算法学习记录

普通平衡树
威佐夫博弈
斐波那契博弈
区间不同数的个数

算法基础

前缀和&差分

一维：

二维：

- 二维前缀和：
- 二位差分：

树上差分：

数据结构

树状数组

一维树状数组

模版（区间+单点）

```
const int maxn = 500000;

int n, q; // n-数组长度，q-询问次数
vector<int>c(maxn + 5, 0);

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

void update(int pos, int num) { // pos-更新位置，num-更新数据
    for(int i = pos; i <= n; i += lowbit(i)) c[i] += num;
}
void range_update(int l, int r, int num) {
    update(l, num); // 以差分形式更新
    update(r + 1, -num);
}
int getsum(int pos) { // 获得 1-pos 的和
    int sum = 0;
    for(int i = pos; i; i -= lowbit(i)) sum += c[i];
    return sum;
}

void solve() {
    cin >> n >> q;
    vector<int>a(n + 1); // 原始数组
    for(int i = 1; i <= n; ++i) cin >> a[i];

    //+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-//
```

```
    //单点修改，区间查询
    for(int i = 1; i <= n; ++i) update(i, a[i]);

    while(m--) {
        int op; cin >> op;
        if(op == 1) {
            int pos, num; cin >> pos >> num;
            update(pos, num);
        }
        else {
            int l, r; cin >> l >> r;
            cout << getsum(r) - getsum(l - 1) << "\n";
        }
    }
    //+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-//
    //区间修改，单点查询（利用差分）
    for(int i = 1; i <= n; ++i) update(i, a[i] - a[i - 1]); // 以差分形式更新

    while(q--) {
        int op; cin >> op;
        if(op == 1) {
            int l, r, num; cin >> l >> r >> num;
            range_update(l, r, num);
        }
        else {
            int pos; cin >> pos;
            cout << getsum(pos) << "\n";
        }
    }
}
```

模版（区间+区间）

```
const int maxn = 500000;
int n, q; // n-数组长度，q-询问次数
vector<int>ta(maxn + 5, 0);
vector<int>tb(maxn + 5, 0);

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

void update(int pos, int num) { // pos-更新位置，num-更新数据
    for(int i = pos; i <= n; i += lowbit(i)) ta[i] += num, tb[i] += num * (pos -
1);
}
void range_update(int l, int r, int num) {
    update(l, num); // 以差分形式更新
    update(r + 1, -num);
}
int getsum(int pos) { // 获得 1-pos 的和
    int sum = 0;
    for(int i = pos; i; i -= lowbit(i)) sum += pos * ta[i] - tb[i];
    return sum;
}

void solve() {
```

```
    cin >> n >> q;
    vector<int>a(n + 1);
    for(int i = 1; i <= n; ++i) cin >> a[i], update(i, a[i] - a[i - 1]); // 以差分形式更新

    while(q--) {
        int op; cin >> op;
        if(op == 1) {
            int l, r, num; cin >> l >> r >> num;
            range_update(l, r, num);
        }
        else {
            int l, r; cin >> l >> r;
            cout << getsum(r) - getsum(l - 1) << "\n";
        }
    }
}
```

关于区间+区间的证明

> 通过对树状数组的了解，会发现树状数组是一种特殊的前缀和形式。
>
> 思考前缀和与差分的可逆关系，可以得到结论。
>
> 记：$a[i]$ 为原数组，$d[i]$ 为数组 $a[i]$ 的差分数组。$(d[i] = a[i] - a[i-1])$
>
> 则，对于 $a[i]$ 在 p 位置的前缀和：
>
> $$\sum_{i=1}^{p} a[i] = \sum_{i=1}^{p} \sum_{j=i}^{p} d[j]$$
>
> 等式右边：
>
> $$\sum_{i=1}^{p} \sum_{j=i}^{p} d[j] = \sum_{i=1}^{p} d[i] * (p - i + 1) = p * \sum_{i=1}^{p} d[i] - \sum_{i=1}^{p} d[i] * (i - 1)$$
>
> 所以我们可以通过树状数组维护两个差分数组的前缀和，求得原数组的前缀和。
>
> 即：
>
> 维护两个数组 $ta[i]$，$tb[i]$：
>
> - 修改时：$ta[i]$ 像原来一样更新，即 $ta[i] + num$；
>
> $tb[i] = ta[i] * (i - 1)$，那么更新后
> $tb[i] = (ta[i] + num) * (i - 1) = ta[i] * (i - 1) + num * (i - 1)$，所以 $tb[i]$ 更新 $num * (i - 1)$ 即可。
> - 查询时：根据求和公式：$\sum_{i=1}^{p} a[i] = p * \sum_{i=1}^{p} d[i] - \sum_{i=1}^{p} d[i] * (i - 1)$，计算即可。

二维树状数组

> 拓展：算法基础 - 前缀和&差分 - 二维
>
> 在此只给出模版，可类比一维自行证明

模版（区间+单点）

```
const int maxn = 5000, maxm = 5000;

int n, m, q; // n, m-数组大小，q-询问次数
vector<vector<int>>c(maxn + 5, vector<int>(maxm + 5, 0));

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

void update(int x, int y, int num) { // (x, y)-更新位置，num-更新数据
    for(int i = x; i <= n; i += lowbit(i))
        for(int j = y; j <= m; j += lowbit(j)) c[i][j] += num;
}
void range_update(int x1, int y1, int x2, int y2, int num) {
    update(x1, y1, num); // 以差分形式更新，二维差分
    update(x1, y2 + 1, -num);
    update(x2 + 1, y1, -num);
    update(x2 + 1, y2 + 1, num);
}
int getsum(int x, int y) { // 获得 (1, 1)-(x, y) 的和
    int sum = 0;
    for(int i = x; i; i -= lowbit(i))
        for(int j = y; j; j -= lowbit(j)) sum += c[i][j];
    return sum;
}

void solve() {
    cin >> n >> m >> q;
    vector<vector<int>>a(n + 1, vector<int>(m + 1, 0)); // 原始数组
    for(int i = 1; i <= n; ++i)
        for(int j = 1; j <= m; ++j) cin >> a[i][j];

    //+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-//
    //单点修改，区间查询
    for(int i = 1; i <= n; ++i)
        for(int j = 1; j <= m; ++j) update(i, j, a[i][j]);

    while(m--) {
        int op; cin >> op;
        if(op == 1) {
            int x, y, num; cin >> x >> y >> num;
            update(x, y, num);
        }
        else {
            int x1, y1, x2, y2; cin >> x1 >> y1 >> x2 >> y2;
            cout << getsum(x2, y2) - getsum(x1 - 1, y2) - getsum(x2, y1 - 1) +
getsum(x1 - 1, y1 - 1) << "\n"; // 二维前缀和
        }
    }
    //+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-//
    //区间修改，单点查询（利用差分）
    for(int i = 1; i <= n; ++i)
        for(int j = 1; j <= m; ++j) // 以差分形式更新
            update(i, j, a[i][j] - a[i - 1][j] - a[i][j - 1] + a[i - 1][j - 1]);
```

```
    while(q--) {
        int op; cin >> op;
        if(op == 1) {
            int x1, y1, x2, y2, num; cin >> x1 >> y1 >> x2 >> y2 >> num;
            range_update(x1, y1, x2, y2, num);
        }
        else {
            int x, y; cin >> x >> y;
            cout << getsum(x, y) << "\n";
        }
    }
}
```

模版（区间+区间）

> 在二维树状数组进行区间修改，区间查询时，需要 4 个维护数组，具体不在此证明。

```
const int maxn = 5000, maxm = 5000;

int n, m, q; // n, m-数组大小，q-询问次数
vector<vector<int>>ta(maxn + 5, vector<int>(maxm + 5, 0));
vector<vector<int>>tb(maxn + 5, vector<int>(maxm + 5, 0));
vector<vector<int>>tc(maxn + 5, vector<int>(maxm + 5, 0));
vector<vector<int>>td(maxn + 5, vector<int>(maxm + 5, 0));

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

void update(int x, int y, int num) { // (x, y)-更新位置，num-更新数据
    for(int i = x; i <= n; i += lowbit(i)) {
        for(int j = y; j <= m; j += lowbit(j)) {
            ta[i][j] += num;
            tb[i][j] += (x - 1) * num;
            tc[i][j] += (y - 1) * num;
            td[i][j] += (x - 1) * (y - 1) * num;
        }
    }
}
void range_update(int x1, int y1, int x2, int y2, int num) {
    update(x1, y1, num); // 以差分形式更新，二维差分
    update(x1, y2 + 1, -num);
    update(x2 + 1, y1, -num);
    update(x2 + 1, y2 + 1, num);
}
int getsum(int x, int y) { // 获得 (1, 1)-(x, y) 的和
    int sum = 0;
    for(int i = x; i; i -= lowbit(i)) {
        for(int j = y; j; j -= lowbit(j)) {
            sum += ta[i][j] * x * y - tb[i][j] * y - tc[i][j] * x + td[i][j];
        }
    }
    return sum;
}

void solve() {
    cin >> n >> m >> q;
```

```
    vector<vector<int>>a(n + 1, vector<int>(m + 1, 0)); // 原始数组
    for(int i = 1; i <= n; ++i)
        for(int j = 1; j <= m; ++j) // 以差分形式更新
            cin >> a[i][j], update(i, j, a[i][j] - a[i - 1][j] - a[i][j - 1] +
a[i - 1][j - 1]);

    while(q--) {
        int op; cin >> op;
        if(op == 1) {
            int x1, y1, x2, y2, num; cin >> x1 >> y1 >> x2 >> y2 >> num;
            range_update(x1, y1, x2, y2, num);
        }
        else {
            int x, y; cin >> x >> y;
            cout << getsum(x, y) << "\n";
        }
    }
}
```

树上的树状数组

> 拓展：算法基础 - 前缀和&差分 - 树上差分；图论 - 树上问题 - dfs序

线段树

基础模版 - 加法线段树

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
const ll N=1e6+7;
const ll mod=2147483647;
ll n,m;
struct node
{
    ll l,r,sum,lz;
}tree[N];
ll arr[N];
void build(ll i,ll l,ll r,ll arr[])
{
    tree[i].lz=0;//初始化的时候肯定都是0
    tree[i].l=l;
    tree[i].r=r;
    if(l==r)
    {
        tree[i].sum=arr[l];//到达底部单节点才把输入的值赋给你
        return ;
    }
    ll mid=(l+r)/2;
    build(i*2,l,mid,arr);
    build(i*2+1,mid+1,r,arr);
    tree[i].sum=tree[i*2].sum+tree[i*2+1].sum;//树已经全部建完了，再从下往上+++，使得上
层的树都有了值
    return ;
}
inline void push_down(ll i)
```

```
{
    if(tree[i].lz!=0)
    {
        tree[i*2].lz+=tree[i].lz;
        tree[i*2+1].lz+=tree[i].lz;
        ll mid=(tree[i].l+tree[i].r)/2;
        tree[i*2].sum+=tree[i].lz*(mid-tree[i*2].l+1);
        tree[i*2+1].sum+=tree[i].lz*(tree[i*2+1].r-mid);
        tree[i].lz=0;
    }
    return ;
}
inline void add(ll i,ll l,ll r,ll k)
{
    if(tree[i].l>=l&&tree[i].r<=r)
    {
        tree[i].sum+=k*(tree[i].r-tree[i].l+1);
        tree[i].lz+=k;
        return ;
    }
    push_down(i);
    if(tree[i*2].r>=l)
        add(i*2,l,r,k);
    if(tree[i*2+1].l<=r)
        add(i*2+1,l,r,k);
    tree[i].sum=tree[i*2].sum+tree[i*2+1].sum;
    return ;
}
inline ll searchs(ll i,ll l, ll r)
{
    if(tree[i].l>=l&&tree[i].r<=r)
        return tree[i].sum;
    push_down(i);
    ll num=0;
    if(tree[i*2].r>=l)
        num+=searchs(i*2,l,r);
    if(tree[i*2+1].l<=r)
        num+=searchs(i*2+1,l,r);
    return num;
}
int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0),cout.tie(0);
    cin>>n>>m;
    for(int i=1;i<=n;++i)
        cin>>arr[i];
    build(1,1,n,arr);//从根节点开始建树
    for(int i=1;i<=m;++i)
    {
        ll tmp;
        cin>>tmp;
        if(tmp==1)
        {
            ll a,b,c;
            cin>>a>>b>>c;
```

```
            add(1,a,b,c);//每次修改都是从编号为1开始的，因为编号1是树的顶端，往下分叉
        }
        if(tmp==2)
        {
            ll a,b;
            cin>>a>>b;
            printf("%lld\n",searchs(1,a,b));//编号i的话，每次都是从1开始
        }
    }
    return 0;
}
```

乘法线段树

我们将 $lazytage$ 分为两种，分别是加法的 plz 和乘法的 mlz ：

mlz 很简单处理， $pushdown$ 时直接乘父亲的就可以了；

plz 需要把原先的 plz 乘父亲的 mlz 再加上父亲的 plz 。

