SICNU ACM Template

renli

2018 年 7 月 30 日

目录

1 头文件

```cpp
#include <bits/stdc++.h>
using namespace std;
#define LL long long
#define INF 0x3f3f3f3f
#define clr(a, x) memset(a, x, sizeof(a))
const double eps = 1e-6;
const LL MOD = 1e9+7;
const int MAXN = 1e5 + 5;

int main()
{
#ifndef ONLINE_JUDGE
    //freopen("in.txt", "r", stdin);
#endif


    return 0;
}
```

2 数学

2.1 素数

2.1.1 素数筛

```cpp
/*
* 素数筛选，判断小于MAXN的数是不是素数。
* O(nloglogn)
* notprime为false表示是素数，true表示不是素数
*/
bool notprime[MAXN];//值为false表示素数，值为true表示非素数
void init()
{
    memset(notprime, false, sizeof(notprime));
    notprime[0] = notprime[1] = true;
    for(int i = 2; i < MAXN; i++)
        if(!notprime[i])
        {
            if(i > MAXN / i)continue; //防止后面i*i溢出(或者i,j用long long)
            //直接从i*i开始就可以，小于i倍的已经筛选过了,注意是j+=i
            for(int j = i * i; j < MAXN; j += i)
                notprime[j] = true;
        }
}
```

2.1.2 区间筛

```cpp
/*
* 对区间[a,b)内的整数执行筛法。
* 函数返回区间内素数个数
* isPrime[i-a]=true表示i是素数
* $a < b <= 10^12, b - a < 10^6
*/
bool isPrime_small[MAXN], isPrime[MAXN];
int prime[MAXN];
int segment_sieve(LL a, LL b)
{
    int tot = 0;
    for (LL i = 0; i * i < b; ++i)
        isPrime_small[i] = true;
    for (LL i = 0; i < b - a; ++i)
        isPrime[i] = true;
    for (LL i = 2; i * i < b; ++i)
        if (isPrime_small[i])
        {
            for (LL j = 2 * i; j * j < b; j += i)
                isPrime_small[j] = false;
            for (LL j = max(2LL, (a + i - 1) / i) * i; j < b; j += i)
                isPrime[j - a] = false;
        }
    for (LL i = 0; i < b - a; ++i)
        if (isPrime[i]) prime[tot++] = i + a;
    return tot;
}
```

2.1.3 Miller Rabin 素数判断

```cpp
/*
* O(slogn) 内判定 2^63 内的数是不是素数, s 为测定次数
*/
LL qmul(LL x, LL y, LL mod) // 乘法防止溢出， 如果p * p不爆LL的话可以直接乘； O(1)乘法或者转化成二进制加法
{
    return (x * y - (long long)(x / (long double)mod * y + 1e-3) * mod + mod) % mod;
}
LL qpow(LL a, LL b, LL n) //快速幂取模 a^b%n
{
    LL ans = 1;
    while (b)
    {
        if (b & 1)
            ans = qmul(ans, a, n);
        a = qmul(a, a, n);
        b >>= 1;
    }
    return ans;
}

bool Miller_Rabin(LL n, int s)
{
    if (n == 2)
        return 1;
    if (n < 2 || !(n & 1))
        return 0;
    int t = 0;
    LL x, y, u = n - 1;
    while ((u & 1) == 0)
        t++, u >>= 1;
    for (int i = 0; i < s; i++)
    {
        LL a = rand() % (n - 1) + 1;
        LL x = qpow(a, u, n);
        for (int j = 0; j < t; j++)
        {
            LL y = qmul(x, x, n);
            if (y == 1 && x != 1 && x != n - 1)
                return

                    x = y;
```

```cpp
        }
        if (x != 1)
            return 0;
    }
    return 1;
}
```

2.2 高斯消元

$A^{-1} * |A| = (A^{*})$

2.2.1 求逆矩阵

```cpp
int a[MAXN][MAXN];
int ni[MAXN][MAXN]; // 逆矩阵

void solve(int n)
{
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= n; j++)
            ni[i][j] = (i == j); // 初始化
    int det = 1; // |A|
    for(int i = 1; i <= n; i++)
    {
        int t = i;
        for(int k = i; k <= n; k++)
            if(a[k][i]) t = k;
        if(t != i) det *= -1;
        for(int j = 1; j <= n; j++)
        {
            swap(a[i][j], a[t][j]);
            swap(ni[i][j], ni[t][j]);
        }
        det = 1ll * a[i][i] * det % MOD;
        int inv = qpow(a[i][i], MOD - 2);
        for(int j = 1; j <= n; j++)
        {
            a[i][j] = 1ll * inv * a[i][j] % MOD;
            ni[i][j] = 1ll * inv * ni[i][j] % MOD;
        }
        for(int k = 1; k <= n; k++)
        {
            if(k == i) continue;
            int tmp = a[k][i];
            for(int j = 1; j <= n; j++)
            {
                a[k][j] = (a[k][j] - 1ll * a[i][j] * tmp % MOD + MOD) % MOD;
                ni[k][j] = (ni[k][j] - 1ll * ni[i][j] * tmp % MOD + MOD) % MOD;
            }
        }
    }
    det = (det + MOD) % MOD; // |A|
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= n; j++)
            ni[i][j] = 1ll * det * ni[i][j] % MOD; // 伴随矩阵
}
```

2.2.2 解 01 方程组

```cpp
//有equ个方程，var个变元。增广矩阵行数为equ,列数为var+1,分别为0到var
int a[MAXN][MAXN];
int b[MAXN][MAXN];
//返回值为-1表示无解，为0是唯一解，否则返回自由变元个数
int Gauss(int equ, int var)
{
    int max_r, col, k;
    for(k = 0, col = 0 ; k < equ && col < var ; k++, col++)
    {
        max_r = k;
        for(int i = k + 1; i < equ; i++)
        {
            if(abs(a[i][col]) > abs(a[max_r][col]))
                max_r = i;
        }
        if(a[max_r][col] == 0)
        {
            k--;
            continue;
        }
        if(max_r != k)
        {
            for(int j = col; j < var + 1; j++)
                swap(a[k][j], a[max_r][j]);
        }
        for(int i = k + 1; i < equ; i++)
        {
            if(a[i][col] != 0)
            {
                for(int j = col; j < var + 1; j++)
                    a[i][j] ^= a[k][j];
            }
        }
    }
    return k;
}
```

2.3 矩阵快速幂

$$F(i) = \begin{cases} F(i-1)+F(i-2)+i^3+i^2+i+1 & i>1 \\ 0 & i=0 \\ 1 & i=1 \end{cases} \text{ 求 F(i)}$$

$$\begin{pmatrix} F(i) \\ F(i-1) \\ i^3 \\ i^2 \\ i \\ 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 & 1 & 4 & 6 & 4 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 3 & 3 & 1 \\ 0 & 0 & 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} F(i-1) \\ F(i-2) \\ (i-1)^3 \\ (i-1)^2 \\ (i-1) \\ 1 \end{pmatrix}$$

```cpp
LL aaa[MAXN] =
{
    1, 1, 1, 4, 6, 4,
    1, 0, 0, 0, 0, 0,
    0, 0, 1, 3, 3, 1,
    0, 0, 0, 1, 2, 1,
    0, 0, 0, 0, 1, 1,
    0, 0, 0, 0, 0, 1
};
struct matrix
{
    LL a[6][6];
};

```

```cpp
matrix mat_mul(matrix x, matrix y)
{
    matrix res;
    memset(res.a, 0, sizeof(res.a));
    for(int i = 0; i < 6; i++)
        for(int j = 0; j < 6; j++)
            for(int k = 0; k < 6; k++)
                res.a[i][j] = (res.a[i][j] + x.a[i][k] * y.a[k][j]) % MOD;
    return res;
}

matrix mat_pow(LL n)
{
    matrix c, res; // res = c ^ n
    int ind = 0;
    for(int i = 0; i < 6; i++)
        for(int j = 0; j < 6; j++)
            c.a[i][j] = aaa[ind++];
    memset(res.a, 0, sizeof(res.a));
    for(int i = 0; i < 6; i++)
        res.a[i][i] = 1;
    while(n)
    {
        if(n & 1)
            res = mat_mul(res, c);
        c = mat_mul(c, c);
        n = n >> 1;
    }
    return res;
}
```

