XCPC算法

一、基础算法

1.排序

快速排序

```
void quick_sort(int q[], int l, int r)
{
    if (l >= r) return;

    int i = l - 1, j = r + 1, x = q[l + r >> 1];
    while (i < j)
    {
        do i ++ ; while (q[i] < x);
        do j -- ; while (q[j] > x);
        if (i < j) swap(q[i], q[j]);
    }
    quick_sort(q, l, j), quick_sort(q, j + 1, r);
}
```

归并排序

```
void merge_sort(int q[], int l, int r)
{
    if (l >= r) return;

    int mid = l + r >> 1;
    merge_sort(q, l, mid);
    merge_sort(q, mid + 1, r);

    int k = 0, i = l, j = mid + 1;
    while (i <= mid && j <= r)
        if (q[i] <= q[j]) tmp[k ++ ] = q[i ++ ];
        else tmp[k ++ ] = q[j ++ ];

    while (i <= mid) tmp[k ++ ] = q[i ++ ];
    while (j <= r) tmp[k ++ ] = q[j ++ ];

    for (i = l, j = 0; i <= r; i ++, j ++ ) q[i] = tmp[j];
}
```

2.二分

整数二分

```
bool check(int x) {/* ... */} // 检查x是否满足某种性质

// 区间[l, r]被划分成[l, mid]和[mid + 1, r]时使用:
int bsearch_1(int l, int r)
{
    while (l < r)
    {
        int mid = l + r >> 1;
        if (check(mid)) r = mid;    // check()判断mid是否满足性质
        else l = mid + 1;
    }
    return l;
}
// 区间[l, r]被划分成[l, mid - 1]和[mid, r]时使用:
int bsearch_2(int l, int r)
{
    while (l < r)
    {
        int mid = l + r + 1 >> 1;
        if (check(mid)) l = mid;
        else r = mid - 1;
    }
    return l;
}

```

浮点数二分

```
bool check(double x) {/* ... */} // 检查x是否满足某种性质

double bsearch_3(double l, double r)
{
    const double eps = 1e-6;   // eps 表示精度，取决于题目对精度的要求
    while (r - l > eps)
    {
        double mid = (l + r) / 2;
        if (check(mid)) r = mid;
        else l = mid;
    }
    return l;
}
```

3.高精度

高精度加法

```
// C = A + B, A >= 0, B >= 0
vector<int> add(vector<int> &A, vector<int> &B)
{
    if (A.size() < B.size()) return add(B, A);

    vector<int> C;
    int t = 0;
    for (int i = 0; i < A.size(); i ++ )
    {
        t += A[i];
        if (i < B.size()) t += B[i];
        C.push_back(t % 10);
        t /= 10;
    }

    if (t) C.push_back(t);
    return C;
}
```

高精度减法

```
// C = A - B, 满足A >= B, A >= 0, B >= 0
vector<int> sub(vector<int> &A, vector<int> &B)
{
    vector<int> C;
    for (int i = 0, t = 0; i < A.size(); i ++ )
    {
        t = A[i] - t;
        if (i < B.size()) t -= B[i];
        C.push_back((t + 10) % 10);
        if (t < 0) t = 1;
        else t = 0;
    }

    while (C.size() > 1 && C.back() == 0) C.pop_back();
    return C;
}
```

高精度乘低精度

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

```

    int t = 0;
    for (int i = 0; i < A.size() || t; i ++ )
    {
        if (i < A.size()) t += A[i] * b;
        C.push_back(t % 10);
        t /= 10;
    }

    while (C.size() > 1 && C.back() == 0) C.pop_back();

    return C;
}
```

高精度除以低精度

```
// A / b = C ... r, A >= 0, b > 0
vector<int> div(vector<int> &A, int b, int &r)
{
    vector<int> C;
    r = 0;
    for (int i = A.size() - 1; i >= 0; i -- )
    {
        r = r * 10 + A[i];
        C.push_back(r / b);
        r %= b;
    }
    reverse(C.begin(), C.end());
    while (C.size() > 1 && C.back() == 0) C.pop_back();
    return C;
}
```

4.双指针算法

```
for (int i = 0, j = 0; i < n; i ++ )
{
    while (j < i && check(i, j)) j ++ ;

    // 具体问题的逻辑
}
常见问题分类:
    (1) 对于一个序列，用两个指针维护一段区间
(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

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

6.离散化

```cpp
vector<int> alls; // 存储所有待离散化的值
sort(alls.begin(), alls.end()); // 将所有值排序
alls.erase(unique(alls.begin(), alls.end()), alls.end());   // 去掉重复元素

// 二分求出x对应的离散化的值
int find(int x) // 找到第一个大于等于x的位置
{
    int l = 0, r = alls.size() - 1;
    while (l < r)
    {
        int mid = l + r >> 1;
        if (alls[mid] >= x) r = mid;
        else l = mid + 1;
    }
    return r + 1; // 映射到1, 2, ...n
}
```

7.区间合并

