Primeritat i factorització

Artur Travesa

(versió 2024-07)

Apendix 0. Manual de les funcions

A0. Introducció

En aquest apèndix es recullen les funcions definides en els capítols del text, així com també una breu guia del seu funcionament.

A0.1. Garbell d'Eratòstenes

A0.1.0. El garbell

ff és la fita per al garbell. Proporciona una lista de zeros i uns. En $i = 0$ hi ha un 1 per a indicar que 2 és primer; per a $i \geq 1$, en el lloc i-èsim hi ha un 1, si, i només si, 2i+1 és primer.

In [ ]:
```
def Eratostenes(ff):
    f=floor((ff+1)/2)
    pr=[1 for i in range(f)]
    i=1
    k=floor((sqrt(ff)+1)/2)
    while i<k:
        if pr[i]==1:
            for j in range(2*i*(i+1),f,2*i+1):
                pr[j]=0
        i=i+1
    return pr

```

A0.1.1. La llista de primers

ff és la fita per al garbell. Proporciona la lista dels nombres naturals primers menors que la fita ff.

Observació: SageMath incorpora la funció prime_range( ) que permet resoldre aquesta tasca.

In [ ]:
```
def LlistaDePrimers(ff):
    f=floor((ff+1)/2)
    pr=[1 for i in range(f)]
    i=1
    k=floor((sqrt(ff)+1)/2)
    while i<k:
        if pr[i]==1:
            for j in range(2*i*(i+1),f,2*i+1):
                pr[j]=0
        i=i+1
    lta=[pr[n]*(2*n+1) for n in range(f) if pr[n]>0]
    lta[0]=2
    return lta

```

A0.2. Tests de primeritat i certificats de composició

A0.2.0. El test de Solovay-Strassen

nn és el nombre que es vol provar; ff és la fita per a la quantitat màxima de bases que es proven. Si nn és compost, la funció proporciona (gairebé segur) false; si nn és 2, 3, 5, o 7, proporciona true; en cas que nn sigui primer més gran que 7 o bé compost però no ho detecti en ff intents, proporciona 'Indeterminat'.

In [ ]:
```
def SolovayStrassenTest(nn,ff):
    if nn==1:
        return false
    if nn in [2,3,5,7]:
        return true
    if is_even(nn):
        return false
    if ff<1:
        return 'Cal fer alguna prova.'
    f=0
    n2=(nn-1)//2
    while f<ff:
        g=ZZ.random_element(2,nn-1)
        x=Mod(g,nn)^n2
        if x==1 or x==nn-1:
            y=Mod(kronecker(g,nn),nn)
            if y!=x:
                return false
        else:
            return false
        f=f+1
    return 'Indeterminat'

```

A0.2.1. El test de Miller-Rabin

nn és el nombre que es vol provar; ff és la fita per a la quantitat màxima de bases que es

proven. Si nn és compost, la funció proporciona (gairebé segur) false; si nn és 2, 3, 5, o 7, proporciona true; en cas que nn sigui primer més gran que 7 o bé compost però no ho detecti en ff intents, proporciona 'Indeterminat'.

In [ ]:
```
def MillerRabinTest(nn,ff):
    if nn==1:
        return false
    if nn in [2,3,5,7]:
        return true
    if is_even(nn):
        return false
    if ff<1:
        return 'Cal fer alguna prova.'
    v=0
    m=nn-1
    while is_even(m):
        v=v+1
        m=m//2
    f=0
    while f<ff:
        g=ZZ.random_element(2,nn-1)
        x=Mod(g,nn)^m
        if x!=1 and x!=nn-1:
            k=1
            x=x^2
            while (x!=nn-1 and k<v-1):
                x=x^2
                k=k+1
            if k>=v-1 and x!=nn-1:
                return false
        f=f+1
    return 'Indeterminat'

```

A0.2.2. El certificat de Solovay-Strassen

nn és el nombre que es vol provar; ff és la fita per a la quantitat màxima de bases que es proven. Si nn és compost, la funció proporciona (gairebé segur) un certificat de composició, [nn,false,g], on g és un valor que certifica que nn és compost; si nn és 2, 3, 5, o 7, proporciona un certificat de primeritat per a nn; en cas que nn sigui primer més gran que 7 o bé compost però la funció no ho detecti en ff intents, proporciona 'Indeterminat'.