3 字符串

3.1 KMP

```cpp
/*
* next[]的含义：x[i-next[i]...i-1]=x[0...next[i]-1]
* next[i]为满足x[i-z...i-1]=x[0...z-1]的最大z值（就是x的自身匹配)
*/
void kmp_pre(char x[], int m, int next[])
{
    int i = 0, j = next[0] = -1;
    while(i < m)
    {
        while(-1 != j && x[i] != x[j])j = next[j];
        next[++i] = ++j;
    }
}
/*
* kmpNext[]的意思：next'[i]=next[next[...[next[i]]]] (直到next'[i]<0或者x[next'[i]]!=x[i])
* 这样的预处理可以快一些
*/
void preKMP(char x[], int m, int kmpNext[])
{
    int i = 0, j = kmpNext[0] = -1;
    while(i < m)
    {
        while(-1 != j && x[i] != x[j])j = kmpNext[j];
        if(x[++i] == x[++j])kmpNext[i] = kmpNext[j];
        else kmpNext[i] = j;
    }
}
/*
* 返回x在y中出现的次数，可以重叠
*/
int next[10010];
int KMP_Count(char x[], int m, char y[], int n)
{
    //x是模式串，y是主串
    int i, j, ans = 0;
    //preKMP(x,m,next);
    kmp_pre(x, m, next);
    i = j = 0;
    while(i < n)
    {
        while(-1 != j && y[i] != x[j])j = next[j];
        i++, j++;
        if(j >= m)
        {
            ans++;
            j = next[j];
        }
    }
    return ans;
}
```

3.2 扩展 KMP

```cpp
//next[i]:x[i...m-1]与x[0...m-1]的最长公共前缀
//extend[i]:y[i...n-1]与x[0...m-1]的最长公共前缀
void pre_EKMP(char x[], int m, int next[])
{
    next[0] = m;
    int j = 0;
    while(j + 1 < m && x[j] == x[j + 1])j++;
    next[1] = j;
    int k = 1;
    for(int i = 2; i < m; i++)
    {
        int p = next[k] + k - 1;
        int L = next[i - k];
        if(i + L < p + 1)next[i] = L;
        else
        {
            j = max(0, p - i + 1);
            while(i + j < m && x[i + j] == x[j])j++;
            next[i] = j;
            k = i;
        }
    }
}
void EKMP(char x[], int m, char y[], int n, int next[], int extend[])
{
    pre_EKMP(x, m, next);
    int j = 0;
    while(j < n && j < m && x[j] == y[j])j++;
    extend[0] = j;
    int k = 0;
    for(int i = 1; i < n; i++)
    {
        int p = extend[k] + k - 1;
        int L = next[i - k];
        if(i + L < p + 1)extend[i] = L;
        else
```

```
        {
            j = max(0, p - i + 1);
            while(i + j < n && j < m && y[i + j] == x[j
                ])j++;
            extend[i] = j;
            k = i;
        }
    }
}
```

3.3 Manacher 最长回文子串

```
/*
* O(n)求最长回文子串
*/
char Ma[MAXN * 2];
int Mp[MAXN * 2];
void Manacher(char s[], int len)
{
    int l = 0;
    Ma[l++] = '$';
    Ma[l++] = '#';
    for(int i = 0; i < len; i++)
    {
        Ma[l++] = s[i];
        Ma[l++] = '#';
    }
    Ma[l] = 0;
    int mx = 0, id = 0;
    for(int i = 0; i < l; i++)
    {
        Mp[i] = mx > i ? min(Mp[2 * id - i], mx - i) :
             1;
        while(Ma[i + Mp[i]] == Ma[i - Mp[i]])Mp[i]++;
        if(i + Mp[i] > mx)
        {
            mx = i + Mp[i];
            id = i;
        }
    }
}
/*
* abaaba
* i: 0 1 2 3 4 5 6 7 8 9 10 11 12 13
* Ma[i]: $ # a # b # a # a $ b # a #
* Mp[i]: 1 1 2 1 4 1 2 7 2 1 4 1 2 1
*/
char s[MAXN];
int main()
{
    while(scanf("%s", s) == 1)
    {
        int len = strlen(s);
        Manacher(s, len);
        int ans = 0;
        for(int i = 0; i < 2 * len + 2; i++)
            ans = max(ans, Mp[i] - 1);
        printf("%d\n", ans);
    }
    return 0;
}
```

3.4 AC 自动机

```
/*
* 求目标串中出现了几个模式串
*/
struct Trie
{
    int next[500010][26], fail[500010], end[500010];
    int root, L;
    int newnode()
    {
        for(int i = 0; i < 26; i++)
            next[L][i] = -1;
        end[L++] = 0;
        return L - 1;
    }
    void init()
    {
        L = 0;
        root = newnode();
    }
    void insert(char buf[])
    {
        int len = strlen(buf);
        int now = root;
        for(int i = 0; i < len; i++)
        {
            if(next[now][buf[i] - 'a'] == -1)
                next[now][buf[i] - 'a'] = newnode();
            now = next[now][buf[i] - 'a'];
        }
        end[now]++;
    }
    void build()
    {
        queue<int>Q;
        fail[root] = root;
        for(int i = 0; i < 26; i++)
            if(next[root][i] == -1)
                next[root][i] = root;
            else
            {
                fail[next[root][i]] = root;
                Q.push(next[root][i]);
            }
        while( !Q.empty() )
        {
            int now = Q.front();
            Q.pop();
            for(int i = 0; i < 26; i++)
                if(next[now][i] == -1)
                    next[now][i] = next[fail[now]][i];
                else
                {
                    fail[next[now][i]] = next[fail[now
                        ]][i];
                    Q.push(next[now][i]);
                }
        }
    }
    int query(char buf[])
    {
        int len = strlen(buf);
        int now = root;
        int res = 0;
```

```
        for(int i = 0; i < len; i++)
        {
            now = next[now][buf[i] - 'a'];
            int temp = now;
            while( temp != root )
            {
                res += end[temp];
                end[temp] = 0;
                temp = fail[temp];
            }
        }
        return res;
    }
};
char buf[1000010];
Trie ac;

int solve(int n)
{
    scanf("%d", &n);
    ac.init();
    for(int i = 0; i < n; i++)
    {
        scanf("%s", buf);
        ac.insert(buf);
    }
    ac.build();
    scanf("%s", buf);
    printf("%d\n", ac.query(buf));
}
```

3.5 后缀数组

```
//倍增算法构造后缀数组,复杂度O(nlogn)
char s[MAXN];
int sa[MAXN], t[MAXN], t2[MAXN], c[MAXN], rank[MAXN], height[MAXN];
//n为字符串的长度,字符集的值为0~m-1
void build_sa(int m, int n)
{
    n++;
    int *x = t, *y = t2;
    //基数排序
    for (int i = 0; i < m; i++) c[i] = 0;
    for (int i = 0; i < n; i++) c[x[i] = s[i]]++;
    for (int i = 1; i < m; i++) c[i] += c[i - 1];
    for (int i = n - 1; ~i; i--) sa[--c[x[i]]] = i;
    for (int k = 1; k <= n; k <<= 1)
    {
        //直接利用sa数组排序第二关键字
        int p = 0;
        for (int i = n - k; i < n; i++) y[p++] = i;
        for (int i = 0; i < n; i++)
            if (sa[i] >= k) y[p++] = sa[i] - k;
        //基数排序第一关键字
        for (int i = 0; i < m; i++) c[i] = 0;
        for (int i = 0; i < n; i++) c[x[y[i]]]++;
        for (int i = 0; i < m; i++) c[i] += c[i - 1];
        for (int i = n - 1; ~i; i--) sa[--c[x[y[i]]]] = y[i];
        //根据sa和y数组计算新的x数组
        swap(x, y);
        p = 1;
        x[sa[0]] = 0;
        for (int i = 1; i < n; i++)
            x[sa[i]] = y[sa[i - 1]] == y[sa[i]] && y[sa[i - 1] + k] == y[sa[i] + k] ? p - 1 : p++;
        if (p >= n) break; //以后即使继续倍增，sa也不会改变，推出
        m = p; //下次基数排序的最大值
    }
    n--;
    int k = 0;
    for (int i = 0; i <= n; i++) rank[sa[i]] = i;
    for (int i = 0; i < n; i++)
    {
        if (k) k--;
        int j = sa[rank[i] - 1];
        while (s[i + k] == s[j + k]) k++;
        height[rank[i]] = k;
    }
}
int dp[MAXN][30];
void initrmq(int n)
{
    for (int i = 1; i <= n; i++)
        dp[i][0] = height[i];
    for (int j = 1; (1 << j) <= n; j++)
        for (int i = 1; i + (1 << j) - 1 <= n; i++)
            dp[i][j] = min(dp[i][j - 1], dp[i + (1 << (j - 1))][j - 1]);
}
int rmq(int l, int r)
{
    int k = 31 - __builtin_clz(r - l + 1);
    return min(dp[l][k], dp[r - (1 << k) + 1][k]);
}
int lcp(int a, int b)
{
    // 求两个后缀的最长公共前缀
    a = rank[a], b = rank[b];
    if (a > b) swap(a, b);
    return rmq(a + 1, b);
}
```