```cpp
// 将所有存在交集的区间合并
void merge(vector<PII> &segs)
{
    vector<PII> res;

    sort(segs.begin(), segs.end());

    int st = -2e9, ed = -2e9;
    for (auto seg : segs)
        if (ed < seg.first)
        {
            if (st != -2e9) res.push_back({st, ed});
            st = seg.first, ed = seg.second;
        }
        else ed = max(ed, seg.second);

    if (st != -2e9) res.push_back({st, ed});

    segs = res;
}

```

8.启发式合并

```
待补充
```

9.最小表示法

```
待补充
```

10.点分治与点分树

```

```

11.CDQ分治

```

```

二、数据结构

1.链表

单链表

```
// 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
int head, e[N], ne[N], idx;

// 初始化
void init()
{
    head = -1;
    idx = 0;
}

// 在链表头插入一个数a
void insert(int a)
{
    e[idx] = a, ne[idx] = head, head = idx ++ ;
}

// 将头结点删除，需要保证头结点存在
void remove()
{
    head = ne[head];
```

```
}
```

双链表

```
// 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
int e[N], l[N], r[N], idx;

// 初始化
void init()
{
    //0是左端点，1是右端点
    r[0] = 1, l[1] = 0;
    idx = 2;
}

// 在节点a的右边插入一个数x
void insert(int a, int x)
{
    e[idx] = x;
    l[idx] = a, r[idx] = r[a];
    l[r[a]] = idx, r[a] = idx ++ ;
}

// 删除节点a
void remove(int a)
{
    l[r[a]] = l[a];
    r[l[a]] = r[a];
}
```

2.栈与队列

栈

```
// tt表示栈顶
int stk[N], tt = 0;

// 向栈顶插入一个数
stk[ ++ tt] = x;

// 从栈顶弹出一个数
tt -- ;

// 栈顶的值
stk[tt];

// 判断栈是否为空，如果 tt > 0，则表示不为空
if (tt > 0)
```

```
{

}
```

队列

```
1. 普通队列:

// hh 表示队头，tt表示队尾
int q[N], hh = 0, tt = -1;

// 向队尾插入一个数
q[ ++ tt] = x;

// 从队头弹出一个数
hh ++ ;

// 队头的值
q[hh];

// 判断队列是否为空，如果 hh <= tt，则表示不为空
if (hh <= tt)
{

}
2. 循环队列

// hh 表示队头，tt表示队尾的后一个位置
int q[N], hh = 0, tt = 0;

// 向队尾插入一个数
q[tt ++ ] = x;
if (tt == N) tt = 0;

// 从队头弹出一个数
hh ++ ;
if (hh == N) hh = 0;

// 队头的值
q[hh];

// 判断队列是否为空，如果hh != tt，则表示不为空
if (hh != tt)
{

}
```

3.单调栈,单调队列

```
常见模型：找出每个数左边离它最近的比它大/小的数
int tt = 0;
for (int i = 1; i <= n; i ++ )
{
    while (tt && check(stk[tt], i)) tt -- ;
    stk[ ++ tt] = i;
}

常见模型：找出滑动窗口中的最大值/最小值
int hh = 0, tt = -1;
for (int i = 0; i < n; i ++ )
{
    while (hh <= tt && check_out(q[hh])) hh ++ ;  // 判断队头是否滑出窗口
    while (hh <= tt && check(q[tt], i)) tt -- ;
    q[ ++ tt] = i;
}
```

4.哈希表

一般哈希

```
(1) 拉链法
    int h[N], e[N], ne[N], idx;

    // 向哈希表中插入一个数
    void insert(int x)
    {
        int k = (x % N + N) % N;
        e[idx] = x;
        ne[idx] = h[k];
        h[k] = idx ++ ;
    }

    // 在哈希表中查询某个数是否存在
    bool find(int x)
    {
        int k = (x % N + N) % N;
        for (int i = h[k]; i != -1; i = ne[i])
            if (e[i] == x)
                return true;

        return false;
    }

(2) 开放寻址法
    int h[N];
```

```

    // 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
    int find(int x)
    {
        int t = (x % N + N) % N;
        while (h[t] != null && h[t] != x)
        {
            t ++ ;
            if (t == N) t = 0;
        }
        return t;
    }
```

字符串哈希

```
typedef unsigned long long ULL;
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 mod 2^64

// 初始化
p[0] = 1;
for (int i = 1; i <= n; i ++ )
{
    h[i] = h[i - 1] * P + str[i];
    p[i] = p[i - 1] * P;
}

// 计算子串 str[l ~ r] 的哈希值
ULL get(int l, int r)
{
    return h[r] - h[l - 1] * p[r - l + 1];
}
```

5.前缀和与差分

一维前缀和

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

二维前缀和

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

一维差分

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

二维差分

```
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：
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

```
int lowbit(int x)
{
    return x&(-x);
}
void add(int x,int c)
{
    for(int i=x;i<=n;i+=lowbit(i))tr[i]+=c;
}
int sum(int x)
{
    int res=0;
    for(int i=x;i;i-=lowbit(i))res+=tr[i];
    return res;
}
```

7.线段树

单点修改

```
struct Node
{
    int l, r;
    int v;  // 区间[l, r]中的最大值
}tr[N * 4];

void pushup(int u)  // 由子节点的信息，来计算父节点的信息
{
    tr[u].v = max(tr[u << 1].v, tr[u << 1 | 1].v);
}

