5
6  int g[N][N]; // 存储每条边
7  int dist[N]; // 存储1号点到每个点的最短距离
8  bool st[N]; // 存储每个点的最短路是否已经确定
9
10 // 求1号点到n号点的最短路, 如果不存在则返回-1
11 int dijkstra()
12 {
13     memset(dist, 0x3f, sizeof dist);
14     dist[1] = 0;
15
16     for (int i = 0; i < n - 1; i++)
17     {
18         int t = -1; // 在还未确定最短路的点中, 寻找距离最小的点
19         for (int j = 1; j <= n; j++)
20             if (!st[j] && (t == -1 || dist[t] > dist[j]))
21                 t = j;
22
23         // 用t更新其他点的距离
24         for (int j = 1; j <= n; j++)
25             dist[j] = min(dist[j], dist[t] + g[t][j]);
26
27         st[t] = true;
28     }
29
30     if (dist[n] == 0x3f3f3f3f) return -1;
31     return dist[n];
32 }
```

堆优化dijkstra

```
1  时间复杂度  $O(m\log n)$ 
2  , n
3  表示点数, m
4  表示边数
5
6  typedef pair<int, int> PII;
7
8  int n; // 点的数量
9  int h[N], w[N], e[N], ne[N], idx; // 邻接表存储所有边
10 int dist[N]; // 存储所有点到1号点的距离
11 bool st[N]; // 存储每个点的最短距离是否已确定
```

```

12
13 // 求1号点到n号点的最短距离，如果不存在，则返回-1
14 int dijkstra()
15 {
16     memset(dist, 0x3f, sizeof dist);
17     dist[1] = 0;
18     priority_queue<PII, vector<PII>, greater<PII>> heap;
19     heap.push({0, 1}); // first存储距离, second存储节点编号
20
21     while (heap.size())
22     {
23         auto t = heap.top();
24         heap.pop();
25
26         int ver = t.second, distance = t.first;
27
28         if (st[ver]) continue;
29         st[ver] = true;
30
31         for (int i = h[ver]; i != -1; i = ne[i])
32         {
33             int j = e[i];
34             if (dist[j] > distance + w[i])
35             {
36                 dist[j] = distance + w[i];
37                 heap.push({dist[j], j});
38             }
39         }
40     }
41
42     if (dist[n] == 0x3f3f3f3f) return -1;
43     return dist[n];
44 }

```

bellman_ford

```

1 时间复杂度  $O(nm)$ 
2  , n表示点数,
3  m表示边数
4
5  注意在模板题中需要对下面的模板稍作修改，加上备份数组，详情见模板题。
6
7  int n, m; // n表示点数, m表示边数
8  int dist[N]; // dist[x]存储1到x的最短路距离
9
10 struct Edge // 边, a表示出点, b表示入点, w表示边的权重
11 {
12     int a, b, w;
13 }edges[M];

```

```

14
15 // 求1到n的最短路距离，如果无法从1走到n，则返回-1。
16 int bellman_ford()
17 {
18     memset(dist, 0x3f, sizeof dist);
19     dist[1] = 0;
20
21     // 如果第n次迭代仍然会松弛三角不等式，就说明存在一条长度是n+1的最短路径，由抽屉原理，路径中至少存在两个相同的点，说明图中存在负权回路。
22     for (int i = 0; i < n; i++)
23     {
24         for (int j = 0; j < m; j++)
25         {
26             int a = edges[j].a, b = edges[j].b, w = edges[j].w;
27             if (dist[b] > dist[a] + w)
28                 dist[b] = dist[a] + w;
29         }
30     }
31
32     if (dist[n] > 0x3f3f3f3f / 2) return -1;
33     return dist[n];
34 }

```

spfa

```

1  时间复杂度 平均情况下  $O(m)$ 
2  , 最坏情况下  $O(nm)$ 
3  , n
4  表示点数, m
5  表示边数
6
7  int n;          // 总点数
8  int h[N], w[N], e[N], ne[N], idx;          // 邻接表存储所有边
9  int dist[N];          // 存储每个点到1号点的最短距离
10 bool st[N];          // 存储每个点是否在队列中
11
12 // 求1号点到n号点的最短路距离，如果从1号点无法走到n号点则返回-1
13 int spfa()
14 {
15     memset(dist, 0x3f, sizeof dist);
16     dist[1] = 0;
17
18     queue<int> q;
19     q.push(1);
20     st[1] = true;
21
22     while (q.size())
23     {
24         auto t = q.front();

```