3.6 后缀自动机

```
struct SAM
{
    int len[MAXN << 1], link[MAXN << 1], ch[MAXN << 1][26];
    int sz, rt, last;
    int newnode(int x = 0)
    {
        len[sz] = x;
        link[sz] = -1;
        clr(ch[sz], -1);
        return sz++;
    }
    void init() { sz = last = 0, rt = newnode(); }
    void extend(int c)
    {
        int np = newnode(len[last] + 1);
        int p;
        for (p = last; ~p && ch[p][c] == -1; p = link[p]) ch[p][c] = np;
        if (p == -1)
```

```cpp
            link[np] = rt;
        else
        {
            int q = ch[p][c];
            if (len[p] + 1 == len[q])
                link[np] = q;
            else
            {
                int nq = newnode(len[p] + 1);
                memcpy(ch[nq], ch[q], sizeof(ch[q]));
                link[nq] = link[q], link[q] = link[np] = nq;
                for (; ~p && ch[p][c] == q; p = link[p])
                    ch[p][c] = nq;
            }
        }
        last = np;
    }
    int topcnt[MAXN], topsam[MAXN << 1];
    void sort()
    { // 加入串后拓扑排序
        clr(topcnt, 0);
        for (int i = 0; i < sz; i++) topcnt[len[i]]++;
        for (int i = 0; i < MAXN - 1; i++) topcnt[i + 1] += topcnt[i];
        for (int i = 0; i < sz; i++) topsam[--topcnt[len[i]]] = i;
    }
};
```

4 数据结构

4.1 RMQ

4.1.1 一维 RMQ

```cpp
/*
* 求区间最大最小值，数组下标从 1 开始。
* 预处理复杂度 O(nlogn),查询O(1)
*/
int mmax[MAXN][30], mmin[MAXN][30];
int a[MAXN], n;
void init()
{
    for (int i = 1; i <= n; i++) mmax[i][0] = mmin[i][0] = a[i];
    for (int j = 1; (1 << j) <= n; j++)
        for (int i = 1; i + (1 << j) - 1 <= n; i++)
        {
            mmax[i][j] = max(mmax[i][j - 1], mmax[i + (1 << (j - 1))][j - 1]);
            mmin[i][j] = min(mmin[i][j - 1], mmin[i + (1 << (j - 1))][j - 1]);
        }
}

int rmqMax(int l, int r)
{
    int k = 31 - __builtin_clz(r - l + 1);
    return max(mmax[l][k], mmax[r - (1 << k) + 1][k]);
}

int rmqMin(int l, int r)
{
    int k = 31 - __builtin_clz(r - l + 1);
    return min(mmin[l][k], mmin[r - (1 << k) + 1][k]);
}
```

4.1.2 二维 RMQ

```cpp
int dp[310][310][9][9];//最大值
int n, m;
void init()
{
    for (int i = 0; (1 << i) <= n; i++)
        for (int j = 0; (1 << j) <= m; j++)
        {
            if (i == 0 && j == 0) continue;
            for (int row = 1; row + (1 << i) - 1 <= n; row++)
                for (int col = 1; col + (1 << j) - 1 <= m; col++)
                    if (i)
                        dp[row][col][i][j] =
max(dp[row][col][i - 1][j], dp[row + (1 << (i - 1))][col][i - 1][j]);
                    else
                        dp[row][col][i][j] =
max(dp[row][col][i][j - 1], dp[row][col + (1 << (j - 1))][i][j - 1]);
        }
}
int rmq(int x1, int y1, int x2, int y2)
{
    int kx = 31 - __builtin_clz(x2 - x1 + 1);
    int ky = 31 - __builtin_clz(y2 - y1 + 1);
    int m1 = dp[x1][y1][kx][ky];
    int m2 = dp[x2 - (1 << kx) + 1][y1][kx][ky];
    int m3 = dp[x1][y2 - (1 << ky) + 1][kx][ky];
    int m4 = dp[x2 - (1 << kx) + 1][y2 - (1 << ky) + 1][kx][ky];
    return max(max(m1, m2), max(m3, m4));
}
```

4.2 线段树

4.2.1 宏定义

```cpp
#define lson rt << 1 // 左儿子
#define rson rt << 1 | 1 // 右儿子
#define Lson l, m, lson // 左子树
#define Rson m + 1, r, rson // 右子树
```

4.2.2 单点修改

```cpp
int sum[MAXN << 2]; // sum[rt]用于维护区间和
void PushUp(int rt)
{
    sum[rt] = sum[lson] + sum[rson];
}
void build(int l, int r, int rt)
{
    if (l == r)
    {
        scanf("%d", &sum[rt]); // 建立的时候直接输入叶节点
```

```
        return;
    }
    int m = (l + r) >> 1;
    build(Lson);
    build(Rson);
    PushUp(rt);
}
void update(int p, int add, int l, int r, int rt)
{
    if (l == r)
    {
        sum[rt] += add;
        return;
    }
    int m = (l + r) >> 1;
    if (p <= m) update(p, add, Lson);
    else update(p, add, Rson);
    PushUp(rt);
}
int query(int L, int R, int l, int r, int rt)
{
    if (L <= l && r <= R)
        return sum[rt];
    int m = (l + r) >> 1;
    int s = 0;
    if (L <= m) s += query(L, R, Lson);
    if (m < R) s += query(L, R, Rson);
    return s;
}
```

4.2.3 区间修改

```
int lazy[MAXN << 2], sum[MAXN << 2];
void PushUp(int rt)
{
    sum[rt] = sum[lson] + sum[rson];
}
void PushDown(int rt, int m)
{
    if (lazy[rt] == 0)
        return;
    lazy[lson] += lazy[rt];
    lazy[rson] += lazy[rt];
    sum[lson] += lazy[rt] * (m - (m >> 1));
    sum[rson] += lazy[rt] * (m >> 1);
    lazy[rt] = 0;
}
void build(int l, int r, int rt)
{
    lazy[rt] = 0;
    if (l == r)
    {
        scanf("%lld", &sum[rt]);
        return;
    }
    int m = (l + r) >> 1;
    build(Lson);
    build(Rson);
    PushUp(rt);
}
void update(int L, int R, int add, int l, int r, int
    rt)
{
    if (L <= l && r <= R)
    {
        lazy[rt] += add;
        sum[rt] += add * (r - l + 1);
        return;
    }
    PushDown(rt, r - l + 1);
    int m = (l + r) >> 1;
    if (L <= m) update(L, R, add, Lson);
    if (m < R) update(L, R, add, Rson);
    PushUp(rt);
}
int query(int L, int R, int l, int r, int rt)
{
    if (L <= l && r <= R)
        return sum[rt];
    PushDown(rt, r - l + 1);
    int m = (l + r) >> 1, ret = 0;
    if (L <= m) ret += query(L, R, Lson);
    if (m < R) ret += query(L, R, Rson);
    return ret;
}
```

4.3 分块

```
int n, block, num, l[MAXN], r[MAXN], belong[MAXN];

void build()
{
    block = sqrt(n);
    num = n / block;
    if(n % block) //除不尽，多出一块
        num++;
    for(int i = 1; i <= num; i++)
    {
        l[i] = (i - 1) * block + 1;
        r[i] = i * block;
    }
    r[num] = n;//制定后一块的右端点为n
    for(int i; i <= n; i++)
        belong[i] = (i - 1) / block + 1;
}
```

5 图论

5.1 最小生成树

5.1.1 并查集

```
int fa[MAXN], ra[MAXN];
void init(int n)
{
    clr(ra, 0);
    for(int i = 1; i <= n; i++)
        fa[i] = i;
}
int find(int x)
{
    return fa[x] != x ? fa[x] = find(fa[x]) : x;
}
void Union(int x, int y)
{
    x = find(x), y = find(y);
```

```
    if (x == y) return;
    if (ra[x] < ra[y])
        fa[x] = y;
    else
    {
        fa[y] = x;
        if (ra[x] == ra[y]) ra[x]++;
    }
}
```

