- For 热身赛
- 日常
  - 文件读入读出
  - cin/cout加速
  - 扩栈
  - 数组作参数
  - 重写比较符
- 字符串操作
  - 字符串判断
  - 回文
    - 可以求出所有str[i][j]是否是回文串的方法
    - 最长回文子序列(O(n^2))
- 数学
  - 常用数字
  - 位运算
    - 判断一个数是否是二的倍数
    - 对二的n次方取余
  - 全排列
  - 欧几里得算法求最大公约数
  - 筛法求素数
    - 一般筛法
    - 线性筛法，O(n)
    - 判断一个数字是否为素数
  - 矩阵快速幂
  - 二分查找
  - 三分查找
  - 康托展开
  - HASH
  - 求两个数公因子个数/求数字的因子个数
  - 大数字求组合数——分解质因子
  - BigInteger
  - 乘积求模/次方求模
    - return (a * b) % m
    - return (a ^ b) % m
  - 欧拉函数
    - 直接求解欧拉函数
    - 筛选法打欧拉函数表
- 数据结构
  - 并查集
  - Tire树

For 热身赛

- 测试栈大小
- 测试是否有 `#define ONLINE_JUDGE`
- 熟练使用PC^2

日常

文件读入读出

```
#define LOCAL
#ifdef LOCAL
freopen("in.txt","r",stdin);
freopen("out.txt","w",stdout);
#endif
```

或者

```
#ifndef ONLINE_JUDGE
freopen("in.txt","r",stdin);
freopen("out.txt","w",stdout);
#endif
```

cin/cout 加速

```
#include <iostream>
int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(0);
    // IO
}
```

扩栈

```
#pragma comment(linker, "/STACK:102400000,102400000")
```

数组作参数

```
//一维数组
void f(int *a)
void f(int a[maxn]).
//二维数组
void f(int a[maxn][maxn])
```

重写比较符

```
bool operator < (const node &a)const{
```

```
        if(x1==a.x1)
            return x2<a.x2;
        return x1<a.x1;
    }
```

字符串操作

字符串判断

```
    strcmp//char的比较
```

回文

可以求出所有str[i][j]是否是回文串的方法

```
    void init() {
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= i; j++) {
                if (i == j || (str[i] == str[j] && (ok[j + 1][i - 1]
    || i - j == 1)))
                    ok[j][i] = 1;
                else ok[j][i] = 0;
            }
    }
```

最长回文子序列(O(n^2))

```
    for(i=l-1;i>=0;i--){
            dp[i][i]=1;
            for(j=i+1;j<l;j++){
                if(c[i]==c[j])
                    dp[i][j]=dp[i+1][j-1]+2;
                else
                    dp[i][j]=max(dp[i+1][j],dp[i][j-1]);
            }
        }
        printf("%d",dp[0][l-1]);
```

数学

常用数字

```
const double PI = acos(-1.0);
fabs//double求绝对值
atan(x)//求的是x的反正切，其返回值为[-pi/2,+pi/2]之间的一个数。
atan2(y,x)//求的是y/x的反正切，其返回值为[-pi,+pi]之间的一个数。
    //另外要注意的是，函数atan2(y,x)中参数的顺序是倒置的，atan2(y,x)计算的值相当于点(x,y)的角度值。
    //第二个能保存坐标的位置...一四象限是正的二三象限是负的。
floor(double x)//x下取整
cell(double x)//x上取整
pow(double x,double y)//x的y次方
```

位运算

判断一个数是否是二的倍数

```
bool isFactorialofTwo(int n){
    return n > 0 ? (n & (n - 1)) == 0 : false;
}
```

对二的n次方取余

```
int quyu(int m,int n){//n为2的次方
    return m & (n - 1);
}
```

全排列

```
next_permutation//生成n个元素的n!种不同排列。
prev_permutation//反着输出
```

欧几里得算法求最大公约数

欧几里德算法又称辗转相除法，用于计算两个正整数 a , b 的最大公约数。

其计算原理依赖于下面的定理：gcd(a,b) = gcd(b,a mod b) (a > b 且 a mod b 不为0)

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

筛法求素数

一般筛法

```
for(int i=2;i<=sqrt(mx);i++){
    if(a[i]==0)
    for(int j=2;i*j<=mx;j++){
        a[i*j]=1;
    }
}
```

线性筛法，O(n)

```
int prime[mx] = {0},num_prime = 0;
int isNotPrime[mx] = {1,1};
void solve() {
    for(int i=2 ; i<mx ; i++) {
        if(!isNotPrime[i])
            prime[num_prime++]=i;
        for(int j=0 ; j<num_prime && i*prime[j]<mx ; j++) {
            isNotPrime[i * prime[j]] = 1;
            if( !(i%prime[j]) )break;
        }
    }
    //for(int i=0;i<num_prime;i++)cout<<prime[i]<<endl;
}
```

判断一个数字是否为素数

```
bool prime(long long n){
    int l = sqrt(n);
    for(int i=2;i<=l;i++){//通常不用i*i的写法避免爆int
        if(n%i==0)return false;
    }
```

```
return n!=1;
}
```

矩阵快速幂

A为单位矩阵，B为构造的矩阵

```
#define LL long long
LL n,mod;
LL A1,A2,A3;
struct matrix{
    LL m[3][3];
};
matrix multiple(matrix A,matrix B){
    matrix ans;
    LL res;
    for(int i=0;i<3;i++)
        for(int j=0;j<3;j++){
            ans.m[i][j]=0;
            res=0;
            for(int k=0;k<3;k++){
                res+=(A.m[i][k]*B.m[k][j])%mod;
            }
            ans.m[i][j]=res;
        }
    return ans;
}
LL solve(){
    matrix A,B;
//init the matrix
    n--;
    while(n){
        if(n%2)
            A=multiple(A,B);
        B=multiple(B,B);
        n/=2;
    }
    return (A.m[0][0]*A1+A.m[0][1]*A2+A.m[0][2]*A3)%mod;
}
```