```
inline void pushdown(long long i){//注意这种级别的数据一定要开long long
    long long k1=tree[i].mlz,k2=tree[i].plz;
    tree[i<<1].sum=(tree[i<<1].sum*k1+k2*(tree[i<<1].r-tree[i<<1].l+1))%p;//
    tree[i<<1|1].sum=(tree[i<<1|1].sum*k1+k2*(tree[i<<1|1].r-
tree[i<<1|1].l+1))%p;
    tree[i<<1].mlz=(tree[i<<1].mlz*k1)%p;
    tree[i<<1|1].mlz=(tree[i<<1|1].mlz*k1)%p;
    tree[i<<1].plz=(tree[i<<1].plz*k1+k2)%p;
    tree[i<<1|1].plz=(tree[i<<1|1].plz*k1+k2)%p;
    tree[i].plz=0;
    tree[i].mlz=1;
    return ;
}
```

根号线段树（除法同理）

```
inline void Sqrt(int i,int l,int r){
    if(tree[i].l>=l && tree[i].r<=r && (tree[i].minn-(long
long)sqrt(tree[i].minn))==(tree[i].maxx-(long long)sqrt(tree[i].maxx))){//如果这个
区间的最大值最小值一样
        long long u=tree[i].minn-(long long)sqrt(tree[i].minn);//计算区间中每个元素
需要减去的
        tree[i].lz+=u;
        tree[i].sum-=(tree[i].r-tree[i].l+1)*u;
        tree[i].minn-=u;
        tree[i].maxx-=u;
            //cout<<"i"<<i<<" "<<tree[i].sum<<endl;
        return ;
    }
    if(tree[i].r<l || tree[i].l>r)  return ;
    push_down(i);
    if(tree[i*2].r>=l)  Sqrt(i*2,l,r);
    if(tree[i*2+1].l<=r)  Sqrt(i*2+1,l,r);
    tree[i].sum=tree[i*2].sum+tree[i*2+1].sum;
    tree[i].minn=min(tree[i*2].minn,tree[i*2+1].minn);//维护最大值和最小值
    tree[i].maxx=max(tree[i*2].maxx,tree[i*2+1].maxx);
```

```
    //cout<<"i"<<i<<" "<<tree[i].sum<<endl;
    return ;
}
```

可持久化线段树

ST 表

解决 **区间可重复贡献问题** 的数据结构，例如「区间最大\最小值」、「区间按位与」、「区间按位或」、「区间 GCD」。

> **可重复贡献问题** 是指对于运算 opt，满足 $x\ opt\ x = x$，则对应的区间询问就是一个可重复贡献问题。另外，opt 还必须满足结合律才能使用 ST 表求解。
>
> 如果分析一下，「可重复贡献问题」一般都带有某种类似 RMQ 的成分。例如「区间按位与」就是每一位取最小值，而「区间 GCD」则是每一个质因数的指数取最小值。

```
#include<iostream>
using namespace std;
int n, q;
const int MAXN = 5e4+5,MAX_L = 20;
int log_2[MAXN];
int stmin[MAXN][MAX_L];
int stmax[MAXN][MAX_L];

int main()
{
    cin >> n >> q;
    log_2[0] = -1;
    for (int i = 1; i <= n; i++)
    {
        cin >> stmax[i][0];
        stmin[i][0] = stmax[i][0];
        log_2[i] = log_2[i >> 1] + 1;
    }
    for (int j = 1; (1 << j) <= n; j++)
    {
        for (int i = 1; i + (1 << j) - 1 <= n; i++)
        {
            stmin[i][j] = min(stmin[i][j - 1], stmin[i + (1 << j - 1)][j - 1]);
            stmax[i][j] = max(stmax[i][j - 1], stmax[i + (1 << j - 1)][j - 1]);
        }
    }
    while(q--)
    {
        int l, r;
        cin >> l >> r;
        int x = log_2[r - l + 1];
        cout << max(stmax[l][x], stmax[r - (1 << x) + 1][x]) - min(stmin[l][x],
stmin[r - (1 << x) + 1][x]) << endl;
    }
}
```

字符串

最长回文子串

```
int findBMstr(string str0) {
    string str;
    str += "$#";
    int n = str0.size();
    for (int i = 0; i < n; i++) {
        str += str0[i];
        str += "#";
    }
    int len = str.size();
    vector<int> p(len + 1);
    int mx = 0, id = 0;
    for (int i = 1; i <= len; i++) {
        if (mx > i)
            p[i] = (p[2 * id - i] < (mx - i) ? p[2 * id - i] : (mx - i));
        else
            p[i] = 1;
        while (str[i - p[i]] == str[i + p[i]])
            p[i]++;
        if (i + p[i] > mx) {
            mx = i + p[i];
            id = i;
        }
    }
    int Max = 0, ii;
    for (int i = 1; i < len; i++) {
        if (p[i] > Max) {
            ii = i;
            Max = p[i];
        }
    }
    Max--;
    return Max; // 返回最长长度
    // string sMax = "";
    // int s = ii - Max;
    // int e = ii + Max;
    // for (int i = s; i <= e; i++) {
    //     if (str[i] != '#')
    //         sMax += str[i];
    // }
    // return sMax; // 返回最长子串
}
```

数组的最小表示法

> 一个数组的循环同构中字典序最小的那个

```
void get_min(int *b)
{
    const int len = 6;  /// 数组长度
    static int a[len<<1];
```

```
    for (int i = 0; i < (len << 1); i++) a[i] = b[i % len];//    复制一份 b在b的后面，把这个结果赋值给 a
    int i = 0, j = 1, k;
    while (i < len && j < len) {
        // k 表示的是当前比对的第几位
        for (k = 0; k < len && a[i + k] == a[j + k]; k++);  // 注意这里只移动指针
        if (k == len) break;

        // 上面的小于下面的
        if (a[i + k] > a[j + k]) {
            i += k + 1;
            if (i == j) i++;
        }
        else {
            j += k + 1;
            if (i == j) j++;
        }
    }
    k = min(i, j);
    for (int i = 0; i < 6; i++)b[i] = a[i + k];
}
```

双哈希

```
struct DoubleHash {
    using i64 = long long;
    const i64 Base1 = 29, MOD1 = 1e9 + 7;
    const i64 Base2 = 131, MOD2 = 1e9 + 9;
    vector<i64> ha1, ha2, pow1, pow2;
    vector<i64> rha1, rha2;
    int len;
    DoubleHash() {}
    DoubleHash(string &s) {
        init(s);
    }
    void init(string &s) {
        len = s.size();
        ha1.resize(len + 1), ha2.resize(len + 1);
        pow1.resize(len + 1), pow2.resize(len + 1);
        rha1.resize(len + 1), rha2.resize(len + 1);
        s = " " + s;
        pow1[0] = pow2[0] = 1;
        for (int i = 1; i <= len; i++) {
            pow1[i] = pow1[i - 1] * Base1 % MOD1;
            pow2[i] = pow2[i - 1] * Base2 % MOD2;
        }
        for (int i = 1; i <= len; i++) {
            ha1[i] = (ha1[i - 1] * Base1 + s[i]) % MOD1;
            ha2[i] = (ha2[i - 1] * Base2 + s[i]) % MOD2;
            rha1[i] = (rha1[i - 1] * Base1 + s[len - i + 1]) % MOD1;
            rha2[i] = (rha2[i - 1] * Base2 + s[len - i + 1]) % MOD2;
        }
    }
    pair<i64, i64> get(int l, int r) {
```

```
        i64 res1 = ((ha1[r] - ha1[l - 1] * pow1[r - l + 1]) % MOD1 + MOD1) %
MOD1;
        i64 res2 = ((ha2[r] - ha2[l - 1] * pow2[r - l + 1]) % MOD2 + MOD2) %
MOD2;
        return {res1, res2};
    }
    //反哈希
    pair<i64, i64> get_rhash(int l, int r) {
        i64 res1 = ((rha1[len - l + 1] - rha1[len - r] * pow1[r - l + 1]) % MOD1
+ MOD1) % MOD1;
        i64 res2 = ((rha2[len - l + 1] - rha2[len - r] * pow2[r - l + 1]) % MOD2
+ MOD2) % MOD2;
        return {res1, res2};
    }
    //判断s[l, r]是否为回文串
    bool is_palindrome(int l, int r) {
        return get(l, r) == get_rhash(l, r);
    }
    pair<i64, i64> add(pair<i64, i64> aa, pair<i64, i64> bb) {
        i64 res1 = (aa.first + bb.first) % MOD1;
        i64 res2 = (aa.second + bb.second) % MOD2;
        return {res1, res2};
    }
    //aa *= Base的k次方
    pair<i64, i64> mul(pair<i64, i64> aa, i64 kk) {
        i64 res1 = aa.first * pow1[kk] % MOD1;
        i64 res2 = aa.second * pow2[kk] % MOD2;
        return {res1, res2};
    }
    //拼接字符串 r1 < l2  s = s1 + s2
    pair<i64, i64> link(int l1, int r1, int l2, int r2) {
        return add(mul(get(l2, r2), r1 - l1 + 1), get(l1, r1));
    }
};
```

图论

链式前向星

```
vector

// head[u] 和 cnt 的初始值都为 -1
void add(int u, int v) {
  nxt[++cnt] = head[u];  // 当前边的后继
  head[u] = cnt;         // 起点 u 的第一条边
  to[cnt] = v;           // 当前边的终点
}

// 遍历 u 的出边
for (int i = head[u]; ~i; i = nxt[i]) {  // ~i 表示 i != -1
  int v = to[i];
}
```

树上问题

dfs 序列

欧拉序列

树链剖分

最近公共祖先

Tarjan 算法（离线）

倍增算法

```
struct Edge {
    int to, w, next;
};
// n-点数，q-询问数，s-根节点，lg-最大倍增
int n, q, s, lg;
// 链式前向星
vector<int>head;
vector<Edge>edge;
// lca
vector<vector<int>>fa;
vector<int>deep;

void init() { // 输入范围后，初始化
    head.clear();
    edge.clear();
    fa.clear();
    deep.clear();

    head.resize(n + 1, -1);
    fa.resize(n + 1, vector<int>(lg + 1));
    deep.resize(n + 1);
}

void add_edge(int u, int v, int w){ // 添加 从 u 到 v 权为 w 的边
    edge.push_back({v, w, head[u]});
    head[u] = edge.size() - 1;
}

void get_deep(int now, int father) { // 预处理树上节点深度，祖先节点

    deep[now] = deep[father] + 1;
    fa[now][0] = father;

    for(int i = 1; (1 << i) <= deep[now]; ++i) {
        fa[now][i] = fa[fa[now][i - 1]][i - 1]; // 倍增法求 lca 关键所在
    }

    for(int i = head[now]; i != -1; i = edge[i].next) {
        if(edge[i].to == father) continue;
        get_deep(edge[i].to, now);
    }
}
```

```cpp
int lca(int u, int v) {

    if(deep[u] != deep[v]) { // 统一高度
        if(deep[u] > deep[v]) swap(u, v);

        int d = deep[v] - deep[u];
        for(int j = 0; j <= lg; ++j) {
            if((1 << j) & d) v = fa[v][j];
        }
    }

    if(u == v) return u;

    for(int j = lg; j >= 0; --j) {
        if(fa[u][j] == fa[v][j]) continue;
        else u = fa[u][j], v = fa[v][j];
    }

    return fa[u][0];
}

void solve() {

    cin >> n >> q >> s;
    lg = log2(n);

    init();

    for(int i = 1; i <= n - 1; ++i) {
        int u, v; cin >> u >> v;
        add_edge(u, v, 1);
        add_edge(v, u, 1);
    }

    get_deep(s, s);

    for(int i = 1; i <= q; ++i) {
        int u, v;
        cin >> u >> v;
        cout << lca(u, v) << "\n";
    }
}
```

欧拉序列转化为 RMQ 问题

> RMQ 问题：Range Maximum/Minimum Query 区间最大（最小）值

详见数据结构 - ST表。

拓扑排序

```cpp
int n, m;
vector<int> G[MAXN];
int in[MAXN];  // 存储每个结点的入度

bool toposort() {
```

```
  vector<int> L;
  queue<int> S;
  for (int i = 1; i <= n; i++)
    if (in[i] == 0) S.push(i);
  while (!S.empty()) {
    int u = S.front();
    S.pop();
    L.push_back(u);
    for (auto v : G[u]) {
      if (--in[v] == 0) {
        S.push(v);
      }
    }
  }
  if (L.size() == n) {
    for (auto i : L) cout << i << ' ';
    return true;
  }
  return false;
}
```

最短路

弗洛伊德算法

