Teoria de Números Computacional visto pelos olhos do Pari/GP

Pedro Patrício, 2008, 2009

LCC & LMat

Resumo

Nestas notas pretendemos exemplificar algumas formas de aplicar a Teoria de Números Computacional num CAS, e em particular no Pari/GP. Estas notas *não substituem* as que por ventura retirará das aulas teóricas. Deve, sempre que possível, experimentar por si as informações descritas neste documento.

Conteúdo

1 Verde de honra

O *software* que usaremos ao longo destas notas é o Pari/GP, distribuído segundo a licença GPL. Pode ser obtido no sítio http://pari.math.u-bordeaux.fr/. Existem binários para MSWindows e está disponível nos repositórios das distribuições mais importantes de Linux. O código fonte está disponível no sítio do Pari/GP. O tradicional `make & make install` deve ser o suficiente para instalar o Pari/GP. Esta é ainda uma solução para o fazer num MacOSX.

Precisaremos, ainda, de editor de texto para construirmos funções. Num MSWindows o Wordpad poderá ser o suficiente. Aconselhamos, no entanto, que instale o *vim*, obtido de

http://www.vim.org/download.php.

Pode não parecer, à primeira vista, muito prático de se usar, mas pode acreditar que tem mais valias que o Wordpad não tem. Aquela que mais nos interessará será o reconhecimento da sintaxe do GP. O mesmo se aplica para os utilizadores de Linux/Unix/FreeBSD/MacOSX, e outros *nix. Por norma, os utilizadores destes sistemas já estão convencidos, pelo que terminamos este parágrafo por aqui[1]. O editor *emacs* tem extensões[2] que lhe permitem reconhecer a sintaxe do GP.

Quanto a documentação, o Guia do Utilizador on-line pode ser conveniente, como

http://pari.math.u-bordeaux.fr/dochtml/html.stable/.

Existe ainda [2] um muito prático *Pari-GP reference card.*

2 Aquecendo os motores

Antes de mais, inicie uma sessão do Pari/GP.

A tecla `tab` mostra os completamentos possíveis para o texto introduzido:

```
? pri
prime     primes    print1    printp1
primepi   print     printp    printtex
```

Escreva `pri` e tecle em `tab`. Para conhecer um comando, faça

```
? ?primepi
primepi(x): the prime counting function pi(x) = #{p <= x, p prime}.
```

Por exemplo, para saber quantos são os primos inferiores a 1000,

```
 ? primepi(1000)
%1 = 168
```