二分查找

函数lower_bound()在first和last中的前闭后开区间进行二分查找，返回大于或等于val的第一个元素位置。如果所有元素都小于val，则返回last的位置

```
lower_bound()//指向ai>=k的ai的最小的指针
upper_bound()//指向满足ai>k的ai的最小的指针
upper_bound( a , a+n , k ) - lower_bound ( a , a+n , k );//求长度为n的有序数组中k的个数
```

```
bool ok(int x){
//...
}
int solve(int l, int r){ /// [l,r)返回l    右边大一点
    int m;
    while (l + 1 < r){
        m = (l + r) >> 1;
        ok(m) ? l = m : r = m;
    }
    return l; //或 return calc(l);
}
int solve(int l, int r){ /// (l,r]返回r     左边小一点
int m;
    while (l + 1 < r){
        m = (l + r) >> 1;
        ok(m) ? r = m : l = m;
    }
 return r; //或 return calc(r);
}
int solve(double l, double r){ /// bool ok(double x)
    double m;
    int i;
    For(i, 100){
        m = (l + r) / 2.0;
        ok(m) ? l = m : r = m;
    }
    return l; //或 return calc(l);
}
```

三分查找

```
double solve(double l, double r){
    double tmp, m1, m2;
    while (l + eps < r){ /// *eps不用开太小
        tmp = (l + r) / 3;
        m1= l + tmp;
        m2= r - tmp;
        /// 若要在内部添加逗号表达式，注意在逗号表达式外面加一层括号
        calc(m1) > calc(m2) + eps ? l = m1 : r = m2; /// 求极小值
        //calc(m1) + eps < calc(m2) ? l = m1: r = m2; /// 求极大值
```

```
    }
return l; //return calc(l);
}
```

康托展开

康托展开指的是全排列到一个自然数的双射，即一个自然数对应全排列的一种情况。康托展开的公式是 X=an*(n-1)!+an-1*(n-2)!+...+ai*(i-1)!+...+a2*1!+a1*0! 其中，ai为当前未出现的元素中是排在第几个（从0开始）。

```
#include<cstdio>
const int fac[] = {1, 1, 2, 6, 24, 120, 720, 5040, 40320};///阶乘
int KT(int s[], int n){
    int i, j, cnt, sum;
    sum = 0;
    for (i = 0; i < n; ++i){
        cnt = 0;
        for (j = i + 1; j < n; ++j)
            if (s[j] < s[i]) ++cnt;
        sum += cnt * fac[n - i - 1];
    }
    return sum;
}
int main(){
    int a[] = {3, 5, 7, 4, 1, 2, 9, 6, 8};
    printf("%d\n", 1 + KT(a, sizeof(a) / sizeof(*a)));
///1+98884
}
```

逆展开：

```
#include<cstdio>
#include<cstring>
const int fac[] = {1, 1, 2, 6, 24, 120, 720, 5040, 40320};///阶乘
bool vis[10];
///n为ans大小，k为全排列的编码
void invKT(int ans[], int n, int k){
    int i, j, t;
    memset(vis, 0, sizeof(vis));
    --k;
    for (i = 0; i < n; ++i){
        t = k / fac[n - i - 1];
        for (j = 1; j <= n; j++)
            if (!vis[j]){
```

```
                if (t == 0) break;
                --t;
            }
        ans[i] = j, vis[j] = true;
        k %= fac[n - i - 1];///余数
    }
}
int main(){
    int a[10];
    invKT(a, 5, 16);
    for (int i = 0; i < 5; ++i)
        printf("%d ", a[i]);///1 4 3 5 2
}
```

HASH

p取一个较大素数，mod取一个大素数。
习惯上，p取一个6到8位的素数即可，mod一般取大素数 1e9+7（1000000007）或1e9+9（1000000009）
hash[l..r]=(hash[r]-hash[l-1]*(p^(r-1+1)))%mod
记得hash[l..r]可能有负数，所以当得到的hash[l..r]<0的时候，hash[l..r]+=mod，就好啦。

```
hash[i]=(hash[i-1]*p+idx(s[i]))%mod
hash[l..r]=(hash[r]-hash[l-1]*(p^(r-1+1)))%mod
```

1. unsigned long long hash[N];``hash[i]=hash[i-1]*p （自动取模）
2. hash[i]=(hash[i-1]*p+idx(s[i]))%mod
3. 双hash
   hash1[i]=(hash1[i-1]*p+idx(s[i]))%mod1
   hash2[i]=(hash2[i-1]*p+idx(s[i]))%mod2
   pair<hash1,hash2> 表示一个字符串！

求两个数公因子个数/求数字的因子个数

```
//这是求两个数a,b的公因子的个数的
//等价于求这两个数的最大公约数的因子的个数
//注意一下数据范围
#include <cstdio>
#include <cstring>
#include <iostream>
using namespace std;
#define N 1000
```

```
int    mark[N+1] = {0};
//【flag】然而并不知道mark数组是什么（质数的mark数组可成链
void InitPrimeTable(){
    int    i,j,k;
    for (i = 2; i-N-1; mark[i] = i + 1, i++);
    for (i = 2; i-N-1; mark[k] = N + 1, i = mark[i])
        for (k = i, j = mark[i]; j-N-1; j = mark[j])
            if (j%i) mark[k] = j, k = j;
}
int gcd(int a, int b){
    return b ? gcd(b, a%b) : a;
}
int main(){
    #ifndef ONLINE_JUDGE
    freopen("t.in","r",stdin);
    freopen("t.out","w",stdout);
    #endif
    int n;
    InitPrimeTable();
    for(int i=1;i<=20;i++)
        cout<<i<<" "<<mark[i]<<endl;
    scanf("%d", &n);//n组数据
    while (n--){
        int a, b, i, curr = 0, result = 1;
        int n;
        scanf("%d%d", &a, &b);
        n = gcd(a, b);//求最大公约数
        for (i = 2; i<N+1 && n >= i; i = mark[i]){
            int c = 0;
            while (n%i==0) ++c,n/=i;
            result *= c+1;
        }//这里在求n的因子个数
        if (n != 1) result *= 2;
        printf("%d\n", result);
    }
    return 0;
}
/*
9=3*3
那么因子个数是3
9=3^2
2+1为所求
如果
n = p1^a1*p2^a2...pk^ak;
那么因子个数为
(a1+1)*(a2+1)*...*(ak+1)
*/
```

