ACM 常用算法模板

风了咕凉

2015/4/20

目录

字符串处理

1. KMP 算法

```
/*next[i]为失配指针*/
void getnext(char s[])
{
    memset(next,0,sizeof(next));
    next[0]=-1;
    int j=-1,k=0;
    int len=strlen(s);
    while(k<len){
        if(j==-1||s[j]==s[k]){
            j++;k++;
            next[k]=j;
        }
        else
            j=next[j];
    }
}
```

```
/*返回值为模式串在主串中匹配的次数*/
int KMP(char s[],char t[])
{
    int i,j,ans;
    ans=i=j=0;
    getnext(s);
    int
len=strlen(t),len1=strlen(s);
    while(i<len){
        if(j==-
1||s[j]==t[i]){i++;j++;}
        else j=next[j];
        if(j==len1){
            ans++;
            //j=nex[j];可重叠匹配
            j=0;//不可重叠匹配
        }
    }
    return ans;
}
```

2. 后缀数组

```
int sa[maxn],c[maxn],y[maxn],x[maxn];//x 为后缀数组
bool cmp(int *p,int a,int b,int l){
    return y[a]==y[b]&&y[a+l]==y[b+l];
}
//从 0 开始，m 为字符上界+1，范围是 0~n-1，n 为字符串长度+1，后缀 0
到后缀 n
void build_sa(int m,int n)
{
     int *x=wa,*y=wb,*t;
     //基数排序
    for(int i=0;i<m;i++)c[i]=0;
    for(int i=0;i<n;i++)c[x[i]=id(s[i])]++;
    for(int i=1;i<m;i++)c[i]+=c[i-1];
    for(int i=n-1;i>=0;i--)sa[--c[x[i]]]=i;
```

```
//相同情况下，后缀前后顺序不变
    for(int k=1;k<=n;k<<=1){
        int p=0;
//直接利用 sa 数组排序第二关键字，长度不够 k 的第二关键字为 0,直接排到前面
//斜线平移 后缀 sa[i]决定了后缀 sa[i]-k 的排名
        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, //后缀 y[i]
        for(int i=0;i<m;i++)c[i]=0;
        for(int i=0;i<n;i++)c[x[y[i]]]++;
        for(int i=1;i<m;i++)c[i]+=c[i-1];
        for(int i=n-1;i>=0;i--)sa[--c[x[y[i]]]]=y[i];
//根据 sa 和 y 数组重新计算新的 x 数组
        t=x;x=y;y=t;
        p=1;x[sa[0]]=0;
        for(int i=1;i<n;i++){
            x[sa[i]]=cmp(y,sa[i],sa[i-1],k)?p-1:p++;
        }
        if(p>=n)break;
        m=p;
    }
}
```

3.Manacher 算法（求最长回文子串）---O(n)

```
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;
//2*id-i 为 i 关于 id 对称位置。翻转后依然对称。或者 p[j]超出范围。
        while(Ma[i+Mp[i]]==Ma[i-Mp[i]])Mp[i]++;
```

```
        if(i+Mp[i]>mx) { //更新最大回文串最右端
            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;
}
```

4.最长公共前缀---（后缀数组解法 O（nlogn））

```
char s[maxn]; int dp[maxn][18],ran[maxn],height[maxn];
int sa[maxn],wa[maxn],wb[maxn],c[maxn];
int id(char c){ if(!c)return 0;return c-'a'+1;}
bool cmp(int *r,int a,int b,int k);
void build_sa(int m,int n);
void calheight(int n){
    int k=0;
    for(int i=0;i<n;i++)ran[sa[i]]=i;
    for(int i=0;i<n;i++){
        if(k)k--;
        if(ran[i]==0)continue;
        int j=sa[ran[i]-1];
        while(s[j+k]==s[i+k])k++;
        height[ran[i]]=k;
    }
```

```
}
void init_rmq(int n,int h[])
{
    for(int i=1;i<n;i++){
        dp[i][0]=h[i];
    }
    for(int j=1;(1<<j)<n;j++){
        for(int i=0;i+(1<<j)-1<n;i++)
            dp[i][j]=MIN(dp[i][j-1],dp[i+(1<<(j-1))][j-1]);
    }
}
int lcp(int L,int R)//返回最长公共前缀长度
{
    L=ran[L],R=ran[R];
    if(R<L)swap(R,L);
    L++;
    int k=0;
    while((1<<(k+1))<=R-L+1)k++;
    return MIN(dp[L][k],dp[R-(1<<k)+1][k]);
}
```

5.Trie

```
char s[maxn];
#define sigma_size 26
struct Trie
{
    int ch[maxnode][sigma_size];
    int val[maxnode];
    int sz;
    Trie(){
        sz=1;
        memset(ch,0,sizeof(ch));
    }
    int idx(char c){return c-'a';}
    void insert(char *s,int v)
    {
        int len=strlen(s),u=0;
        for(int i=0;i<len;i++){
            int c=idx(s[i]);
            if(!ch[u][c]){
                val[sz]=0;
                ch[u][c]=sz++;
            }
            u=ch[u][c];
        }
        val[u]=v;
    }
    bool query(char *s){
        int len=strlen(s),u=0;
        for(int i=0;i<len;i++){
            int c=idx(s[i]);
            if(!ch[u][c])
                return false;
            u=ch[u][c];
        }
        if(!val[u])return true;
    }
};
```

6.AC 自动机

```
const int SIGMA_SIZE = 26;
const int MAXNODE = 11000;
const int MAXS = 150 + 10;
map<string,int> ms;
//ms 是为了满足特殊要求,比如模板串相同时
struct ACautomata {
  int ch[MAXNODE][SIGMA_SIZE];
  int f[MAXNODE];    // fail 函数
  int val[MAXNODE]; // 每个字符串的结尾结点都有一个非 0 的 val
  int last[MAXNODE]; // 输出链表的下一个结点
  int cnt[MAXS];
  int sz;
  void init() {
    sz = 1; ms.clear();
    memset(ch[0], 0, sizeof(ch[0]));
    memset(cnt, 0, sizeof(cnt));
  }
  int idx(char c) {return c-'a';}  // 字符 c 的编号
  void insert(char *s, int v) {  // 插入字符串。v 必须非 0
    int u = 0, n = strlen(s);
    for(int i = 0; i < n; i++) {
      int c = idx(s[i]);
      if(!ch[u][c]) {
        memset(ch[sz], 0, sizeof(ch[sz]));
        val[sz] = 0;ch[u][c] = sz++;
      }
      u = ch[u][c];
    }
    val[u] = v;  ms[string(s)] = v;
  }
  // 递归打印匹配文本串 str[i]结尾的后缀,以结点 j 结尾的所有字符串
  void print(int i,int j) {
    if(j) {
      cnt[val[j]]++;
      print(i,last[j]);
    }
  }
```

```
  int find(char* T) { // 在 T 中找模板
   int n = strlen(T);
   int j = 0; // 当前结点编号，初始为根结点
   for(int i = 0; i < n; i++) { // 文本串当前指针
     int c = idx(T[i]);
     j = ch[j][c];
     if(val[j]) print(i,j);
     else if(last[j]) print(i,last[j]); // 找到了!
   }
  }
  // 计算 fail 函数
  void getFail() {
   queue<int> q;
  f[0] = 0;
  // 初始化队列
  for(int c = 0; c < SIGMA_SIZE; c++) {
     int u = ch[0][c];
     if(u) { f[u] = 0; q.push(u); last[u] = 0; }
  }
  // 按 BFS 顺序计算 fail
   while(!q.empty()) {
     int r = q.front(); q.pop();
     for(int c = 0; c < SIGMA_SIZE; c++) {
       int u = ch[r][c];
       if(!u) {
           ch[r][c]=ch[f[r]][c];
           continue;
       }
       q.push(u);
       int v = f[r];
       while(v && !ch[v][c]) v = f[v];
       f[u] = ch[v][c];
       last[u] = val[f[u]] ? f[u] : last[f[u]];
     }
   }
  }
};
```