5.1.2 Kruskal

```
const int MAXM = 10000; //最大边数

struct Edge
{
    int u, v, w;
} edge[MAXM]; //存储边的信息，包括起点/终点/权值
int tol;//边数，加边前赋值为0
void addedge(int u, int v, int w)
{
    edge[tol].u = u;
    edge[tol].v = v;
    edge[tol++].w = w;
}
bool cmp(Edge a, Edge b)
{
    //排序函数，讲边按照权值从小到大排序
    return a.w < b.w;
}

int Kruskal(int n)//传入点数，返回最小生成树的权值，如果
    不连通返回-1
{
    init(n);
    sort(edge, edge + tol, cmp);
    int cnt = 0; //计算加入的边数
    int ans = 0;
    for(int i = 0; i < tol; i++)
    {
        int u = edge[i].u;
        int v = edge[i].v;
        int w = edge[i].w;
        if(find(u) != find(v))
        {
            Union(u,v);
            ans += w;
            cnt++;
        }
        if(cnt == n - 1)break;
    }
    if(cnt < n - 1)return -1; //不连通
    else return ans;
}
```

5.1.3 Prim

```
/*
* Prim求MST
* 耗费矩阵cost[][]，标号从0开始，0~n-1
* 返回最小生成树的权值，返回-1表示原图不连通
*/
bool vis[MAXN];
int lowc[MAXN];
int Prim(int cost[][MAXN], int n) //点是0~n-1
{
    int ans = 0;
    memset(vis, false, sizeof(vis));
    vis[0] = true;
    for(int i = 1; i < n; i++)
        lowc[i] = cost[0][i];
    for(int i = 1; i < n; i++)
    {
        int minc = INF;
        int p = -1;
        for(int j = 0; j < n; j++)
            if(!vis[j] && minc > lowc[j])
            {
                minc = lowc[j];
                p = j;
            }
        if(minc == INF)return -1; //原图不连通
        ans += minc;
        vis[p] = true;
        for(int j = 0; j < n; j++)
            if(!vis[j] && lowc[j] > cost[p][j])
                lowc[j] = cost[p][j];
    }
    return ans;
}
```

5.2 最短路

5.2.1 Dijkstra

```
/*
* 复杂度O(ElogE)
* 注意对vector<Edge>E[MAXN]进行初始化后加边
*/
struct qnode
{
    int v, c;
    qnode(int _v = 0, int _c = 0): v(_v), c(_c) {}
    bool operator <(const qnode &r) const
    {
        return c > r.c;
    }
};
struct Edge
{
    int v, cost;
    Edge(int _v = 0, int _cost = 0): v(_v), cost(_cost
        ) {}
};
vector<Edge>E[MAXN];
bool vis[MAXN];
int dist[MAXN];
void Dijkstra(int n, int start) //点的编号从1开始
{
    memset(vis, false, sizeof(vis));
    for(int i = 1; i <= n; i++)dist[i] = INF;
    priority_queue<qnode>que;
    while(!que.empty())que.pop();
    dist[start] = 0;
    que.push(qnode(start, 0));
    qnode tmp;
    while(!que.empty())
```

```
    {
        tmp = que.top();
        que.pop();
        int u = tmp.v;
        if(vis[u])continue;
        vis[u] = true;
        for(int i = 0; i < E[u].size(); i++)
        {
            int v = E[tmp.v][i].v;
            int cost = E[u][i].cost;
            if(!vis[v] && dist[v] > dist[u] + cost)
            {
                dist[v] = dist[u] + cost;
                que.push(qnode(v, dist[v]));
            }
        }
    }
}
void addedge(int u, int v, int w)
{
    E[u].push_back(Edge(v, w));
}
```

5.2.2 SPFA

```
/*
*单源最短路SPFA
*时间复杂度 O(kE)
*这个是队列实现，有时候改成栈实现会更加快，很容易修改
*这个复杂度是不定的
*/
const int MAXN = 1010;
const int INF = 0x3f3f3f3f;
struct Edge
{
    int v;
    int cost;
    Edge(int _v = 0, int _cost = 0): v(_v), cost(_cost
        ) {}
};
vector<Edge>E[MAXN];
void addedge(int u, int v, int w)
{
    E[u].push_back(Edge(v, w));
}
bool vis[MAXN];//在队列标志
int cnt[MAXN];//每个点的入队列次数
int dist[MAXN];
bool SPFA(int start, int n)
{
    memset(vis, false, sizeof(vis));
    for(int i = 1; i <= n; i++)dist[i] = INF;
    vis[start] = true;
    dist[start] = 0;
    queue<int>que;
    que.push(start);
    memset(cnt, 0, sizeof(cnt));
    cnt[start] = 1;
    while(!que.empty())
    {
        int u = que.front();
        que.pop();
        vis[u] = false;
        for(int i = 0; i < E[u].size(); i++)
        {
            int v = E[u][i].v;
            if(dist[v] > dist[u] + E[u][i].cost)
            {
                dist[v] = dist[u] + E[u][i].cost;
                if(!vis[v])
                {
                    vis[v] = true;
                    que.push(v);
                    if(++cnt[v] > n)return false;
                    //cnt[i]为入队列次数，用来判定是否存在
                        负环回路
                }
            }
        }
    }
    return true;
}
```

5.2.3 Floyd

```
// ---
// O(n^3)求出任意两点间最短路
// 邻接矩阵存图需注意判断重边
// ---
int G[MAXN][MAXN];
void init(int n)
{
    clr(G, 0x3f);
    for (int i = 0; i < n; i++) G[i][i] = 0;
}
void add_edge(int u, int v, int w) { G[u][v] = min(G[
    u][v], w); }
void Floyd(int n)
{
    for (int k = 0; k < n; k++)
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                G[i][j] = min(G[i][j], G[i][k] + G[k][j
                    ]);
}
```

5.3 LCA

5.3.1 离线 Tarjan

```
/**
* Tarjan离线算法
* 时间复杂度O(n+q)
*/
int par[MAXN]; //并查集
int ans[MAXN]; //存储答案
vector<int> G[MAXN]; //邻接表
vector<int> query[MAXN], num[MAXN]; //存储查询信息
bool vis[MAXN]; //是否被遍历
inline void init(int n)
{
    for (int i = 1; i <= n; i++)
    {
        G[i].clear();
        query[i].clear();
        num[i].clear();
        par[i] = i;
```

```
        vis[i] = 0;
    }
}
int find(int x)
{
    return par[x] != x ? par[x] = find(par[x]) : x;
}
void Union(int x, int y)
{
    x = find(x), y = find(y);
    if (x == y) return;
    par[y] = x;
}

inline void addEdge(int u, int v)
{
    G[u].push_back(v);
}
inline void addQuery(int id, int u, int v)
{
    query[u].push_back(v), query[v].push_back(u);
    num[u].push_back(id), num[v].push_back(id);
}
void tarjan(int u)
{
    vis[u] = 1;
    for(int i = 0; i < G[u].size(); i++)
    {
        int v = G[u][i];
        if (vis[v]) continue;
        tarjan(v);
        Union(u, v);
    }
    for(int i = 0; i < query[u].size(); i++)
    {
        int v = query[u][i];
        if (!vis[v]) continue;
        ans[num[u][i]] = find(v);
    }
}
```