```

```
void build(int u, int l, int r)
{
    tr[u] = {l, r};
    if (l == r) return;
    int mid = l + r >> 1;
build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
pushup(u);
}

int query(int u, int l, int r)
{
    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

    int mid = tr[u].l + tr[u].r >> 1;
    int v = 0;
    if (l <= mid) v = query(u << 1, l, r);
    if (r > mid) v = max(v, query(u << 1 | 1, l, r));

    return v;
}

void modify(int u, int x, int v)
{
    if (tr[u].l == x && tr[u].r == x) tr[u].v = v;
    else
    {
        int mid = tr[u].l + tr[u].r >> 1;
        if (x <= mid) modify(u << 1, x, v);
        else modify(u << 1 | 1, x, v);
        pushup(u);
    }
}
```

区间修改(lazy tag)

```
struct Node
{
    int l,r,sum,add;
}tr[N*4];
void pushup(Node &u,Node &l,Node &r)
{
    u.sum=l.sum+r.sum;
}
void pushup(int u)
{
    pushup(tr[u],tr[u<<1],tr[u<<1|1]);
}
void pushdown(int u)
```

```
{
    Node &root=tr[u],&left=tr[u<<1],&right=tr[u<<1|1];
    if(root.add)
    {
        left.add+=root.add,left.sum+=(left.r-left.l+1)*root.add;
        right.add+=root.add,right.sum+=(right.r-right.l+1)*root.add;
        root.add=0;
    }
}
void build(int u,int l,int r)
{
    if(l==r)
    {
        tr[u]={l,r,a[l],0};
        return;
    }else
    {
        tr[u]={l,r};
        int mid=(l+r)>>1;
        build(u<<1,l,mid),build(u<<1|1,mid+1,r);
        pushup(u);
    }
}
void modify(int u,int l,int r,int d)
{
    if(tr[u].l>=l&&tr[u].r<=r)
    {
        tr[u].sum+=(tr[u].r-tr[u].l+1)*d;
        tr[u].add+=d;
    }else
    {
        pushdown(u);
        int mid=(tr[u].l+tr[u].r)>>1;
        if(l<=mid)modify(u<<1,l,r,d);
        if(r>mid)modify(u<<1|1,l,r,d);
        pushup(u);
    }
}
Node query(int u,int l,int r)
{
    if(tr[u].l>=l&&tr[u].r<=r)return tr[u];
    pushdown(u);
    int mid=(tr[u].l+tr[u].r)>>1;
    if(l>mid)return query(u<<1|1,l,r);
    if(r<=mid)return query(u<<1,l,r);
    else
    {
        auto left=query(u<<1,l,r);
        auto right=query(u<<1|1,l,r);
```

```
        Node res;
        pushup(res,left,right);
        return res;
    }
}
```

可持久化线段树

```
#include <cstdio>
#include <cstring>
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

const int N = 100010, M = 10010;

int n, m;
int a[N];
vector<int> nums;

struct Node
{
    int l, r;
    int cnt;
}tr[N * 4 + N * 17];

int root[N], idx;

int find(int x)
{
    return lower_bound(nums.begin(), nums.end(), x) - nums.begin();
}

int build(int l, int r)
{
    int p = ++ idx;
    if (l == r) return p;
    int mid = l + r >> 1;
    tr[p].l = build(l, mid), tr[p].r = build(mid + 1, r);
    return p;
}

int insert(int p, int l, int r, int x)
{
    int q = ++ idx;
    tr[q] = tr[p];
    if (l == r)
```

```
    {
        tr[q].cnt ++ ;
        return q;
    }
    int mid = l + r >> 1;
    if (x <= mid) tr[q].l = insert(tr[p].l, l, mid, x);
    else tr[q].r = insert(tr[p].r, mid + 1, r, x);
    tr[q].cnt = tr[tr[q].l].cnt + tr[tr[q].r].cnt;
    return q;
}

int query(int q, int p, int l, int r, int k)
{
    if (l == r) return r;
    int cnt = tr[tr[q].l].cnt - tr[tr[p].l].cnt;
    int mid = l + r >> 1;
    if (k <= cnt) return query(tr[q].l, tr[p].l, l, mid, k);
    else return query(tr[q].r, tr[p].r, mid + 1, r, k - cnt);
}

int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i ++ )
    {
        scanf("%d", &a[i]);
        nums.push_back(a[i]);
    }

    sort(nums.begin(), nums.end());
    nums.erase(unique(nums.begin(), nums.end()), nums.end());

    root[0] = build(0, nums.size() - 1);

    for (int i = 1; i <= n; i ++ )
        root[i] = insert(root[i - 1], 0, nums.size() - 1, find(a[i]));

    while (m -- )
    {
        int l, r, k;
        scanf("%d%d%d", &l, &r, &k);
        printf("%d\n", nums[query(root[r], root[l - 1], 0, nums.size() - 1, k)]);
    }

    return 0;
}
```