```

25     q.pop();
26
27     st[t] = false;
28
29     for (int i = h[t]; i != -1; i = ne[i])
30     {
31         int j = e[i];
32         if (dist[j] > dist[t] + w[i])
33         {
34             dist[j] = dist[t] + w[i];
35             if (!st[j]) // 如果队列中已存在j，则不需要将j重复插入
36             {
37                 q.push(j);
38                 st[j] = true;
39             }
40         }
41     }
42 }
43
44 if (dist[n] == 0x3f3f3f3f) return -1;
45 return dist[n];
46 }

```

spfa判断负环

```

1  .spfa判断图中是否存在负环
2  时间复杂度是  $O(nm)$ 
3  , n
4  表示点数, m
5  表示边数
6
7  int n; // 总点数
8  int h[N], w[N], e[N], ne[N], idx; // 邻接表存储所有边
9  int dist[N], cnt[N]; // dist[x]存储1号点到x的最短距离, cnt[x]存储1到x的最短路中经过
   的点数
10 bool st[N]; // 存储每个点是否在队列中
11
12 // 如果存在负环, 则返回true, 否则返回false。
13 bool spfa()
14 {
15     // 不需要初始化dist数组
16     // 原理: 如果某条最短路径上有n个点(除了自己), 那么加上自己之后一共有n+1个点, 由抽屉原理一定
   有两个点相同, 所以存在环。
17
18     queue<int> q;
19     for (int i = 1; i <= n; i++)
20     {
21         q.push(i);
22         st[i] = true;

```

```

23     }
24
25     while (q.size())
26     {
27         auto t = q.front();
28         q.pop();
29
30         st[t] = false;
31
32         for (int i = h[t]; i != -1; i = ne[i])
33         {
34             int j = e[i];
35             if (dist[j] > dist[t] + w[i])
36             {
37                 dist[j] = dist[t] + w[i];
38                 cnt[j] = cnt[t] + 1;
39                 if (cnt[j] >= n) return true; // 如果从1号点到x的最短路中包含至少n
个点（不包括自己），则说明存在环
40                 if (!st[j])
41                 {
42                     q.push(j);
43                     st[j] = true;
44                 }
45             }
46         }
47     }
48
49     return false;
50 }

```

floyd

```

1  时间复杂度是  $O(n^3)$ ,  $n$ 表示点数
2
3  初始化:
4      for (int i = 1; i <= n; i ++ )
5          for (int j = 1; j <= n; j ++ )
6              if (i == j) d[i][j] = 0;
7              else d[i][j] = INF;
8
9  // 算法结束后, d[a][b]表示a到b的最短距离
10 void floyd()
11 {
12     for (int k = 1; k <= n; k ++ )
13         for (int i = 1; i <= n; i ++ )
14             for (int j = 1; j <= n; j ++ )
15                 d[i][j] = min(d[i][j], d[i][k] + d[k][j]);

```

```
16 }
```

差分约束

```
1 待补充
```

5.最小生成树

Prim

```
1 时间复杂度是  $O(n^2+m)$ 
2 , n
3 表示点数, m
4 表示边数
5
6 int n;          // n表示点数
7 int g[N][N];     // 邻接矩阵, 存储所有边
8 int dist[N];     // 存储其他点到当前最小生成树的距离
9 bool st[N];      // 存储每个点是否已经在生成树中
10
11
12 // 如果图不连通, 则返回INF(值是0x3f3f3f3f), 否则返回最小生成树的树边权重之和
13 int prim()
14 {
15     memset(dist, 0x3f, sizeof dist);
16
17     int res = 0;
18     for (int i = 0; i < n; i ++ )
19     {
20         int t = -1;
21         for (int j = 1; j <= n; j ++ )
22             if (!st[j] && (t == -1 || dist[t] > dist[j]))
23                 t = j;
24
25         if (i && dist[t] == INF) return INF;
26
27         if (i) res += dist[t];
28         st[t] = true;
29
30         for (int j = 1; j <= n; j ++ ) dist[j] = min(dist[j], g[t][j]);
31     }
32
33     return res;
34 }
```

Kruskal