5.3.2 LCA 倍增法

```
/*
* LCA 在线算法
*/
const int DEG = 20;
struct Edge
{
    int to, next;
    int val;
} edge[MAXN * 2];
int head[MAXN], tot;
void addedge(int u, int v, int k)
{
    edge[tot].to = v;
    edge[tot].next = head[u];
    edge[tot].val = k;
    head[u] = tot++;
}
void init()
{
    tot = 0;
    memset(head, -1, sizeof(head));
}
int fa[MAXN][DEG];//fa[i][j]表示结点i的第2^j个祖先
int deg[MAXN];//深度数组
int dis[MAXN];//距离数组
void BFS(int root)
{
    queue<int> que;
    deg[root] = 0;
    fa[root][0] = root;
    que.push(root);
    while(!que.empty())
    {
        int tmp = que.front();
        que.pop();
        for(int i = 1; i < DEG; i++)
            fa[tmp][i] = fa[fa[tmp][i - 1]][i - 1];
        for(int i = head[tmp]; i != -1; i = edge[i].next)
        {
            int v = edge[i].to;
            if(v == fa[tmp][0])continue;
            deg[v] = deg[tmp] + 1;
            dis[v] = dis[tmp]+edge[i].val;
            fa[v][0] = tmp;
            que.push(v);
        }
    }
}
int LCA(int u, int v)
{
    if(deg[u] > deg[v])swap(u, v);
    int hu = deg[u], hv = deg[v];
    int tu = u, tv = v;
    for(int det = hv - hu, i = 0; det ; det >>= 1, i++)
        if(det & 1)
            tv = fa[tv][i];
    if(tu == tv)return tu;
    for(int i = DEG - 1; i >= 0; i--)
    {
        if(fa[tu][i] == fa[tv][i])
            continue;
        tu = fa[tu][i];
        tv = fa[tv][i];
    }
    return fa[tu][0];
}
bool flag[MAXN];

void solve()
{
    int n, q;
    scanf("%d%d", &n, &q);
    init();
    clr(flag, 0);
    for(int i = 1; i < n; i++)
    {
        int a, b, k;
        scanf("%d%d%d", &a, &b, &k);
        addedge(a, b, k);
        addedge(b, a, k);
    }
    BFS(1);
    for(int i = 1; i <= q; i++)
    {
```

```cpp
        int u, v;
        scanf("%d%d", &u, &v);
        printf("%d\n", dis[u]+dis[v]-2*dis[LCA(u, v)])
            ;
    }
}
```

5.4 拓扑排序

```cpp
// ---
// 存图前记得初始化
// Ans排序结果，G邻接表，deg入度，map用于判断重边
// 排序成功返回1，存在环返回0
// ---
int Ans[MAXN];
vector<int> G[MAXN];
int deg[MAXN];
map<pair<int, int>, bool> S;
void init(int n)
{
    S.clear();
    for (int i = 0; i < n; i++) G[i].clear();
    clr(deg, 0), clr(Ans, 0);
}
void add_edge(int u, int v)
{
    if (S[make_pair(u, v)]) return;
    G[u].push_back(v), S[make_pair(u, v)] = 1, deg[v
        ]++;
}
bool Toposort(int n)
{
    int tot = 0;
    queue<int> q;
    for (int i = 0; i < n; ++i)
        if (deg[i] == 0) q.push(i);
    while (!q.empty())
    {
        int u = q.front();
        q.pop();
        Ans[tot++] = u;
        for(int i = 0; i < G[u].size(); i++)
        {
            int v = G[u][i];
            if(--deg[v] == 0)
                q.push(v);
        }
    }
    if (tot < n - 1) return false;
    return true;
}
```

5.5 网络流

5.5.1 建模技巧

建模技巧

二分图带权最大独立集。给出一个二分图，每个结点上有一个正权值。要求选出一些点，使得这些点之间没有边相连，且权值和最大。

解： 在二分图的基础上添加源点 S 和汇点 T，然后从 S 向所有 X 集合中的点连一条边，所有 Y 集合中的点向 T 连一条边，容量均为该点的权值。X 结点与 Y 结点之间的边的容量均为无穷大。这样，对于图中的任意一个割，将割中的边对应的结点删掉就是一个符合要求的解，权和为所有权减去割的容量。因此，只需要求出最小割，就能求出最大权和。

公平分配问题。把 m 个任务分配给 n 个处理器。其中每个任务有两个候选处理器，可以任选一个分配。要求所有处理器中，任务数最多的那个处理器所分配的任务数尽量少。不同任务的候选处理器集 $\{p_1, p_2\}$ 保证不同。

解： 本题有一个比较明显的二分图模型，即 X 结点是任务，Y 结点是处理器。二分答案 x，然后构图，首先从源点 S 出发向所有的任务结点引一条边，容量等于 1，然后从每个任务结点出发引两条边，分别到达它所能分配到的两个处理器结点，容量为 1，最后从每个处理器结点出发引一条边到汇点 T，容量为 x，表示选择该处理器的任务不能超过 x。这样网络中的每个单位流量都是从 S 流到一个任务结点，再到处理器结点，最后到汇点 T。只有当网络中的总流量等于 m 时才意味着所有任务都选择了一个处理器。这样，我们通过 $O(\log m)$ 次最大流便算出了答案。

区间 k 覆盖问题。数轴上有一些带权值的左闭右开区间。选出权和尽量大的一些区间，使得任意一个数最多被 k 个区间覆盖。

解： 本题可以用最小费用流解决，构图方法是把每个数作为一个结点，然后对于权值为 w 的区间 $[u, v)$ 加边 $u \to v$，容量为 1，费用为 $-w$。再对所有相邻的点加边 $i \to i+1$，容量为 k，费用为 0。最后，求最左点到最右点的最小费用最大流即可，其中每个流量对应一组互不相交的区间。如果数值范围太大，可以先进行离散化。

最大闭合子图。给定带权图 G（权值可正可负），求一个权和最大的点集，使得起点在该点集中的任意弧，终点也在该点集中。

解： 新增附加源 s 和附加汇 t，从 s 向所有正权点引一条边，容量为权值；从所有负权点向汇点引一条边，容量为权值的相反数。求出最小割以后，$S - \{s\}$ 就是最大闭合子图。

5.5.2 Edge

```cpp
// ---
// 最大流
// ---
struct Edge
{
    int from, to, cap, flow;
    Edge(int u, int v, int c, int f)
        : from(u), to(v), cap(c), flow(f) {}
};
// ---
// 最小费用流
// ---
struct Edge
{
    int from, to, cap, flow, cost;
    Edge(int u, int v, int c, int f, int w)
        : from(u), to(v), cap(c), flow(f), cost(w) {}
};
```

5.5.3 Dinic

```cpp
struct Dinic
{
    int n, m, s, t; //结点数，边数（包括反向弧），源点编
        号和汇点编号
```

```
    vector<Edge> edges; //边表。edge[e]和edge[e^1]互为反向弧
    vector<int> G[MAXN]; //邻接表，G[i][j]表示节点i的第j条边在e数组中的序号
    bool vis[MAXN]; //BFS使用
    int d[MAXN]; //从起点到i的距离
    int cur[MAXN]; //当前弧下标
    void init(int n)
    {
        this->n = n;
        for (int i = 0; i < n; i++) G[i].clear();
        edges.clear();
    }
    void AddEdge(int from, int to, int cap)
    {
        edges.push_back(Edge(from, to, cap, 0));
        edges.push_back(Edge(to, from, 0, 0));
        m = edges.size();
        G[from].push_back(m - 2);
        G[to].push_back(m - 1);
    }
    bool BFS()
    {
        clr(vis, 0);
        clr(d, 0);
        queue<int> q;
        q.push(s);
        d[s] = 0;
        vis[s] = 1;
        while (!q.empty())
        {
            int x = q.front();
            q.pop();
            for (int i = 0; i < G[x].size(); i++)
            {
                Edge& e = edges[G[x][i]];
                if (!vis[e.to] && e.cap > e.flow)
                {
                    vis[e.to] = 1;
                    d[e.to] = d[x] + 1;
                    q.push(e.to);
                }
            }
        }
        return vis[t];
    }
    int DFS(int x, int a)
    {
        if (x == t || a == 0) return a;
        int flow = 0, f;
        for (int& i = cur[x]; i < G[x].size(); i++)
        {
            //从上次考虑的弧
            Edge& e = edges[G[x][i]];
            if (d[x] + 1 == d[e.to] && (f = DFS(e.to, min(a, e.cap - e.flow))) > 0)
            {
                e.flow += f;
                edges[G[x][i] ^ 1].flow -= f;
                flow += f;
                a -= f;
                if (a == 0) break;
            }
        }
        return flow;
    }
    int Maxflow(int s, int t)
    {
        this->s = s;
        this->t = t;
        int flow = 0;
        while (BFS())
        {
            clr(cur, 0);
            flow += DFS(s, INF);
        }
        return flow;
    }
};
```