8.堆

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

9.平衡树

Treap

```
#include<bits/stdc++.h>
using namespace std;
const int N=1e5+5,INF=0x3f3f3f3f;
int n;
struct Node
{
    int l,r;
    int key,val;
    int cnt,size;
}tr[N];
int root,idx;
void pushup(int p)
{
    tr[p].size=tr[tr[p].l].size+tr[tr[p].r].size+tr[p].cnt;
}
int get_node(int key)
{
    tr[++idx].key=key;
    tr[idx].val=rand();
    tr[idx].cnt=tr[idx].size=1;
    return idx;
}
void zig(int &p)
{
    int q=tr[p].l;
    tr[p].l=tr[q].r,tr[q].r=p,p=q;
    pushup(tr[p].r),pushup(p);
}
void zag(int &p)
{
    int q=tr[p].r;
    tr[p].r=tr[q].l,tr[q].l=p,p=q;
    pushup(tr[p].l),pushup(p);
}
void build()
{
    get_node(-INF),get_node(INF);
    root=1,tr[1].r=2;
    pushup(root);
```

```
}
void insert(int &p,int key)
{
    if(!p)p=get_node(key);
    else if(tr[p].key==key)tr[p].cnt++;
    else if(tr[p].key>key)
    {
        insert(tr[p].l,key);
        if(tr[tr[p].l].val>tr[p].val)zig(p);
    }
    else
    {
        insert(tr[p].r,key);
        if(tr[tr[p].r].val>tr[p].val)zag(p);
    }
    pushup(p);
}
void remove(int &p,int key)
{
    if(!p)return;
    if(tr[p].key==key)
    {
        if(tr[p].cnt>1)tr[p].cnt--;
        else if(tr[p].l||tr[p].r)
        {
            if(!tr[p].r||tr[tr[p].l].val>tr[tr[p].r].val)
            {
                zig(p);
                remove(tr[p].r,key);
            }else
            {
                zag(p);
                remove(tr[p].l,key);
            }
        }
        else p=0;
    }
    else if(tr[p].key>key)remove(tr[p].l,key);
    else remove(tr[p].r,key);
    pushup(p);
}
int get_rank_by_key(int p,int key)
{
    if(!p)return 0;
    if(tr[p].key==key)return tr[tr[p].l].size;
    if(tr[p].key>key)return get_rank_by_key(tr[p].l,key);
    return tr[tr[p].l].size+tr[p].cnt+get_rank_by_key(tr[p].r,key);
}
int get_key_by_rank(int p,int rank)
```

```
{
    if(!p)return INF;
    if(tr[tr[p].l].size>=rank)return get_key_by_rank(tr[p].l,rank);
    if(tr[tr[p].l].size+tr[p].cnt>=rank)return tr[p].key;
    return get_key_by_rank(tr[p].r,rank-tr[tr[p].l].size-tr[p].cnt);
}
int get_prev(int p,int key)
{
    if(!p)return -INF;
    if(tr[p].key>=key)return get_prev(tr[p].l,key);
    return max(tr[p].key,get_prev(tr[p].r,key));
}
int get_next(int p,int key)
{
    if(!p)return INF;
    if(tr[p].key<=key)return get_next(tr[p].r,key);
    return  min(tr[p].key,get_next(tr[p].l,key));
}
int main()
{
    build();
    int T;
    cin>>T;
    while(T--)
    {
        int c,x;
        cin>>c>>x;
        if(c==1)insert(root,x);
        else if(c==2)remove(root,x);
        else if(c==3)cout<<get_rank_by_key(root,x)<<endl;
        else if(c==4)cout<<get_key_by_rank(root,x+1)<<endl;
        else if(c==5)cout<<get_prev(root,x)<<endl;
        else cout<<get_next(root,x)<<endl;
    }
    return 0;
}
```

Splay

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

using namespace std;

const int N = 100010;

int n, m;
struct Node
```

```
{
    int s[2], p, v;
    int size, flag;

    void init(int _v, int _p)
    {
        v = _v, p = _p;
        size = 1;
    }
}tr[N];
int root, idx;

void pushup(int x)
{
    tr[x].size = tr[tr[x].s[0]].size + tr[tr[x].s[1]].size + 1;
}

void pushdown(int x)
{
    if (tr[x].flag)
    {
        swap(tr[x].s[0], tr[x].s[1]);
        tr[tr[x].s[0]].flag ^= 1;
        tr[tr[x].s[1]].flag ^= 1;
        tr[x].flag = 0;
    }
}

void rotate(int x)
{
    int y = tr[x].p, z = tr[y].p;
    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
    tr[z].s[tr[z].s[1] == y] = x, tr[x].p = z;
    tr[y].s[k] = tr[x].s[k ^ 1], tr[tr[x].s[k ^ 1]].p = y;
    tr[x].s[k ^ 1] = y, tr[y].p = x;
    pushup(y), pushup(x);
}

void splay(int x, int k)
{
    while (tr[x].p != k)
    {
        int y = tr[x].p, z = tr[y].p;
        if (z != k)
            if ((tr[y].s[1] == x) ^ (tr[z].s[1] == y)) rotate(x);
            else rotate(y);
        rotate(x);
    }
    if (!k) root = x;
```