```
for (k = 1; k <= n; k++) {
    for (x = 1; x <= n; x++) {
        for (y = 1; y <= n; y++) {
            f[x][y] = min(f[x][y], f[x][k] + f[k][y]);
        }
    }
}
```

迪杰斯塔拉

```
struct edge {
    int v, w;
};
struct node {
    int dis, u;

    bool operator>(const node& a) const { return dis > a.dis; }
};

vector<edge> e[MAXN];
int dis[MAXN], vis[MAXN];
priority_queue<node, vector<node>, greater<node>> q;

void dijkstra(int n, int s) {
    memset(dis, 0x3f, (n + 1) * sizeof(int));
    dis[s] = 0;
    q.push({0, s});
    while (!q.empty()) {
        int u = q.top().u;
        q.pop();
```

```
        if (vis[u])
            continue;
        vis[u] = 1;
        for (auto ed : e[u]) {
            int v = ed.v, w = ed.w;
            if (dis[v] > dis[u] + w) {
                dis[v] = dis[u] + w;
                q.push({dis[v], v});
            }
        }
    }
}
```

最小生成树

克鲁斯卡尔算法

```
struct node{    //存储边及边权
    int u,v,dis;
}e[M];
bool cmp(node x,node y){    //sort的比较函数
    return x.dis < y.dis;
}
int father[N];  //存放每个节点的父亲节点的father数组
int findfather(int v){  //查找节点的父亲节点
    if(father[v] == v)  return v;
    else{
        int F = findfather(father[v]);
        father[v] = F;
        return F;
    }
}
int Kruskal(int n,int m){
    int ans = 0,num_edge = 0;
    //ans为所求最小生成树的边权之和，num_edge为当前生成树的边数
    for(int i = 0;i< n;i++){        //father[]初始化，注意节点的范围（以0 - n为例）
        father[i] = i;
    }
    sort(e,e+m,cmp);    //对边的权值按照从小到大排序
    for(int i = 0;i< m;i++){
        int fu = findfather(e[i].u);
        int fv = findfather(e[i].v);
        if(fu != fv){   //如果两个节点的父亲节点不同，则将次边加入最小生成树
            ans+= e]i].dis; //加上边权
            father[fu] = fv;
            num_edge++;
            if(num_edge == n-1) //如果最小生成树的边数目达到节点数减一，就退出
                break;
        }
    }
    return ans;
}
```

联通性相关

强联通分量 + 缩点

```
int dfn[N], low[N], dfncnt, s[N], in_stack[N], tp;
int scc[N], sc;  // 结点 i 所在 SCC 的编号
int sz[N];       // 强连通 i 的大小

void tarjan(int u) {
    low[u] = dfn[u] = ++dfncnt, s[++tp] = u, in_stack[u] = 1;
    for (int i = h[u]; i; i = e[i].nex) {
        const int& v = e[i].t;
        if (!dfn[v]) {
            tarjan(v);
            low[u] = min(low[u], low[v]);
        } else if (in_stack[v]) {
            low[u] = min(low[u], dfn[v]);
        }
    }
    if (dfn[u] == low[u]) {
        ++sc;
        while (s[tp] != u) {
            scc[s[tp]] = sc;
            sz[sc]++;
            in_stack[s[tp]] = 0;
            --tp;
        }
        scc[s[tp]] = sc;
        sz[sc]++;
        in_stack[s[tp]] = 0;
        --tp;
    }
}

// 缩点代码
// for (int i = 1; i <= m; i++)                //循环每一条边
//     if (scc[from[i]] != scc[to[i]])     //如果这条边的出发点和终止点不在同一个强连通
分量中
//         add(scc[from[i]], scc[to[i]]);    //就连一条边
```

割点

```
int dfsn[MAXN], low[MAXN], cnt;
vector<int> cut; // 存储所有割点
void tarjan(int p, bool root = true)
{
    int tot = 0;
    low[p] = dfsn[p] = ++cnt;
    for (auto q : edges[p])
    {
        if (!dfsn[q])
        {
            tarjan(q, false);
            low[p] = min(low[p], low[q]);
            tot += (low[q] >= dfsn[p]); // 统计满足low[q] >= dfsn[p]的子节点数目
```

```
        }
        else
            low[p] = min(low[p], dfsn[q]);
    }
    if (tot > root) // 如果是根，tot需要大于1；否则只需大于0
        cut.push_back(p);
}
```

割边（桥）

```
vector<pair<int, int>> bridges;
int dfsn[MAXN], low[MAXN], fa[MAXN], cnt;
void tarjan(int p)
{
    low[p] = dfsn[p] = ++cnt;
    for (auto to : edges[p])
    {
        if (!dfsn[to])
        {
            fa[to] = p; // 记录父节点
            tarjan(to);
            low[p] = min(low[p], low[to]);
            if (low[to] > dfsn[p])
                bridges.emplace_back(p, to);
        }
        else if (fa[p] != to) // 排除父节点
            low[p] = min(low[p], dfsn[to]);
    }
}
```

数学

零碎知识点

快速幂

```
int ksm(int base, int power, int p) {
    int ans = 1;
    while (power) {
        if (power & 1) ans = ans * base % p;
        power >>= 1;
        base = base * base % p;
    }
    return ans;
}
```

唯一质因子分解定理

- 合数 N 仅能以一种方式，写成如下的乘积的形式： $N = p_1^{e_1} p_2^{e_2} \dots p_n^{e_n}$ ，其中 p_i 为质数，$p_1 < p_2 < \dots < p_n$ ，且 e_i 为正整数。

> 推广：
>
> N 的因子个数为： $(1+e_1)(1+e_2)\dots(1+e_n)$

> N 的所有因子的和为：
> $(1+p_1+p_1^2+\cdots+p_1^{e_1})(1+p_2+p_2^2+\cdots+p_2^{e_2})\ldots(1+p_n+p_n^2+\cdots+p_n^{e_n})$

线性求所有因子的和

```
const ll N = 1e7 + 10;

ll primes[N], mul[N], d[N], cnt = 0;
bool st[N];

void init() { // d[x], 即为 x 的所有因子的和
    cnt = 0;
    st[0] = st[1] = 1;
    mul[1] = 1, d[1] = 1;
    for (ll i = 2; i < N; ++i) {
        if (!st[i]) {
            primes[++cnt] = i;
            mul[i] = i + 1;
            d[i] = i + 1;
        }
        for (ll j = 1; primes[j] * i < N; ++j) {
            st[primes[j] * i] = 1;

            if (i % primes[j] == 0) {
                mul[i * primes[j]] = mul[i] * primes[j] + 1;
                d[i * primes[j]] = d[i] / mul[i] * mul[i * primes[j]];
                break;
            } else {
                mul[i * primes[j]] = primes[j] + 1;
                d[i * primes[j]] = d[i] * d[primes[j]];
            }
        }
    }
}
```

线性基

```
const int MN = 63;
ll a[65], tmp[65];
bool flag;
void ins(ll x) {  // 构造
    for (int i = MN; ~i; i--)
        if (x & (1ll << i))
            if (!a[i]) {
                a[i] = x;
                return;
            } else
                x ^= a[i];
    flag = true;
}
bool check(ll x) {  // 检查能否异或出 x
    for (int i = MN; ~i; i--)
        if (x & (1ll << i))
            if (!a[i])
                return false;
```

```
            else
                x ^= a[i];
    return true;
}
ll qmax(ll res = 0) {  // 异或最大值
    for (int i = MN; ~i; i--)
        res = max(res, res ^ a[i]);
    return res;
}
ll qmin() {  // 异或最小值
    if (flag)
        return 0;
    for (int i = 0; i <= MN; i++)
        if (a[i])
            return a[i];
}
ll query(ll k) {  // 查询第 k 小值
    ll res = 0;
    int cnt = 0;
    k -= flag;
    if (!k)
        return 0;
    for (int i = 0; i <= MN; i++) {
        for (int j = i - 1; ~j; j--)
            if (a[i] & (1ll << j))
                a[i] ^= a[j];
        if (a[i])
            tmp[cnt++] = a[i];
    }
    if (k >= (1ll << cnt))
        return -1;
    for (int i = 0; i < cnt; i++)
        if (k & (1ll << i))
            res ^= tmp[i];
    return res;
}
```

数论

质数（素数）

欧拉筛

时间复杂度：$O(n)$

```cpp
const int maxn = 100000000;
vector<int> prime;
vector<bool> isprime(maxn + 10, 1);
void euler(){
    isprime[1] = 0;
    for(int i = 2; i < maxn; i++){
        if(isprime[i]) prime.push_back(i);
        for(int j = 0; j < prime.size() && i * prime[j] < maxn; j++) {
            isprime[i * prime[j]] = 0;
            if (i % prime[j] == 0) break;
        }
    }
}
```

Pollard-Rho 附带 Miller Rabin

> 对于每个数字检验是否是质数，是质数就输出 $Prime$；如果不是质数，输出它最大的质因子。

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

int t;
long long max_factor, n;

long long gcd(long long a, long long b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}
long long quick_pow(long long x, long long p, long long mod) {  // 快速幂
    long long ans = 1;
    while (p) {
        if (p & 1) ans = (__int128)ans * x % mod;
        x = (__int128)x * x % mod;
        p >>= 1;
    }
    return ans;
}
bool Miller_Rabin(long long p) {  // 判断素数
    if (p < 2) return 0;
    if (p == 2) return 1;
    if (p == 3) return 1;
    long long d = p - 1, r = 0;
    while (!(d & 1)) ++r, d >>= 1;  // 将d处理为奇数
    for (long long k = 0; k < 10; ++k) {
        long long a = rand() % (p - 2) + 2;
        long long x = quick_pow(a, d, p);
        if (x == 1 || x == p - 1) continue;
        for (int i = 0; i < r - 1; ++i) {
            x = (__int128)x * x % p;
            if (x == p - 1) break;
        }
        if (x != p - 1) return 0;
    }
    return 1;
}
```

```
long long Pollard_Rho(long long x) {
  long long s = 0, t = 0;
  long long c = (long long)rand() % (x - 1) + 1;
  int step = 0, goal = 1;
  long long val = 1;
  for (goal = 1;; goal *= 2, s = t, val = 1) {  // 倍增优化
    for (step = 1; step <= goal; ++step) {
      t = ((__int128)t * t + c) % x;
      val = (__int128)val * abs(t - s) % x;
      if ((step % 127) == 0) {
        long long d = gcd(val, x);
        if (d > 1) return d;
      }
    }
    long long d = gcd(val, x);
    if (d > 1) return d;
  }
}
void fac(long long x) {
    if (x <= max_factor || x < 2) return;
    if (Miller_Rabin(x)) {              // 如果x为质数
        max_factor = max(max_factor, x);  // 更新答案
        return;
    }
    long long p = x;
    while (p >= x) p = Pollard_Rho(x);  // 使用该算法
    while ((x % p) == 0) x /= p;
    fac(x), fac(p);  // 继续向下分解x和p
}
int main() {
    cin >> t;
    while (t--) {
       srand((unsigned)time(NULL));
       max_factor = 0;
      cin >> n;
      fac(n);
      if (max_factor == n) cout << "Prime\n"; // 最大的质因数即自己
      else cout << max_factor << "\n";
    }
    return 0;
}
```

逆元

求法

费马小定理

求单个数逆元的时间复杂度： $O(log(n))$

```
//由费马小定理可得： a * a^{-1} == 1(mod p)
const long long mod = 1000000000 + 7;
long long ksm(long long a, long long b) {
  long long ans = 1;
  a = (a % mod + mod) % mod;
  for (; b; b >>= 1) {
```

```
    if (b & 1) ans = (a * ans) % mod;
    a = (a * a) % mod;
  }
  return ans;
}
long long inv(long long n) {
    return ksm(n, mod - 2);
}
```

exgcd求逆元

适用于 n 个数不多，但是 mod 很大的时候， mod 不需要为质数。

时间复杂度： $O(log(n))$

```
int mod = 1000000007;
int exgcd(int a, int b, int &x, int &y){ //扩展欧几里得算法
    if(a == 0 && b == 0) return -1;
    if(b == 0) {
        x = 1,y = 0;
        return a;
    }
    int res = exgcd(b, a % b, y, x);
    y -= a / b * x;
    return res;
}
int inv(int a){ // 求a在mod下的逆元，不存在逆元返回-1
    int x, y;
    int ans = exgcd(a, mod, x, y);
    return (ans == 1 ? (x % mod + mod) % mod : -1);
}
```

欧拉函数求逆元

mod 为质数时才可用。

时间复杂度： $O(log(mod))$

```
int mod = 1000000000 + 7;
int ksm(int x, int n){
    int res = 1;
    while(n){
        if(n & 1) res = res * x % mod;
    x = x * x % mod;
    n >>= 1;
    }
    return res;
}
int inv(int a, int mod){
    return ksm(a, mod - 2);
}
```