大数字求组合数——分解质因子

```
map <int, LL> m;
//分解质因数
//k为1或-1
void fun(int n, int k){
    for (int i = 2; i <= sqrt(n * 1.0); i++){
        while (n % i == 0){
            n /= i;
            m[i] += k;
        }
    }
    if (n > 1){
        m[n] += k;
    }
}
//大数快速幂取模
LL quick_pow(LL a, LL b){
    LL ret = 1;
    while (b){
        if (b & 1){
            ret *= a;
            ret %= MOD;
        }
        b >>= 1;
        a *= a;
        a %= MOD;
    }
    return ret;
}
//求组合数
LL C(LL a, LL b){
    if (a < b || a < 0 || b < 0)
        return 0;
    m.clear();
    LL ret = 1;
    b = min(a - b, b);
    for (int i = 0; i < b; i++){
        fun(a - i, 1);
    }
    for (int i = b; i >= 1; i--){
        fun(i, -1);
    }
    ///以下计算出了具体的数
    for (__typeof(m.begin()) it = m.begin(); it != m.end();
it++){
        if ((*it).second != 0){
            ret *= quick_pow((*it).first, (*it).second);
            ret %= MOD;
        }
    }
```

```
    return ret;
}
```

BigInteger

```
#include<iostream>
#include<cstdio>
#include<cstring>
using namespace std;
const int maxlen = 10000;
class HP {
    public:
        int len,s[maxlen];
        HP() {(*this)=0;};
        HP(int inte) {(*this)= inte;};
        HP(const char *str) {(*this)=str;};
        friend ostream& operator<<(ostream &cout,const HP &x);
        HP operator=(int inte);
        HP operator=(const char *str);
        HP operator*(const HP &b);
        HP operator+(const HP &b);
        HP operator-(const HP &b);
        HP operator/(const HP &b);
        HP operator%(const HP &b);
        int Compare(const HP &b);
};
ostream& operator<<(ostream &cout,const HP &x){
    for (int i=x.len;i>=1;i--)
        cout<<x.s[i];
    return cout;
}
HP HP::operator=(const char *str){
    len=strlen(str);
    for (int i=1; i<=len;i++)
        s[i]=str[len-i]-'0';
    return *this;
}
HP HP::operator=(int inte){
    if(inte==0){
    len=1;
    s[1]=0;
    return (*this);
    };
    for(len=0;inte>0;){
        s[++len]=inte%10;
        inte/=10;
    };
```

```
    return (*this);
}
HP HP::operator*(const HP &b){
    int i,j;
    HP c;
    c.len=len+b.len;
    for (i=1;i<=c.len;i++)
        c.s[i] =0;
    for (i=1;i<=len;i++)
        for (j=1;j<=b.len;j++)
            c.s[i+j-1]+=s[i]*b.s[j];
    for ( i=1;i <c.len; i++){
        c.s[i+1]+=c.s[i]/10;
        c.s[i]%=10;
    }
    while (c.s[i]){
        c.s[i+1]=c.s[i]/10;
        c.s[i]%=10;
        i++;
    }
    while (i>1 && !c.s[i]) i--;
    c.len=i;
    return c;
}
HP HP::operator+(const HP &b){
    int i; HP c; c.s[1] =0;
    for (i=1; i<=len || i<=b.len || c.s[i] ;i++){
    if (i<=len) c.s[i]+=s[i];
    if (i<=b.len) c.s[i]+=b.s[i];
    c.s[i+1]=c.s[i]/10; c.s[i]%=10;
    }
    c.len=i-1;
    if (c.len==0) c.len=1;
    return c;
}
HP HP::operator-(const HP &b) {
    int i,j;
    HP c;
    for(i=1,j=0;i<=len;i++){
        c.s[i]= s[i]-j;
        if(i<=b.len) c.s[i]-=b.s[i];
        if (c.s[i]<0){
            j=1;
            c.s[i]+=10;
        }
        else j =0;
    }
    c.len=len;
    while (c.len>1 && !c.s[c.len]) c.len--;
    return c;
```

```
}
int HP::Compare(const HP &y){
    if ( len >y.len ) return 1;
    if ( len <y.len ) return -1;
    int i=len;
    while ((i>1)&&(s[i]==y.s[i])) i--;
    return s[i]-y.s[i];
}
HP HP::operator/(const HP &b){
    int i,j;
    HP d(0),c;
    for (i=len ; i>0; i--){
        if (!(d.len ==1&& d.s[1]==0)) {
            for (j=d.len; j>0; j--)
                d.s[j+1]=d.s[j];
            ++d.len;
        }
        d.s[1]=s[i];
        c.s[i]=0;
        while ((j=d.Compare(b))>=0){
            d=d-b;
            c.s[i]++;
            if (j==0) break;
        }
    }
    c.len=len;
    while((c.len>1)&&(c.s[c.len]==0))c.len--;
    return c;
}
HP HP::operator%(const HP &b){
    int i,j;
    HP d(0);
    for ( i=len ; i >0; i--){
        if(!(d.len ==1&& d.s[1]==0)){
            for ( j=d.len; j>0; j--)
                d.s[j+1]=d.s[j];
            ++d.len;
        }
        d.s[1]=s[i];
        while ((j=d.Compare(b))>=0){
            d=d-b;
            if (j==0) break;
        }
    }
    return d;
}
int main(){
    HP a=1051,b="20";
    cout<<a+b<<endl;
    cout<<a-b<<endl;
```

```
        cout<<a*b<<endl;
        cout<<a/b<<endl;
        cout<<a%b<<endl;
        if(a.Compare(b)>0)
            cout<<"a is bigger than b"<<endl;
        if(b.Compare(a)>0)
            cout<<"b is bigger than a"<<endl;
        return 0;
    }
```