关于 KMP

next[k]就是后缀与前缀的最大匹配。

n-next[k]就是最小覆盖子串。

统计不同前缀(后缀)的数量　复杂度为 O(n)

dp[i]表示以 i 结尾的不同前缀的数量。

转移方程就为 dp[i]=dp[next[i]]+1

统计一个字符串中不相同的回文子串数量的 O(nlogn)算法

插入最少的字符使字符串成为回文 （两端 dp)

数学部分

1.素数

1)埃氏筛法 时间复杂度 O(nlog(logn))

```
void getprime(int n)
{

memset(isprime,1,sizeof(isprime));
    isprime[0]=isprime[1]=0;
    for(int i=0;i*i<n;i++){
        if(isprime[i]){
            for(int j=i*i;j<=n;j+=i){
                isprime[j]=0;
            }
        }
    }
}
```

2)欧拉筛法(线性筛)O(n)(严格)

```
void getprime(int n)
{

  memset(isprime,1,sizeof(isprime));
    isprime[0]=isprime[1]=0;
    for(int i=2;i<=n;i++){
        if(isprime[i]){
            prime[cnt++]=i;
        }
        for(int j=0;j<cnt;j++){
            isprime[prime[j]*i]=0;
            if(i%prime[j]==0)break;
        }
    }
}
```

2.GCD 最大公约数

```
int gcd(int a,int b){
    if(!b)return a; return gcd(b,a%b);
}
```

快速 GCD

```
/*stein 算法
补充 gcd 的性质
a，b 均为偶数时，gcd(a,b)=gcd(a>>1,b>>1)<<1;
a 为奇数，b 为偶数时，gcd(a,b)=gcd(a,b>>1);
a 为偶数，b 为奇数时，gcd(a,b)=gcd(a>>1,b);
a，b 均为奇数时，gcd(a,b)=(abs(a-b),min(a,b));*/
int sgcd(int a,int b)
{
    if(!a)return b;
    if(!b)return a;
```

```
    if(!(a&1)&&!(b&1)) return sgcd(a>>1,b>>1)<<1;
    if(!(a&1))return sgcd(a>>1,b);
    if(!(b&1)) return sgcd(a,b>>1);
    return sgcd(abs(a-b),min(a,b));
}
```

扩展 GCD

已知 a, b 求解一组 x，y 使得 a*x+b*y=Gcd(a,b)

```
void exgcd(int a,int b,int &x,int &y)
{
    if(!b){
        x=1;y=0;return;//可同时返回最大公约数
    }
    exgcd(b,a%b,x,y);
    int t=x;
    x=y;
    y=t-(a/b)*y;
}
/*   a * x + b * y = c 的整数解
 *   先计算 Gcd(a,b)，若 n 不能被 Gcd(a,b)整除，则方程无整数解；否
 *   则，在方程两边同时除以 Gcd(a,b)，得到新的不定方程
 *   a' * x + b' * y = c'，此时 Gcd(a',b')=1；
 *   所有整数解为：
 *   x = c'*x0 + b'*t ;  y = c'*y0 - a'*t (t 为整数)  */
```

3.线性求每个数的约数个数 ---时间复杂度严格 O(n)

```
// e[i]表示 i 的最小素数因子的个数
//T[i]为 i 的约数个数，一次更新
int prime[M],e[M],T[M];
bool isprime[M];
void get_prime()
{
    int i,j,k;
    memset(isprime,true,sizeof(isprime));
    k=0;
    for(i=2;i<M;i++){
        if(isprime[i]){
            prime[k++]=i;
```

```
                e[i]=1;
                T[i]=2;          //素数的约数个数为 2
            }
            for(j=0;j<k&&i*prime[j]<M;j++){
                isprime[i*prime[j]]=false;
                if(i%prime[j]==0){//prime[j]等于 i 的最小素因子
                    T[i*prime[j]]=T[i]/(e[i]+1)*(e[i]+2);
                    e[i*prime[j]]=e[i]+1;
                    break;
                }
                else{//prime[j]小于 i 的最小素因子
                    T[i*prime[j]]=T[i]*T[prime[j]];
                    e[i]=1;
                }
            }
        }
}
```

4.欧拉函数

```
1.欧拉函数是积性函数,但不是完全积性函数,即 φ(mn)=φ(n)*φ(m)只在
(n,m)=1 时成立.
2.对于一个正整数 N 的素数幂分解 N=P1^q1*P2^q2*...*Pn^qn.
   φ(N)=N*(1-1/P1)*(1-1/P2)*...*(1-1/Pn).
3.除了 N=2,φ(N)都是偶数.
4.设 N 为正整数,∑φ(d)=N  (d|N).
求小于等于 N 的与 N 互质的数的和 : ans=N*phi(N)/2;
/* 欧拉 phi(x)函数等于不超过 x 且和 x 互素的整数个数。 */
int phi[MAXN];
/* 单个欧拉函数值*/
int euler_phi(int n)
{
    int m=sqrt(n+0.5),ans=n;
    for(int i=2;i<=m;i++){
        if(n%i==0) ans=ans/i*(i-1);
        while(n%i==0)n/=i;
    }
    if(n>1)ans=ans/n*(n-1);
    return ans;
}
```

```
/*欧拉函数 表*/
void phi_table(int n)
{
    memset(phi,0,sizeof(phi));
    phi[1]=1;
    for(int i=2;i<=n;i++)
    {
        if(!phi[i]){
            for(int j=i;j<=n;j+=i){
                if(!phi[j])phi[j]=j;
                phi[j]=phi[j]/i*(i-1);
            }
        }
    }
}
```

5.线性同余方程

```
void exgcd(int a,int b,int &d,int &x,int &y)
{
    if(!b){
        d=a;x=1;y=0;return;
    }
    exgcd(b,a%b,d,y,x);
    y-=x*(a/b);
}
//当 a 与 m 互素时，a 关于模 m 的乘法逆元有唯一解。如果不互素，则无解。
/*解 ax ≡ b (mod n)    x ≡ (a^-1)b (mod n) */
int rev(int a,int n) //求逆元
{
    int x,y,d;
    exgcd(a,n,d,x,y);
    return d==1?(x+n)%n:-1;
}
int mod_equ(int a,int b,int n) //返回方程解
{
    int d=rev(a,n);
    if(d==-1)return -1;
    return d*b;
}
```

6.线性同余方程组

```
//ai X≡bi （mod mi）
int liner_equs(int n)
{
    int x=0,m=1;
    for(int i=0;i<n;i++) {
        int a=A[i]*m,b=B[i]-A[i]*x,d=gcd(a,M[i]);
        if(b%d) return -1;
        a/=d;b/=d;int cur_m=M[i]/d;
        int t=b*inv(a,cur_m)%cur_m;
        x+=m*t;m*=cur_m;
        x%=m;
    }
    return x;
}
```

7.中国剩余定理

```
/*令 m1,…,mr 两两互素, b1,…,br 为整数,
x≡b1(mod m1),x≡b2(mod m2)……x≡br(mod mr);
有唯一正整数解 x，其形式为：x=∑biMi’Mi(mod M) (1<=i<=r)
M=∏mi(1<=i<=r) ;Mi=M(mod mi);MiMi’ ≡1(mod mi);
 b[0...n-1],m[0..n-1]*/
/*返回值为方程的解*/
int Chi_r()
{
    for(int i=0;i<n;i++)
        M*=m[i];
    int x,y;
    for(int i=0;i<n;i++){
        Mi=M/m[i];
        exgcd(Mi,m[i],x,y);
        Mi1=x;
        ans+=(b[i]*Mi*Mi1)%M;
        ans%=M;
    }
    return ans;
}
```