```
}

void insert(int v)
{
    int u = root, p = 0;
    while (u) p = u, u = tr[u].s[v > tr[u].v];
    u = ++ idx;
    if (p) tr[p].s[v > tr[p].v] = u;
    tr[u].init(v, p);
    splay(u, 0);
}

int get_k(int k)
{
    int u = root;
    while (true)
    {
        pushdown(u);
        if (tr[tr[u].s[0]].size >= k) u = tr[u].s[0];
        else if (tr[tr[u].s[0]].size + 1 == k) return u;
        else k -= tr[tr[u].s[0]].size + 1, u = tr[u].s[1];
    }
    return -1;
}

void output(int u)
{
    pushdown(u);
    if (tr[u].s[0]) output(tr[u].s[0]);
    if (tr[u].v >= 1 && tr[u].v <= n) printf("%d ", tr[u].v);
    if (tr[u].s[1]) output(tr[u].s[1]);
}

int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 0; i <= n + 1; i ++ ) insert(i);
    while (m -- )
    {
        int l, r;
        scanf("%d%d", &l, &r);
        l = get_k(l), r = get_k(r + 2);
        splay(l, 0), splay(r, l);
        tr[tr[r].s[0]].flag ^= 1;
    }
    output(root);
    return 0;
}
```

fhq treap

```
待补充
```

树套树

```
待补充
```

动态树

```
待补充
```

左偏树

三、图论

1.DFS

```
深度优先遍历

int dfs(int u)
{
    st[u] = true; // st[u] 表示点u已经被遍历过

    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!st[j]) dfs(j);
    }
}
```

2.BFS

```
宽度优先遍历

queue<int> q;
st[1] = true; // 表示1号点已经被遍历过
q.push(1);

while (q.size())
{
    int t = q.front();
```

```
    q.pop();

    for (int i = h[t]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!st[j])
        {
            st[j] = true; // 表示点j已经被遍历过
            q.push(j);
        }
    }
}
```

3.拓扑排序

```
bool topsort()
{
    int hh = 0, tt = -1;

    // d[i] 存储点i的入度
    for (int i = 1; i <= n; i ++ )
        if (!d[i])
            q[ ++ tt] = i;

    while (hh <= tt)
    {
        int t = q[hh ++ ];

        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (-- d[j] == 0)
                q[ ++ tt] = j;
        }
    }

    // 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。
    return tt == n - 1;
}
```

4.最短路

dijkstra

```
时间复杂是 O(n2+m)
, n
 表示点数, m
 表示边数

int g[N][N];  // 存储每条边
int dist[N];  // 存储1号点到每个点的最短距离
bool st[N];   // 存储每个点的最短路是否已经确定

// 求1号点到n号点的最短路，如果不存在则返回-1
int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    for (int i = 0; i < n - 1; i ++ )
    {
        int t = -1;     // 在还未确定最短路的点中，寻找距离最小的点
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;

        // 用t更新其他点的距离
        for (int j = 1; j <= n; j ++ )
            dist[j] = min(dist[j], dist[t] + g[t][j]);

        st[t] = true;
    }

    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

堆优化dijkstra

```
时间复杂度 O(mlogn)
, n
 表示点数, m
 表示边数

typedef pair<int, int> PII;

int n;      // 点的数量
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
int dist[N];        // 存储所有点到1号点的距离
bool st[N];     // 存储每个点的最短距离是否已确定

```

```cpp

// 求1号点到n号点的最短距离，如果不存在，则返回-1
int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    priority_queue<PII, vector<PII>, greater<PII>> heap;
    heap.push({0, 1});      // first存储距离，second存储节点编号

    while (heap.size())
    {
        auto t = heap.top();
        heap.pop();

        int ver = t.second, distance = t.first;

        if (st[ver]) continue;
        st[ver] = true;

        for (int i = h[ver]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > distance + w[i])
            {
                dist[j] = distance + w[i];
                heap.push({dist[j], j});
            }
        }
    }

    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

bellman_ford

```cpp
时间复杂度 O(nm)
, n表示点数,
m表示边数

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。

int n, m;       // n表示点数，m表示边数
int dist[N];        // dist[x]存储1到x的最短路距离

struct Edge     // 边，a表示出点，b表示入点，w表示边的权重
{
    int a, b, w;
}edges[M];
```

```

// 求1到n的最短路距离，如果无法从1走到n，则返回-1。
int bellman_ford()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    // 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。
    for (int i = 0; i < n; i ++ )
    {
        for (int j = 0; j < m; j ++ )
        {
            int a = edges[j].a, b = edges[j].b, w = edges[j].w;
            if (dist[b] > dist[a] + w)
                dist[b] = dist[a] + w;
        }
    }

    if (dist[n] > 0x3f3f3f3f / 2) return -1;
    return dist[n];
}
```

spfa

```
时间复杂度 平均情况下 O(m)
, 最坏情况下 O(nm)
, n
 表示点数, m
 表示边数

int n;      // 总点数
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
int dist[N];        // 存储每个点到1号点的最短距离
bool st[N];     // 存储每个点是否在队列中

// 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
int spfa()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    queue<int> q;
    q.push(1);
    st[1] = true;

    while (q.size())
    {
        auto t = q.front();
```

```
        q.pop();

        st[t] = false;

        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                if (!st[j])     // 如果队列中已存在j，则不需要将j重复插入
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }

    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

spfa判断负环

```
.spfa判断图中是否存在负环
时间复杂度是 O(nm)
, n
 表示点数, m
 表示边数

int n;      // 总点数
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
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
bool st[N];     // 存储每个点是否在队列中

// 如果存在负环，则返回true，否则返回false。
bool spfa()
{
    // 不需要初始化dist数组
    // 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。

    queue<int> q;
    for (int i = 1; i <= n; i ++ )
    {
        q.push(i);
        st[i] = true;
```

```
    }

    while (q.size())
    {
        auto t = q.front();
        q.pop();

        st[t] = false;

        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                cnt[j] = cnt[t] + 1;
                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
                if (!st[j])
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }

    return false;
}
```

floyd

```
时间复杂度是 O(n3), n表示点数

初始化:
    for (int i = 1; i <= n; i ++ )
        for (int j = 1; j <= n; j ++ )
            if (i == j) d[i][j] = 0;
            else d[i][j] = INF;

// 算法结束后，d[a][b]表示a到b的最短距离
void floyd()
{
    for (int k = 1; k <= n; k ++ )
        for (int i = 1; i <= n; i ++ )
            for (int j = 1; j <= n; j ++ )
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
```

```
}
```

差分约束

```
待补充
```

5.最小生成树

Prim

```
时间复杂度是 O(n2+m)
, n
 表示点数, m
 表示边数

int n;      // n表示点数
int g[N][N];        // 邻接矩阵，存储所有边
int dist[N];        // 存储其他点到当前最小生成树的距离
bool st[N];     // 存储每个点是否已经在生成树中


// 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
int prim()
{
    memset(dist, 0x3f, sizeof dist);

    int res = 0;
    for (int i = 0; i < n; i ++ )
    {
        int t = -1;
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;

        if (i && dist[t] == INF) return INF;

        if (i) res += dist[t];
        st[t] = true;

        for (int j = 1; j <= n; j ++ ) dist[j] = min(dist[j], g[t][j]);
    }

    return res;
}
```

Kruskal

```
时间复杂度是 O(mlogm)
, n
 表示点数, m
 表示边数

int n, m;       // n是点数, m是边数
int p[N];       // 并查集的父节点数组

struct Edge     // 存储边
{
    int a, b, w;

    bool operator< (const Edge &W)const
    {
        return w < W.w;
    }
}edges[M];

int find(int x)     // 并查集核心操作
{
    if (p[x] != x) p[x] = find(p[x]);
    return p[x];
}

int kruskal()
{
    sort(edges, edges + m);

    for (int i = 1; i <= n; i ++ ) p[i] = i;    // 初始化并查集

    int res = 0, cnt = 0;
    for (int i = 0; i < m; i ++ )
    {
        int a = edges[i].a, b = edges[i].b, w = edges[i].w;

        a = find(a), b = find(b);
        if (a != b)     // 如果两个连通块不连通, 则将这两个连通块合并
        {
            p[a] = b;
            res += w;
            cnt ++ ;
        }
    }

    if (cnt < n - 1) return INF;
    return res;
}
```

朱刘算法

```

```

6.二分图

染色法判定二分图

```
时间复杂度是 O(n+m)
, n
 表示点数, m
 表示边数

int n;      // n表示点数
int h[N], e[M], ne[M], idx;      // 邻接表存储图
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

// 参数:u表示当前节点,c表示当前点的颜色
bool dfs(int u, int c)
{
    color[u] = c;
    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (color[j] == -1)
        {
            if (!dfs(j, !c)) return false;
        }
        else if (color[j] == c) return false;
    }

    return true;
}

bool check()
{
    memset(color, -1, sizeof color);
    bool flag = true;
    for (int i = 1; i <= n; i ++ )
        if (color[i] == -1)
            if (!dfs(i, 0))
            {
                flag = false;
                break;
            }
    return flag;
```

```
}
```

匈牙利算法

```
时间复杂度是 O(nm)
  n表示点数
m表示边数

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
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
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
bool st[N];     // 表示第二个集合中的每个点是否已经被遍历过

bool find(int x)
{
    for (int i = h[x]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!st[j])
        {
            st[j] = true;
            if (match[j] == 0 || find(match[j]))
            {
                match[j] = x;
                return true;
            }
        }
    }

    return false;
}

// 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
int res = 0;
for (int i = 1; i <= n1; i ++ )
{
    memset(st, false, sizeof st);
    if (find(i)) res ++ ;
}
```

7.最近公共祖先

```
待补充
```

8.有向图的强连通分量

```
待补充
```

9.无向图的双连通分量

```
待补充
```

10.欧拉路径和欧拉回路

```
待补充
```

11.网络流

最大流

```
待补充
```

最小割

```
待补充
```

费用流

```
待补充
```

12.2-SAT

```

```

13.prufer编码

```

```

四、字符串

1.kmp

```
// s[]是长文本，p[]是模式串，n是s的长度，m是p的长度
求模式串的Next数组：
for (int i = 2, j = 0; i <= m; i ++ )
{
    while (j && p[i] != p[j + 1]) j = ne[j];
    if (p[i] == p[j + 1]) j ++ ;
    ne[i] = j;
}

// 匹配
for (int i = 1, j = 0; i <= n; i ++ )
{
    while (j && s[i] != p[j + 1]) j = ne[j];
    if (s[i] == p[j + 1]) j ++ ;
    if (j == m)
    {
        j = ne[j];
        // 匹配成功后的逻辑
    }
}
```