线性逆元预处理

时间复杂度：$O(n)$

```
int maxn = 1000000000, mod = 1000000007;
int inv[maxn];
void init_inv(){
    inv[1] = 1;
    for(int i = 2; i < N; i++)
    inv[i] = (mod - mod / i) * inv[mod % i] % mod;
}
```

组合数学

排列组合基础

排列数

- 计算公式：$A_n^m = \dfrac{n!}{(n-m)!}$

组合数

- 计算公式：$C_n^m = \dfrac{n!}{m!(n-m)!}$

> 组合数更常用的符号为 $\binom{n}{m}$，读作“n 选 m”，$\binom{n}{m} = C_n^m$。

```
// 逆元求组合数模版
const ll maxn = 1e5 + 5;
const ll mod = 998244353;
ll inv[maxn], fac[maxn];  //分别表示阶乘的逆元和阶乘
ll quickPow(ll a, ll b) {
    ll ans = 1;
    while(b){
        if(b & 1) ans = (ans * a) % mod;
        b >>= 1;
        a=(a * a) % mod;
    }
    return ans;
}
void init() {
    fac[0] = 1;
    for(int i = 1; i < maxn; i++) fac[i] = fac[i - 1] * i % mod; //求阶乘
    inv[maxn - 1] = quickPow(fac[maxn - 1], mod - 2);
    for(int i = maxn - 2; i >= 0; i--) inv[i] = inv[i + 1] * (i + 1) % mod; //求阶
乘的逆元
}
ll C(ll n, ll m){
    if(m > n) return 0;
    if(m == 0) return 1;
    return fac[n] * inv[m] % mod * inv[n - m] % mod;
}
```

组合数性质

1. $\binom{n}{m}=\binom{n}{n-m}$。对称性。
2. $\binom{n}{k}=\frac{n}{k}\binom{n-1}{k-1}$。由定义推得递推式。
3. $\binom{n}{m}=\binom{n-1}{m}+\binom{n-1}{m-1}$。组合数的递推式，杨辉三角的公式表达。
4. $\sum\limits_{i=0}^{n}\binom{n}{i}=2^n$。
5. $\sum\limits_{i=0}^{n}(-1)^i\binom{n}{i}=[n=0]$。
6. $\sum\limits_{i=0}^{m}\binom{n}{i}\binom{m}{m-i}=\binom{n+m}{m}$。$n\geq m$ ，拆解组合数的式子。
7. $\sum\limits_{i=0}^{n}\binom{n}{i}^2=\binom{2n}{n}$。$(6)$ 的特殊情况，取 $n=m$。
8. $\sum\limits_{i=0}^{n}i\binom{n}{i}=n2^{n-1}$。带权和的式子。
9. $\sum\limits_{i=0}^{n}i^2\binom{n}{i}=n(n+1)2^{n-2}$。带权和的式子，与 (8) 类似。
10. $\sum\limits_{l=0}^{n}\binom{l}{k}=\binom{l+1}{k+1}$。考虑集合 $S=\{a_1,a_2,\ldots,a_{n+1}\}$ 的 $(k+1)$ 子集数可以得证，在恒等式证明中比较常用。
11. $\binom{n}{r}\binom{r}{k}=\binom{n}{k}\binom{n-k}{r-k}$。可通过定义证明。
12. $\sum\limits_{i=0}^{n}\binom{n-i}{i}=F_{n+1}$。其中 F 为斐波那契数列。

前置知识

斯特林数（Stirling Number）

> 第二类斯特林数比第一类斯特林数常用的多

- **第二类斯特林数**（斯特林子集数） $\left\{{n\atop k}\right\}$，也可记做 $S_2(n,k)$，表示将 n 个两两不同的元素，划分为 k 个互不区分的非空子集的方案数。
  - 递推式：$S_2(n,k)=S_2(n-1,k-1)+k\cdot S_2(n-1,k)$，边界：$S_2(n,0)=[n=0]$。

    > 对于想要新增的第 n 个元素：
    >
    > 如果把这个元素放在**独立的**一个子集里面，那么就是 $S_2(n-1,k-1)$。
    > 如果把这个元素放在**现有的**一个子集里面，也就是在 $S_2(n-1,k)$ 的情况下，找一个子集插入一个元素，因为有 k 个子集所以要乘 k。

  - 通项式：$S_2(n,k)=\frac{1}{k!}\sum\limits_{i=0}^{k}(-1)^iC_m^i(k-i)^n$。

    > 证明略。

  - 一些性质：

1. $S_2(0,0)=1$
2. $S_2(n,0)=0,n>0$
3. $S_2(n,n)=1$
4. $S_2(n,2)=S_2(n-1,1)+2*S_2(n-1,2)=1+2*S_2(n-1,2)=2^{n-1}-1$
5. $S_2(n,3)=\frac{1}{2}(3^{n-1}+1)-2^{n-1}$
6. $S_2(n,n-1)=C_n^2$
7. $S_2(n,n-2)=C_n^3+3C_n^4$

    > 简单巧妙的证明：我们分成两种情况，把 n 个不同的元素分成 $n-2$ 个集合有两种情况，分别是有一个集合有三个元素和有两个集合有两个元素。对于前者，我们选出三个元素为一个集合，其他的各成一个集合，这种情况的方案数就是 C_n^3 。对于后者，先选出四个元素来，考虑怎么分配。当其中一个元素选定另一个元素形成同一个集合时，这种情况就确定了，所以是 $3C_n^4$ 。加法原理计算和即得证。

8. $S_2(n,n-3)=C_n^4+10C_n^5+15C_n^6$

    > 同理 7.

- **第一类斯特林数**（斯特林轮换数） $\begin{bmatrix}n\\k\end{bmatrix}$ ，也可记做 $S_1(n,k)$ ，表示将 n 个两两不同的元素，划分为 k 个互不区分的非空轮换的方案数。

  > 一个轮换就是一个首尾相接的环形排列。我们可以写出一个轮换 $[A,B,C,D]$ ，并且我们认为 $[A,B,C,D]=[D,A,B,C]=[C,D,A,B]=[B,C,D,A]$ ，即，两个可以通过旋转而互相得到的轮换是等价的。

  > 更清晰的理解，第一类斯特林数表示把 n 个不同元素构成 m 个圆的排列方案数。

  - 递推式：$S_1(n,k)=S_1(n-1,k-1)+(n-1)\cdot S_1(n-1,k)$ ，边界：$S_1(n,0)=[n=0]$ 。

    > 对于想要新增的第 n 个元素：
    >
    > 如果把这个元素放在**独立的**一个圆里面，那么就有 $S_1(n-1,k-1)$ 种方案。
    > 如果把这个元素放在**现有的**一个圆里面，也就是在 $S_1(n-1,k)$ 的情况下，这个元素可以放在前 $(n-1)$ 个元素中任意一个的前面，所以要乘 $(n-1)$ 。

  - 通项公式：第一类斯特林数没有实用的通项公式。
  - 一些性质：
    1. $S_1(0,0)=1$
    2. $S_1(n,0)=0$
    3. $S_1(n,1)=(n-1)!$
    4. $S_1(n,n-1)=C_n^2$
    5. $S_1(n,2)=(n-1)!\sum\limits_{i=1}^{n-1}\frac{1}{i}$
    6. $S_1(n,n-2)=2C_n^3+3C_n^4$
    7. $\sum\limits_{k=0}^{n}S_1(n,k)=n!$

贝尔数

- 定义：B_n 是基数为 n 的集合的划分方法的数目。集合 S 的一个划分是定义为 S 的两两不相交的非空子集的族，它们的并是 S 。

> 例如 $B_3 = 5$ ，是因为大小为 3 的集合 $\{a, b, c\}$ 有一下 5 种划分办法：
> $$\{\{a\},\{b\},\{c\}\}$$
> $$\{\{a,b\},\{c\}\}$$
> $$\{\{a,c\},\{b\}\}$$
> $$\{\{b,c\},\{a\}\}$$
> $$\{\{a,b,c\}\}$$

- 递推公式：$B_{n+1} = \sum_{k=0}^{n} C_n^k B_k$

> 假设， B_{n+1} 是含有 $n+1$ 个元素集合的划分个数。
> 先单独拿出一个一个元素：
> 这个元素单独分为一类，剩下 n 个元素，有 $C_n^n * B_n$ 种方案。
> 这个元素和某 1 个单独元素分成一类，有 $C_n^{n-1} * B_n$ 种方案。
> 这个元素和某 2 个单独元素分成一类，有 $C_n^{n-2} * B_n$ 种方案。
>
> ……
>
> 求和可得。

- 与第二类斯特林数的关系：$B_n = \sum_{k=0}^{n} S_2(n,k)$

错位排列

- **定义**：如果定义全排列 n ~ 1 ，那么一个排列满足任意的 i 都满足 $a[i] \neq i$ ，称之为错位排列。定义集合元素个数为 n 的错位排列个数为 D_n
- **递推公式**：$D_n = (n-1)(D_{n-1} + D_{n-2})$ ，初始化 $D_1 = 0 D_2 = 1$ 。

> 假设 $n \geq 3$ ，考虑 $\{12 \ldots n\}$ 的 D_n 个错位排列
> 来看看第一个位置的情况：
> 它可以是除 1 以外的任何数字，那么一共有 $n-1$ 种情况并且每一种情况所产生的排列数都应该相同，我们设为 d_n
> 也就是说 $D_n = (n-1)d_n$
> 现在来看看 d_n ：
> 因为第 1 个位置可以是除 1 以外的任何数，并且产生排列数相同。
> 出于方便，我们假设第 1 个位置的数是 2 ：
> 确定了第一个位置之后我们发现只需要讨论第二个位置是不是 1 这是一个特殊的点。
>
> 如果第 2 个位置是 1 ，那接下来的 $n-2$ 个位置等价于 n 个元素中有两个元素调换了位置。
> 那我们可以把他们踢出去，他们已经不影响问题了。
> 而剩下的元素将继续进行错位排列，也就是 D_{n-2} 。
>
> 如果第 2 个位置不是 1 ，这时候来重新陈述一下问题：
> 第 2 个位置不能是 1 ，第 3 个位置不能是 3 ，第 4 个位置不能是 4 ，……，第 n 个位置不能是 n
> 这个问题是不是似曾相识?
> 是的，又是一个错排，这个错排只少了一个位置，所以他是 $D_n - 1$ 。
>
> 得到：$d_n = D_{n-1} + D_{n-2}$
> 联立之前的式子就是 $D_n = (n-1)(D_{n-1} + D_{n-2})$

盒子与球

- **问题描述**：想将个 n 球放入 m 个盒子中，有几种放入方法？（将球是否相同，盒子是否相同，是否可以有空盒分为八种情况）
- **具体解决**：
  1. 球相同，盒子不同，不可空盒：
     - 挡板法，相当于将 n 个球分成 m 组，相当于在 $n-1$ 个空隙中插入 $m-1$ 块板子。
     - 结论：C_{n-1}^{m-1}
  2. 球相同，盒子不同，可以空盒：
     - 预先在每个盒子中放入 1 个球，总共放入 m 个球。现在等价于 (1) 的问题。
     - 结论：C_{n+m-1}^{m-1}
  3. 球相同，盒子相同，不可空盒：
     - 动态规划。
     - 假设 $f[n][m]$ 为 n 个球放到 m 个盒子里的方案数。

       如果 $n<m$，此时 m 个盒子必然装不满，可得 $f[n][m]=f[n][n]$。

       如果 $n \geq m$，此时可以选择将盒子放满或者不放满：

       （1）如果没放满，就减掉一个盒子，此时 $f[i][j]=f[i][j-1]$。

       （2）如果放满了，那就在每个盒子里放一个球，此时 $f[i][j]=f[i-j][j]$。

       综上，转移方程为：$f[i][j]=f[i][j-1]+f[i-j][j]$。

       得到转移方程之后考虑边界条件：

       如果没有球或者只有一个盒子，此时方案数为 1，即 $f[0][j]=f[i][1]=1$。
  4. 球相同，盒子相同，不可空盒：
     - 类比问题 (3) 的情况，此时问题的答案为 $f[n-m][m]$ 的值。
  5. 球不同，盒子不同，可以空盒：
     - 对于每一个球，我们可以放在任意一个位置，也就是说每一个球都有 m 种取法。故一共有 m^n 种方案数。
  6. 球不同，盒子相同，可以空盒：

     > 拓展：数学 - 组合数学 - 前置知识 - 斯特林数

     - 结论：第二类斯特林数：$S_2(n,m)$
  7. 球不同，盒子不同，不可空盒：
     - 与问题 (6) 基本一致，但因为盒子两两不同，所以需要乘 $m!$。
     - 结论：$S_2(n,m)*m!$
  8. 球不同，盒子相同，可以空盒：