乘积求模/次方求模

```
return (a * b) % m
```

```
    LL mod_mult(LL a, LL b, LL m){
        LL res = 0;
        LL exp = a % m;
        while (b){
            if (b & 1){
                res += exp;
                if (res > m) res -= m;
            }
            exp <<= 1;
            if (exp > m) exp -= m;
            b >>= 1;
        }
        return res;
    }
```

```
return (a ^ b) % m
```

```
    LL mod_exp(LL a, LL b, LL m){
        LL res = 1;
        LL exp = a % m;
        while (b){
            if (b & 1)
                res = mod_mult(res, exp, m);//Or res = res * exp %
    m;
            exp = mod_mult(exp, exp, m);//Or exp = exp * exp % m;
            b >>= 1;
        }
        return res;
    }
```

欧拉函数

直接求解欧拉函数

```
int euler(int n){ //返回euler(n)
     int res=n,a=n;
     for(int i=2;i*i<=a;i++){
         if(a%i==0){
             res=res/i*(i-1);
             //先进行除法是为了防止中间数据的溢出
             while(a%i==0) a/=i;
         }
     }
     if(a>1) res=res/a*(a-1);
     return res;
}
```

筛选法打欧拉函数表

```
#define Max 1000001
int euler[Max];
void Init(){
     euler[1]=1;
     for(int i=2;i<Max;i++)
       euler[i]=i;
     for(int i=2;i<Max;i++)
        if(euler[i]==i)
           for(int j=i;j<Max;j+=i)
              euler[j]=euler[j]/i*(i-1);//先进行除法是为了防止中间数
据的溢出
}
```

数据结构

并查集

```
int p[mx],h[mx],cas;
void init(){
    for(int i=1;i<=mx;i++){
    p[i]=i;
    h[i]=1;
    }
```

```
}
int get_fa(int x){
    return p[x] == x ? x : p[x]=get_fa(p[x]);
}
void get_union(int a,int b){
    int x = get_fa(a);
    int y = get_fa(b);
    if(x==y)return;
    if(h[x]<h[y])
        p[x]=y;
    else
        p[y]=x;
    h[x]+=h[y];
    h[y]=h[x];
}
```

Tire树

```
struct node{
    int num;
    node *next[26];
}root;
void insert(char *word) {
    node *p = &root;
    int l=strlen(word);
    int temp;
    for(int i=0; i<l; i++) {
        temp=word[i]-'a';
        if(!p->next[temp]) {
            p->next[temp] = new node;
            p->next[temp]->num = 1;
            memset(p->next[temp]->next,0,sizeof(p->next[temp]-
>next));
        }
        else
            p->next[temp]->num++;
        p=p->next[temp];
    }
}
int num(char *word) {
    node *p = &root;
    int ans,temp;
    int l = strlen(word);
    for(int i=0; i<l; i++) {
        temp=word[i]-'a';
        if(!p->next[temp]) {
            return 0;
```

```
            }
            p=p->next[temp];
        }
        ans=p->num;
        return ans;
    }
```

博弈

威佐夫博奕

两堆各若干个物品，两个人轮流从某一堆或同时从两堆中取同样多的物品，规定每次至少取一个，多者不限，最后取光者得胜。
那么任给一个局势(a,b)，怎样判断它是不是奇异局势呢？我们有如下公式:
ak=[k(1+√5)/2],bk= ak+k (k=0,1,2,…,n方括号表示取整函数
(http://blog.csdn.net/ljd4305/article/details/12167775)

Nim博弈_SG函数

首先定义mex(minimal excludant)运算，这是施加于一个集合的运算，表示最小的不属于这个集合的非负整数。例如mex{0,1,2,4}=3、mex{2,3,5}=0、mex{}=0。对于一个给定的有向无环图，定义关于图的每个顶点的Sprague-Grundy函数g如下：g(x)=mex{ g(y) | y是x的后继 },这里的g(x)即sg[x]。
打表:

```
//f[]：可以取走的石子个数
//sg[]:0~n的SG函数值
//hash[]:mex{}
int f[N],sg[N],hash[N];
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;
            }
        }
    }
}
```

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;
}
```

anti-nim

先手必胜当且仅当
1、所有堆的石子数量都为1且游戏的SG值为0；
2、存在大于一的石子堆数并且游戏的SG值不为0。

anti-SG

先手必胜当且仅当：
1、游戏的SG函数不为0且游戏中的某一个单一游戏的SG函数大于1；
2、游戏的SG函数为0且游戏中没有单一游戏的SG函数大于1。

Trie图/AC自动机

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

```
#include <queue>
using namespace std;
const int MAX_N=26;
const int MAX_NODE=500010;
struct NODE{
    int cnt;
    NODE *next[MAX_N],*fail;
    NODE(){
        cnt=0;
        fail=NULL;
        memset(next,0,sizeof(next));
    }
}*head;
void Build_trie(char *s,NODE *head){
    int len = strlen(s);
    for(int i=0;i<len;i++){
        int k = s[i]-'a';
        if(head->next[k]==NULL)
            head->next[k] = new NODE();
        head = head->next[k];
    }
    head->cnt++;
}
queue<NODE*> q;
void Build_fail(NODE *head){
    head->fail=NULL;
    q.push(head);
    while(!q.empty()){
        NODE *now = q.front();
        q.pop();
        for(int i=0;i<MAX_N;i++)
            if(now->next[i]){
                NODE *p = now->fail;
            while(p){
                if(p->next[i]){
                    now->next[i]->fail = p->next[i];
                    break;
                }
                p=p->fail;
            }
            if(p==NULL)
                now->next[i]->fail = head;
            q.push(now->next[i]);
            }
    }
}
bool AC_find(NODE *head, char *s){
    int len = strlen(s),sum=0;
    NODE* p = head;
    for(int i=0;i<len;i++){
```