In [ ]:

```
def SolovayStrassenCert(nn,ff):
    if nn==1:
        return [nn,false,1]
    if nn==2 or nn==3:
        return [nn,true,nn-1,[nn-1]]
    if nn==5:
        return [nn,true,2,[2]]
    if nn==7:
        return [nn,true,3,[2,3]]
    if is_even(nn):
        return [nn,false,2]
    if ff<1:
        return 'Cal fer alguna prova.'
    f=0
    n2=(nn-1)//2
    while f<ff:
        g=ZZ.random_element(2,nn-1)
        x=Mod(g,nn)^n2
        if x ==1 or x==nn-1:
            y=Mod(kronecker(g,nn),nn)
            if y!=x:
                return [nn,false,g]
        else:
            return [nn,false,g]
        f=f+1
    return 'Indeterminat'

```

A0.2.3. El certificat de Miller-Rabin

nn és el nombre que es vol provar; ff és la fita per a la quantitat màxima de bases que es proven. Si nn és compost, la funció proporciona (gairebé segur) un certificat de composició, [nn,false,g], on g és un valor que certifica que nn és compost; si nn és 2, 3, 5, o 7, proporciona un certificat de primeritat per a nn; en cas que nn sigui primer més gran que 7 o bé compost però la funció no ho detecti en ff intents, proporciona 'Indeterminat'.

In [ ]:

```
def MillerRabinCert(nn,ff):
    if nn==1:
        return [nn,false,1]
    if nn==2 or nn==3:
        return [nn,true,nn-1,[nn-1]]
    if nn==5:
        return [nn,true,2,[2]]
    if nn==7:
        return [nn,true,3,[2,3]]
    if is_even(nn):
        return [nn,false,2]
    if ff<1:
        return 'Cal fer alguna prova.'
    v=0
    m=nn-1
    while is_even(m):
        v=v+1
        m=m//2
    f=0
    while f<ff:
        g=ZZ.random_element(2,nn-1)
        x=Mod(g,nn)^m
        if x!=1 and x!=nn-1:
            k=1
            x=x^2
            while (x!=nn-1 and k<v-1):
                x=x^2
                k=k+1
            if k>=v-1 and x!=nn-1:
                return [nn,false,g]
        f=f+1
    return 'Indeterminat'

```

A0.3. Certificats de primeritat

A0.3.0. Un certificat bàsic de primeritat

pp és el nombre que es vol certificar com a primer; ff és la fita per a la quantitat màxima de bases que es proven i fppmu és una llista de nombres (que haurien de ser els primers que divideixen pp-1). Si pp és compost, la funció proporciona (gairebé segur) un certificat de composició, [pp,false,g]. En cas que pp sigui primer i la llista sigui la dels divisors primers de pp-1, la funció en proporciona (gairebé segur) un certificat de primeritat, [pp,true,g,fppmu], amb la llista fppmu ordenada en sentit creixent. I en cas que pp sigui primer, o bé compost, però la funció no ho detecti en ff intents, proporciona 'Indeterminat'.

In [ ]:

```
def Certifica(pp,fppmu,ff):
    if pp==1:
        return [pp,false,1]
    if pp==2 or pp==3:
        return [pp,true,pp-1,[pp-1]]
    if is_even(pp):
        return [pp,false,2]
    if ff<1:
        return ["Cal fer alguna prova."]
    if len(fppmu)==0:
        lta1=factor(pp-1)
        lta=[lta1[i][0] for i in range(len(lta1))]
    else:
        lta=sorted(fppmu)
    l=len(lta)
    f=0
    while f<ff:
        g=ZZ.random_element(2,pp-2)
        if (s:=Mod(g,pp)^((pp-1)//2))==pp-1:
            i=1
            while i<= l-1 and Mod(g,pp)^((pp-1)//lta[i])!=1:
                i=i+1
            if i==l:
                return [pp,true,g,lta]
        else:
            if s!=1:
                return [pp,false,g]
        f=f+1
    return [pp,'Indeterminat']

```

A0.3.1. El certificat de primeritat de Pocklington

pp és el nombre que es vol certificar com a primer; tt és una llista de factors primers de pp-1, i ff és una fita per a la quantitat màxima de valors de g que provarem a l'atzar. Si la llista tt és correcta, la funció proporciona (gairebé segur) un certificat de primeritat per a pp, o bé 'Indeterminat' si en ff proves no l'aconsegueix, o un certificat de composició si pot provar que pp és compost.