2.Trie

```
int son[N][26], cnt[N], idx;
// 0号点既是根节点，又是空节点
// son[][]存储树中每个节点的子节点
// cnt[]存储以每个节点结尾的单词数量

// 插入一个字符串
void insert(char *str)
{
    int p = 0;
    for (int i = 0; str[i]; i ++ )
    {
        int u = str[i] - 'a';
        if (!son[p][u]) son[p][u] = ++ idx;
        p = son[p][u];
    }
    cnt[p] ++ ;
}

// 查询字符串出现的次数
int query(char *str)
{
    int p = 0;
    for (int i = 0; str[i]; i ++ )
```

```
    {
        int u = str[i] - 'a';
        if (!son[p][u]) return 0;
        p = son[p][u];
    }
    return cnt[p];
}
```

3.AC自动机

```
待补充
```

4.后缀数组

```
待补充
```

5.后缀自动机（SAM）

```
待补充
```

6. manacher算法

```
待补充
```

7.回文自动机（PAM）

```
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

```
bool is_prime(int x)
{
    if (x < 2) return false;
    for (int i = 2; i <= x / i; i ++ )
        if (x % i == 0)
            return false;
    return true;
}
```

试除法分解质因数

```cpp
void divide(int x)
{
    for (int i = 2; i <= x / i; i ++ )
        if (x % i == 0)
        {
            int s = 0;
            while (x % i == 0) x /= i, s ++ ;
            cout << i << ' ' << s << endl;
        }
    if (x > 1) cout << x << ' ' << 1 << endl;
    cout << endl;
}
```

线性筛法求素数

```cpp
int primes[N], cnt;     // primes[]存储所有素数
bool st[N];         // st[x]存储x是否被筛掉

void get_primes(int n)
{
    for (int i = 2; i <= n; i ++ )
    {
        if (!st[i]) primes[cnt ++ ] = i;
        for (int j = 0; primes[j] <= n / i; j ++ )
        {
            st[primes[j] * i] = true;
            if (i % primes[j] == 0) break;
        }
    }
}
```

2.约数

试除法求所有约数

```
vector<int> get_divisors(int x)
{
    vector<int> res;
    for (int i = 1; i <= x / i; i ++ )
        if (x % i == 0)
        {
            res.push_back(i);
            if (i != x / i) res.push_back(x / i);
        }
    sort(res.begin(), res.end());
    return res;
}
```

约数个数与约束之和

```
如果 N = p1^c1 * p2^c2 * ... *pk^ck
约数个数: (c1 + 1) * (c2 + 1) * ... * (ck + 1)
约数之和: (p1^0 + p1^1 + ... + p1^c1) * ... * (pk^0 + pk^1 + ... + pk^ck)
```

3.欧拉函数

求欧拉函数

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

    return res;
}
```

筛法求欧拉函数

```
int primes[N], cnt;      // primes[]存储所有素数
int euler[N];            // 存储每个数的欧拉函数
bool st[N];         // st[x]存储x是否被筛掉


void get_eulers(int n)
{
```

```
    euler[1] = 1;
    for (int i = 2; i <= n; i ++ )
    {
        if (!st[i])
        {
            primes[cnt ++ ] = i;
            euler[i] = i - 1;
        }
        for (int j = 0; primes[j] <= n / i; j ++ )
        {
            int t = primes[j] * i;
            st[t] = true;
            if (i % primes[j] == 0)
            {
                euler[t] = euler[i] * primes[j];
                break;
            }
            euler[t] = euler[i] * (primes[j] - 1);
        }
    }
}
```

4.快速幂

```
求 m^k mod p，时间复杂度 O(logk)。

int qmi(int m, int k, int p)
{
    int res = 1 % p, t = m;
    while (k)
    {
        if (k&1) res = res * t % p;
        t = t * t % p;
        k >>= 1;
    }
    return res;
}
```

5.扩展欧几里得算法

欧几里德算法

```
int gcd(int a, int b)
{
    return b ? gcd(b, a % b) : a;
}
```

扩展欧几里得算法

```
// 求x, y，使得ax + by = gcd(a, b)
int exgcd(int a, int b, int &x, int &y)
{
    if (!b)
    {
        x = 1; y = 0;
        return a;
    }
    int d = exgcd(b, a % b, y, x);
    y -= (a/b) * x;
    return d;
}
```

6.中国剩余定理

7.高斯消元

```
// a[N][N]是增广矩阵
int gauss()
{
    int c, r;
    for (c = 0, r = 0; c < n; c ++ )
    {
        int t = r;
        for (int i = r; i < n; i ++ )   // 找到绝对值最大的行
            if (fabs(a[i][c]) > fabs(a[t][c]))
                t = i;

        if (fabs(a[t][c]) < eps) continue;

        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
        for (int i = n; i >= c; i -- ) a[r][i] /= a[r][c];      // 将当前行的首位变成1
        for (int i = r + 1; i < n; i ++ )       // 用当前行将下面所有的列消成0
            if (fabs(a[i][c]) > eps)
                for (int j = n; j >= c; j -- )
                    a[i][j] -= a[r][j] * a[i][c];

        r ++ ;
    }

    if (r < n)
    {
        for (int i = r; i < n; i ++ )
            if (fabs(a[i][n]) > eps)
                return 2; // 无解
```

```
            return 1; // 有无穷多组解
    }

    for (int i = n - 1; i >= 0; i -- )
        for (int j = i + 1; j < n; j ++ )
            a[i][n] -= a[i][j] * a[j][n];

    return 0; // 有唯一解
}
```