     > 拓展：数学 - 组合数学 - 前置知识 - 斯特林数，贝尔数

     - 从斯特林数角度：
       - 只需要枚举非空盒子的数量即可
       - 结论：$\sum_{i=1}^{m} S_2(n,i)$
     - 从贝尔数角度：

- 结论：贝尔数： B_n

博弈论

> 博弈论题型概述：
>
> 1. 有两名选手 $Alice$ 和 Bob 交替进行预先规定好的操作。
> 2. 任意时刻，可以执行的合法操作只取决于情况本身，与选手无关。
> 3. 失败取决于选手无法进行合法操作。

博弈论简介

- **博弈论**，是经济学的一个分支，主要研究具有竞争或对抗性质的对象，在一定规则下产生的各种行为。博弈论考虑游戏中的个体的预测行为和实际行为，并研究它们的优化策略。
- **主要分类**：
  - 公平组合游戏：

    > 公平组合游戏定义如下：
    >
    > - 游戏有两个人参与，二者轮流做出决策，双方均知道游戏的完整信息；
    > - 任意一个游戏者在某一确定状态可以作出的决策集合只与当前的状态有关，而与游戏者无关；
    > - 游戏中的同一个状态不可能多次抵达，游戏以玩家无法行动为结束，且游戏一定会在有限步后以非平局结束。

  - 非公平组合游戏：

    > 非公平组合游戏与公平组合游戏的区别在于在非公平组合游戏中，游戏者在某一确定状态可以做出的决策集合与游戏者有关。大部分的棋类游戏都 **不是** 公平组合游戏，如国际象棋、中国象棋、围棋、五子棋等（因为双方都不能使用对方的棋子）。

  - 反常游戏：

    > 反常游戏按照传统的游戏规则进行游戏，但是其胜者为第一个无法行动的玩家。以 Nim 游戏为例，Nim 游戏中取走最后一颗石子的为胜者，而反常 Nim 游戏中取走最后一刻石子的为败者。

巴什博弈（Bash Game）

> 一堆 n 个物品，两个人轮流从中取出 $1 \sim m$ 个，不能继续取的人输。

- 结论： $(m+1) \mid n$ 时，先手必败。

> 考虑两种特殊的状态：
> 状态1：考虑 $n \leq m$ 的情况，先手可以直接取完，此时先手必胜。
> 状态2：考虑 $n = m + 1$ 的情况，先手必然不能取完，此时先手必败。
>
> 显然存在 $n = k * (m+1) + r$
>
> 假如 $(m+1) \mid n$ ，即 $r = 0$ ，先手取走 x 个，后手可以取 $(m+1) - x$ 个，即无论先手取多少个，后手一定可以保证石子个数整除 $m+1$ 。所以最后一定可以回归到状态二，此时先手必败。
> 反之， $r \neq 0$ ，先手拿 r 个石子，后手陷入上述的必败局面，所以先手必胜。

尼姆博弈（Nim Game）

> 有 n 堆石子，两个人可以从任意一堆石子中拿任意多个石子（不能不拿），取光者胜利。

- 结论：定义 Nim 和 $= a_1 \otimes a_2 \otimes a_3 \otimes \cdots \otimes a_n$ 。当且仅当 Nim 和为 0 时，该状态为必败状态；否则该状态为必胜状态。

SG函数

```
int N = 1000001;
vector<int>sg(N);
vector<int>vis(N);
void init_SG() {
    for(int i = 1; i < N; ++i) {
        vector<int>f; // 记录可转移到的状态
        for(int j = 0; j < f.size(); ++j) {
            vis[sg[i - f[j]]] = i;
        }
        while(vis[sg[i]] == i) sg[i]++;
    }
}
```

计算几何

Hoppz 板子

```
using namespace std;

const double eps = 1e-18;
int sgn(double x)
{
    if(fabs(x) < eps ) return 0;
    if( x < 0 ) return -1;
    else return 1;
}
const int N = 1e5+10;
const int maxp = 10;
const double pi = acos(-1);
const double inf = 1e14;
//square of a double
inline double sqr(double x){return x*x;}

struct Point
{
    double x,y;
    double angle;
    int id;
    Point(){}                                    /// 无参构造
    Point(double _x,double _y){
        x = _x; y = _y;
    }   /// 有参构造
    /// 向量加法
    Point operator + (const Point b) const{
        return Point( x+b.x,y+b.y );
```

```
    }
    /// 向量乘法
    Point operator - (const Point b) const{
        return Point( x-b.x,y-b.y );
    }
    /// 向量数乘
    Point operator * (const double k) const{
        return Point(k*x,k*y);
    }
    /// 向量数除
    Point operator / (const double k) const{
        return Point(x/k , y/k);
    }
    bool operator == (const Point b) const{
        return sgn(x-b.x) == 0 && sgn(y-b.y) == 0;
    }
    bool operator < (const Point b) const {
        return sgn(x-b.x)==0?sgn(y-b.y)<0:x<b.x;
    }
    /// 点积
    double operator * (const Point b) const{
        return x*b.x + y*b.y;
    }
    /// 叉积
    double operator ^ (const Point b) const{
        return x*b.y - y*b.x;
    }
    /// 两点之间的距离
    double distance(const Point P) const {
        return hypot(x-P.x,y-P.y); /// 库函数，求根号下，两数平方和
    }
    /// 向量长度
    double len(){
        return hypot(x,y);
    }
    /// 长度的平方，便于之后的计算
    double len2(){
        return x*x + y*y;
    }
    /// 化为长度为 r 的向量
    Point trunc(double r){
        double l = len();
        r /= l;
        return Point(x*r,y*r);
    }
    /// 以 p 为中心点，pa,pb的夹角大小
    double rad(Point a,Point b){
        Point p = *this;
        return fabs( atan2( fabs( (a-p)^(b-p) )  , (a-p)*(b-p) ) );
    }
    /// 绕 p点 逆时针选择 angle 度
    Point rotate(Point p,double angle){
        Point v = (*this) - p;
        double c = cos(angle) , s = sin(angle);
        return Point(p.x + v.x * c - v.y * s , p.y + v.x *s + v.y * c);
        /// 绕原点旋转后，加上原来P点的偏移量
```

```
    }
    ///逆时针旋转90度
    Point rotleft(){
        return Point(y,-x);
    }
    ///顺时针旋转90度
    Point rotright(){
        return Point(y,-x);
    }
};

struct Line
{
    Point s,e;
    Line(){}
    Line(Point _s,Point _e){
        s =_s; e= _e;
    }

    /// 点斜
    Line(Point p,double angle){
        s = p;
        if( sgn(angle - pi/2) == 0 )    e = (s + Point(0,1));
        else e = (s + Point(1,tan(angle)));
    }
    ///ax + by + c = 0;
    Line(double a,double b,double c){
        if( sgn(a) == 0 ) {
            s = Point(0,-c/b);
            e = Point(1,-c/b);
        }
        else if(sgn(b) == 0) {
            s = Point(-c/a,0);
            e = Point(-c/a,1);
        }
        else {
            s = Point(0,-c/b);
            e = Point(1,(-c-a)/b);
        }
    }
    /// 线段长度
    double length(){
        return s.distance(e);
    }
    /// 返回 0 <= angle <= pi 的基于 x轴 的斜倾角
    double angle(){
        double k = atan2(e.y-s.y , e.x-s.x);
        if( sgn(k) < 0 ) k += pi;
        if( sgn(k-pi) == 0 ) k -=pi;
        return k;
    }
    ///点与直线关系
    //1 在左侧
    //2 在右侧
    //3 在直线
    int relation(Point p){
```

```
        int c = sgn( (p-s) ^ (e -s) );
        if(c < 0) return 1;
        else if(c > 0) return 2;
        else return 3;
    }
    /// 点到直线的距离
    double dispointtoline(Point p){
        return fabs( (p-s)^(e-s) ) /length();
    }
    /// 返回点p在直线上的投影点(垂足)
    Point lineprog(Point p){
        return s + ( ( (e-s)*((e-s)*(p-s)) ) / ( (e-s).len2() ) );
    }
    /// 返回点p在直线上的对称点
    Point symmetrypoint(Point p){
        Point q = lineprog(p);
        return Point(2*q.x-p.x,2*q.y-p.y);
    }
    /// 点与线段的位置关系
    bool point_on_seg(Point p){
        return sgn((p-s)^(e-s) ) == 0 && sgn( (p-s)*(p-e) ) <= 0 ;
    }
    /// 点到线段的距离
    double dispointtoseg(Point p){
        if( sgn((p-s)*(e-s)) < 0 || sgn((p-e)*(s-e))<0 )
            return min( p.distance(s),p.distance(e) );
        return dispointtoline(p);
    }
    /// 判断平行
    bool parallel(Line v){
        return sgn( (e-s)^(v.e-v.s) ) == 0 ;
    }
    /// 判断两直线的位置关系
    //0 平行
    //1 共线
    //2 相交
    int linecrossline(Line v){
        if( v.parallel(*this) ) return v.relation(s) == 3;
        return 2;
    }
    /// 直线与线段位置关系 -*this line -v seg
    //2 规范相交
    //1 非规范相交(顶点处相交)
    //0 不相交
    int linecrossseg(Line v){
        int d1 = sgn( (e-s)^(v.s-s) );
        int d2 = sgn( (e-s)^(v.e-s) );
        if( (d1 ^ d2) == -2 ) return 2;
        return (d1==0||d2==0);
    }

    ///两线段相交判断
    ///2 规范相交
    ///1 非规范相交
    ///0 不想交
    int segcrossseg(Line v) {
```

```
        int d1 = sgn((e - s) ^ (v.s - s));
        int d2 = sgn((e - s) ^ (v.e - s));
        int d3 = sgn((v.e - v.s) ^ (s - v.s));
        int d4 = sgn((v.e - v.s) ^ (e - v.s));
        if ((d1 ^ d2) == -2 && (d3 ^ d4) == -2)return 2;

        return (d1 == 0 && sgn((v.s - s) * (v.s - e)) <= 0) ||
            (d2 == 0 && sgn((v.e - s) * (v.e - e)) <= 0) ||
            (d3 == 0 && sgn((s - v.s) * (s - v.e)) <= 0) ||
            (d4 == 0 && sgn((e - v.s) * (e - v.e)) <= 0);
    }
    /// 线段到线段的距离
    double disssegtoseg(Line v){
        return min( min( dispointtoseg(v.s),dispointtoseg(v.e)) ,
min(v.dispointtoseg(s), v.dispointtoseg(e) ) );
    }

    /// 求两直线的交点
    Point crosspoint(Line v){
        double a1 = (v.e-v.s)^(s-v.s);
        double a2 = (v.e-v.s)^(e-v.s);
        return Point( (s.x*a2-e.x*a1)/(a2-a1) , (s.y*a2-e.y*a1)/(a2-a1) );
    }
};

struct Polygon
{
    int n;
    Point p[maxp];
    Line l[maxp];

    /// 在多边形中添加点
    void add(Point q){
        p[n++] = q;
    }
    /// 获取所有的线段
    void getLine(){
        for(int i = 0;  i < n ; i++){
            l[i] = Line(p[i],p[(i+1)%n]);
        }
    }
    /// 判断多边形是不是凸包
    /// 如果是直接对点集效验的话，要先极角排序
    bool isconvex(){
        bool s[2];
        memset(s,0,sizeof s);
        for(int i = 0 ; i < n ; i++){
            int j = (i+1)%n;
            int k = (j+1)%n;
            s[ sgn( (p[j]-p[i])^(p[k]-p[i]))+1 ] = true;
            if( s[0] && s[2] ) return false;
        }
        return true;
    }
    /// 多边形周长
    double getcircumference(){
```

```
        double sum = 0;
        for(int i = 0 ; i < n ; i++){
            sum += p[i].distance(p[(i+1)%n]);
        }
        return sum;
    }
    /// 多边形面积
    double getarea(){
        double sum = 0;
        for(int i = 0; i < n ; i++){
            sum += (p[i]^p[(i+1)%n]);
        }
        return fabs(sum)/2;
    }
    /// 重心
    Point getbarycenter(){
        Point ret(0,0);
        double area = 0 ;
        for(int i = 1; i < n-1 ; i++){
            double tmp = (p[i]-p[0])^(p[i+1]-p[0]);
            if( sgn(tmp) == 0 ) continue;
            area += tmp;
            ret.x += (p[0].x + p[i].x + p[i+1].x)/3*tmp;
            ret.y += (p[0].y + p[i].y + p[i+1].y)/3*tmp;
        }
        if( sgn(area)) ret = ret/area;
        return ret;
    }
    /// 判断点与任意多边形的关系
    //3 点上
    //2 边上
    //1 内部
    //0 外部
    int relationpoint(Point q){
        /// 在点上
        for(int i = 0; i < n; i++){
            if( p[i] == q ) return 3;
        }
//        getLine();  /// 之前getline了的话，可以注释节约时间
        /// 在边上
        for(int i = 0; i < n ; i++){
            if( l[i].point_on_seg(q) ) return 2;
        }
        /// 在内部
        int cnt = 0;
        for(int i = 0 ; i < n ; i++){
            int j = (i+1)%n;
            int k = sgn( (q-p[j])^(p[i]-p[j]) );
            int u = sgn( p[i].y - q.y );
            int v = sgn( p[j].y - q.y );
            if( k > 0 && u < 0 && v >= 0 ) cnt ++;
            if( k < 0 && v < 0 && u >= 0 ) cnt--;
        }
        return cnt != 0;
    }
    /// 判断多边形是否与多边形相离
```

```
    int relationpolygon(Polygon Poly){
        getLine();
        Poly.getLine();
        for(int i1 = 0 ; i1 < n ; i1++){
            int j1 = (i1+1)%n;
            for(int i2 = 0 ; i2 <= Poly.n ; i2++){
                int j2 = (i2+1)%Poly.n;
                Line l1 = Line(p[i1],p[j1]);
                Line l2 = Line(Poly.p[i2],Poly.p[j2]);
                if( l1.segcrossseg(l2) ) return 0;
                if( Poly.relationpoint(p[i1]) || relationpoint(Poly.p[i2]) )
return 0;
            }
        }
        return 1;
    }

};

struct Circle
{
    Point p;
    double r;
    Circle(){}
    Circle(Point _p,double _r):p(_p),r(_r){}
    Circle(double x,double y,double _r){    /// 点的坐标，圆心
        p = Point(x,y); r = _r;
    }

    bool operator == (Circle v){
        return (p == v.p) && sgn(r-v.r) == 0;
    }   /// 以圆心为主排序，半径为副排序
    bool operator < (Circle v) const{
        return ( (p<v.p) || (p == v.p) && sgn(r-v.r)<0 );
    }
    /// 面积
    double area(){
        return pi*r*r;
    }
    /// 周长
    double circumference(){
        return 2*pi*r;
    }
    /// 点和圆的位置关系
    //0 圆外
    //1 圆上
    //2 圆内
    int relationPoint(Point b){
        double dst = b.distance(p);
        if( sgn(dst-r)<0 ) return 2;
        if( sgn(dst-r)==0 ) return 1;
        return 0;
    }
    /// 直线和圆的关系
    //0 相离
    //1 相切
```

```
    //2 相交
    int relationLine(Line v){
        double dst = v.dispointtoline(p);
        if( sgn(dst-r) < 0 ) return 2;
        if( sgn(dst-r) == 0 ) return 1;
        return 0;
    }
    /// 线段和圆的关系
    int relationSeg(Line v){
        double dst = v.dispointtoseg(p);
        if( sgn(dst-r) < 0 ) return 2;
        if( sgn(dst-r) == 0 ) return 1;
        return 0;
    }
    /// 两圆的关系
    //5 相离
    //4 外切
    //3 相交
    //2 内切
    //1 内含
    int relationcircle(Circle v){
        double d = p.distance(v.p);
        if( sgn( d-r-v.r ) > 0 ) return 5;
        if( sgn( d-r-v.r ) == 0 ) return 4;
        double l = fabs(r-v.r);
        if( sgn( d-r-v.r ) < 0 && sgn(d-l)  > 0 ) return 3;
        if( sgn( d-l ) == 0 ) return 2;
        if( sgn( d-l ) < 0 ) return 1;
    }

};


/// 用来指定排序的极点
Point cmpPoint = Point(inf,inf);

/// 需重载 Point 的小于符号
bool cmpAtan2(Point a,Point b)
{
    if( sgn(a.angle - b.angle) == 0 )    return a.distance(cmpPoint) <
b.distance(cmpPoint );
    return a.angle < b.angle;
}

/// 慎用，第一个基本上就够了
bool cmpCross(const Point a,const Point b)
{

    double d =  sgn((a-cmpPoint)^(b-cmpPoint))  ;
    if( d == 0 ) return sgn( a.distance(cmpPoint) - b.distance(cmpPoint) ) < 0;
    return d > 0;
}

///--------------------- 凸包 --------------------///

/// 求之前的点集
```

```
Point convexP[N];
int Sizep;      /// 点集大小
void getconvex(Polygon &convex)
{
    /// 按照 x 从小到大排序，如果 x 相同，按 y 排序。
    sort(convexP,convexP+Sizep);
    convex.n = Sizep;
    for(int i = 0 ; i < min(Sizep,2) ; i ++){
        convex.p[i] = convexP[i];
    }
    /// 特判
    if( convex.n == 2 && ( convex.p[0] == convex.p[1] ) ) convex.n--;
    if( Sizep <= 2 ) return ;
    int &top = convex.n;
    top = 1;
    for(int i = 2; i < Sizep ; i++){
        while( top && sgn( (convex.p[top] - convexP[i])^(convex.p[top-1]-
convexP[i]) ) <= 0  ) top--;
        convex.p[++top] = convexP[i];
    }
    int temp = top;
    convex.p[++top] = convexP[Sizep-2];

    for(int i = Sizep-3 ; i >= 0 ; i--){
        while( top != temp && sgn( (convex.p[top] -convexP[i])^(convex.p[top-1]-
convexP[i])) <= 0 ) top --;
        convex.p[++top] = convexP[i];
    }
    if( convex.n == 2 && ( convex.p[0] == convex.p[1] ) ) convex.n--;
    /// 这样求出来的顺时针的凸包，如果要逆时针的话，来一次极角排序就好了。
    /// 极角排序
//    cmpPoint = convex.p[0];
//    sort(convex.p,convex.p+convex.n,cmpCross);
}
```