5.5.4 ISAP

```
struct ISAP
{
    int n, m, s, t; //结点数，边数（包括反向弧），源点编号和汇点编号
    vector<Edge> edges; //边表。edges[e]和edges[e^1]互为反向弧
    vector<int> G[MAXN]; //邻接表，G[i][j]表示结点i的第j条边在e数组中的序号
    bool vis[MAXN]; //BFS使用
    int d[MAXN]; //起点到i的距离
    int cur[MAXN]; //当前弧下标
    int p[MAXN]; //可增广路上的一条弧
    int num[MAXN]; //距离标号计数
    void init(int n)
    {
        this->n = n;
        for (int i = 0; i < n; i++) G[i].clear();
        edges.clear();
    }
    void AddEdge(int from, int to, int cap)
    {
        edges.push_back(Edge(from, to, cap, 0));
        edges.push_back(Edge(to, from, 0, 0));
        int m = edges.size();
        G[from].push_back(m - 2);
        G[to].push_back(m - 1);
    }
    int Augumemt()
    {
        int x = t, a = INF;
        while (x != s)
        {
            Edge& e = edges[p[x]];
            a = min(a, e.cap - e.flow);
            x = edges[p[x]].from;
        }
        x = t;
        while (x != s)
        {
            edges[p[x]].flow += a;
            edges[p[x] ^ 1].flow -= a;
            x = edges[p[x]].from;
        }
        return a;
    }
    void BFS()
    {
```

```cpp
        clr(vis, 0);
        clr(d, 0);
        queue<int> q;
        q.push(t);
        d[t] = 0;
        vis[t] = 1;
        while (!q.empty())
        {
            int x = q.front();
            q.pop();
            int len = G[x].size();
            for (int i = 0; i < len; i++)
            {
                Edge& e = edges[G[x][i]];
                if (!vis[e.from] && e.cap > e.flow)
                {
                    vis[e.from] = 1;
                    d[e.from] = d[x] + 1;
                    q.push(e.from);
                }
            }
        }
    }
    int Maxflow(int s, int t)
    {
        this->s = s;
        this->t = t;
        int flow = 0;
        BFS();
        clr(num, 0);
        for (int i = 0; i < n; i++)
            if (d[i] < INF) num[d[i]]++;
        int x = s;
        clr(cur, 0);
        while (d[s] < n)
        {
            if (x == t)
            {
                flow += Augumemt();
                x = s;
            }
            int ok = 0;
            for (int i = cur[x]; i < G[x].size(); i++)
            {
                Edge& e = edges[G[x][i]];
                if (e.cap > e.flow && d[x] == d[e.to] + 1)
                {
                    ok = 1;
                    p[e.to] = G[x][i];
                    cur[x] = i;
                    x = e.to;
                    break;
                }
            }
            if (!ok) //Retreat
            {
                int m = n - 1;
                for (int i = 0; i < G[x].size(); i++)
                {
                    Edge& e = edges[G[x][i]];
                    if (e.cap > e.flow) m = min(m, d[e.to]);
                }
                if (--num[d[x]] == 0) break; //gap优化
                num[d[x] = m + 1]++;
                cur[x] = 0;
                if (x != s) x = edges[p[x]].from;
            }
        }
        return flow;
    }
};
```

5.5.5 MCMF

```cpp
struct MCMF
{
    int n, m;
    vector<Edge> edges;
    vector<int> G[MAXN];
    int inq[MAXN]; //是否在队列中
    int d[MAXN]; //bellmanford
    int p[MAXN]; //上一条弧
    int a[MAXN]; //可改进量
    void init(int n)
    {
        this->n = n;
        for (int i = 0; i < n; i++) G[i].clear();
        edges.clear();
    }
    void AddEdge(int from, int to, int cap, int cost)
    {
        edges.push_back(Edge(from, to, cap, 0, cost));
        edges.push_back(Edge(to, from, 0, 0, -cost));
        m = edges.size();
        G[from].push_back(m - 2);
        G[to].push_back(m - 1);
    }
    bool BellmanFord(int s, int t, int& flow, LL& cost)
    {
        for (int i = 0; i < n; i++) d[i] = INF;
        clr(inq, 0);
        d[s] = 0;
        inq[s] = 1;
        p[s] = 0;
        a[s] = INF;
        queue<int> q;
        q.push(s);
        while (!q.empty())
        {
            int u = q.front();
            q.pop();
            inq[u] = 0;
            for (int i = 0; i < G[u].size(); i++)
            {
                Edge& e = edges[G[u][i]];
                if (e.cap > e.flow && d[e.to] > d[u] + e.cost)
                {
                    d[e.to] = d[u] + e.cost;
                    p[e.to] = G[u][i];
                    a[e.to] = min(a[u], e.cap - e.flow);
                    if (!inq[e.to])
                    {
                        q.push(e.to);
                        inq[e.to] = 1;
                    }
```

```
                }
            }
        }
        if (d[t] == INF) return false; // 当没有可增广的路时退出
        flow += a[t];
        cost += (LL)d[t] * (LL)a[t];
        for (int u = t; u != s; u = edges[p[u]].from)
        {
            edges[p[u]].flow += a[t];
            edges[p[u] ^ 1].flow -= a[t];
        }
        return true;
    }
    int MincostMaxflow(int s, int t, LL& cost)
    {
        int flow = 0;
        cost = 0;
        while (BellmanFord(s, t, flow, cost));
        return flow;
    }
};
```

6 计算几何

6.1 基本函数

6.1.1 定义点和线

```
const double PI = acos(-1.0);
#define zero(x) ((fabs(x) < eps ? 1 : 0))
int sgn(double x)
{
    if(fabs(x) < eps)return 0;
    if(x < 0)return -1;
    else return 1;
}

struct point
{
    double x, y;
    point(double a = 0, double b = 0)
    {
        x = a, y = b;
    }
    point operator-(const point &b) const
    {
        return point(x - b.x, y - b.y);
    }
    point operator+(const point &b) const
    {
        return point(x + b.x, y + b.y);
    }
    // 两点是否重合
    bool operator==(point &b)
    {
        return zero(x - b.x) && zero(y - b.y);
    }
    // 点积(以原点为基准)
    double operator*(const point &b) const
    {
        return x * b.x + y * b.y;
    }
    // 叉积(以原点为基准)
    double operator^(const point &b) const
    {
        return x * b.y - y * b.x;
    }
    // 绕P点逆时针旋转a弧度后的点
    point rotate(point b, double a)
    {
        double dx, dy;
        (*this - b).split(dx, dy);
        double tx = dx * cos(a) - dy * sin(a);
        double ty = dx * sin(a) + dy * cos(a);
        return point(tx, ty) + b;
    }
    // 点坐标分别赋值到a和b
    void split(double &a, double &b)
    {
        a = x, b = y;
    }
};
struct line
{
    point s, e;
    line() {}
    line(point _s, point _e)
    {
        s = _s;
        e = _e;
    }
    //两直线相交求交点
    //第一个值为0表示直线重合，为1表示平行，为0表示相交,为2是相交
    //只有第一个值为2时，交点才有意义
    pair<int, point> operator &(const line &b)const
    {
        point res = s;
        if(sgn((s - e) ^ (b.s - b.e)) == 0)
        {
            if(sgn((s - b.e) ^ (b.s - b.e)) == 0)
                return make_pair(0, res); //重合
            else return make_pair(1, res); //平行
        }
        double t = ((s - b.s) ^ (b.s - b.e)) / ((s - e) ^ (b.s - b.e));
        res.x += (e.x - s.x) * t;
        res.y += (e.y - s.y) * t;
        return make_pair(2, res);
    }
};
```

6.1.2 两点间距离

```
//*两点间距离
double dist(point a, point b)
{
    return sqrt((a - b) * (a - b));
}
```

6.1.3 线段相交

```
//*判断线段相交
bool inter(line l1, line l2)
{
    return
```

```
        max(l1.s.x, l1.e.x) >= min(l2.s.x, l2.e.x) &&
        max(l2.s.x, l2.e.x) >= min(l1.s.x, l1.e.x) &&
        max(l1.s.y, l1.e.y) >= min(l2.s.y, l2.e.y) &&
        max(l2.s.y, l2.e.y) >= min(l1.s.y, l1.e.y) &&
        sgn((l2.s - l1.e) ^ (l1.s - l1.e)) * sgn((l2.e-l1.e) ^ (l1.s - l1.e)) <= 0 &&
        sgn((l1.s - l2.e) ^ (l2.s - l2.e)) * sgn((l1.e-l2.e) ^ (l2.s - l2.e)) <= 0;
}
```

6.1.4 直线和线段相交

```
//判断直线和线段相交
bool Seg_inter_line(line l1, line l2) //判断直线l1和线段l2是否相交
{
    return sgn((l2.s - l1.e) ^ (l1.s - l1.e)) * sgn((l2.e-l1.e) ^ (l1.s - l1.e)) <= 0;
}
```