Observació. Si la llista tt no és correcta, el resultat del certificat pot ser erroni.

In [ ]:

```
def Pocklington(pp,tt,ff):
    if not pp in ZZ or pp<1:
        return ['Cal que el nombre P sigui enter positiiu.']
    if pp==1:
        return [pp,false,1]
    if pp==2 or pp==3:
        return [pp,true,pp-1,[pp-1]]
    if is_even(pp):
        return [pp,false,2]
    if ff<1:
        return 'Cal fer alguna prova.'
# Comprovació que la llista tt és de divisors de pp-1, i càlcul d
# però no que són primers.
    if false in [(r in ZZ and r>1) for r in tt]:
        return 'La llista T no és de nombres enters >1.'
# Si 2 no pertany a la llista tt, li afegim (per a millora del ca
    t=tt
    if not (2 in t):
        t=[2]+t
    x=prod(t)
    q,r=divmod(pp-1,x)
    if r:
        return 'La llista T no és correcta.'
    d=gcd(q,x)
    while d>1:
        q=q//d
        d=gcd(q,x)
    uu=q
    q=uu^2
    if q==pp:
        return [pp,false,uu]
    if q>pp:
        return 'U és massa gran.'
    t=sorted(t)
# Si hem arribat aquí, és que P, T, F i U són correctes (excepte
# potser, que alguns elements de T no siguin primers).
    l=len(t)
    f=0
    while f<ff:
        g=ZZ.random_element(2,pp-2)
        if (s:=Mod(g,pp)^((pp-1)//2))==pp-1:
            i=1
            while i<= l-1 and gcd((s:=Mod(g,pp)^((pp-1)//t[i]))-1
                i=i+1
            if i==l:
                return [pp,true,g,t]
        else:
            if s!=1:
                return [pp,false,g]
        f=f+1
    return [pp,'Indeterminat']

```

A0.3.2. Una funció per a comprovar certificats

La funció s'aplica a un certificat cert de composició o de primeritat. Si cert és correcte, proporciona true; si cert no és correcte, proporciona false; i en cas que cert sigui

'indeterminat' o bé 'Indeterminat', la funció proporciona 'Indeterminat'.

In [ ]:
```sage
def ComprovaCert(cert):
    pp=cert[0]
    if not pp in ZZ or pp<1:
        return 'S\'ha d\'aplicar a un nombre enter positiu.'
    if pp==1:
        return cert==[pp,false,1]
    if pp==2 or pp==3:
        return cert==[pp,true,pp-1,[pp-1]]
    if is_even(pp):
        return cert==[pp,false,2]
    if cert[1]==false:
        return 1<cert[2]<pp and mod(cert[2],pp)^(pp-1)!=1
    if cert[1]=='Indeterminat' or cert[1]=='indeterminat':
        return'Indeterminat'
    if cert[1]!=true:
        return 'La llista no és cap certificat.'
# Sembla que la llista pot ser un certificat de primeritat de P.
# Comprovem que la llista ho és de nombres naturals entre 1 i n-
# i divisors de n-1.
    tt=cert[3]
    if false in [v in ZZ and 1<v and v<pp and ((pp-1)%v)==0 for v
        return 'La llista T és incorrecta.'
# Suposem que els elements de la llista són primers.
# Comprovem que g és un natural entre 1 i n-1.
    g=cert[2]
    if not g in ZZ or g<2 or g>pp-1:
        return 'La llista no és cap certificat: g incorrecte.'
# Calculem el cofactor U.
    x=prod(tt)
    q,r=divmod(pp-1,x)
    if r:
        return 'La llista T no és correcta.'
    d=gcd(q,x)
    while d>1:
        q=q//d
        d=gcd(q,x)
    uu=q
    q=uu^2
    if q==pp:
        return 'P és un quadrat.'
    if q>pp:
        return 'U és massa gran.'
# Comprovem les propietats de g.
    if Mod(g,pp)^(pp-1)!=1:
        return False, 'L\'ordre de g no divideix p-1.'
    if false in [gcd(Mod(g,pp)^((pp-1)//tt[v])-1,pp)==1 for v in
        return False, 'Algun g és incorrecte.'
    return true