最小圆覆盖

```
inline int PIC(Circle C,Point a){return dcmp(Len(a-C.O)-C.r)<=0;}//判断点A是否在圆C
内

inline Circle Min_Circle(Point *P,Re n){//【求点集P的最小覆盖圆】
//  random_shuffle(P+1,P+n+1);
    for(int i = 0 ; i < n ; i++) swap(P[i],P[rand()%n+1]);
    Circle C=Circle(P[0],0);
    for(int i=1;i<n;++i)if(!C.relationPoint(P[i])){
        C=Circle(P[i],0);
        for(Rinte j=1;j<i;++j)if(!C.relationPoint(P[j])){
            C.p=(P[i]+P[j])*0.5,C.r=P[j].distance(C.p);
            for(int
k=1;k<j;++k)if(!C.relationPoint(P[j]))C=getcircle(P[i],P[j],P[k]);
        }
    }
    return C;
}
```

其他技巧

关于 __int128

输入输出函数

```
#define int __int128
inline void read(int &n){
    int x=0,f=1;
    char ch=getchar();
    while(ch<'0'||ch>'9'){
        if(ch=='-') f=-1;
        ch=getchar();
    }
    while(ch>='0'&&ch<='9'){
        x=(x<<1)+(x<<3)+(ch^48);
        ch=getchar();
    }
    n=x*f;
}
inline void print(int n){
    if(n<0){
        putchar('-');
        n*=-1;
    }
    if(n>9) print(n/10);
    putchar(n % 10 + '0');
}
```

超大整数转换

```
inline __int128 to_int128(string s){
    int l=s.length();
    __int128 m=0;
    for(int i=0;i<l;i++){
        m*=10;
        m+=s[i]-48;
    }
    return m;
}
```

一些公式定理技巧

排序不等式

设有两数列 $a_1, a_2, \ldots, a_n$ 和 $b_1, b_2, \ldots, b_n$，满足 $a_1 \le a_2 \le \cdots \le a_n$ 和 $b_1 \le b_2 \le \cdots \le b_n$，$c_1, c_2, \ldots, c_n$ 是 $b_1, b_2, \ldots, b_n$ 的乱序排列，则有：

$$\sum_{i=1}^{n} a_i b_{n+i-1} \le \sum_{i=1}^{n} a_i c_i \le \sum_{i=1}^{n} a_i b_i$$

当且仅当 $a_i = a_j$ 或 $a_i = a_j$ $(1 \le i, j \le n)$ 时等号成立。

分治求 $1+p^1+p^2+...+p^c$

1. 若 c 为奇数： $sum(p,c)=(1+p^{\frac{c+1}{2}})\times sum(p,\frac{c-1}{2})$
2. 若 c 为偶数： $sum(p,c)=1+p\times sum(p,c-1)$

平方和

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n}(a_i+a_j)^2=(n-2)\times(\sum_{i=1}^{n}a_i^2)+(\sum_{i=1}^{n}a_i)^2$$

康托展开

> 求 $1\sim N$ 的一个给定全排列在所有 $1\sim N$ 全排列中的排名。结果对 998244353 取模。

$$ans=1+\sum_{i=1}^{n}A[i]\times(n-i)!$$

麦乐鸡定理

> 两个整数的正线性组合所不能表示的最大数是多少?

$$A_{max}=lcm(a,b)-a-b$$

经典模型

括号匹配问题

> 合法的括号序列：
>
> 1、左括号数 == 右括号数
>
> 2、在所有前缀中，满足：左括号数 >= 右括号数

最长合法括号子串

> 我们把括号抽象到二维平面坐标上，起点在 `(0,0)` ，遇到 `(` 纵坐标 `+1`，遇到 `)` 纵坐标 `-1`

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e6 + 10;
int st[N];
int main()
{
    ios::sync_with_stdio(false);cin.tie(0); cout.tie(0);
    string s;
    cin >> s;

    stack<int> sta;
    int ans = 0;

    int cnt = 0, ma =0;
    for (int i = 0; i < s.length(); i++) {
        if (!sta.empty() && s[i] == ')' && s[sta.top()] == '(') { sta.pop(); }
        else sta.push(i);

        int res = 0;
```

```
        if (!sta.empty()) res = i - sta.top();
        else res = i + 1;

        if (res == ma) cnt++;
        else if(res > ma) { ma = res; cnt = 1; }
    }
    if (ma)cout << ma << ' ' << cnt << endl;
    else cout << "0 1" << endl;
    return 0;
}
```

最长合法括号子序列

直接贪心。

奇数码

> 在一个 $n*n$ 的网格中进行，其中 n 为奇数， 1 个空格和 1 到 $n\times n-1$ 个数恰好不重不漏地分布在 $n*n$ 的网格中。空格移动可与四个方向相邻数码互换。
>
> 问是否能从一个局面，转移到另一个局面?

结论： 奇数码游戏两个局面可达，当且仅当两个局面下网格中的数依次写成 1 行 $n\times n-1$ 个元素后（不考虑空格），**逆序对个数的奇偶性相同。**

> 推广： n 为偶数的情况，两个局面可达，当且仅当写成序列后，`逆序对数之差` 和 `两个局面下空格所在的行数之差` 奇偶性相同。

滑动窗口

约瑟夫问题

> n 个人标号 $0,1,2,\ldots,n-1$ 。逆时针站一圈，从 0 号开始，每一次从当前的人逆时针数个 k ，然后让这个人出局。问最后剩下的人是谁。

```
int josephus(int n, int k) {
  int res = 0;
  for (int i = 1; i <= n; ++i) res = (res + k) % i;
  return res;
}
```

未整理题目

判断负环

> 给定一个 n 个点的有向图，请求出图中是否存在**从顶点 1 出发能到达**的负环。
>
> 负环的定义是：一条边权之和为负数的回路。

```
const int N = 2e6 + 1000;
int cnt = 0;
struct node{
    int x , y , v;
}e[N];
void add(int x ,int y , int v) {e[++cnt] = {x , y , v};}
void addd(int x, int y , int v) {
    if(v < 0)add(x , y , v);
    if(v >= 0) add(x , y , v) , add(y , x , v);
}
int n;
bool bellman() {
    static int d[N];
    d[1] = 0;
    for(int i = 2; i <= n; i++)
        d[i] = 0x7fffffff;
    for(int i = 1 ; i <= n - 1; i++)
        for(int j = 1; j <= cnt; j++) {
            if(d[e[j].x] != 0x7fffffff &&
            d[e[j].x] + e[j].v < d[e[j].y])
                d[e[j].y] = d[e[j].x] + e[j].v;
        }
    for(int i = 1; i <= cnt; i++) {
        if(d[e[i].x] == 0x7fffffff || d[e[i].y] == 0x7fffffff)continue;
        if(d[e[i].x] + e[i].v < d[e[i].y])return true;// 负权回路
    }
    return false;
}
signed main() {
    int t;scanf("%d" , &t);
    while(t--) {
        memset(e , 0 , sizeof(e));
        cnt = 0;
        int m;scanf("%d%d" , &n, &m);
        for(int i = 1; i <= m; i++) {
            int x , y , v;scanf("%d%d%d" , &x , &y , &v);
            addd(x , y , v);
        }
        if(bellman())printf("YES\n");
        else printf("NO\n");
    }
    return 0;
}
```

SPFA

> 可负权值单源最短路