```
1  时间复杂度是  $O(m \log m)$ 
2  , n
3  表示点数, m
4  表示边数
5
6  int n, m;          // n是点数, m是边数
7  int p[N];          // 并查集的父节点数组
8
9  struct Edge        // 存储边
10 {
11     int a, b, w;
12
13     bool operator< (const Edge &W) const
14     {
15         return w < W.w;
16     }
17 }edges[M];
18
19 int find(int x)      // 并查集核心操作
20 {
21     if (p[x] != x) p[x] = find(p[x]);
22     return p[x];
23 }
24
25 int kruskal()
26 {
27     sort(edges, edges + m);
28
29     for (int i = 1; i <= n; i++) p[i] = i;    // 初始化并查集
30
31     int res = 0, cnt = 0;
32     for (int i = 0; i < m; i++)
33     {
34         int a = edges[i].a, b = edges[i].b, w = edges[i].w;
35
36         a = find(a), b = find(b);
37         if (a != b)    // 如果两个连通块不连通, 则将这两个连通块合并
38         {
39             p[a] = b;
40             res += w;
41             cnt++;
42         }
43     }
44
45     if (cnt < n - 1) return INF;
46     return res;
47 }
```


6.二分图

染色法判定二分图

```
1  时间复杂度是  $O(n+m)$ 
2  , n
3  表示点数, m
4  表示边数
5
6  int n;          // n表示点数
7  int h[N], e[M], ne[M], idx;    // 邻接表存储图
8  int color[N];    // 表示每个点的颜色, -1表示未染色, 0表示白色, 1表示黑色
9
10 // 参数: u表示当前节点, c表示当前点的颜色
11 bool dfs(int u, int c)
12 {
13     color[u] = c;
14     for (int i = h[u]; i != -1; i = ne[i])
15     {
16         int j = e[i];
17         if (color[j] == -1)
18         {
19             if (!dfs(j, !c)) return false;
20         }
21         else if (color[j] == c) return false;
22     }
23
24     return true;
25 }
26
27 bool check()
28 {
29     memset(color, -1, sizeof color);
30     bool flag = true;
31     for (int i = 1; i <= n; i++)
32     {
33         if (color[i] == -1)
34             if (!dfs(i, 0))
35             {
36                 flag = false;
37                 break;
38             }
39     }
40     return flag;
```

匈牙利算法

```

1  时间复杂度是  $O(nm)$ 
2  n表示点数
3  m表示边数
4
5  int n1, n2;      // n1表示第一个集合中的点数, n2表示第二个集合中的点数
6  int h[N], e[M], ne[M], idx;    // 邻接表存储所有边, 匈牙利算法中只会用到从第一个集合指向第
   二个集合的边, 所以这里只用存一个方向的边
7  int match[N];    // 存储第二个集合中的每个点当前匹配的第一个集合中的点是哪个
8  bool st[N];      // 表示第二个集合中的每个点是否已经被遍历过
9
10 bool find(int x)
11 {
12     for (int i = h[x]; i != -1; i = ne[i])
13     {
14         int j = e[i];
15         if (!st[j])
16         {
17             st[j] = true;
18             if (match[j] == 0 || find(match[j]))
19             {
20                 match[j] = x;
21                 return true;
22             }
23         }
24     }
25
26     return false;
27 }
28
29 // 求最大匹配数, 依次枚举第一个集合中的每个点能否匹配第二个集合中的点
30 int res = 0;
31 for (int i = 1; i <= n1; i ++ )
32 {
33     memset(st, false, sizeof st);
34     if (find(i)) res ++ ;
35 }

```

7.最近公共祖先

1 | 待补充

8.有向图的强连通分量

1 | 待补充

9.无向图的双连通分量

1 | 待补充

10.欧拉路径和欧拉回路

1 | 待补充

11.网络流

最大流

1 | 待补充

最小割

1 | 待补充

费用流

1 | 待补充

12.2-SAT

1 |

13.prufer编码

1 |

四、字符串

1.kmp