6.1.5 点到直线距离

```
//点到直线距离
//返回为result,是点到直线最近的点
point PointToLine(point P, line L)
{
    point result;
    double t = ((P - L.s) * (L.e-L.s)) / ((L.e-L.s) * (L.e-L.s));
    result.x = L.s.x + (L.e.x - L.s.x) * t;
    result.y = L.s.y + (L.e.y - L.s.y) * t;
    return result;
}
```

6.1.6 点到线段距离

```
//点到线段的距离
//返回点到线段最近的点
point NearestPointToLineSeg(point P, line L)
{
    point result;
    double t = ((P - L.s) * (L.e-L.s)) / ((L.e-L.s) * (L.e-L.s));
    if(t >= 0 && t <= 1)
    {
        result.x = L.s.x + (L.e.x - L.s.x) * t;
        result.y = L.s.y + (L.e.y - L.s.y) * t;
    }
    else
    {
        if(dist(P, L.s) < dist(P, L.e))
            result = L.s;
        else result = L.e;
    }
    return result;
}
```

6.1.7 判断点在线段上

```
//*判断点在线段上
bool OnSeg(point P, line L)
{
    return
        sgn((L.s - P) ^ (L.e-P)) == 0 &&
        sgn((P.x - L.s.x) * (P.x - L.e.x)) <= 0 &&
        sgn((P.y - L.s.y) * (P.y - L.e.y)) <= 0;
}
```

6.2 多边形

6.2.1 计算多边形面积

```
//计算多边形面积
//点的编号从0~n-1
double CalcArea(point p[], int n)
{
    double res = 0;
    for(int i = 0; i < n; i++)
        res += (p[i] ^ p[(i + 1) % n]) / 2;
    return fabs(res);
}
```

6.2.2 判断点在凸多边形内

```
//*判断点在凸多边形内
//点形成一个凸包，而且按逆时针排序（如果是顺时针把里面的<0改为>0)
//点的编号:0~n-1
//返回值:
//-1:点在凸多边形外
//0:点在凸多边形边界上
//1:点在凸多边形内
int inConvexPoly(point a, point p[], int n)
{
    for(int i = 0; i < n; i++)
    {
        if(sgn((p[i] - a) ^ (p[(i + 1) % n] - a)) < 0)
            return -1;
        else if(OnSeg(a, line(p[i], p[(i + 1) % n])))
            return 0;
    }
    return 1;
}
```

6.2.3 判断点在任意多边形内

```
//*判断点在任意多边形内
//射线法，poly[]的顶点数要大于等于3,点的编号0~n-1
//返回值
//-1:点在凸多边形外
//0:点在凸多边形边界上
//1:点在凸多边形内
int inPoly(point p, point poly[], int n)
{
    int cnt;
    line ray, side;
    cnt = 0;
    ray.s = p;
```

```
    ray.e.y = p.y;
    ray.e.x = -100000000000.0;//-INF,注意取值防止越界
    for(int i = 0; i < n; i++)
    {
        side.s = poly[i];
        side.e = poly[(i + 1) % n];
        if(OnSeg(p, side))return 0;
        //如果平行轴则不考虑
        if(sgn(side.s.y - side.e.y) == 0)
            continue;
        if(OnSeg(side.s, ray))
        {
            if(sgn(side.s.y - side.e.y) > 0)cnt++;
        }
        else if(OnSeg(side.e, ray))
        {
            if(sgn(side.e.y - side.s.y) > 0)cnt++;
        }
        else if(inter(ray, side))
            cnt++;
    }
    if(cnt % 2 == 1)return 1;
    else return -1;
}
```

6.2.4 判断凸多边形

```
//判断凸多边形
//允许共线边
//点可以是顺时针给出也可以是逆时针给出
//点的编号1~n-1
bool isconvex(point poly[], int n)
{
    bool s[3];
    memset(s, false, sizeof(s));
    for(int i = 0; i < n; i++)
    {
        s[sgn( (poly[(i + 1) % n] - poly[i]) ^ (poly[(
             i + 2) % n] - poly[i]) ) + 1] = true;
        if(s[0] && s[2])return false;
    }
    return true;
}
```

6.2.5 凸包

```
/*
* 求凸包, Graham算法
* 点的编号0~n-1
* 返回凸包结果Stack[0~top-1]为凸包的编号
*/
point lst[MAXN];
int Stack[MAXN], top;
//相对于lst[0]的极角排序
bool _cmp(point p1, point p2)
{
    double tmp = (p1 - lst[0]) ^ (p2 - lst[0]);
    if(sgn(tmp) > 0)return true;
    else if(sgn(tmp) == 0 && sgn(dist(p1, lst[0]) -
        dist(p2, lst[0])) <= 0)
        return true;
    else return false;
}
void Graham(int n)
{
    point p0;
    int k = 0;
    p0 = lst[0];
    //找最下边的一个点
    for(int i = 1; i < n; i++)
    {
        if( (p0.y > lst[i].y) || (p0.y == lst[i].y &&
            p0.x > lst[i].x) )
        {
            p0 = lst[i];
            k = i;
        }
    }
    swap(lst[k], lst[0]);
    sort(lst + 1, lst + n, _cmp);
    if(n == 1)
    {
        top = 1;
        Stack[0] = 0;
        return;
    }
    if(n == 2)
    {
        top = 2;
        Stack[0] = 0;
        Stack[1] = 1;
        return ;
    }
    Stack[0] = 0;
    Stack[1] = 1;
    top = 2;
    for(int i = 2; i < n; i++)
    {
        while(top > 1 && sgn((lst[Stack[top - 1]] -
            lst[Stack[top - 2]]) ^ (lst[i] - lst[Stack
            [top - 2]])) <= 0)
            top--;
        Stack[top++] = i;
    }
}
```

6.3 圆

6.3.1 外心

```
//过三点求圆心坐标
point waixin(point a, point b, point c)
{
    double a1 = b.x - a.x, b1 = b.y - a.y, c1 = (a1 *
        a1 + b1 * b1) / 2;
    double a2 = c.x - a.x, b2 = c.y - a.y, c2 = (a2 *
        a2 + b2 * b2) / 2;
    double d = a1 * b2 - a2 * b1;
    return point(a.x + (c1 * b2 - c2 * b1) / d, a.y +
        (a1 * c2 - a2 * c1) / d);
}
```

6.3.2 两圆相交的面积

```
//两个圆的公共部分面积
```

```cpp
double Area_of_overlap(point c1, double r1, point c2, double r2)
{
    double d = dist(c1, c2);
    if(r1 + r2 < d + eps)return 0;
    if(d < fabs(r1 - r2) + eps)
    {
        double r = min(r1, r2);
        return PI * r * r;
    }
    double x = (d * d + r1 * r1 - r2 * r2) / (2 * d);
    double t1 = acos(x / r1);
    double t2 = acos((d - x) / r2);
    return r1 * r1 * t1 + r2 * r2 * t2 - d * r1 * sin(t1);
}
```

7 动态规划

7.1 最长上升子序列

```cpp
// 序列下标从1开始，LIS()返回长度，序列存在lis[]中
int len, a[MAXN], b[MAXN], f[MAXN];
int Find(int p, int l, int r)
{
    while (l <= r)
    {
        int mid = (l + r) >> 1;
        if (a[p] > b[mid])
            l = mid + 1;
        else
            r = mid - 1;
    }
    return f[p] = l;
}
int LIS1(int lis[], int n)
{
    int len = 1;
    f[1] = 1, b[1] = a[1];
    for (int i = 2; i <= n; i++)
    {
        if (a[i] > b[len])
            b[++len] = a[i], f[i] = len;
        else
            b[Find(i, 1, len)] = a[i];
    }
    for (int i = n, t = len; i >= 1 && t >= 1; i--)
        if (f[i] == t) lis[--t] = a[i];
    return len;
}

// 简单写法(下标从0开始,只返回长度)
int dp[MAXN];
int LIS(int a[], int n)
{
    clr(dp, 0x3f);
    for (int i = 0; i < n; i++) *lower_bound(dp, dp + n, a[i]) = a[i];
    return lower_bound(dp, dp + n, INF) - dp;
}
```