```

A0.4. Construcció de primers

A0.4.0. Construcció d'un primer de 112 bits

La funció Primer112( ) genera un primer aleatori de 112 bits, que certifica, tot i que no proporciona el certificat.

In [ ]:
```
def Primer112():
    fita=500
    n=0
    while not SolovayStrassenCert((q:=(2*ZZ.random_element(2^110
        n=n+1
    if n<fita and Certifica(q,[],50)[1]:
        return q
    return 0

```

A0.4.1. Construcció d'un primer de 480 bits

La funció Primer480(ltanp) genera un primer p aleatori de 480 bits, que certifica, de la forma $p = 2kn_0n_1^2n_2 + 1$, on $ltanp = [n_0, n_1, n_2, \ldots]$, és una llista de nombres primers dels quals se suposa que el producte $n_0n_1^2n_2$ és prou gran (i prou petit) a fi que k pugui ésser calculat aleatòriament i permeti la certificació de p amb el certificat de Pocklington.

En el cas (poc probable) que no s'aconsegueixi un tal nombre primer p, la funció retorna (0,'Cal tornar-hi!'). Si l'aconsegueix, retorna una llista [p,k,cert], amb els valors de p i de k i un certificat de Pocklington de la primeritat de p.

In [ ]:
```
def Primer480(ltanp):
    np=ltanp[0]*ltanp[1]^2*ltanp[2]
    infkp=ceil((2^479-1)/(2*np))
    supkp=floor((2^480-1)/(2*np))
    kp=ZZ.random_element(infkp,supkp)
    p=2*kp*np+1
    fita=1000
    while fita >0 and SolovayStrassenTest(p,50)==false:
        kp=ZZ.random_element(infkp,supkp)
        p=2*kp*np+1
        fita=fita-1
    if fita==0:
        return 0, 'Cal tornar-hi!'
    else:
        cert=Pocklington(p,ltanp[0:2],50)
    return [p,kp,cert]
```

A0.4.2. Construcció d'un primer de 1024 bits

La funció Primer1024(ltanp) genera un primer p aleatori de 1024 bits, que certifica, de la forma $p = 2kn_0n_1 + 1$, on $ltanp = [n_0, n_1]$, és una llista formada per una parella de nombres primers de 480 bits.

En el cas (poc probable) que no s'aconsegueixi un tal nombre primer p, la funció retorna (0,'Cal tornar-hi!'). Si l'aconsegueix, retorna una llista [p,k,cert], amb els valors de p i de k i un certificat de Pocklington de la primeritat de p.

In [ ]:
```
def Primer1024(ltanp):
    np=ltanp[0]*ltanp[1]
    infkp=ceil((2^1023-1)/(2*np))
    supkp=floor((2^1024-1)/(2*np))
    kp=ZZ.random_element(infkp,supkp)
    p=2*kp*np+1
    fita=2000
    while fita >0 and SolovayStrassenTest(p,50)==false:
        kp=ZZ.random_element(infkp,supkp)
        p=2*kp*np+1
        fita=fita-1
    if fita==0:
        return 0, 'Cal tornar-hi!'
    else:
        cert=Pocklington(p,ltanp,50)
    return [p,kp,cert]
```

A0.5. Refinament i repartició

A0.5.0. Funció per a repartir factors primers o compostos

La funció Reparteix(lta) reparteix els nombres de la llista lta en dues llistes, que proporciona com a sortida, prm, per als factors primers (amb un tercer paràmetre, si cal), i altres per als factors compostos (sense tercer paràmetre).