扩展（非互质版）

```
/*给定 n 个方程 X≡b1(mod m1) ，X≡b2(mod m2)…………X≡bn(mod mn).
m1，m2,……，mn 可以任意取，不一定是互质的。。。*/
bool Chi_r()//方程组是否有解
{
    int m1=m[0],b1=b[0],k1,y;
    for(int i=1;i<n;i++){
        int m2=m[i],b2=b[i];
        int g=exgcd(m1,m2,k1,y);
        if((b2-b1)%g!=0) return 0;
        k1=(k1*((b2-b1)/g)%m2+m2)%m2;
        int t=m2/d;
        int K=(k1%t+t)%t;
        b1+=K*m1;m1=m1/g*m2;
    }
     return 1;//b1%m1 为最小正整数解
}
```

8.离散对数(a^x≡ b(mod n))

```
int log_mod(int a,int b,int n) //a^x=b(mod n)无解返回-1
{
    int e=1;
    int m=sqrt(n+0.5);
    int v=rev(pow_mod(a,m,n),n);//a^(-m)
    map<int,int>x;
    x[1]=0;
    for(int i=1;i<m;i++){
        e=mul_mod(e,a,n);
        if(!x.count(e))x[e]=i;
    }
    for(int i=0;i<m;i++){
        if(x.count(b))return i*m+x[b];
        b=mul_mod(b,v,n);
    }
    return -1;
}
```

9.矩阵快速幂

```
//对于递推关系 利用矩阵快速幂 o(n)的复杂度可以降到 o（logn）
struct matrix
{
    int M[maxn][maxn];int n,m;
    matrix(){   memset(M,0,sizeof(M));}
    matrix multiply(matrix &a,matrix &b)
    //aij=sum(aik*akj)矩阵乘法
    {
        matrix z;
        for(int i=0;i<a.n;i++)
            for(int j=0;j<a.m;j++)
                for(int k=0;k<b.n;k++)
                    z.M[i][j]+=a.M[i][k]*b.M[k][j];
        return z;
    }
    void powmod(matrix &u,matrix a,int n)//矩阵快速幂
    {
        while(n>0){
            if(n&1) u=multiply(u,a);
            a=multiply(a,a);
            n>>=1;
        }
    }
};
```

矩阵快速幂处理斐波那契数列

```
int main()
{
    ll n;
    while(scanf("%lld",&n)!=EOF)// 1≤N≤100,000,000
    {
        Matrix A,B;
        B.m=B.n=2;
        B.a[0][0]=0;
        B.a[0][1]=B.a[1][0]=B.a[1][1]=1;
        A.m=1;A.n=2;
```

```
        A.a[0][0]=1;A.a[0][1]=1;
        B=B.quickpow(B,n-1);
        A=A.multiply(B);
        printf("%lld\n",A.a[0][1]%M);//模 M
    }
    return 0;
}
```

10. 生成子集

```
int C[1010][1010];//所有结果
int p[1010],s[1010];
int n,cnt;
void Copy()
{
    C[cnt][0]=-1;//-1 表示空集
    int j=0;
    for(int i=0;i<n;i++){if(p[i])C[cnt][j++]=i+1;}
    j=0;
    while(C[cnt][j++])printf("%d ",C[cnt][j-1]);
    printf("\n");
    cnt++;
}
void Deal1()//压缩序  二进制序
{
    s[0]=1;cnt=0;
    int num=1<<n;//O(2^n)
    for(int i=0;i<num;i++){
        Copy();
        if(i==num)break;
        for(int j=0;j<n;j++){
            p[j]+=s[j];
            if(p[j]>1){
                p[j+1]++; p[j]-=2;
            }
        }
    }
}
```

```
void Deal2()//生成集合的子集，反射 Gray 码序，相邻子集差异尽可能小。
{
    memset(C,0,sizeof(C));
    memset(p,0,sizeof(p));
    int num=1<<n,sum=0,j;cnt=0;
    for(int i=0;i<num;i++){
        Copy();
        if(sum){
            for(j=0;j<n;j++){if(p[j])break;}
            p[j+1]=!p[j+1]; sum=0;
        }
        else{ p[0]=!p[0]; sum=1;}
    }
}
```

生成 n 元素集合的 r 子集(字典序)

```
/* 设a1a2....ar 是{1,2,...,n}的 r 子集。在字典序中，第一个 r 子集
是 12...r.最后一个 r 子集是（n-r+1）（n-r+2）....n.
假设 a1a2...ar≠（n-r+1）(n-r+2)....n.设 k 是满足 ak<n 且 ak+1 不
等于 ak+1，...,ar 中任一个数的最大整数。那么，在字典序中，
a1a2...ar 的直接后继 r 子集是：a1..ak-1(ak+1)(ak+2)...(ak+r-
k+1)  *//*n 元素的 r 子集 cnt 为子集数 字典序*/
int n,r,cnt;int C[1010][1010],p[1010];
void Copy()
{
    for(int i=1;i<=r;i++){
        C[cnt][i]=p[i];
        printf("%d ",p[i]);
    }
    printf("\n");
    cnt++;
}
void Deal()
{
    cnt=0;
    for(int i=1;i<=r;i++)p[i]=i;
    while(1){
        int k;
```

```
        Copy();
        if(p[1]==n-r+1)break;
        for(k=r;k>=1;k--){
            if(p[k]+1<=n&&p[k]+1!=p[k+1])break;
        }
        int temp=p[k];
        for(int i=k;i<=r;i++){ p[i]=temp+i-k+1;}
    }
}
```

11.生成排列

生成排列(字典序)

```
int c[1010][1010];
int p[1010];
int n,cnt;
void Deal()
{
    for(intj=1;j<=n;j++)
    p[j]=j,printf("%d",p[j]),
    c[cnt][j]=p[j];
    printf("\n");
    int i=n-1;cnt=1;
    while(1){
        while(i>0&&p[i+1]<p[i])i--;
        if(i==0)break;
        int j=n;
        for(;j>i;j--){
            if(p[j]>p[i])break;
        }
        swap(p[i],p[j]);
        for(i=i+1,j=n;i<=j;i++,j--)
            swap(p[i],p[j]);
        for(j=1;j<=n;j++){
            printf("%d",p[j]);
            c[cnt][j]=p[j];
        }
        printf("\n");
        cnt++;
        i=n-1;
    }
}
```

由逆序列构建原排列

//b 为逆序列 a 为原排列

```
int b[10010],a[10010];
int main()
{
    int n;
    while(scanf("%d",&n)!=EOF){
        for(int i=0;i<n;i++)
            scanf("%d",&b[i]);
        for(int i=0;i<n;i++)
        {
            int cnt=0;
            for(int j=0;j<n;j++)
            {
                if(a[j]==0)cnt++;
                if(cnt==b[i]+1)
                {
                    a[j]=i+1;break;
                }
            }
        }
            for(int i=0;i<n-1;i++)
              printf("%d ",a[i]);
            printf("%d\n",a[n-1]);
        }
}
```

12.求一个数的质因子

单个数计算，时间复杂度并不会算

```
ll fac[100],fac_num;
void getfactor(int num)
{
    fac_num=0;
     for(int i=2;i*i<=num;i++){
        if(num%i==0){
            fac[fac_num++]=i;
            while(num%i==0)num/=i;
        }
    }
    if(num>1) fac[fac_num++]=num;
}
```