```
const int N=1e5+10,M=1e5+10,INF=0x3f3f3f3f;
int h[N],e[M],w[M],ne[M],idx;
int dis[N];
int q[N],hh,tt=-1;//手写队列
bool vis[N]; //判断顶点是否已经加入队列
int n,m;
void add(int a,int b,int c) {
    e[idx]=b,w[idx]=c,ne[idx]=h[a],h[a]=idx++;
}
void spfa(int s) {
    memset(dis,0x3f,sizeof dis);
    dis[s]=0;
    q[++tt]=s,vis[s]=true;  //默认第一个距离已经被修改，加入队列
    while (hh<=tt) {
        int v=q[hh++];
        vis[v]=false;
        for (int i=h[v];i!=-1;i=ne[i]) {//更新所有的dis[b]
            int j=e[i];  //v->j的边，权值为w[i]
            if (dis[j] > dis[v]+w[i]) {
                dis[j]=dis[v]+w[i];
                if (!vis[j]) q[++tt]=j,vis[j]=true;
            }
        }

    }
}
int main() {
    memset(h,-1,sizeof h);
    scanf("%d%d",&n,&m);
    int a,b,c;
    for (int i=0;i<m;i++) {
        scanf("%d%d%d",&a,&b,&c);
        add(a,b,c); //有向图：a->b，权值为c
    }
    spfa(1);//求起点为1号结点到其它所有结点的距离
    dis[n]==INF?puts("impossible"):printf("%d",dis[n]);
    return 0;
}
```

AC自动机

> 给你一个文本串 S 和 n 个模式串 $T_1, T_2, \ldots, T_n$，请你分别求出每个模式串 T_i 在 S 中出现的次数。

```
#define maxn 2000001
char s[maxn],T[maxn];
int n,cnt,vis[200051],ans,in[maxn],Map[maxn];
struct kkk{
    int son[26],fail,flag,ans;
    void clear(){memset(son,0,sizeof(son)),fail=flag=ans=0;}
}trie[maxn];
```

```
queue<int>q;
void insert(char* s,int num){
    int u=1,len=strlen(s);
    for(int i=0;i<len;i++){
        int v=s[i]-'a';
        if(!trie[u].son[v])trie[u].son[v]=++cnt;
        u=trie[u].son[v];
    }
    if(!trie[u].flag)trie[u].flag=num;
    Map[num]=trie[u].flag;
}
void getFail(){
    for(int i=0;i<26;i++)trie[0].son[i]=1;
    q.push(1);
    while(!q.empty()){
        int u=q.front();q.pop();
        int Fail=trie[u].fail;
        for(int i=0;i<26;i++){
            int v=trie[u].son[i];
            if(!v){trie[u].son[i]=trie[Fail].son[i];continue;}
            trie[v].fail=trie[Fail].son[i]; in[trie[v].fail]++;
            q.push(v);
        }
    }
}
void topu(){
    for(int i=1;i<=cnt;i++)
    if(in[i]==0)q.push(i);
    while(!q.empty()){
        int u=q.front();q.pop();vis[trie[u].flag]=trie[u].ans;
        int v=trie[u].fail;in[v]--;
        trie[v].ans+=trie[u].ans;
        if(in[v]==0)q.push(v);
    }
}
void query(char* s){
    int u=1,len=strlen(s);
    for(int i=0;i<len;i++)
    u=trie[u].son[s[i]-'a'],trie[u].ans++;
}
int main(){
    scanf("%d",&n); cnt=1;
    for(int i=1;i<=n;i++){
        scanf("%s",s);
        insert(s,i);
    }getFail();scanf("%s",T);
    query(T);topu();
    for(int i=1;i<=n;i++)printf("%d\n",vis[Map[i]]);
}
```

KMP

> 给出两个字符串 S_1 和 S_2，若 S_1 的区间 $[l,r]$ 子串与 S_2 完全相同，则称 S_2 在 S_1 中出现了，其出现位置为 l。
> 现在请你求出 S_2 在 S_1 中所有出现的位置。
>
> 定义一个字符串 S 的 $border$ 为 S 的一个非 S 本身的子串 T，满足 T 既是 S 的前缀，又是 S 的后缀。
> 对于 S_2，你还需要求出对于其每个前缀 S 的最长 $border$ 的长度。

```
#define MAXN 1000010
using namespace std;
int kmp[MAXN];
int la, lb, j;
char a[MAXN], b[MAXN];
int main() {
    cin >> a + 1;
    cin >> b + 1;
    la = strlen(a + 1);
    lb = strlen(b + 1);
    for (int i = 2; i <= lb; i++) {
        while (j && b[i] != b[j + 1]) j = kmp[j];
        if (b[j + 1] == b[i]) j++;
        kmp[i] = j;
    }
    j = 0;
    for (int i = 1; i <= la; i++) {
        while (j > 0 && b[j + 1] != a[i]) j = kmp[j];
        if (b[j + 1] == a[i]) j++;
        if (j == lb) {
            cout << i - lb + 1 << endl;
            j = kmp[j];
        }
    }
    for (int i = 1; i <= lb; i++) cout << kmp[i] << " ";
    return 0;
}
```

二分图最大匹配

> 给定一个二分图，其左部点的个数为 n，右部点的个数为 m，边数为 e，求其最大匹配的边数。
>
> 左部点从 1 至 n 编号，右部点从 1 至 m 编号。

```
#define N 1001
using namespace std;
int n, m, e, ans, match[N];
bool a[N][N], vis[N];
bool dfs(int x){
    for (int i = 1; i <= m; i++)
        if (!vis[i] && a[x][i]) {
            vis[i] = 1;
            if (!match[i] || dfs(match[i])) {
                match[i] = x;
                return 1;
```

```
            }
        }
    return 0;
}
int main(){
    cin >> n >> m >> e;
    for (int i = 1; i <= e; i++) {
        int u, v;
        scanf("%d%d", &u, &v);
        if (v <= m) a[u][v] = 1;
    }
    for (int i = 1; i <= n; i++) {
        ans += dfs(i);
        memset(vis, 0, sizeof(vis));
    }
    cout << ans;
    return 0;
}
```

树链剖分

> 如题，已知一棵包含 N 个结点的树（连通且无环），每个节点上包含一个数值，需要支持以下操作：
>
> - `1 x y z`，表示将树从 x 到 y 结点最短路径上所有节点的值都加上 z。
> - `2 x y`，表示求树从 x 到 y 结点最短路径上所有节点的值之和。
> - `3 x z`，表示将以 x 为根节点的子树内所有节点值都加上 z。
> - `4 x` 表示求以 x 为根节点的子树内所有节点值之和。

```
using namespace std;
typedef long long LL;
#define mid ((l+r)>>1)
#define lson rt<<1,l,mid
#define rson rt<<1|1,mid+1,r
#define len (r-l+1)
const int maxn=200000+10;
int n,m,r,mod; // 树的结点个数，操作个数，根节点序号，取模数
int e,beg[maxn],nex[maxn],to[maxn],w[maxn],wt[maxn];
int a[maxn<<2],laz[maxn<<2];
int son[maxn],id[maxn],fa[maxn],cnt,dep[maxn],siz[maxn],top[maxn];
int res=0;

void add(int x,int y){
    to[++e]=y;
    nex[e]=beg[x];
    beg[x]=e;
}
//------------------------------------ 以下为线段树
void pushdown(int rt,int lenn){
    laz[rt<<1]+=laz[rt];
    laz[rt<<1|1]+=laz[rt];
    a[rt<<1]+=laz[rt]*(lenn-(lenn>>1));
    a[rt<<1|1]+=laz[rt]*(lenn>>1);
    a[rt<<1]%=mod;
```

```
    a[rt<<1|1]%=mod;
    laz[rt]=0;
}
void build(int rt,int l,int r){
    if(l==r){
        a[rt]=wt[l];
        if(a[rt]>mod)a[rt]%=mod;
        return;
    }
    build(lson);
    build(rson);
    a[rt]=(a[rt<<1]+a[rt<<1|1])%mod;
}
void query(int rt,int l,int r,int L,int R){
    if(L<=l&&r<=R){res+=a[rt];res%=mod;return;}
    else{
        if(laz[rt])pushdown(rt,len);
        if(L<=mid)query(lson,L,R);
        if(R>mid)query(rson,L,R);
    }
}
void update(int rt,int l,int r,int L,int R,int k){
    if(L<=l&&r<=R){
        laz[rt]+=k;
        a[rt]+=k*len;
    }
    else{
        if(laz[rt])pushdown(rt,len);
        if(L<=mid)update(lson,L,R,k);
        if(R>mid)update(rson,L,R,k);
        a[rt]=(a[rt<<1]+a[rt<<1|1])%mod;
    }
}
//--------------------------------以上为线段树
int qRange(int x,int y){
    int ans=0;
    while(top[x]!=top[y]){
        if(dep[top[x]]<dep[top[y]])swap(x,y);
        res=0;
        query(1,1,n,id[top[x]],id[x]);
        ans+=res;
        ans%=mod;
        x=fa[top[x]];
    }
    if(dep[x]>dep[y])swap(x,y);
    res=0;
    query(1,1,n,id[x],id[y]);
    ans+=res;
    return ans%mod;
}
void updRange(int x,int y,int k){
    k%=mod;
    while(top[x]!=top[y]){
        if(dep[top[x]]<dep[top[y]])swap(x,y);
        update(1,1,n,id[top[x]],id[x],k);
        x=fa[top[x]];
```

```
    }
    if(dep[x]>dep[y])swap(x,y);
    update(1,1,n,id[x],id[y],k);
}
int qSon(int x){
    res=0;
    query(1,1,n,id[x],id[x]+siz[x]-1);
    return res;
}
void updSon(int x,int k){
    update(1,1,n,id[x],id[x]+siz[x]-1,k);
}
void dfs1(int x,int f,int deep){
    dep[x]=deep;
    fa[x]=f;
    siz[x]=1;
    int maxson=-1;
    for(int i=beg[x];i;i=nex[i]){
        int y=to[i];
        if(y==f)continue;
        dfs1(y,x,deep+1);
        siz[x]+=siz[y];
        if(siz[y]>maxson)son[x]=y,maxson=siz[y];
    }
}
void dfs2(int x,int topf){
    id[x]=++cnt;
    wt[cnt]=w[x];
    top[x]=topf;
    if(!son[x]) return;
    dfs2(son[x],topf);
    for(int i=beg[x];i;i=nex[i]){
        int y=to[i];
        if(y==fa[x]||y==son[x])continue;
        dfs2(y,y);
    }
}
int main(){
    cin >> n >> m >> r >> mod;
    for(int i=1;i<=n;i++) cin >> w[i];
    for(int i=1;i<n;i++){
        int a,b; cin >> a >> b;
        add(a,b);add(b,a);
    }
    dfs1(r,0,1); dfs2(r,r); build(1,1,n);
    while(m--){
        int k,x,y,z; cin >> k;
        if(k==1){ // 将树从 x 到 y 节点最短路径上所有节点的值都加上 z
            cin >> x >> y >> z;
            updRange(x,y,z);
        }
        else if(k==2){ // 表示树从 x 到 y 节点最短路径上所有节点的值的和
            cin >> x >> y;
            printf("%d\n",qRange(x,y));
        }
        else if(k==3){ // 将以 x 为根节点的子树内所有节点的值加上 y
```

```
            cin >> x >> y;
            updSon(x,y);
        }
        else{ // 表示以 x 为根节点的子树的所有节点值的和
            cin >> x;
            printf("%d\n",qSon(x));
        }
    }
}
```

二元一次不定方程

对方程 $ax+by=c$，首先用 $exgcd$ 求解方程的一组特解 x,y。

其整数通解为：（k 为任意整数）

$$\begin{cases} x = x_0 + \frac{b}{gcd(a,b)} \times k \\ y = y_0 - \frac{a}{gcd(a,b)} \times k \end{cases}$$

差分约束

> 给出一组包含 m 个不等式，有 n 个未知数的形如：
>
> $$\begin{cases} x_{c_1} - x_{c_1'} \le y_1 \\ x_{c_2} - x_{c_2'} \le y_2 \\ \cdots \\ x_{c_m} - x_{c_m'} \le y_m \end{cases}$$
>
> 的不等式组，求任意一组满足这个不等式组的解。
>
> **输入**
>
> 第一行为两个正整数 n,m 代表未知数的数量和不等式的数量。
>
> 接下来 m 行，每行包含三个整数 c,c',y 代表一个不等式 $x_c - x_{c'} \le y$。