No es fa cap intent de veure si els nombres de l'entrada són o no naturals no nuls, ni coprimers dos a dos. Se suposa que la funció és cridada per una altra que porta el control dels seus paràmetres.

In [ ]:
```
def Reparteix(lta):
    aux=lta
    prm=[]
    altres=[]
    FitaSolovayStrassen=1024
    while len(aux)>0:
        n=aux[0][0]
        e=aux[0][1]
        aux=aux[1:len(aux)]
        fSS=min(FitaSolovayStrassen,max(20,1+log(n,2)))
        res=SolovayStrassenTest(n,fSS)
        if res=='Indeterminat':
            prm=prm+[[n,e,'?']]
        if res==true:
            prm=prm+[[n,e]]
        if res==false:
            altres=altres+[[n,e]]
    return prm,altres

```

A0.5.1. Funció per a refinar llistes de factors

La funció Refina(lta) s'aplica a una llista lta de parelles, [n,e], de nombres naturals n>1 amb

multiplicitats e>0, i retorna una llista de parelles, [m,f], de factors coprimers dos a dos amb multiplicitats, de manera que el producte dels n^e coincideix amb el producte dels m^f, i que els m són divisors dels n.

De fet, la funció canvia (recursivament) dos factors n i n' que no són coprimers pels factors d:=gcd(n,n'), n/d, i n'/d, amb les multiplicitats respectives. En particular, pot retornar factoritzacions més precises (o sigui, amb factors més petits).

Aquesta funció és útil en el cas que un mètode de factorització retorni dos (o més) factors sense comprovar-ne la primeritat o si són o no coprimers dos a dos.

Per exemple, per a la llista lta=[[6,2],[10,3],[15,1]], el retorn és la llista lta=[[2,5],[3,3],[5,4]].

In [ ]:
```
def Refina(lta):
    if len(lta)<2:
        return lta
    aux=lta
    ref=[]
    while len(aux)>1:
        test=aux[0]
        aux=aux[1:len(aux)]
        aux2=[]
        while len(aux)>0:
            test2=aux[0]
            aux=aux[1:len(aux)]
            d=gcd(test[0],test2[0])
            if d>1:
                a=test[0]//d
                v=test[1]
                b=test2[0]//d
                w=test2[1]
                if b>1:
                    aux=[[b,w]]+aux
                if a>1:
                    aux=[[a,v]]+aux
                test=[d,v+w]
            else:
                aux2=aux2+[test2]
        ref=ref+[test]
        aux=aux2
    ref=ref+aux
    return sorted(ref)

```

A0.6. Mètode de factorització de Fermat

A0.6.0. La funció Fermat(nn,ff)

La funció Fermat(nn,ff) aplica el mètode de factorització de Fermat a un nombre enter nn i una fita ff per al nombre de passos.

Si nn=2m és parell amb m>0, la funció retorna la parella [2,m]; si nn=-2m és parell amb m>0, la funció retorna la parella [-2,m]; si la fita és petita, proporciona nn i un missatge; i si aconsegueix trencar nn retorna els dos factors més propers a l'arrel quadrada (per sota i

per sobre), amb el signe en el primer factor.

Observació. Notem que la funció no retorna cap factorització tal com l'hem definida en el text, essencialment, perqùe no està pensada per a ésser incorporada a l'algoritme general de factorització. En qualsevol cas, si s'hi incorporés, no caldria tenir en compte els nombres negatius, perquè ja els té en compte la funció de Factoritza(nn), i els exponents dels valors de la sortida serien els mateixos que els del valor d'entrada, i només caldria refinar i repartir.