```
            int k = s[i]-'a';
            while(p->next[k]==NULL&&p!=head)
                p=p->fail;
            p = p->next[k]==NULL ? head : p->next[k];
        NODE *tmp = p;
            while(tmp!=head&&tmp->cnt != -1){
                sum+=tmp->cnt;
                tmp->cnt = -1;
                tmp=tmp->fail;
            }
        }
        return sum>0;
    }
    char s[1000005];
    char ss[1000];
    int main(){
        int n;
        head = new NODE();
        scanf("%d",&n);
        while(n--){
            scanf("%s",ss);
            Build_trie(ss,head);
        }
        Build_fail(head);
        scanf("%s",s);
        if(AC_find(head,s))printf("YES\n");
        else printf("NO\n");
    }
```

图论

链式前向星

```
    int head[n],tot;
    struct EdgeNode{
        int to;
        int w;
        int next;
    };
    EdgeNode edges[m];
    void init(){
        memset(head,-1,sizeof(head));
        tot=0;
    }
```

```
//add edge from x to y，w是权值
void addedge(int x,int y,int w){
    edges[tot].to=y;
    edges[tot].w=w;
    edges[tot].next=head[x];
    head[x]=tot++;
}
//遍历从k出发的所有边
for(k=head[i];k!=-1;k=edges[k].next){
    //do what you want.
}
```

最小生成树—kruskal

```
//u[i],v[i],w[i]分别保存两个端点和权值。
//n为点的数量，m为边的数量。
int cmp(const int i,const int j){//间接排序函数
    return w[i]<w[j];
}
int find(int x){//并查集的find
    return p[x]==x ? p[x] : find(p[x]);
}
int kruskal(){
    int ans = 0;
    for(int i=0;i<n;i++)p[i]=i;//初始化并查集
    for(int i=0;i<m;i++)r[i]=i;//初始化边序号
    sort(r,r+m,cmp);
    for(int i=0;i<m;i++){
        int e=r[i];
        int x=find(u[e]);
        int y=find(v[e]);
        if(x!=y){
            ans+=w[e];
            p[x]=y;
        }
    }
    return ans;
}
```

二分图判定

判断一个图是不是二分图，思路当然就是染色法，首先给一个顶点然色，然后与它相邻的顶点全部染相反的颜色，如果过程中发现要染的点已经染色了，而且是和现在点相同的颜色的话，那么就说明不是一个二分图。

要注意是否是一个连通图。
非联通图代码，如果是连通图可以直接dfs(1)。

```
const int mx = 10005;
vector<int> node[mx];
int col[mx];
bool vis[mx];
bool dfs(int i){
    queue<int> q;
    q.push(i);
    col[i]=1;
    vis[i]=true;
    while(!q.empty()){
        int temp = q.front();
        q.pop();
        for(int i=0;i<node[temp].size();i++){
            int num = node[temp][i];
            int cc = col[temp] == 1 ? 2 : 1;
            if(col[num]==col[temp])return false;
            col[num]=cc;
            if(!vis[num]){
                q.push(num);
                vis[num]=true;
            }
        }
    }
    return true;
}
int main(){
int t,n,m,a,b;
bool jud,judd;
    judd=true;
    scanf("%d%d",&n,&m);
    memset(col,0,sizeof(col));
    memset(vis,0,sizeof(vis));
    while(m--){
        scanf("%d%d",&a,&b);
        node[a].push_back(b);
        node[b].push_back(a);
    }
    for(int i=1;i<=n;i++){
        if(!vis[i])
            jud = dfs(i);
        if(!jud)judd=false;
    }
    for(int i=0;i<=n;i++)
        node[i].clear();
return 0;
}
```

```
#include<cstdio>
#include<vector>
#include<queue>
#define MAXV 10000
#define INF 1000000000
struct Edge{
    int v; //边权
    int to; //连接的点
};
vector<Edge> e[MAXV]; //由于一般情况下E<<V*V,故在此选用了vector动态数组存储,也可以使用链表存储
int dist[MAXV]; //存储到原点0的距离,可以开二维数组存储每对节点之间的距离
int cnt[MAXV]; //记录入队次数,超过V则退出
queue<int> buff; //队列,用于存储在SPFA算法中的需要松弛的节点
bool done[MAXV]; //用于判断该节点是否已经在队列中
int V; //节点数
int E; //边数
bool spfa(const int st){ //返回值:TRUE为找到最短路返回,FALSE表示出现负环退出
    for(int i=0;i<V;i++){ //初始化:将除了原点st的距离外的所有点到st的距离均赋上一个极大值
        if(i==st){
            dist[st]=0; //原点距离为0;
            continue;
        }
        dist[i]=INF; //非原点距离无穷大
    }
    buff.push(st); //原点入队
    done[st]=1; //标记原点已经入队
    cnt[st]=1; //修改入队次数为1
    while(!buff.empty()){ //队列非空,需要继续松弛
        int tmp=buff.front(); //取出队首元素
        for(int i=0;i<(int)e[tmp].size();i++){ //枚举该边连接的每一条边
            Edge *t=&e[tmp][i]; //由于vector的寻址速度较慢,故在此进行一次优化
            if(dist[tmp]+(*t).v<dist[(*t).to]){ //更改后距离更短,进行松弛操作
                dist[(*t).to]=dist[tmp]+(*t).v; //更改边权值
                if(!done[(*t).to]){ //没有入队,则将其入队
                    buff.push((*t).to); //将节点压入队列
                    done[(*t).to]=1; //标记节点已经入队
                    cnt[(*t).to]+=1; //节点入队次数自增
                    if(cnt[(*t).to]>V){//已经超过V次,出现负环
                        while(!buff.empty())buff.pop(); //清空队
```