打质因子表

```
vector<int> fac[10000];
int vis[100011];
void getfactor()
{
    int i,j;
    for(i=0;i<10010;i++)
       fac[i].clear();//vector 的清空
    memset(vis,0,sizeof(vis));
    for(i=2;i*i<=10000;i++){
        if(vis[i]==0){//i 是素数
            fac[i].push_back(i);
            for(j=i+i;j<=10000;j+=i){
                vis[j]=1;
                fac[j].push_back(i);
            }
        }
    }
}
```

13.容斥原理

$$|A_1 \cup A_2 \cup \cdots \cup A_m| = \sum_{1\le i\le m} |A_i| - \sum_{1\le i<j\le m} |A_i \cap A_m| + \sum_{1\le i<j<k\le m} |A_i \cap A_j \cap A_k| - \cdots + (-1)^{m-1} |A_1 \cap A_2 \cap \cdots \cap A_m|$$

(1,r)区间与 n 互质/不互质的数的个数

```
void getfactor(__int64 num);//单个
```

二进制

```
__int64 coprime(__int64 num,__int64 r)
{
    int mm=(1<<fac.size());
    __int64 ans=0;
    for(int i=1;i<mm;i++){
        int cnt=0;__int64 temp=1;
        for(int j=0;j<fac.size();j++){
            if(i&(1<<j)){
                cnt++;
                temp*=fac[j];
            }
        }
        if(cnt&1)
                ans+=r/temp;
            else ans-=r/temp;
     }
     return r-ans;//返回互质数
}
```

dfs 版：

```
//cnt 为质因子个数，num 为所用因子个数，val 为因子值
int x;//x 为非互质数
void dfs(int idx,int num,int val,int R,int cnt)
{
    if(idx==cnt)
    {
        if(val==1)return;
        if(num&1)x+=R/val;
        else x-=R/val;
        return;
    }
    dfs(idx+1,num+1,val*p[idx],R,cnt);
    dfs(idx+1,num,val,R,cnt);
}
```

a1~an 与 ai 不互质的个数。

```
//同色三角形模型
#define MAXN 100010
int a[MAXN],ss[MAXN];
//ss[i]统计含 i 因子的数个数
int p[MAXN];bool isprime[MAXN];
void getprime();
void getfactor();
void init(int n){//ai 初始化
   getfactor(n);
    int mm=1<<c;
    for(int i=1;i<mm;i++){
        int temp=1;
        for(int j=0;j<c;j++){
            if(i&(1<<j)){
                temp*=fac[j];
            }
        }
        ss[temp]++;
    }
}
```

```
__int64 coprime(int num,int n)
{
   getfactor(num);
    int mm=1<<c;
    for(int i=1;i<mm;i++){
        int cnt=0,temp=1;
        for(int j=0;j<c;j++){
            if(i&(1<<j)){
                cnt++;
                temp*=fac[j];
            }
        }
        if(cnt&1) sum+=ss[temp];
        else  sum-=ss[temp];
    }
    if(sum==0)return 0;
    return (sum-1)*(n-sum);
    //sum 包含了 ai 本身
}
```

1~r 中被 m 集合任意一个数整除的数的个数

```
int p[20]; //p 为m 集合
int n,m;int cnt=0;
int gcd(int a,int b){
    if(!b)return a;
    return gcd(b,a%b);
}
int LCM(int a,int b)
{
    return a/gcd(a,b)*b;
}
int ans=0;//答案
```

```
void dfs(int num,int val,int sign)
{
    if(num>=0)
    {
        ans+=sign*(n-1)/val;
    }
    for(int i=num+1;i<cnt;i++)
    {
        dfs(i,LCM(p[i],val),-sign);
    }
}
```

给定 (1,b),(1,d),求 gcd(x,y)=k 的对数。x 属于(1,b),y 属于(1,c)。

```
//考虑正反即(x,y),(y,x)都算
#define MAXN 100000
int p[MAXN+10]; //素因子
__int64 phi[MAXN+10];
//phi[]为 phi 前缀和
int n,m;int cnt=0;
int x=0;
void dfs(int idx,int num,int val,int
R);
void getphi();
void getfac(int x);
int main()
{
    getphi();
    int t;
    int ica=1;
    scanf("%d",&t);
    while(t--){
        int m,n;
        scanf("%d%d",&m,&n);
        int max=0,min=0;
```

```
 if(m>n)max=m,min=n;
        else max=n,min=m;
        __int64 ans=2*phi[min];
        for(int i=min+1;i<=max;i++)
        {
            getfac(i);
            x=0;
            dfs(0,0,1,min);
            ans+=min-x;
        }
        printf("%I64d\n",ans-1);
    }
    return 0;
}
```

不考虑正反

```
#define MAXN 100000
int p[MAXN+10];
__int64 phi[MAXN+10];
int n,m;int cnt=0;
int x=0;
void dfs(int idx,int num,int val,int
R);
void getphi();
void getfac(int x);
int main()
{
    getphi();
    int t;
    int ica=1;
    scanf("%d",&t);
while(t--){
        int m,n;
        scanf("%d%d",&m,&n);
        int max=0,min=0;
        if(m>n)max=m,min=n;
        else max=n,min=m;
        __int64 ans=phi[min];
        for(int i=min+1;i<=max;i++)
        {
            getfac(i);
            x=0;
            dfs(0,0,1,min);
            ans+=min-x;
        }
        printf("%I64d\n",ans);
    }
    return 0;
}
```

14.高斯消元

整数方程组

```
const int MAXN=50;
int a[MAXN][MAXN];//增广矩阵
int x[MAXN];//解集
bool free_x[MAXN];//标记是否是不确定的变元
inline int gcd(int a,int b){
    int t;
    while(b!=0){
        t=b; b=a%b;
        a=t;
    }
    return a;
}
inline int lcm(int a,int b){
    return a/gcd(a,b)*b;//先除后乘防溢出
}
//高斯消元法解方程组(Gauss-Jordan elimination).(-2 表示有浮点数解，但无整数解，
//-1 表示无解，0 表示唯一解，大于 0 表示无穷解，并返回自由变元的个数)
```