In [ ]:
```
def FermatFact(nn,ff):
    if not nn in ZZ:
        return 'Cal que el nombre sigui enter.'
    if nn==0:
        return [0]
    if nn==1:
        return [1]
    if nn==-1:
        return [-1]
    if nn<0:
        n=-nn
        signe=-1
    else:
        n=nn
        signe=1
    if n==2 or n==3 or n==5 or n==7:
        return [nn]
    if is_even(n):
        return [signe*2, n//2]
    if ff<1:
        return 'Cal fer alguna prova.'
    f=2*(floor(ff)+1)
    an=floor(sqrt(n))
    if n==an^2:
        return [signe*an,an]
    v=1
    u=2*(an+1)+1
    r=(an+1)^2-n
    while r>0 and v<f:
        r=r-v
        v=v+2
    while r<0:
        r=r+u
        u=u+2
        while r>0 and v<f:
            r=r-v
            v=v+2
    if v<f:
        return [signe*(u-v)//2,(u+v-2)//2]
    return [nn,'fita petita']

```

A0.7. Mètode de factorització rho de Pollard

A0.7.0. La funció PollardRho(nn,tt,ff)

S'aplica a un nombre natural nn i dues fites. D'una banda, tt és el nombre màxim de comparacions que farem per a cadascuna de les, com a màxim, ff funcions (de la forma $x \mapsto x^2 + a \pmod{nn}$) que utilitzarem.

Si troba un factor no trivial d de nn, retorna la parella [d,nn/d].

In [ ]:
```
def PollardRho(nn,tt,ff):
    f=ff
    while f>0:
        a=ZZ.random_element(nn)
        x=ZZ.random_element(nn)
        y=x
        t=tt
        while t>0:
            x=(x^2+a)%nn
            y=((y^2+a)^2+a)%nn
            d=gcd(x-y,nn)
            if d>1 and d<nn:
                # Cal mantenir enters els paràmetres!!!
                return [d,nn//d]
            if d==1:
                t=t-1
            if d==nn:
                t=0
        f=f-1
    return nn

```

A0.8. Mètode de factorització p-1 de Pollard

A0.8.0. La funció PollardPmU(nn,ff)

Anomenarem nn el nombre que volem factoritzar, i ff la fita màxima per a l'exponent $r!$, $1 \leq r \leq f$, per al qual calcularem amb el mètode $p-1$ de Pollard.

Observació. La funció no fa cap control dels paràmetres d'entrada; per exemple, no comprova que nn sigui un nombre natural compost i no divisible per primers menors que ff, cosa que la funció suposa.

Si aconsegueix factoritzar, la funció retorna una parella [d,nn/d], on d és un factor no trivial de nn.

In [ ]:
```
def PollardPmU(nn,ff):
    a=2
    x=Mod(a,nn)
    r=2
    while r<ff:
        x=x^r
        d=gcd(x-1,nn)
        if d==nn:
            return nn
        if d>1:
# Cal que d i nn//d siguin enters, però d no ho és.
            d=Integer(d)
            return [d,nn//d]
        r=r+1
    return nn

```

A0.9. Algoritme general de factorització

A0.9.0. Descripció de l'algoritme

L'algoritme segueix els passos següents, mentre siguin necessaris.

1. Primerament, es fa una comprovació del nombre nn i es descarten els casos més trivials.

2. Es té en compte el signe.

3. Es calcula un garbell d'Eratòstenes, amb una fita màxima de 10^5, però que es pot canviar (és el valor FitaEratostenes de la funció).

4. S'aplica el mètode de factorització per divisió pels primers de la llista calculada.

5. En cas que quedi algun factor, es mira si és compost amb un test de Solovay-Strassen.

6. S'aplica el mètode de factorització rho de Pollard, amb una fita màxima de 10^6 i un màxim de 8 funcions. Aquests valors es poden canviar (són els valors FitaRho i IteracionsRho de la funció).

7. S'aplica el mètode de factorització p-1 de Pollard, amb una fita màxima de FitaPmU=FitaEratostenes (que també es pot canviar).

La funció retorna la factorització en la forma habitual d'una llista de llistes, de la forma [nombre,exponent], si nombre és -1 o bé primer, [nombre,exponent,?], si nombre és primer però no s'ha certificat (podria ser compost), o bé [nombre,exponent, **], si el factor corresponent és compost i no s'ha aconseguit factoritzar.

Observació. Notem que encara es podria tornar a provar els mètodes anteriors per als nombres compostos que resten per a la factoritació completa; però probablement no obtindríem res de nou.