7.2 数位 dp

```cpp
int a[20];
ll dp[20][state];//不同题目状态不同
ll dfs(int pos,/*state变量*/, bool lead/*前导零*/, bool limit/*数位上界变量*/) //不是每个题都要判断前导零
{
    //递归边界，既然是按位枚举，最低位是0，那么pos==-1说明这个数我枚举完了
    if(pos == -1) return 1; /*这里一般返回1，表示你枚举的这个数是合法的，那么这里就需要你在枚举时必须每一位都要满足题目条件，也就是说当前枚举到pos位，一定要保证前面已经枚举的数位是合法的。不过具体题目不同或者写法不同的话不一定要返回1 */
    //第二个就是记忆化(在此前可能不同题目还能有一些剪枝)
    if(!limit && !lead && dp[pos][state] != -1) return dp[pos][state];
    /*常规写法都是在没有限制的条件记忆化，这里与下面记录状态是对应，具体为什么是有条件的记忆化后面会讲*/
    int up = limit ? a[pos] : 9; //根据limit判断枚举的上界up;这个的例子前面用213讲过了
    ll ans = 0;
    //开始计数
    for(int i = 0; i <= up; i++) //枚举，然后把不同情况的个数加到ans就可以了
    {
        if() ...
            else if()...
                ans += dfs(pos - 1,/*状态转移*/, lead && i == 0, limit && i == a[pos]) //最后两个变量传参都是这样写的
                /*这里还算比较灵活，不过做几个题就觉得这里也是套路了
                大概就是说，我当前数位枚举的数是i，然后根据题目的约束条件分类讨论
                去计算不同情况下的个数，还有要根据state变量来保证i的合法性，比如题目
                要求数位上不能有62连续出现,那么就是state就是要保存前一位pre,然后分类,
                前一位如果是6那么这意味就不能是2，这里一定要保存枚举的这个数是合法*/
    }
    //计算完，记录状态
    if(!limit && !lead) dp[pos][state] = ans;
    /*这里对应上面的记忆化，在一定条件下时记录，保证一致性，当然如果约束条件不需要考虑lead，这里就是lead就完全不用考虑了*/
    return ans;
}
ll solve(ll x)
{
    int pos = 0;
    while(x)//把数位都分解出来
    {
        a[pos++] = x % 10; //个人老是喜欢编号为[0,pos),看不惯的就按自己习惯来，反正注意数位边界就行
        x /= 10;
    }
    return dfs(pos - 1/*从最高位开始枚举*/,/*一系列状态 */, true, true); //刚开始最高位都是有限制并且有前导零的，显然比最高位还要高的一位视为0嘛
}
```

8 其他

8.1 Java

```java
import java.io.BufferedInputStream;
import java.math.BigInteger;
import java.util.Scanner;

public class Main {
public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    Scanner cin = new Scanner(new BufferedInputStream(System.in));
    // 使用cin进行输入的时候可能会比in快一些。
    while (in.hasNext()) { // 多组输入
        int x = in.nextInt();
        int n = in.nextInt();
        int k = in.nextInt();
        // 大数BigInteger
        BigInteger b = new BigInteger(x + "");
        BigInteger ans = BigInteger.valueOf(k);
        b = b.add(b); // b = b + b
        b = b.subtract(b); // b = b - b
        b = b.multiply(b); // b = b * b
        b = b.divide(b); // b = b / b
        b = b.remainder(b); // b = b % b
        b = b.pow(n); // b ^ n
        if (ans.compareTo(b) == 0) { // 比较两个大数
            System.out.println("equ");
        }
        // 二维
        BigInteger c[][] = new BigInteger[110][110];
        for (int i = 0; i <= 100; i++)
            for (int j = 0; j <= 100; j++)
                c[i][j] = BigInteger.valueOf(0); // 别忘了初始化
        String s = b.toString(k); // 大数转化成k进制的字符串
        System.out.println(s.charAt(0)); // s[0]
        char[] ch = s.toCharArray(); // String转化成char数组
        for (int i = 0; i < ch.length; i++) {
            System.out.println(ch[i]);
        }

    }
}
}
```

8.2 STL

8.2.1 优先队列

```cpp
// empty() 如果队列为空返回真
// pop() 删除对顶元素
// push() 加入一个元素
// size() 返回优先队列中拥有的元素个数
// top() 返回优先队列队顶元素
// 在默认的优先队列中，优先级高的先出队。在默认的 int 型中先出队的为较大的数。
priority_queue<int> q1;//大的先出队
priority_queue<int, vector<int>, greater<int> > q2; //小的先出队
// 自定义比较函数：
struct cmp
{
    bool operator()(int x, int y)
    {
        return x > y; // x小的优先级高
        //也可以写成其他方式，如： return p[x] > p[y];表示p[i]小的优先级高
    }
};
priority_queue<int, vector<int>, cmp> q;//定义方法
// 其中，第二个参数为容器类型。第三个参数为比较函数。
// 结构体排序：
struct node
{
    int x, y;
    friend bool operator < (node a, node b)
    {
        return a.x > b.x; //结构体中，x小的优先级高
    }
};
priority_queue<node> q;//定义方法
//在该结构中，y为值，x为优先级。
//通过自定义operator<操作符来比较元素中的优先级。
//在重载”<”时，最好不要重载”>”，可能会发生编译错误
```

8.3 SG 函数

8.3.1 解题模型

1. 把原游戏分解成多个独立的子游戏，则原游戏的 SG 函数值是它的所有子游戏的 SG 函数值的异或。
即 $sg(G) = sg(G1) \wedge sg(G2) \wedge ... \wedge sg(Gn)$。

2. 分别考虑没一个子游戏，计算其 SG 值。SG 值的计算方法：（重点）
1. 可选步数为 $1 - m$ 的连续整数，直接取模即可，$SG(x) = x\%(m+1)$;
2. 可选步数为任意步，$SG(x) = x$;
3. 可选步数为一系列不连续的数，用模板计算。

一般 DFS 只在打表解决不了的情况下用，首选打表预处理。

8.3.2 打表

```cpp
//f[]：可以取走的石子个数
//sg[]:0~n的SG函数值
//hash[]:mex{}
int f[MAXN], sg[MAXN], hash[MAXN];
void getSG(int n)
{
    int i, j;
    memset(sg, 0, sizeof(sg));
    for(i = 1; i <= n; i++)
    {
        memset(hash, 0, sizeof(hash));
        for(j = 1; f[j] <= i; j++)
            hash[sg[i - f[j]]] = 1;
        for(j = 0; j <= n; j++) //求mes{}中未出现的最小的非负整数
        {
            if(hash[j] == 0)
            {
                sg[i] = j;
                break;
            }
```

```
        }
    }
}
```

8.3.3 dfs

```
//注意 S数组要按从小到大排序 SG函数要初始化为-1 对于每个
    集合只需初始化1遍
//n是集合s的大小 S[i]是定义的特殊取法规则的数组
int s[110], sg[10010], n;
int SG_dfs(int x)
{
    int i;
    if(sg[x] != -1)
        return sg[x];
    bool vis[110];
    memset(vis, 0, sizeof(vis));
    for(i = 0; i < n; i++)
    {
        if(x >= s[i])
        {
            SG_dfs(x - s[i]);
            vis[sg[x - s[i]]] = 1;
        }
    }
    int e;
    for(i = 0;; i++)
        if(!vis[i])
        {
            e = i;
            break;
        }
    return sg[x] = e;
}
```

8.4 战术研究

- 读新题的优先级高于一切
- 读完题之后必须看一遍 clarification
- 交题之前必须看一遍 clarification
- 可能有 SPJ 的题目提交前也应该尽量做到与样例输出完全一致
- A 时需要检查 INF 是否设小
- 构造题不可开场做
- 每道题需至少有两个人确认题意
- 上机之前做法需得到队友确认
- 带有猜想性质的算法应放后面写
- 当发现题目不会做但是过了一片时应冲一发暴力
- 将待写的题按所需时间放入小根堆中，每次选堆顶的题目写
- 交完题目后立马打印随后让出机器
- 写题超过半小时应考虑是否弃题
- 细节、公式等在上机前应在草稿纸上准备好，防止上机后越写越乱
- 提交题目之前应检查 $solve(n,m)$ 是否等于 $solve(m,n)$
- 检查是否所有东西都已经清空
- 对于中后期题应该考虑一人写题，另一人在一旁辅助，及时发现手误
- 最后半小时不能慌张
- 对于取模的题，在输出之前一定要再取模一次进行保险

8.5 打表找规律方法

- 直接找规律
- 差分后找规律
- 找积性
- 点阵打表
- 相除
- 找循环节
- 凑量纲
- 猜想满足 $P(n)f(n) = Q(n)f(n-2) + R(n)f(n-1) + C$，其中 P,Q,R 都是关于 n 的二次多项式