```
列,释放内存
                        return false; //返回FALSE
                    }
                }
            }
        }
        buff.pop();//弹出队首节点
        done[tmp]=0;//将队首节点标记为未入队
    }
    return true; //返回TRUE
}//算法结束
int main(){ //主函数
    scanf("%d%d",&V,&E); //读入点数和边数
    for(int i=0,x,y,l;i<E;i++){
        scanf("%d%d%d",&x,&y,&l); //读入x,y,l表示从x->y有一条有向边
长度为l
        Edge tmp; //设置一个临时变量,以便存入vector
        tmp.v=l; //设置边权
        tmp.to=y; //设置连接节点
        e[x].push_back(tmp); //将这条边压入x的表中
    }
    if(!spfa(0)){ //出现负环
        printf("出现负环,最短路不存在\n");
    }else{ //存在最短路
        printf("节点0到节点%d的最短距离为%d",V-1,dist[V-1]);
    }
    return 0;
}
```

Tarjan算法_有向图强联通分量

```
void tarjan(int i){
    int j;
    DFN[i]=LOW[i]=++Dindex;
    instack[i]=true;
    Stap[++Stop]=i;
    for (edge *e=V[i];e;e=e->next){
        j=e->t;
    if (!DFN[j]){
        tarjan(j);
        if (LOW[j]<LOW[i])
            LOW[i]=LOW[j];
        }
    else if (instack[j] && DFN[j]<LOW[i])
            LOW[i]=DFN[j];
    }
```

```
        if (DFN[i]==LOW[i]){
            Bcnt++;
            do{
                j=Stap[Stop--];
                instack[j]=false;
                Belong[j]=Bcnt;
            }while (j!=i);
        }
    }
    void solve(){
        int i;
        Stop=Bcnt=Dindex=0;
        memset(DFN,0,sizeof(DFN));
        for (i=1;i<=N;i++)
            if (!DFN[i])
                tarjan(i);
    }
```

二分图最大匹配

匈牙利算法DFS版本

适用于边比较多的图，DFS寻找增广路径快。
每次寻找增广路径的复杂度是O(m)，最多需要寻找O(n)次，因此复杂度为O(n*m)，由于采用了邻接矩阵，空间复杂度为O(n^2)。（n为点的个数，m为边的个数）

```
const int MAXN = 1000;//最大顶点数
bool bmap[MAXN][MAXN];//二分图
bool bmask[MAXN];//寻找增广路径时的标志数组
int nx,ny;//nx为左集合的顶点数目，ny为右集合的顶点数目
int cx[MAXN];//cx[i]表示左集合i顶点所匹配到的右集合的顶点序号
int cy[MAXN];//cy[i]表示右集合i顶点所匹配到的左集合的顶点序号
//寻找增广路径
int findpath(int u){
    for(int i=0;i<ny;i++){
        if(bmap[u][i]&&!bmask[i]){
            bmask[i]=1;
            if(cy[i]==-1||findpath(cy[i])){
                cy[i]=u;
                cx[u]=i;
                return 1;
            }
        }
    }
    return 0;
}
//求最大匹配
```

```
int MaxMatch(){
    int res=0;
    for(int i=0;i<nx;i++)
        cx[i]=-1;
    for(int i=0;i<ny;i++)
        cy[i]=-1;
    for(int i=0;i<nx;i++){
        if(cx[i]==-1){
            for(int j=0;j<ny;j++)
                bmask[j]=0;
            res+=findpath(i);
        }
    }
return res;
}
```

匈牙利算法BFS版本

```
const int MAXN = 1000;//最大顶点数
bool bmap[MAXN][MAXN];//二分图
int bmask[MAXN];//寻找增广路径时的标志数组
int nx,ny;//nx为左集合的顶点数目，ny为右集合的顶点数目
int cx[MAXN];//cx[i]表示左集合i顶点所匹配到的右集合的顶点序号
int cy[MAXN];//cy[i]表示右集合i顶点所匹配到的左集合的顶点序号
int Queue[MAXN];//队列保存待扩展顶点
int bprev[MAXN];//记录前置顶点
//求最大匹配
int MaxMatch(){
    int res=0;
    int qs,qe;
    memset(cx,-1,sizeof(cx));
    memset(cy,-1,sizeof(cy));
    memset(bmask,-1,sizeof(bmask));
    for(int i=0;i<nx;i++){
        if(cx[i]==-1){
            qs=qe=0;
            Queue[qe++]=i;
            bprev[i]=-1;
            bool flag = 0;
            while(qs<qe&&!flag){
                int u = Queue[qs];
                for(int v = 0;v<ny&&!flag;v++){
                    if(way[u][v]&&bmask[v]!=i){
                        bmask[v]=i;
                        Queue[qe++]=cy[v];
                        if(cy[v]>=0){
                            bprev[cy[v]]=u;
                        }
```

```
                        else{
                            flag = 1;
                            int d = u,e = v;
                            while(d!=-1){
                                int t = cx[d];
                                cx[d] = e;
                                cy[e] = d;
                                d = bprev[d];
                                e = t;
                            }
                        }
                    }
                }
                qs++;
            }
            if(cx[i]!=-1)
                res++;
        }
    }
return res;
}
```

Hopcroft-Karp算法

时间复杂度为O(n^(1/2)*m)，空间复杂度为O(m^2)。

```
#include <queue>
const int mx = 3005;//最大顶点数
const int inf = 1<<28;//距离初始值
bool bmap[mx][mx];//二分图
bool bmask[mx];//寻找增广路径时的标志数组
int cx[mx];//cx[i]表示左集合i顶点所匹配到的右集合的顶点序号
int cy[mx];//cy[i]表示右集合i顶点所匹配到的左集合的顶点序号
int nx,ny;//nx为左集合的顶点数目，ny为右集合的顶点数目
int dx[mx];//dx[i]表示左集合i顶点的距离标号
int dy[mx];//dy[i]表示右集合i顶点的距离标号
int dis;
bool searchpath(){
    queue<int> Q;
    dis=inf;
    memset(dx,-1,sizeof(dx));
    memset(dy,-1,sizeof(dy));
    for(int i=0;i<nx;i++){
        if(cx[i]==-1){
            Q.push(i);
            dx[i]=0;
        }
```