A0.9.1. La funció Factoritza(nn)

In [ ]:

```
def Factoritza(nn):
# Control del paràmetre d'entrada i factoritzacions trivials.
    if not nn in ZZ:
        return 'El paràmetre ha de ser un nombre enter.'
    if nn==0:
        return [0]
    if nn==1:
        return [1]
    if nn==-1:
        return [[-1,1]]
# Creació de les llistes pendents, primers i compostos.
    if nn<0:
        primers=[[-1,1]]
        pendents=[[-nn,1]]
    else:
        primers=[]
        pendents=[[nn,1]]
    compostos=[]
# Repartició dels pendents. Si no en queda cap, retorn.
    [pr,cp]=Reparteix(pendents)
    cp=[cp[i]+[' ** '] for i in range(len(cp))]
    primers=primers+pr
    pr=[]
    pendents=cp
    cp=[]
    if len(pendents)==0:
        return primers
# Calculem la llista de primers petits per a la divisió.
    FitaEratostenes=10^5
    n=pendents[0][0]
    e=pendents[0][1]  # Notem que aquí ha de ser e=1.
    pendents=[]
    ff=min(FitaEratostenes,max(10,ceil(sqrt(n))))
    pr=LlistaDePrimers(ff)
    l=len(pr)
 # Comencem la divisió.
    i=0
    p=pr[0]
    while n>=p^2 and i<l-1:
        a,b=divmod(n,p)
        if b==0:
            v=0
            while b==0:
                n=a
                v=v+1
                a,b=divmod(n,p)
            primers=primers+[[p,v]]
        i=i+1
        p=pr[i]
    if n>=p^2 and i==l-1:
        a,b=divmod(n,p)
        if b==0:
            v=0
            while b==0:
                n=a
                v=v+1
                a,b=divmod(n,p)
            primers=primers+[[p,v]]
    if n<p^2 and n>1:
```

```
60          primers=primers+[[n,1]]
61          n=1
62          fact=primers
63          return fact
64      if n==1:
65          fact=primers
66          return fact
67  # Si som aquí, és que queda un factor. Mirem si és primer.
68      FitaSolovayStrassen=1024
69      fSS=min(FitaSolovayStrassen,max(20,1+log(n,2)))
70      if SolovayStrassenTest(n,fSS)=='Indeterminat':
71          primers=primers+[[n,1,'?']]
72      else:
73          pendents=pendents+[[n,1,'**']]
74      compostos=[]
75  # Si som aquí, queda un factor compost, en pendents.
76  # Comencem amb el mètode Rho de Pollard.
77      FitaRho=10^6    # Es pot augmentar, probablement fins a 10^8
78      IteracionsRho=8 # Es pot augmentar, però no millora gaire.
79      while len(pendents)>0:
80          n=pendents[0][0]
81          e=pendents[0][1]
82          pendents=pendents[1:len(pendents)]
83          rho=PollardRho(n,min(FitaRho,floor(10*sqrt(n))),Iteracio
84          if rho==n:
85              compostos=compostos+[[n,e,'**']]
86          else:
87              [prm,cp]=Reparteix(Refina([[rho[0],e],[rho[1],e]]))
88              primers=sorted(primers+prm)
89              prm=[]
90              cp=[cp[i]+[' ** '] for i in range(len(cp))]
91              pendents=pendents+cp
92              cp=[]
93      pendents=compostos
94      compostos=[]
95  # Hem acabat el mètode rho.
96  # Comencem el mètode p-1.
97      FitaPmU=FitaEratostenes  # Es pot augmentar, probablement fi
98      while len(pendents)>0:
99          n=pendents[0][0]
100         e=pendents[0][1]
101         pendents=pendents[1:len(pendents)]
102         PmU=PollardPmU(n,FitaPmU)
103         if PmU==n:
104             compostos=compostos+[[n,e,'**']]
105         else:
106             [prm,cp]=Reparteix(Refina([[PmU[0],e],[PmU[1],e]]))
107             primers=sorted(primers+prm)
108             prm=[]
109             cp=[cp[i]+[' ** '] for i in range(len(cp))]
110             pendents=pendents+cp
111             cp=[]
112     pendents=compostos
113     compostos=[]
114 # Hem acabat el mètode p-1.
115     fact=primers+pendents
116     return fact
117
```

Fi de l'apèndix A0