```
1 // s[]是长文本, p[]是模式串, n是s的长度, m是p的长度
2 求模式串的Next数组:
3 for (int i = 2, j = 0; i <= m; i ++ )
4 {
5     while (j && p[i] != p[j + 1]) j = ne[j];
6     if (p[i] == p[j + 1]) j ++ ;
7     ne[i] = j;
8 }
9
10 // 匹配
11 for (int i = 1, j = 0; i <= n; i ++ )
12 {
13     while (j && s[i] != p[j + 1]) j = ne[j];
14     if (s[i] == p[j + 1]) j ++ ;
15     if (j == m)
16     {
17         j = ne[j];
18         // 匹配成功后的逻辑
19     }
20 }
```

2.Trie

```
1 int son[N][26], cnt[N], idx;
2 // 0号点既是根节点, 又是空节点
3 // son[][]存储树中每个节点的子节点
4 // cnt[]存储以每个节点结尾的单词数量
5
6 // 插入一个字符串
7 void insert(char *str)
8 {
9     int p = 0;
10    for (int i = 0; str[i]; i ++ )
11    {
12        int u = str[i] - 'a';
13        if (!son[p][u]) son[p][u] = ++ idx;
14        p = son[p][u];
15    }
16    cnt[p] ++ ;
17 }
18
19 // 查询字符串出现的次数
20 int query(char *str)
21 {
22     int p = 0;
23     for (int i = 0; str[i]; i ++ )
```

```
24     {
25         int u = str[i] - 'a';
26         if (!son[p][u]) return 0;
27         p = son[p][u];
28     }
29     return cnt[p];
30 }
```

3.AC自动机

1 | 待补充

4.后缀数组

1 | 待补充

5.后缀自动机（SAM）

1 | 待补充

6. manacher算法

1 | 待补充

7.回文自动机（PAM）

1 | 待补充

五、动态规划（DP）

1.数字三角形模型

2.最大上升子序列模型

3.背包模型

4.状态机模型

5.状态压缩dp

6.区间dp

7.树形dp/记忆化搜索

8.数位dp

9.基环树dp

10.插头dp

11.四边形不等式优化

12.数据结构优化dp

单调队列优化dp

六、数学

1.质数

试除法判定质数

```
1  bool is_prime(int x)
2  {
3      if (x < 2) return false;
4      for (int i = 2; i <= x / i; i ++ )
5          if (x % i == 0)
6              return false;
7      return true;
8  }
```

试除法分解质因数

```

1 void divide(int x)
2 {
3     for (int i = 2; i <= x / i; i ++ )
4         if (x % i == 0)
5             {
6                 int s = 0;
7                 while (x % i == 0) x /= i, s ++ ;
8                 cout << i << ' ' << s << endl;
9             }
10    if (x > 1) cout << x << ' ' << 1 << endl;
11    cout << endl;
12 }

```

线性筛法求素数

```

1 int primes[N], cnt;    // primes[]存储所有素数
2 bool st[N];           // st[x]存储x是否被筛掉
3
4 void get_primes(int n)
5 {
6     for (int i = 2; i <= n; i ++ )
7     {
8         if (!st[i]) primes[cnt ++ ] = i;
9         for (int j = 0; primes[j] <= n / i; j ++ )
10            {
11                st[primes[j] * i] = true;
12                if (i % primes[j] == 0) break;
13            }
14     }
15 }

```

2.约数

试除法求所有约数

```

1  vector<int> get_divisors(int x)
2  {
3      vector<int> res;
4      for (int i = 1; i <= x / i; i ++ )
5          if (x % i == 0)
6              {
7                  res.push_back(i);
8                  if (i != x / i) res.push_back(x / i);
9              }
10     sort(res.begin(), res.end());
11     return res;
12 }

```

约数个数与约数之和

```

1  如果  $N = p_1^{c_1} * p_2^{c_2} * \dots * p_k^{c_k}$ 
2  约数个数:  $(c_1 + 1) * (c_2 + 1) * \dots * (c_k + 1)$ 
3  约数之和:  $(p_1^0 + p_1^1 + \dots + p_1^{c_1}) * \dots * (p_k^0 + p_k^1 + \dots + p_k^{c_k})$ 

```

3.欧拉函数

求欧拉函数

```

1  {
2      int res = x;
3      for (int i = 2; i <= x / i; i ++ )
4          if (x % i == 0)
5              {
6                  res = res / i * (i - 1);
7                  while (x % i == 0) x /= i;
8              }
9      if (x > 1) res = res / x * (x - 1);
10
11     return res;
12 }

```

筛法求欧拉函数