8.组合计数

递推法求组合数

```
// c[a][b] 表示从a个苹果中选b个的方案数
for (int i = 0; i < N; i ++ )
    for (int j = 0; j <= i; j ++ )
        if (!j) c[i][j] = 1;
        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

```
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]
如果取模的数是质数，可以用费马小定理求逆元
int qmi(int a, int k, int p)    // 快速幂模板
{
    int res = 1;
    while (k)
    {
        if (k & 1) res = (LL)res * a % p;
        a = (LL)a * a % p;
        k >>= 1;
    }
    return res;
}

// 预处理阶乘的余数和阶乘逆元的余数
fact[0] = infact[0] = 1;
for (int i = 1; i < N; i ++ )
{
    fact[i] = (LL)fact[i - 1] * i % mod;
    infact[i] = (LL)infact[i - 1] * qmi(i, mod - 2, mod) % mod;
}
```

Lucas定理

```
若p是质数，则对于任意整数 1 <= m <= n，有：
    C(n, m) = C(n % p, m % p) * C(n / p, m / p) (mod p)

int qmi(int a, int k, int p)  // 快速幂模板
{
    int res = 1 % p;
    while (k)
    {
        if (k & 1) res = (LL)res * a % p;
        a = (LL)a * a % p;
        k >>= 1;
    }
    return res;
}

int C(int a, int b, int p)  // 通过定理求组合数C(a, b)
{
    if (a < b) return 0;

    LL x = 1, y = 1;  // x是分子，y是分母
    for (int i = a, j = 1; j <= b; i --, j ++ )
    {
        x = (LL)x * i % p;
        y = (LL) y * j % p;
    }

    return x * (LL)qmi(y, p - 2, p) % p;
}

int lucas(LL a, LL b, int p)
{
    if (a < p && b < p) return C(a, b, p);
    return (LL)C(a % p, b % p, p) * lucas(a / p, b / p, p) % p;
}
```

分解质因数法求组合数

```
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：
    1. 筛法求出范围内的所有质数
    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 + ...
    3. 用高精度乘法将所有质因子相乘

```

```cpp
int primes[N], cnt;     // 存储所有质数
int sum[N];     // 存储每个质数的次数
bool st[N];     // 存储每个数是否已被筛掉


void get_primes(int n)      // 线性筛法求素数
{
    for (int i = 2; i <= n; i ++ )
    {
        if (!st[i]) primes[cnt ++ ] = i;
        for (int j = 0; primes[j] <= n / i; j ++ )
        {
            st[primes[j] * i] = true;
            if (i % primes[j] == 0) break;
        }
    }
}


int get(int n, int p)       // 求n! 中的次数
{
    int res = 0;
    while (n)
    {
        res += n / p;
        n /= p;
    }
    return res;
}


vector<int> mul(vector<int> a, int b)       // 高精度乘低精度模板
{
    vector<int> c;
    int t = 0;
    for (int i = 0; i < a.size(); i ++ )
    {
        t += a[i] * b;
        c.push_back(t % 10);
        t /= 10;
    }

    while (t)
    {
        c.push_back(t % 10);
        t /= 10;
    }

    return c;
```

```
}

get_primes(a);  // 预处理范围内的所有质数

for (int i = 0; i < cnt; i ++ )     // 求每个质因数的次数
{
    int p = primes[i];
    sum[i] = get(a, p) - get(b, p) - get(a - b, p);
}

vector<int> res;
res.push_back(1);

for (int i = 0; i < cnt; i ++ )     // 用高精度乘法将所有质因子相乘
    for (int j = 0; j < sum[i]; j ++ )
        res = mul(res, primes[i]);
```

9.卡特兰数

```
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

```
**NIM游戏**
给定N堆物品，第i堆物品有Ai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。

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。
所谓采取最优策略是指，若在某一局面下存在某种行动，使得行动后对面面临必败局面，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，即两人均无失误，都采取最优策略行动时游戏的结果。
NIM博弈不存在平局，只有先手必胜和先手必败两种情况。

定理： NIM博弈先手必胜，当且仅当 A1 ^ A2 ^ … ^ An != 0

**公平组合游戏ICG**

若一个游戏满足：

由两名玩家交替行动；
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；
不能行动的玩家判负；
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mex(S)为求出不属于集合S的最小非负整数的运算，即：
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y1, y2, …, yk，定义SG(x)为x的后继节点y1, y2, …, yk 的SG函数值构成的集合再执行mex(S)运算的结果，即：
SG(x) = mex({SG(y1), SG(y2), …, SG(yk)})
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SG(G) = SG(s)。

有向图游戏的和 —— 模板题 AcWing 893. 集合-Nim游戏

设G1, G2, …, Gm 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Gi，并在Gi上行动一步。G被称为有向图游戏G1, G2, …, Gm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：
SG(G) = SG(G1) ^ SG(G2) ^ … ^ SG(Gm)

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

```

```

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

```

```

10.自适应辛普森积分

```

```