```
using namespace std;
int n,m;
const int N=5e3+5;//点数
const int M=1e4+5;//边数，注意虚拟源点连的边
int End[M],Last[N],Next[M],Len[M],e;
void add_edge(int x,int y,int z) {
    End[++e]=y;
    Next[e]=Last[x];
    Last[x]=e;
    Len[e]=z;
}
int dis[N],cnt[N];
queue<int> q;
bool inq[N];
bool spfa(int s) {
    for(int i=0;i<=n;i++) dis[i]=-1e9;//最长路，初始值为-inf
    dis[s]=0;
    q.push(s);
    inq[s]=true;
    cnt[s]++;
    while(q.size()) {
```

```
        int x=q.front(); q.pop();
        inq[x]=false;
        for(int i=Last[x];i;i=Next[i]) {
            int y=End[i];
            if(dis[y]<dis[x]+Len[i]) {
                dis[y]=dis[x]+Len[i];
                if(!inq[y]) {
                    q.push(y);
                    inq[y]=true;
                    cnt[y]++;
                    if(cnt[y]>n+1) return false;//返回 false，意味着无解
                }
            }
        }
    }
    return true;
}
int main() {
    scanf("%d%d",&n,&m);
    for(int i=1;i<=n;i++) add_edge(0,i,0);//虚拟源点连边
    for(int i=1;i<=m;i++) {
        int x,y,z; scanf("%d%d%d",&x,&y,&z);
        add_edge(x,y,-z);
    }
    if(spfa(0)) {
        for(int i=1;i<=n;i++) printf("%d%c",dis[i],i<n?' ':'\n');
    }
    else printf("NO\n");
    return 0;
}
```

可持久化线段树

> 如题，你需要维护这样的一个长度为 N 的数组，支持如下几种操作
>
> 1. 在某个历史版本上修改某一个位置上的值
> 2. 访问某个历史版本上的某一位置的值
>
> **输入**
>
> 输入的第一行包含两个正整数 N, M 分别表示数组的长度和操作的个数。
>
> 第二行包含 N 个整数，依次为初始状态下数组各位的值（依次为 a_i， $1 \leq i \leq N$ ）。
>
> 接下来 M 行每行包含3或4个整数，代表两种操作之一（ i 为基于的历史版本号）：
>
> 1. 对于操作1，格式为 $v_i\ 1\ loc_i\ value_i$ ，即为在版本 v_i 的基础上，将 a_{loc_i} 修改为 $value_i$ 。
> 2. 对于操作2，格式为 $v_i\ 2\ loc_i$ ，即访问版本 v_i 中的 a_{loc_i} 的值。

```
using namespace std;
const int maxn = 1000009;
int rt[maxn],a[maxn];
struct sg_tree{int val,lson,rson;}node[maxn*33];
int n,m,tot;
void build(int &now,int l,int r) {
    now=++tot;
```

```
    if(l==r){node[now].val=a[l];return ;}
    int mid=(l+r)>>1;
    build(node[now].lson,l,mid);
    build(node[now].rson,mid+1,r);
}
void insert(int &now,int last,int l,int r,int pos,int val) {
    now=++tot;node[now]=node[last];
    if(l==r){node[now].val=val;return ;}
    int mid=(l+r)>>1;
    if(pos<=mid) insert(node[now].lson,node[last].lson,l,mid,pos,val);
    else insert(node[now].rson,node[last].rson,mid+1,r,pos,val);
}
int query(int now,int l,int r,int pos) {
    if(l==r) return node[now].val;
    int mid=(l+r)>>1;
    if(pos<=mid) return query(node[now].lson,l,mid,pos);
    else return query(node[now].rson,mid+1,r,pos);
}
int main() {
    cin >> n >> m;
    for(int i=1;i<=n;i++) cin >> a[i];
    build(rt[0],1,n);
    for(int i=1;i<=m;i++)
    {
        int ver,opt,pos; cin >> ver >> opt >> pos;
        if(opt==1) {
            int val; cin >> val;
            insert(rt[i],rt[ver],1,n,pos,val);
        }
        else {
            cout << query(rt[ver],1,n,pos) << "\n";
            rt[i]=rt[ver];
        }
    }
    return 0;
}
```

静态区间第 k 小

> 如题，给定 n 个整数构成的序列 a，将对于指定的闭区间 $[l,r]$ 查询其区间内的第 k 小值。

```
#define M 200010
using namespace std;
int node_cnt, n, m;
int sum[M<<5], rt[M], lc[M<<5], rc[M<<5];//线段树相关
int a[M], b[M];//原序列和离散序列
int p;//修改点
void build(int &t, int l, int r) {
    t = ++node_cnt;
    if(l == r) return;
    int mid = (l + r) >> 1;
    build(lc[t], l, mid);
    build(rc[t], mid+1, r);
}
int modify(int o, int l, int r) {
```

```
    int oo = ++node_cnt;
    lc[oo] = lc[o]; rc[oo] = rc[o]; sum[oo] = sum[o] + 1;
    if(l == r) return oo;
    int mid = (l + r) >> 1;
    if(p <= mid) lc[oo] = modify(lc[oo], l, mid);
    else rc[oo] = modify(rc[oo], mid+1, r);
    return oo;
}
int query(int u, int v, int l, int r, int k) {
    int ans, mid = ((l + r) >> 1), x = sum[lc[v]] - sum[lc[u]];
    if(l == r) return l;
    if(x >= k) ans = query(lc[u], lc[v], l, mid, k);
    else ans = query(rc[u], rc[v], mid+1, r, k-x);
    return ans;
}
int main() {
    int l, r, k, q, ans;
    scanf("%d%d", &n, &m);
    for(register int i = 1; i <= n; i += 1) scanf("%d", &a[i]), b[i] = a[i];
    sort(b+1, b+n+1);
    q = unique(b+1, b+n+1) - b - 1;
    build(rt[0], 1, q);
    for(register int i = 1; i <= n; i += 1) {
        p = lower_bound(b+1, b+q+1, a[i])-b;//可以视为查找最小下标的匹配值，核心算法是二分查找
        rt[i] = modify(rt[i-1], 1, q);
    }
    while(m--) {
        scanf("%d%d%d", &l, &r, &k);
        ans = query(rt[l-1], rt[r], 1, q, k);
        printf("%d\n", b[ans]);
    }
    return 0;
}
```

普通平衡树

> 您需要写一种数据结构（可参考题目标题），来维护一些数，其中需要提供以下操作：
>
> 1. 插入一个数 x 。
> 2. 删除一个数 x （若有多个相同的数，应只删除一个）。
> 3. 定义**排名**为比当前数小的数的个数 $+1$ 。查询 x 的排名。
> 4. 查询数据结构中排名为 x 的数。
> 5. 求 x 的前驱（前驱定义为小于 x，且最大的数）。
> 6. 求 x 的后继（后继定义为大于 x，且最小的数）。
>
> 对于操作 $3,5,6$，**不保证**当前数据结构中存在数 x 。

```
typedef long long LL;
using namespace std;
const int maxn = 1000019,INF = 1e9;
int na;
int ch[maxn][2];
```

```
int val[maxn],dat[maxn];
int sz[maxn],cnt[maxn];
int tot,root;
int New(int v){
    val[++tot] = v;
    dat[tot] = rand();
    sz[tot] = 1;
    cnt[tot] = 1;
    return tot;
}
void pushup(int id){
    sz[id] = sz[ch[id][0]] + sz[ch[id][1]] + cnt[id];
}
void build(){
    root = New(-INF),ch[root][1] = New(INF);
    pushup(root);
}
void Rotate(int &id,int d){
    int temp = ch[id][d ^ 1];
    ch[id][d ^ 1] = ch[temp][d];
    ch[temp][d] = id;
    id = temp;
    pushup(ch[id][d]),pushup(id);
}
void insert(int &id,int v){
    if(!id){
        id = New(v);
        return ;
    }
    if(v == val[id])cnt[id]++;
    else{
        int d = v < val[id] ? 0 : 1;
        insert(ch[id][d],v);
        if(dat[id] < dat[ch[id][d]])Rotate(id,d ^ 1);
    }
    pushup(id);
}
void Remove(int &id,int v){
    if(!id)return ;
    if(v == val[id]){
        if(cnt[id] > 1){cnt[id]--,pushup(id);return ;}
        if(ch[id][0] || ch[id][1]){
            if(!ch[id][1] || dat[ch[id][0]] > dat[ch[id][1]]){
                Rotate(id,1),Remove(ch[id][1],v);
            }
            else Rotate(id,0),Remove(ch[id][0],v);
            pushup(id);
        }
        else id = 0;
        return ;
    }
    v < val[id] ? Remove(ch[id][0],v) : Remove(ch[id][1],v);
    pushup(id);
}
int get_rank(int id,int v){
    if(!id)return 1;
```

```
    if(v == val[id])return sz[ch[id][0]] + 1;
    else if(v < val[id])return get_rank(ch[id][0],v);
    else return sz[ch[id][0]] + cnt[id] + get_rank(ch[id][1],v);
    }
int get_val(int id,int rank){
    if(!id)return INF;
    if(rank <= sz[ch[id][0]])return get_val(ch[id][0],rank);
        else if(rank <= sz[ch[id][0]] + cnt[id])return val[id];
    else return get_val(ch[id][1],rank - sz[ch[id][0]] - cnt[id]);
}
int get_pre(int v){
    int id = root,pre;
    while(id) {
        if(val[id] < v)pre = val[id],id = ch[id][1];
        else id = ch[id][0];
    }
    return pre;
}
int get_next(int v){
    int id = root,next;
    while(id){
        if(val[id] > v)next = val[id],id = ch[id][0];
        else id = ch[id][1];
        }
    return next;
}
int main(){
    build();
    cin >> na;
    for(int i = 1;i <= na;i++){
        int cmd, x; cin >> cmd >> x;
        if(cmd == 1)insert(root,x);
        else if(cmd == 2)Remove(root,x);
        else if(cmd == 3)printf("%d\n",get_rank(root,x) - 1);
        else if(cmd == 4)printf("%d\n",get_val(root,x + 1));
        else if(cmd == 5)printf("%d\n",get_pre(x));
        else if(cmd == 6)printf("%d\n",get_next(x));
        }
    return 0;
}
```

威佐夫博弈

> 有两堆各若干个物品，两个人轮流从任意一堆中取出至少一个或者同时从两堆中取出同样多的物品，规定每次至少取一个，至多不限，最后取光者胜利

> 结论：
>
> - 若两堆物品的初始值为 $(x，y)$，且 $x < y$，则另 $z = y - x$；
> - 记 $w = (int)[((sqrt(5) + 1)/2) \times z]$；
> - 若 $w = x$，则先手必败，否则先手必胜。

```
int n, m;
const double lorry = (sqrt(5.0) + 1.0) / 2.0;
int main() {
    cin >> n >> m;
    if(n < m) swap(n, m);
    int a = n - m;
    if(m == int(lorry * (double)a))
        cout << 0 << endl;
    else
        cout << 1 << endl;
}
```

斐波那契博弈

> 有一堆物品，两人轮流取物品，先手最少取一个，至多无上限，但不能把物品取完，之后每次取的物品数不能超过上一次取的物品数的二倍且至少为一件，取走最后一件物品的人获胜。

> 结论是：先手胜当且仅当n不是斐波那契数（n为物品总数）

区间不同数的个数

> 给出一个长度为 n 的数列，$a_1, a_2, \ldots, a_n$，有 q 个询问，每个询问给出数对 (i, j)，需要你给出 $a_i, a_{i+1}, \ldots, a_j$ 这一段中有多少不同的数字。

```
const int N = 1e6+5;
int n,m;
int tr[N],ap[N];
int flag[N],ans[N];
struct seq{
    int l,r,id;
}q[N];
bool cmp(const seq &a,const seq &b) {return a.r < b.r;}
int lowbit(int x) {return x & (-x);}
void add(int x,int v) {
    while(x <= n){
        tr[x] += v;
        x += lowbit(x);
    }
}
int query(int x) {
    int res = 0;
    while(x > 0){
        res += tr[x];
        x -= lowbit(x);
    }
    return res;
}
int main() {
    cin >> n;
    for(int i = 1;i <= n;++ i) cin >> ap[i];
    cin >> m;
    for(int i = 1;i <= m;++ i){
        cin >> q[i].l >> q[i].r;
        q[i].id = i;
    }
```

```
    sort(q+1,q+m+1,cmp);
    int nxt = 1;
    for(int i = 1;i <= m;++ i){
        for(int j = nxt;j <= q[i].r;++ j){
            if(flag[ap[j]]) add(flag[ap[j]],-1);
            flag[ap[j]] = j;
            add(j,1);
        }
        nxt = q[i].r + 1;
        ans[q[i].id] = query(q[i].r) - query(q[i].l-1);
    }
    for(int i = 1;i <= m;++ i) printf("%d\n",ans[i]);
    return 0;
}
```