//有 equ 个方程，var 个变元。增广矩阵行数为 equ,分别为 0 到 equ-1,列数为 var+1,分别为 0 到 var.

```
int Gauss(int equ,int var)
{
    int i,j,k;int max_r;// 当前这列绝对值最大的行.
    int col;//当前处理的列
    int ta,tb,LCM,temp,free_x_num,free_index;
    for(int i=0;i<=var;i++) {//初始化，均为自由变元
        x[i]=0;free_x[i]=true;
    }
    //转换为阶梯阵.
    col=0; // 当前处理的列
    for(k = 0;k < equ && col < var;k++,col++){ // 枚举当前处理的行.
      // 找到该 col 列元素绝对值最大的那行与第 k 行交换.(为了在除法时减小误差)
        max_r=k;
        for(i=k+1;i<equ;i++){
            if(abs(a[i][col])>abs(a[max_r][col])) max_r=i;
        }
        if(max_r!=k){// 与第 k 行交换.
            for(j=k;j<var+1;j++) swap(a[k][j],a[max_r][j]);
        }
// 说明该 col 列第 k 行以下全是 0 了，则处理当前行的下一列.
        if(a[k][col]==0) {
            k--; continue;
        }
        for(i=k+1;i<equ;i++){// 枚举要删去的行.
            if(a[i][col]!=0) {
                LCM = lcm(abs(a[i][col]),abs(a[k][col]));
                ta = LCM/abs(a[i][col]);
                tb = LCM/abs(a[k][col]);
                if(a[i][col]*a[k][col]<0)tb=-tb;//异号的情况是相加
                for(j=col;j<var+1;j++){
                    a[i][j] = a[i][j]*ta-a[k][j]*tb;
                }
            }
        }
    }
```

```
// 1. 无解的情况: 化简的增广阵中存在(0, 0, ..., a)这样的行(a != 0).
    for (i = k; i < equ; i++){
        if (a[i][col] != 0) return -1;
    }
    // 2. 无穷解的情况: 在 var * (var + 1)的增广阵中出现(0, 0, ..., 0)这样的行，即说明没有形成严格的上三角阵.
    // 且出现的行数即为自由变元的个数.
    if (k < var){
        // 首先，自由变元有 var - k 个，即不确定的变元至少有 var - k 个.
        for (i = k - 1; i >= 0; i--) {
     //第 i 行一定不会是(0, 0, ..., 0)的情况，因为这样的行是在第 k 行到第 equ 行.
     // 同样，第 i 行一定不会是(0, 0, ..., a), a != 0 的情况，这样的无解的.
            free_x_num = 0; // 用于判断该行中的不确定的变元的个数，如果超过 1 个，则无法求解，它们仍然为不确定的变元.
            for (j = 0; j < var; j++){
                if (a[i][j] != 0 && free_x[j])
                free_x_num++,free_index = j;
            }
            if (free_x_num > 1) continue; // 无法求解出确定的变元.
            // 说明就只有一个不确定的变元 free_index，那么可以求解出该变元，且该变元是确定的.
            temp = a[i][var];
            for (j = 0; j < var; j++){
                if (a[i][j] != 0 && j != free_index)
                    temp -= a[i][j] * x[j];
            }
            x[free_index] = temp / a[i][free_index]; // 求出该变元.
            free_x[free_index] = 0; // 该变元是确定的.
        }
        return var - k; // 自由变元有 var - k 个.
    }
    // 3. 唯一解的情况: 在 var * (var + 1)的增广阵中形成严格的上三角阵.
    // 计算出 Xn-1, Xn-2 ... X0.
    for (i = var - 1; i >= 0; i--){
        temp = a[i][var];
        for (j = i + 1; j < var; j++){
            if (a[i][j] != 0) temp -= a[i][j] * x[j];
        }
```

```
        if (temp % a[i][i] != 0) return -2; // 说明有浮点数解，但无整数解.
        x[i] = temp / a[i][i];
    }
    return 0;
}
int main(void)
{
    int i, j;
    int equ,var;
    while (scanf("%d%d", &equ, &var) != EOF){
        memset(a, 0, sizeof(a));
        for (i = 0; i < equ; i++){
            for (j = 0; j < var + 1; j++){
                scanf("%d", &a[i][j]);
            }
        }
        int free_num = Gauss(equ,var);
        if (free_num == -1) printf("无解!\n");
        else if (free_num == -2) printf("有浮点数解，无整数解!\n");
        else if (free_num > 0){
            printf("无穷多解! 自由变元个数为%d\n", free_num);
            for (i = 0; i < var; i++){
                if (free_x[i]) printf("x%d 是不确定的\n", i + 1);
                else printf("x%d: %d\n", i + 1, x[i]);
            }
        }
        else{
            for (i = 0; i < var; i++){
                printf("x%d: %d\n", i + 1, x[i]);
            }
        }
        printf("\n");
    }
    return 0;
}
```

//精简版

```
#define eps 1e-9
const int MAXN=220;
double a[MAXN][MAXN],x[MAXN];
//方程的左边的矩阵和等式右边的值，求解之后
x 存的就是结果
int equ,var;//方程数和未知数个数
/* *返回 0 表示无解，1 表示有解*/
int Gauss()
{
    int i,j,k,col,max_r;
   for(k=0,col=0;k<equ&&col<var;k++,col++){
       max_r=k;
        for(i=k+1;i<equ;i++)
            if(fabs(a[i][col])>fabs(a[max_r][col]))  max_r=i;
        if(fabs(a[max_r][col])<eps)return 0;
        if(k!=max_r){
            for(j=col;j<var;j++)
              swap(a[k][j],a[max_r][j]);
            swap(x[k],x[max_r]);
        }
        x[k]/=a[k][col];
        for(j=col+1;j<var;j++)a[k][j]/=a[k][col];
            a[k][col]=1;
        for(i=0;i<equ;i++)
        if(i!=k){
            x[i]-=x[k]*a[i][k];
            for(j=col+1;j<var;j++)a[i][j]-=a[k][j]*a[i][col];
              a[i][col]=0;
          }
    }
    return 1;
}
```

常用定理与结论

欧拉定理：

费马小定理：

假如 p 是质数，且 gcd(a,p)=1，那么 a^(p-1) ≡1（mod p）。即：假如 a 是整数，p 是质数，且 a,p 互质(即两者只有一个公约数 1)，那么 a 的(p-1)次方除以 p 的余数恒等于 1。

数据结构

RMQ

```
#define maxn 100010
#define maxm 6  //种类数
typedef long long ll;
ll n,m,k;
ll Array[maxn][maxm],dp[maxm][maxn][20];
void initRMQ()
{
    for(int i=0;i<m;i++)
        for(int j=0;j<n;j++)
            dp[i][j][0]=Array[j][i];
    for(int k=0;k<m;k++){
        for(int j=1;(1<<j)<=n;j++) {
            for(int i=0;n-i>=(1<<j);i++)
              dp[k][i][j]=MAX(dp[k][i][j-1],dp[k][i+(1<<(j-1))][j-1]);
        }
    }
}
ll RMQ(int kind,int l,int r){
    int k=0;
    while(1<<(k+1)<=r-l+1)k++;
    return MAX(dp[kind][l][k],dp[kind][r-(1<<k)+1][k]);
}
```

线段树—区间最值

```
#define lson 2*o,l,mid
#define rson 2*o+1,mid+1,r
int n,m,k;int tree[4*maxn];int
Array[maxn];
void build_tree(int o,int l,int r){
    if(r<l)return;
    if(l==r){tree[o]=Array[l-
1];return;}
    int mid=(l+r)>>1;
    build_tree(lson); build_tree(rson);
    int a=2*o;
    tree[o]=MAX(tree[a],tree[a+1]);
}
```

```
int query(int o,int l,int r,int a,int b)
{
    if(a>r||b<l||r<l)return 0;
    if(a<=l&&r<=b)  { return tree[o];}
    int mid=(l+r)>>1;
    int aa=query(lson,a,b);
    int bb=query(rson,a,b);
    return MAX(aa,bb);
}
```

线段树—区间更新（lazy 标记）

```
#define lson 2*o,l,mid
#define rson 2*o+1,mid+1,r
typedef long long ll;
ll tree[1000100];ll addc[1000100];
int N,Q;
void pushdown(int o,int l,int r){
    if(addc[o]){
        int mid=(l+r)/2;
        addc[2*o]+=addc[o];
        addc[2*o+1]+=addc[o];
        tree[2*o]+=(mid-l+1)*addc[o];
        tree[2*o+1]+=(r-mid)*addc[o];
        addc[o]=0;
    }
}
```

```
void insert(int o,int l,int r,int a,int b,ll c){//c 为增量
    if(b<l||a>r)return;
    if(a<=l&&r<=b){
        addc[o]+=c;
        tree[o]+=(r-l+1)*c;
        return;
    }
    int mid=(l+r)/2;
    //不是完整节点，标记下传
    pushdown(o,l,r);
    insert(lson,a,b,c);
    insert(rson,a,b,c);
    tree[o]=tree[2*o]+tree[2*o+1];
}
```

```
ll query(int o,int l,int r,int a,int b){
    if(b<l||a>r)return 0;
    if(a<=l&&r<=b){return tree[o];}
    pushdown(o,l,r);
    int mid=(l+r)/2;
    return query(lson,a,b)+query(rson,a,b);
}
```