```

1  int primes[N], cnt;          // primes[]存储所有素数
2  int euler[N];                // 存储每个数的欧拉函数
3  bool st[N];                  // st[x]存储x是否被筛掉
4
5
6  void get_eulers(int n)
7  {

```



```

8     euler[1] = 1;
9     for (int i = 2; i <= n; i ++ )
10    {
11        if (!st[i])
12        {
13            primes[cnt ++ ] = i;
14            euler[i] = i - 1;
15        }
16        for (int j = 0; primes[j] <= n / i; j ++ )
17        {
18            int t = primes[j] * i;
19            st[t] = true;
20            if (i % primes[j] == 0)
21            {
22                euler[t] = euler[i] * primes[j];
23                break;
24            }
25            euler[t] = euler[i] * (primes[j] - 1);
26        }
27    }
28 }

```

4.快速幂

```

1  求  $m^k \bmod p$ , 时间复杂度  $O(\log k)$ 。
2
3  int qmi(int m, int k, int p)
4  {
5      int res = 1 % p, t = m;
6      while (k)
7      {
8          if (k&1) res = res * t % p;
9          t = t * t % p;
10         k >>= 1;
11     }
12     return res;
13 }

```

5.扩展欧几里得算法

欧几里德算法

```

1  int gcd(int a, int b)
2  {
3      return b ? gcd(b, a % b) : a;
4  }

```

扩展欧几里得算法

```
1 // 求x, y, 使得ax + by = gcd(a, b)
2 int exgcd(int a, int b, int &x, int &y)
3 {
4     if (!b)
5     {
6         x = 1; y = 0;
7         return a;
8     }
9     int d = exgcd(b, a % b, y, x);
10    y -= (a/b) * x;
11    return d;
12 }
```

6.中国剩余定理

7.高斯消元

```
1 // a[N][N]是增广矩阵
2 int gauss()
3 {
4     int c, r;
5     for (c = 0, r = 0; c < n; c++)
6     {
7         int t = r;
8         for (int i = r; i < n; i++) // 找到绝对值最大的行
9             if (fabs(a[i][c]) > fabs(a[t][c]))
10                t = i;
11
12         if (fabs(a[t][c]) < eps) continue;
13
14         for (int i = c; i <= n; i++) swap(a[t][i], a[r][i]); // 将绝对值最大的
行换到最顶端
15         for (int i = n; i >= c; i--) a[r][i] /= a[r][c]; // 将当前行的首位变成1
16         for (int i = r + 1; i < n; i++) // 用当前行将下面所有的列消成0
17             if (fabs(a[i][c]) > eps)
18                 for (int j = n; j >= c; j--)
19                     a[i][j] -= a[r][j] * a[i][c];
20
21         r++;
22     }
23
24     if (r < n)
25     {
26         for (int i = r; i < n; i++)
27             if (fabs(a[i][n]) > eps)
28                 return 2; // 无解
```

```

29         return 1; // 有无穷多组解
30     }
31
32     for (int i = n - 1; i >= 0; i -- )
33         for (int j = i + 1; j < n; j ++ )
34             a[i][n] -= a[i][j] * a[j][n];
35
36     return 0; // 有唯一解
37 }

```

8.组合计数

递推法求组合数

```

1 // c[a][b] 表示从a个苹果中选b个的方案数
2 for (int i = 0; i < N; i ++ )
3     for (int j = 0; j <= i; j ++ )
4         if (!j) c[i][j] = 1;
5         else c[i][j] = (c[i - 1][j] + c[i - 1][j - 1]) % mod;

```

通过预处理逆元的方式求组合数

```

1 首先预处理出所有阶乘取模的余数fact[N]，以及所有阶乘取模的逆元infact[N]
2 如果取模的数是质数，可以用费马小定理求逆元
3 int qmi(int a, int k, int p) // 快速幂模板
4 {
5     int res = 1;
6     while (k)
7     {
8         if (k & 1) res = (LL)res * a % p;
9         a = (LL)a * a % p;
10        k >>= 1;
11    }
12    return res;
13 }
14
15 // 预处理阶乘的余数和阶乘逆元的余数
16 fact[0] = infact[0] = 1;
17 for (int i = 1; i < N; i ++ )
18 {
19     fact[i] = (LL)fact[i - 1] * i % mod;
20     infact[i] = (LL)infact[i - 1] * qmi(i, mod - 2, mod) % mod;
21 }

```

Lucas定理