```
        }
        while(!Q.empty()){
            int u=Q.front();
            Q.pop();
            if(dx[u]>dis) break;
            for(int v=0;v<ny;v++){
                if(bmap[u][v]&&dy[v]==-1){
                    dy[v]=dx[u]+1;
                    if(cy[v]==-1) dis=dy[v];
                    else{
                        dx[cy[v]]=dy[v]+1;
                        Q.push(cy[v]);
                    }
                }
            }
        }
        return dis!=inf;
    }
    int findpath(int u){
        for(int v=0;v<ny;v++){
            if(!bmask[v]&&bmap[u][v]&&dy[v]==dx[u]+1){
                bmask[v]=1;
                if(cy[v]!=-1&&dy[v]==dis)
                    continue;
                if(cy[v]==-1||findpath(cy[v])){
                    cy[v]=u;cx[u]=v;
                    return 1;
                }
            }
        }
        return 0;
    }
    int MaxMatch(){
        int res=0;
        memset(cx,-1,sizeof(cx));
        memset(cy,-1,sizeof(cy));
        while(searchpath()){
            memset(bmask,0,sizeof(bmask));
            for(int i=0;i<nx;i++){
                if(cx[i]==-1)
                    res+=findpath(i);
            }
        }
        return res;
    }
```

网络流_Edmond-Karp算法

对于点数较少时比较有利。
时间复杂度为O（n×m^2），空间复杂度为O（n^2）。

```
#include <queue>
const int mx = 410;//最大点个数
const int inf=1<<28;
int map[mx][mx];//邻接数组
int node;//点数
int p[mx];//前驱数组
bool EK_Bfs(int start,int end){
    queue<int> Q;
    bool flag[mx];
    memset(flag,0,sizeof(flag));
    memset(p,-1,sizeof(p));
    Q.push(start);
    flag[start]=true;
    while(!Q.empty()){
        int e = Q.front();
        if(e==end)
            return true;
        Q.pop();
        for(int i=1;i<=node;i++){
            if(map[e][i]&&!flag[i]){
                flag[i]=true;
                p[i]=e;
                Q.push(i);
            }
        }
    }
    return false;
}
int EK_Max_Flow(int start,int end){
    int u,flow_ans=0,mn;
    while(EK_Bfs(start,end)){
        mn=inf;
        u=end;
        while(p[u]!=-1){
            mn=min(mn,map[p[u]][u]);
            u=p[u];
        }
        flow_ans+=mn;
        u=end;
        while(p[u]!=-1){
            map[p[u]][u]-=mn;
            map[u][p[u]]+=mn;
            u=p[u];
        }
    }
    return flow_ans;
```

```
}
```

最小费用最大流

```
#include<algorithm>
#include<queue>
using namespace std;
const int MAXN = 2200;
const int MAXM =400010;
const int INF = 0x7fffffff;
struct Edge{
    int to, next, cap, flow, cost;
} edge[MAXM];
int head[MAXN], tol;
int pre[MAXN], dis[MAXN];
bool vis[MAXN];
int N;
void init(int n){
    N = n;
    tol = 0;
    memset(head, -1, sizeof(head));
}
void addedge(int u, int v, int cap, int cost){
    edge[tol].to = v;
    edge[tol].cap = cap;
    edge[tol].cost = cost;
    edge[tol].flow = 0;
    edge[tol].next = head[u];
    head[u] = tol++;
    edge[tol].to = u;
    edge[tol].cap = 0;
    edge[tol].cost = -cost;
    edge[tol].flow = 0;
    edge[tol].next = head[v];
    head[v] = tol++;
}
bool spfa(int s, int t){
    queue<int>q;
    for (int i = 0; i < N; i++){
        dis[i] = INF;
        vis[i] = false;
        pre[i] = -1;
    }
    dis[s] = 0;
    vis[s] = true;
    q.push(s);
    while (!q.empty()){
```

```
            int u = q.front();
            q.pop();
            vis[u] = false;
            for (int i = head[u]; i != -1; i = edge[i].next){
                int v = edge[i].to;
                if (edge[i].cap > edge[i].flow &&
                        dis[v] > dis[u] + edge[i].cost ){
                    dis[v] = dis[u] + edge[i].cost;
                    pre[v] = i;
                    if (!vis[v]){
                        vis[v] = true;
                        q.push(v);
                    }
                }
            }
        }
        if (pre[t] == -1)return false;
        else return true;
    }
    //返回的是最大流,cost存的是最小费用
    int MinCostMaxFlow(int s, int t, int &cost){
        int flow = 0;
        cost = 0;
        while (spfa(s, t)){
            int Min = INF;
            for (int i = pre[t]; i != -1; i = pre[edge[i ^ 1].to]){
                if (Min > edge[i].cap - edge[i].flow)
                    Min = edge[i].cap - edge[i].flow;
            }
            for (int i = pre[t]; i != -1; i = pre[edge[i ^ 1].to]){
                edge[i].flow += Min;
                edge[i^1].flow -= Min;
                cost += edge[i].cost * Min;
            }
            flow += Min;
        }
    return flow;
    }
```

基本DP

DAG上的最长路

给定N个M维的箱子（或者是被叫做箱子一类的东西），问最多能嵌套多少（当且仅当箱子a的所有维的长度均小于箱子b时可嵌套）。
显然可知，如果箱子b可嵌套a，则把ab边长都按照从小到达排好序，满足对应项bi>ai.

用G[a][b]=true;表示b可嵌套a。
用数组d[i]表示从节点i出发的最长路的长度。