函数化线段树

多次询问区间第 K 大

```
#include<cstdio>
#include<iostream>
#include<algorithm>
#include<map>
using namespace std;
#define maxn 100005
#define lson l,mid
#define rson mid+1,r
map<int,int>ms;int node=0;
int head[maxn],a[maxn],b[maxn],sum[maxn<<5],L[maxn<<5],R[maxn<<5];
//函数化线段树就是在保留之前节点的基础上新建节点，并且利用之前的节点进行压缩空间。
```

```
//此处是 n*logn 的空间复杂度。利用了区间数值数目的可加性。
//[1,l] 的线段树存储的是权值在到 al 为止（1，num）范围内数值的出现次数。
//两个线段树相减就是在区间范围内某些数字的出现次数。进而统计第 K 大的数字。
void init(){
    memset(head,0,sizeof(head));
    node=0;ms.clear();
}
void build(int &o,int l,int r) {//dfs 序建立基础线段树
    o=++node;sum[o]=0;
    if(l>=r)return;
    int mid=(l+r)>>1;
    build(L[o],lson); build(R[o],rson);
}//pre 表示同样位置的上一棵树节点
void update(int pre,int &o,int l,int r,int x) {
    o=++node;
    L[o]=L[pre];R[o]=R[pre];sum[o]=sum[pre]+1;
    int mid=(l+r)>>1;
    if(l>=r)return;
    if(x<=mid) update(L[pre],L[o],lson,x);
    else  update(R[pre],R[o],rson,x);
}
int query(int l,int r,int a,int b,int k){
    if(l==r)return l;
    int mid=(l+r)>>1;
    int x=sum[L[b]]-sum[L[a]];
    if(x>=k) return query(lson,L[a],L[b],k);
    else return query(rson,R[a],R[b],k-x);
}
int main(){
    int t,l,r,k; int n,m;
    scanf("%d",&t);
    while(t--){
        init();
        scanf("%d%d",&n,&m);
        for(int i=1;i<=n;i++){  scanf("%d",&a[i]);  b[i]=a[i];}
        sort(b+1,b+n+1);
        int num=unique(b+1,b+n+1)-b-1;
        for(int i=1;i<=num;i++)ms[b[i]]=i;
```

```
        build(head[0],1,n);
    for(int i=1;i<=n;i++){
        update(head[i-1],head[i],1,num,ms[a[i]]);
    }
    while(m--){
        scanf("%d%d%d",&l,&r,&k);
        printf("%d\n",b[query(1,num,head[l-1],head[r],k)]);
    }
  }
  return 0;
}
```

树状数组

```
int lowbit(int x){
    return x & (-x);
}
```

一维

```
void update(int x,int add)
{
    while(x<=MAXN)  {
        a[x]+=add;
        x+=lowbit(x);
    }
}
int getsum(int x)
{
    int ret=0;
    while(x!=0)  {
        ret+=a[x];
        x-=lowbit(x);
    }
    return ret;
}
```

二维

```
int tree[maxn][maxn];
void update(int x,int y,int w){
    for(int i=x;i<maxn;i+=lowbit(i)){
        for(int j=y;j<maxn;j+=lowbit(j)){
            tree[i][j]+=w;
        }
    }
}
int getsum(int x,int y){
    int sum=0;
    for(int i=x;i>0;i-=lowbit(i)){
        for(int j=y;j>0;j-=lowbit(j)){
            sum+=tree[i][j];
        }
    }
    return sum;
}
```

图论

1.最短路

```
Dijstra(N,sx,ex);//复杂度O(n*n)
int map[210][210], dist[210];
void Dijstra(int n,int x,int y)
{
    bool mark[210]={false};
    int i,j,p;
    for(i=0;i<n;i++)dist[i]=map[x][i];
    dist[x]=0;mark[x]=true;
    for(i=0;i<n;i++){
        int min=INF;
        for(j=0;j<n;j++){
            if(!mark[j]&&dist[j]<min){
                min=dist[j]; p=j;
            }
        }
        if(min==INF)break;
        mark[p]=true;
        for(j=0;j<n;j++){
            if(!mark[j]&&dist[j]>dist[p]+map[p][j]){
                dist[j]=dist[p]+map[p][j];
            }
        }
    }
}
```

SPFA

```
queue<int>q;
int dist[210],map[210][210];
void SPFA(int citynum,int x)
{
    int i,k;
    bool visit[110]={false};
    for(i=0;i<=citynum;i++){
        dist[i]=INF;
    }
    dist[x]=0;visit[x]=true;
```

```
    q.push(x); q.pop();
    while(!q.empty()){
        k=q.front();
        for(i=0;i<citynum;i++){
            if(dist[i]>dist[k]+map[k][i]){
                dist[i]=dist[k]+map[k][i];
                if(!visit[i]){
                    q.push(i); visit[i]=true;
                }
            }
        }
    visit[k]=false;
    }
}
```

Floyed

```
void Floyd(int n,int x,int y)
{
    int i,j,k;
    for(k=0;k<n;k++)
        for(i=0;i<n;i++)
            for(j=0;j<n;j++)
                map[i][j]=min(map[i][j],map[i][k]+map[k][j]);

}
```

2. 最小生成树

2.1 Prim 算法

```
/ * 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
  */
const int INF=0x3f3f3f3f;
const int MAXN=110;
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;
}
```

2.2 Kruskal 算法

```
const int MAXN=110;//最大点数
const int MAXM=10000;//最大边数
int F[MAXN];//并查集使用
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 find(int x){
    if(F[x]==-1)return x;
    else return F[x]=find(F[x]);
}
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
```

```
{
    memset(F,-1,sizeof(F));
    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;
        int t1=find(u);int t2=find(v);
        if(t1!=t2){
            ans+=w;F[t1]=t2;cnt++;
        }
        if(cnt==n-1)break;
    }
    if(cnt<n-1)return -1;//不连通
    else return ans;
}
```

3.有向图的强连通分量

Tarjan 算法（DAG）

```
vector<int>e[1010];//原图
vector<int>e2[1010];//缩点后的图
stack<int>s;//s 存节点
int dfn[1010],low[1010];
//dfn 开始时间 low 为到达节点的最早开始时间
int indx,col;//indx 为节点编号 col 为染色
int vis[1010],sta[1010];//vis 全局标记节点是否访问过，sta 标记是否在栈中
int flag[1010];
int num[1010];//每种颜色点个数
void Tarjan(int u)//重点
{
    vis[u]=1;
    dfn[u]=low[u]=indx++;//最早到达时间
    s.push(u);  sta[u]=1;
    int l=e[u].size();
    for(int i=0;i<l;i++){
        int k=e[u][i];
        if(!vis[k]){
            Tarjan(k);
            low[u]=MIN(low[u],low[k]);
        }
        else if(sta[k]){
            low[u]=MIN(low[u],dfn[k]);
        }
    }
    if(dfn[u]==low[u]){
        int k=s.top();
        while(k!=u){//同一强连通分量染色
            flag[k]=col;num[col]++;
            s.pop();
            sta[k]=0;
            k=s.top();
        }
        flag[u]=col;s.pop();
        sta[u]=0;num[col]++;col++;
    }
}
void init(int n)
{
    for(int i=0;i<=n;i++)
        e[i].clear(),e2[i].clear();
    memset(dfn,0,sizeof(dfn));
    memset(low,0,sizeof(low));
    memset(vis,0,sizeof(vis));
    memset(sta,0,sizeof(sta));
    memset(flag,0,sizeof(flag));
    memset(num,0,sizeof(num));
    while(!s.empty())s.pop();
}
int main()
{
        int n,m;
        scanf("%d%d",&n,&m);
        init(n);
        while(m--){//建原图
            int a,b;
            scanf("%d%d",&a,&b);
            e[a].push_back(b);
        }
        indx=1;col=1;
        for(int i=1;i<=n;i++) {//防止图不连通
            if(!vis[i]) Tarjan(i);
        }
        for(int i=1;i<=n;i++){//缩点
            int l=e[i].size();
            for(int j=0;j<l;j++){
                int a=i,b=e[i][j];
                if(flag[a]!=flag[b]){
                    e2[flag[a]].push_back(flag[b]);
                }
            }
        }
    return 0;
}
```