```
1  若p是质数, 则对于任意整数  $1 \leq m \leq n$ , 有:
2       $C(n, m) = C(n \% p, m \% p) * C(n / p, m / p) \pmod{p}$ 
3
4  int qmi(int a, int k, int p)  // 快速幂模板
5  {
6      int res = 1 % p;
7      while (k)
8      {
9          if (k & 1) res = (LL)res * a % p;
10         a = (LL)a * a % p;
11         k >>= 1;
12     }
13     return res;
14 }
15
16 int C(int a, int b, int p)  // 通过定理求组合数C(a, b)
17 {
18     if (a < b) return 0;
19
20     LL x = 1, y = 1;  // x是分子, y是分母
21     for (int i = a, j = 1; j <= b; i --, j ++ )
22     {
23         x = (LL)x * i % p;
24         y = (LL)y * j % p;
25     }
26
27     return x * (LL)qmi(y, p - 2, p) % p;
28 }
29
30 int lucas(LL a, LL b, int p)
31 {
32     if (a < p && b < p) return C(a, b, p);
33     return (LL)C(a % p, b % p, p) * lucas(a / p, b / p, p) % p;
34 }
```

分解质因数法求组合数

- 1 当我们需要求出组合数的真实值, 而非对某个数的余数时, 分解质因数的方式比较好用:
- 2 1. 筛法求出范围内的所有质数
- 3 2. 通过 $C(a, b) = a! / b! / (a - b)!$ 这个公式求出每个质因子的次数。 $n!$ 中p的次数是 $n / p + n / p^2 + n / p^3 + \dots$
- 4 3. 用高精度乘法将所有质因子相乘
- 5

```

6  int primes[N], cnt;      // 存储所有质数
7  int sum[N];             // 存储每个质数的次数
8  bool st[N];             // 存储每个数是否已被筛掉
9
10
11 void get_primes(int n)    // 线性筛法求素数
12 {
13     for (int i = 2; i <= n; i ++ )
14     {
15         if (!st[i]) primes[cnt ++ ] = i;
16         for (int j = 0; primes[j] <= n / i; j ++ )
17         {
18             st[primes[j] * i] = true;
19             if (i % primes[j] == 0) break;
20         }
21     }
22 }
23
24
25 int get(int n, int p)     // 求n! 中的次数
26 {
27     int res = 0;
28     while (n)
29     {
30         res += n / p;
31         n /= p;
32     }
33     return res;
34 }
35
36
37 vector<int> mul(vector<int> a, int b)    // 高精度乘低精度模板
38 {
39     vector<int> c;
40     int t = 0;
41     for (int i = 0; i < a.size(); i ++ )
42     {
43         t += a[i] * b;
44         c.push_back(t % 10);
45         t /= 10;
46     }
47
48     while (t)
49     {
50         c.push_back(t % 10);
51         t /= 10;
52     }
53
54     return c;

```