```
#include<iostream>
#include<cstring>
#include<cstdio>
#include<algorithm>
using namespace std;
bool G[32][32];
int arr[32][32], d[32], n, k;
bool first;
inline bool jud(int a, int b){
    for(int i=0; i<k; ++i)
        if(arr[a][i]>=arr[b][i])return false;
    return true;
}
int dp(int i){
    if(d[i]!=-1)return d[i];
    d[i]=1;//其长度至少为1
    for(int j=0; j<n; j++)if(G[i][j]){
        d[i] = max(d[i], dp(j)+1);//因为多了一个所以dp(j)需要加上1
    }
    return d[i];
}
void Print(int i){
    if(first) printf(" %d",i+1);
    else {printf("%d",i+1); first=true;}
  //  printf("%d ", i+1);
    for(int j=0; j<n; j++)if(G[i][j]&&d[j]+1==d[i]){
        Print(j);
        break;
    }
}
int main(){
    while(~scanf("%d%d",&n,&k)){
        for(int i=0; i<n; i++){
            for(int j=0; j<k; j++)
                scanf("%d",&arr[i][j]);
            sort(arr[i], arr[i]+k);
        }
        memset(G, false, sizeof(G));
        for(int i=0; i<n-1; i++){
            for(int j=i+1; j<n; j++){
                if(jud(j,i))
                    G[j][i]=true;
                else if(jud(i,j))
                    G[i][j]=true;
            }
        }
```

```
            memset(d, -1, sizeof(d));
            int ans=-1, pos;
            for(int i=0; i<n; ++i){
                int t=dp(i);
                if(t>ans){
                    ans=t;
                    pos=i;
                }
            }
            printf("%d\n", ans);
            first=false;
            Print(pos);
        //输出到最后一个数字没有空格等价于第一个数字没有空格其他的数字空格前置
            printf("\n");
        }
        return 0;
    }
```

状压DP-TSP问题

状压DP可以解决TSP(Travelling Salesman Problem)问题，时间复杂度为2^n*n^2。不过有些题可以用全排列 do{}while(next_permutation(a,a+n)); 水过去。

```
#include<cstdio>
#include<cstring>
#include<iostream>
#include<algorithm>
using namespace std;
const int INF=0x3f3f3f3f;
int ans,n,mp[13][13],dp[1030][13];//dp[state][i]表示到达i点状态为state的最小距离
//dp[state][i]=min(dp[state][i],dp[state'][j]+dis[j][i]);
int main() {
    while(scanf("%d",&n)&&n) {
        for(int i=0; i<=n; i++)
            for(int j=0; j<=n; j++)
                scanf("%d",&mp[i][j]);
        for(int i=0; i<=n; i++)
            for(int j=0; j<=n; j++)
                for(int k=0; k<=n; k++)
                    if(mp[j][i]+mp[i][k]<mp[j][k])
                        mp[j][k]=mp[j][i]+mp[i][k];
        memset(dp,0,sizeof(dp));
        for(int state=0; state<(1<<n); state++) {//枚举所有状态
            for(int i=1; i<=n; i++) {
                if(state&(1<<(i-1))) { //状态state中经过了城市i
                    if(state==(1<<(i-1)))dp[state][i]=mp[0]
```

```
[i];//边界条件，即从原点只到了这一个点处。
                    else {
                        dp[state][i]=INF;
                        for(int j=1; j<=n; j++)
                            //枚举从开始到达j再从j到达i的路径。
                            if(i!=j&&state&(1<<(j-1)))
                                dp[state][i]=min(dp[state]
[i],dp[state^(1<<(i-1))][j]+mp[j][i]);
                                //state^(1<<(i-1))一定小于state。
                    }
                }
            }
        }
        ans=dp[(1<<n)-1][1]+mp[1][0];
        for(int i=2;i<=n;i++)
            ans=min(ans,dp[(1<<n)-1][i]+mp[i][0]);
        cout<<ans<<endl;
    }
    return 0;
}
```

数位DP

数字处理函数

```
int f(int num){
    int ans;
    int pos = 0;
    while(num){
        digit[++pos]=num%10;
        num=num/10;
    }
    return dfs( pos, s , true );
}
```

其中：
digit为处理串的每个数位的数值。
s为初始的状态。
如果有其他限定条件， dfs 中可以增加参数。

DFS 函数

```
int dfs(int l, int s, bool jud) {
    if ( l==-1 ) return s == target_s;
    if ( !e && ~f[l][s] ) return f[l][s];
```

```
        int ans = 0;
        int nex = e ? digit[i] : 9;
        for (int d = 0; d <=  nex; ++d)
            ans += dfs( l-1 , new_s( s,d ) , e && d==nex );
        return jud ? ans : f[l][s] = ans;
    }
```

其中：
f为记忆化数组；
l为当前处理串的第l位（权重表示法，也即后面剩下l+1位待填数）；
s为之前数字的状态（如果要求后面的数满足什么状态，也可以再记一个目标状态t之类，for的时候枚举下t）；
jud表示之前的数是否是上界的前缀（即后面的数能否任意填）。
for循环枚举数字时，要注意是否能枚举0，以及0对于状态的影响，有的题目前导0和中间的0是等价的，但有的不是，对于后者可以在dfs时再加一个状态变量z，表示前面是否全部是前导0，也可以看是否是首位，然后外面统计时候枚举一下位数。It depends.

01分数规划

1. 二分法
   二分一个答案k，令ci=ai-k*bi，并将ci排序，选出最大的m个，如果∑（ci）{1<=i<=m}>=0，那么提高答案k的下界，否则降低上界，直到k的精度满足要求为止。
2. Dinkelbach迭代法
   随意构造一个答案k，令ci=ai-k*bi，并将ci排序，选出最大的m个，令q=∑（ai）/∑(bi){1<=i<=m}与k的差在精度范围内就输出，否则令k=q，直至满足精度要求。
   两种方法各有利弊，二分法精度较高，容易理解和证明，然而速度较慢。而Dinkelbach迭代法速度较快，但精度低，不易理解和证明。可视情况选择解题方法。