4. 无向图求桥和割点

```
vector<int>e[5100];
int index;
//low[u]定义为 u 或者 u 的子树中能够通过
//非父子边追溯到的最早的节点的 DFS 开
//始时间
int dfn[5100],low[5100];
int vis[5100];
bool cut[10010];//割点
int root;
void tarjan(int u,int f)
{
    vis[u]=1;
    dfn[u]=low[u]=index++;
    int l=e[u].size();
    for(int i=0;i<l;i++){
        int k=e[u][i];
        if(!vis[k]){
            tarjan(k,u);
            low[u]=MIN(low[u],low[k]);
            if(low[k]>=dfn[u]){
                cut[u]=true;
                if(low[k]>dfn[u])//桥
                    ...
            }
        }
        else {
            if(k!=f)
                low[u]=MIN(low[u],dfn[k]);
        }
    }
    if(u==root){
        if(l>=2)cut[u]=true;
    }
}
```

5.点双连通分量（不含割点的极大连通子图）

```
struct Edge{int f,t,col;}E[10100];
vector<int>e[10100];stack<Edge>s;//存储的边
int dfn[10010],low[10010];
int vis[10010];int node[10010];
int index,circle;int n,m;
int cal(){//计算点双连通图中节点数
    int ans=0;
    for(int i=0;i<n;i++){if(node[i])ans++;}
    return ans;
}
void tarjan(int u,int f){
    vis[u]=1; Edge edge;
dfn[u]=low[u]=index++;int l=e[u].size();
    for(int i=0;i<l;i++) {
        int v=e[u][i];
        if(!vis[v]){
            edge.f=u;edge.t=v;s.push(edge);
            tarjan(v,u);
            low[u]=MIN(low[u],low[v]);
            if(low[v]>=dfn[u]){//u 是割点
                int num=0;
                memset(node,0,sizeof(node));
                while(1){//点双连通分量
                    Edge ed; ed=s.top();
                    node[ed.f]++;node[ed.t]++;
                    s.pop();num++;//边数
                    if(ed.f==u)break;
                }
                int nodenum=cal();//节点数
                //边数 num 大于点数 nodenum 有环。
            }
        }
        else{
            if(v!=f&&dfn[u]>dfn[v]){
            Edge edge;
            edge.f=u;edge.t=v;s.push(edge);
            low[u]=MIN(low[u],dfn[v]);}
        }
    }
}
```

6. 求无向连通图边双连通分支(不包含桥的极大连通子图）：

只需在求出所有的桥以后，把桥边删除，原图变成了多个连通块，则每个连通块就是一个边双连通分支。桥不属于任何一个边双连通分支，其余的边和每个顶点都属于且只属于一个边双连通分支。

7. 构造双连通图

一个有桥的连通图，如何把它通过加边变成边双连通图？方法为首先求出所有的桥，然后删除这些桥边，剩下的每个连通块都是一个双连通子图。把每个双连通子图收缩为一个顶点，再把桥边加回来，最后的这个图一定是一棵树，边连通度为 1。

统计出树中度为 1 的节点的个数，即为叶节点的个数，记为 leaf。则至少在树上添加(leaf+1)/2 条边，就能使树达到边二连通，所以至少添加的边数就是(leaf+1)/2。具体方法为，首先把两个最近公共祖先最远的两个叶节点之间连接一条边，这样可以把这两个点到祖先的路径上所有点收缩到一起，因为一个形成的环一定是双连通 的。然后再找两个最近公共祖先最远的两个叶节点，这样一对一对找完，恰好是(leaf+1)/2 次，把所有点收缩到了一起。

```
#include<stdio.h>
#include<string.h>
#include<vector>
#include<stack>
#include<algorithm>
#define MIN(a,b) a>b?b:a
using namespace std;
vector<int>e[5100];
stack<int>s;
int index,color;
int dfn[5100],low[5100];
int vis[5100],flag[5100];
int d[5100];
void tarjan(int u,int f)
{
   vis[u]=1;
   dfn[u]=low[u]=index++;
   s.push(u);
   int l=e[u].size();
   for(int i=0;i<l;i++){
      int k=e[u][i];
if(!vis[k]){
         tarjan(k,u);
         low[u]=MIN(low[u],low[k]);
         if(low[k]>dfn[u]){//桥
            while(1){//双连通子图
               int v=s.top();s.pop();
               flag[v]=color;
               if(k==v)break;
            }
            color++;
         }
      }
      else{
         if(k!=f)
            low[u]=MIN(low[u],dfn[k]);
      }
   }
}
```

```
void init(int n)
{
    for(int i=0;i<=n;i++)
        e[i].clear();
    while(!s.empty())s.pop();
    memset(d,0,sizeof(d));
    memset(vis,0,sizeof(vis));
    memset(flag,0,sizeof(flag));
    memset(low,0,sizeof(low));
    memset(dfn,0,sizeof(dfn));
}
int main()
{
    int n,m;
    while(scanf("%d%d",&n,&m)!=EOF){
        init(n);
        while(m--){
            int a,b;
            scanf("%d%d",&a,&b);
            int l=e[a].size();
            int flag1=1;
            for(int i=0;i<l;i++){
                if(e[a][i]==b)
                    flag1=0;
            }
            if(flag1){
                e[a].push_back(b);
                e[b].push_back(a);
            }
        }
        index=color=1;
        tarjan(1,1);
        for(int i=1;i<=n;i++)//缩点 {
            int l=e[i].size();
            for(int j=0;j<l;j++){
                int a=i,b=e[i][j];
                if(flag[a]!=flag[b]){
                    d[flag[a]]++;
                }
            }
        }
        int cnt=0;
        for(int i=0;i<color;i++){
            if(d[i]==1)
                cnt++;
        }
        printf("%d\n",(cnt+1)/2);
    }
    return 0;
}
```

8.最近公共祖先(离线 Tarjan)

```
struct EDGE{  int u,v,next,w;}edge[501000];
int head[10100], parent[10100],ancestor[10100];
bool checked[10100];int inde[10010];
int cnt;int fr,to;
void addedge(int u,int v){
      edge[cnt].u=u;edge[cnt].v=v;edge[cnt].w=1;
      edge[cnt].next=head[u];head[u]=cnt++;
}
int find(int x){
     if(parent[x]==x)return x;
     return parent[x]=find(parent[x]);
}
```

```
void merge(int a,int b){
    int m=find(a),n=find(b);
    parent[m]=n;
}
int ans;//两点最近公共祖先
void TarjanLCA(int x){
    checked[x]=true; parent[x]=x;ancestor[x]=x;
    for(int i=head[x];i!=-1;i=edge[i].next){
        int v=edge[i].v;
        if(!checked[v]){
            TarjanLCA(v); merge(x,v);
            ancestor[find(x)]=x;
        }
    }
    if(x==fr&&checked[to]){ans=ancestor[find(to)]; return;}
    else if(x==to&&checked[fr]){ans=ancestor[find(fr)]; return;}
}
void init(){
    cnt=0;
    memset(head,-1,sizeof(head));
    memset(checked,false,sizeof(checked));
    memset(inde,0,sizeof(inde));
}
```