```

55 }
56
57 get_primes(a); // 预处理范围内的所有质数
58
59 for (int i = 0; i < cnt; i ++ ) // 求每个质因数的次数
60 {
61     int p = primes[i];
62     sum[i] = get(a, p) - get(b, p) - get(a - b, p);
63 }
64
65 vector<int> res;
66 res.push_back(1);
67
68 for (int i = 0; i < cnt; i ++ ) // 用高精度乘法将所有质因子相乘
69     for (int j = 0; j < sum[i]; j ++ )
70         res = mul(res, primes[i]);

```

9.卡特兰数

- 1 给定 n 个0和 n 个1，它们按照某种顺序排成长度为 $2n$ 的序列，满足任意前缀中0的个数都不少于1的个数的序列的数量为： $Cat(n) = C(2n, n) / (n + 1)$

10.容斥原理

11.博弈论

- 1 ****NIM游戏****
- 2 给定 N 堆物品，第 i 堆物品有 A_i 个。两名玩家轮流行动，每次可以任选一堆，取走任意多个物品，可把一堆取光，但不能不取。取走最后一件物品者获胜。两人都采取最优策略，问先手是否必胜。
- 3
- 4 我们把这种游戏称为NIM博弈。把游戏过程中面临的状态称为局面。整局游戏第一个行动的称为先手，第二个行动的称为后手。若在某一局面下无论采取何种行动，都会输掉游戏，则称该局面必败。
- 5 所谓采取最优策略是指，若在某一局面下存在某种行动，使得行动后对方面临必败局面，则优先采取该行动。同时，这样的局面被称为必胜。我们讨论的博弈问题一般都只考虑理想情况，即两人都无失误，都采取最优策略行动时游戏的结果。
- 6 NIM博弈不存在平局，只有先手必胜和先手必败两种情况。
- 7
- 8 定理： NIM博弈先手必胜，当且仅当 $A_1 \oplus A_2 \oplus \dots \oplus A_n \neq 0$
- 9
- 10 ****公平组合游戏ICG****
- 11
- 12 若一个游戏满足：
- 13
- 14 由两名玩家交替行动；
- 15 在游戏进程的任意时刻，可以执行的合法行动与轮到哪名玩家无关；
- 16 不能行动的玩家判负；
- 17 则称该游戏为一个公平组合游戏。

NIM博弈属于公平组合游戏，但城建的棋类游戏，比如围棋，就不是公平组合游戏。因为围棋交战双方分别只能落黑子和白子，胜负判定也比较复杂，不满足条件2和条件3。

有向图游戏

给定一个有向无环图，图中有一个唯一的起点，在起点上放有一枚棋子。两名玩家交替地把这枚棋子沿有向边进行移动，每次可以移动一步，无法移动者判负。该游戏被称为有向图游戏。

任何一个公平组合游戏都可以转化为有向图游戏。具体方法是，把每个局面看成图中的一个节点，并且从每个局面向沿着合法行动能够到达的下一个局面连有向边。

Mex运算

设 S 表示一个非负整数集合。定义 $\text{mex}(S)$ 为求出不属于集合 S 的最小非负整数的运算，即：

$\text{mex}(S) = \min\{x\}$ ， x 属于自然数，且 x 不属于 S

SG函数

在有向图游戏中，对于每个节点 x ，设从 x 出发共有 k 条有向边，分别到达节点 y_1, y_2, \dots, y_k ，定义 $\text{SG}(x)$ 为 x 的后继节点 y_1, y_2, \dots, y_k 的SG函数值构成的集合再执行 $\text{mex}(S)$ 运算的结果，即：

$\text{SG}(x) = \text{mex}(\{\text{SG}(y_1), \text{SG}(y_2), \dots, \text{SG}(y_k)\})$

特别地，整个有向图游戏 G 的SG函数值被定义为有向图游戏起点 s 的SG函数值，即 $\text{SG}(G) = \text{SG}(s)$ 。

有向图游戏的和 — 模板题 AcWing 893. 集合-Nim游戏

设 G_1, G_2, \dots, G_m 是 m 个有向图游戏。定义有向图游戏 G ，它的行动规则是任选某个有向图游戏 G_i ，并在 G_i 上行动一步。 G 被称为有向图游戏 G_1, G_2, \dots, G_m 的和。

有向图游戏的和的SG函数值等于它包含的各个子游戏SG函数值的异或和，即：

$\text{SG}(G) = \text{SG}(G_1) \oplus \text{SG}(G_2) \oplus \dots \oplus \text{SG}(G_m)$

定理

有向图游戏的某个局面必胜，当且仅当该局面对应节点的SG函数值大于0。

有向图游戏的某个局面必败，当且仅当该局面对应节点的SG函数值等于0。

12.同余

1 |

13.矩阵乘法

1 |

14.概率与数学期望

1

15.积性函数

1

16.BSGS

1

17.FFT

1

18.生成函数

1

19.Burnside引理和Polya定理

1

20.斯特林数

1

21.线性基

1

七、计算几何

1.二维计算几何基础

1 |

2.凸包

1 |

3.半平面交

1 |

4.最小圆覆盖

1 |

5.三维计算几何基础

1 |

6.三维凸包

1 |

7.旋转卡壳

1 |

8.三角剖分

1 |

9.扫描线

1 |

10.自适应辛普森积分

1