在线算法（欧拉序列+RMQ）

欧拉序列即为 dfs 节点遍历序列

```
const int MAXN = 10010;
int rmq[2*MAXN];//rmq 数组，就是欧拉序列对应的深度序列
struct ST{
    int mm[2*MAXN];int dp[2*MAXN][20];//最小值对应的下标
    void init(int n){
        mm[0] = -1;
        for(int i = 1;i <= n;i++){
            mm[i] = ((i&(i-1)) == 0)?mm[i-1]+1:mm[i-1];
            dp[i][0] = i;
        }
        for(int j = 1; j <= mm[n];j++)
        for(int i = 1; i + (1<<j) - 1 <= n; i++)
```

```
        dp[i][j] = rmq[dp[i][j-1]] < rmq[dp[i+(1<<(j-1))][j-1]]?dp[i][j-1]:dp[i+(1<<(j-1))][j-1];
    }
    int query(int a,int b){//查询[a,b]之间最小值的下标
        if(a > b)swap(a,b);
        int k = mm[b-a+1];
        return rmq[dp[a][k]] <= rmq[dp[b-(1<<k)+1][k]]?dp[a][k]:dp[b-(1<<k)+1][k];
    }
};
struct Edge{int to,next;};//边的结构体定义
Edge edge[MAXN*2]; int tot,head[MAXN];
int F[MAXN*2];//欧拉序列，就是 dfs 遍历的顺序，长度为 2*n-1,下标从 1 开始
int P[MAXN];//P[i]表示点 i 在 F 中第一次出现的位置
int cnt;//节点数
ST st;
void init(){//邻接表初始化
    tot = 0; memset(head,-1,sizeof(head));
}
void addedge(int u,int v){//加边，无向边需要加两次
    edge[tot].to = v;edge[tot].next = head[u]; head[u] = tot++;
}
void dfs(int u,int pre,int dep){
    F[++cnt] = u;rmq[cnt] = dep;  P[u] = cnt;
    for(int i = head[u];i != -1;i = edge[i].next){
        int v = edge[i].to;
        if(v == pre)continue;
        dfs(v,u,dep+1); F[++cnt] = u; rmq[cnt] = dep;
    }
}
void LCA_init(int root,int node_num){//查询 LCA 前的初始化
    cnt = 0; dfs(root,root,0); st.init(2*node_num-1);
}
int query_lca(int u,int v){//查询 u,v 的 lca 编号
    return F[st.query(P[u],P[v])];
}
bool flag[MAXN];//用来找树根
```

```
int main()
{
    int T;int N;int u,v; int root;
    scanf("%d",&T);
    while(T--){
        scanf("%d",&N);
        init();memset(flag,false,sizeof(flag));
        for(int i = 1; i < N;i++){
            scanf("%d%d",&u,&v);
            addedge(u,v); addedge(v,u);
            flag[v] = true;
        }
        for(int i = 1; i <= N;i++)
        if(!flag[i]){root = i;break;}
        LCA_init(root,N);
        scanf("%d%d",&u,&v);
        printf("%d\n",query_lca(u,v));
      }
      return 0;
}
```

9.最大流

Dinic 算法

```
int S,T;int n,m,cnt,sum;//S 为源点，T 为汇点
int mark[MAXN];int head[MAXN];
struct node{int u,v,w,next;}edge[MAXN*MAXN];
void addedge(int u,int v,int w){
    edge[cnt].u=u; edge[cnt].v=v;edge[cnt].w=w;
    edge[cnt].next=head[u];head[u]=cnt++;
     edge[cnt].u=v;edge[cnt].v=u;edge[cnt].w=0;
    edge[cnt].next=head[v];head[v]=cnt++;
}
bool bfs(int x)
{
    memset(mark,-1,sizeof(mark));
    mark[x]=1; queue<int>q;q.push(x);
    while(!q.empty()){
```

```
        int k=q.front();q.pop();
        for(int i=head[k];i!=-1;i=edge[i].next) {
            int u=edge[i].v;
            if(mark[u]==-1&&edge[i].w){
                mark[u]=mark[k]+1;q.push(u);
            }
        }
    }
    return mark[et]!=-1;
}
int dfs(int x,int delta)
{
    int min,cost=0;
    if(x==et)return delta;
    for(int i=head[x];i!=-1;i=edge[i].next){
        int u=edge[i].v;
        if(mark[x]==mark[u]-1&&edge[i].w){
            min=dfs(u,MIN(delta-cost,edge[i].w));
            if(min>0){
                edge[i].w-=min;
                edge[i^1].w+=min;
                cost+=min;
                if(cost==delta)break;
            }
            else{
                mark[u]=-1;
            }
        }
    }
    return cost;
}
void Dinic()//建图后直接调用
{
    int ans=0;
    while(bfs(S))  ans+=dfs(S,INF);
    printf("%d\n".ans);
}
```

10.最小费用最大流

```
//maxm 为点数，maxn 为边数
int N,M,K,cnt;
int src,des;
struct node{
    int u,v,w,c,next;//u 是起点，v 是终点，w 是容量，c 为费用
}edge[maxn];
int head[maxm],pre[maxm],dist[maxm],delta[maxm];
void addedge(int u,int v,int w,int c){
    edge[cnt].u=u;edge[cnt].v=v;edge[cnt].w=w;
      edge[cnt].c=c;edge[cnt].next=head[u];head[u]=cnt++;
    edge[cnt].u=v;edge[cnt].v=u;edge[cnt].w=0;
      edge[cnt].c=-c;edge[cnt].next=head[v];head[v]=cnt++;
}
void init()
{
    cnt=0;
    memset(head,-1,sizeof(head));
    memset(edge,-1,sizeof(edge));
    memset(pre,-1,sizeof(pre));
}
bool SPFA(int x){//在残留网络上寻找最小费用增流链
    memset(pre,-1,sizeof(pre));//不能少
    queue<int>q;
    q.push(x);
    bool vis[maxm]={false};
    vis[x]=true;
    for(int i=0;i<maxm-1;i++)
        dist[i]=INF;
    dist[x]=0;         //费用
    delta[x]=INF;     //最小限制流量
    while(!q.empty()){
        int k=q.front();q.pop();vis[k]=0;
        for(int i=head[k];i!=-1;i=edge[i].next){
            int u=edge[i].v;
            if(edge[i].w>0&&dist[u]>dist[k]+edge[i].c){
                dist[u]=edge[i].c+dist[k];//更新最小费用
                pre[u]=i;//记录路径
```

```
                delta[u]=MIN(delta[k],edge[i].w);
                if(!vis[u]){
                    vis[u]=1;
                    q.push(u);
                }
            }
        }
    }
    return pre[des]!=-1;//找不到增流链
}
void update()
{
    int x=des;
    while(x!=src){
        edge[pre[x]].w-=delta[des];
        edge[pre[x]^1].w+=delta[des];
        x=edge[pre[x]].u;
    }
}
int MCMF()
{
    int flow=0,Cost=0;//flow 为总流量，Cost 为总费用
    while(SPFA(src)){
        Cost+=dist[des];
        flow+=delta[des];
        update();     //更新残留网络
    }
   return Cost;
}
```

11.