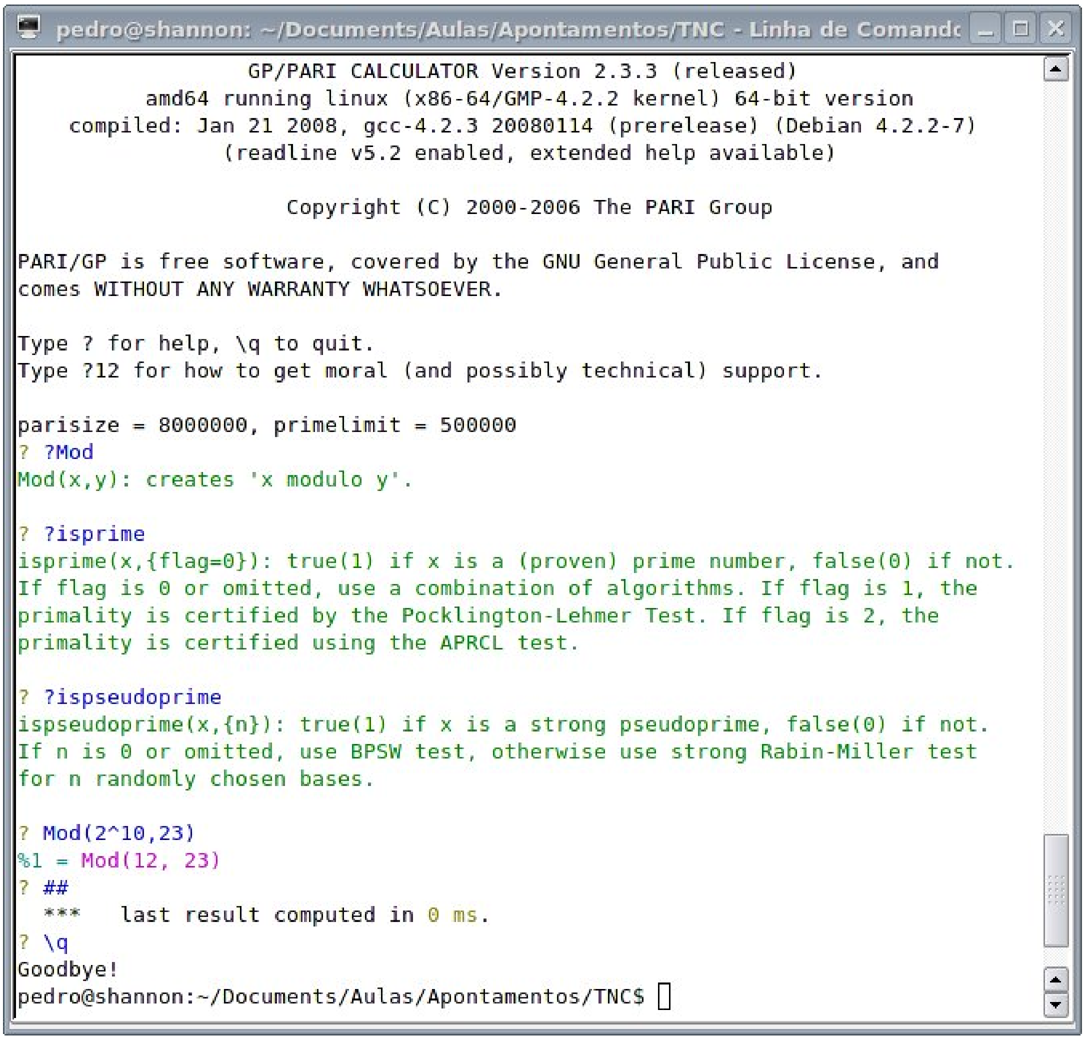
A atribuição de um valor a uma variável é feita, por exemplo, `a = 1` , colocando-se ";" no fim dependendo se se pretende que seja mostrado o valor ou não. Os símbolos

```
+, -, *, /, <, >, <=, >=, ==, !=
```

são tratados da forma usual. Por exemplo,

[1]... não antes sem relembrar que deve pressionar "i" para começar a escrever, e "Esc" para sair do modo de edição...

[2]Ver http://math.univ-lille1.fr/ ramare/ServeurPerso/GP-PARI/.



```
pedro@shannon: ~/Documents/Aulas/Apontamentos/TNC - Linha de Comandc

                  GP/PARI CALCULATOR Version 2.3.3 (released)
          amd64 running linux (x86-64/GMP-4.2.2 kernel) 64-bit version
    compiled: Jan 21 2008, gcc-4.2.3 20080114 (prerelease) (Debian 4.2.2-7)
                 (readline v5.2 enabled, extended help available)

                     Copyright (C) 2000-2006 The PARI Group

PARI/GP is free software, covered by the GNU General Public License, and
comes WITHOUT ANY WARRANTY WHATSOEVER.

Type ? for help, \q to quit.
Type ?12 for how to get moral (and possibly technical) support.

parisize = 8000000, primelimit = 500000
? ?Mod
Mod(x,y): creates 'x modulo y'.

? ?isprime
isprime(x,{flag=0}): true(1) if x is a (proven) prime number, false(0) if not.
If flag is 0 or omitted, use a combination of algorithms. If flag is 1, the
primality is certified by the Pocklington-Lehmer Test. If flag is 2, the
primality is certified using the APRCL test.

? ?ispseudoprime
ispseudoprime(x,{n}): true(1) if x is a strong pseudoprime, false(0) if not.
If n is 0 or omitted, use BPSW test, otherwise use strong Rabin-Miller test
for n randomly chosen bases.

? Mod(2^10,23)
%1 = Mod(12, 23)
? ##
  ***   last result computed in 0 ms.
? \q
Goodbye!
pedro@shannon:~/Documents/Aulas/Apontamentos/TNC$
```

Figura 1: Uma sessão do Pari/GP

```
? a=3; b=4; a==b-1
% = 1
? a=3; b=4; a!=b
% = 1
? a=3; b=4; a!<=b
% = 0
```

Os resultados apresentados correspondem a "verdadeiro" ou "falso".

Os símbolos `&&, ||` indicam, respectivamente, os operadores lógicos $\wedge$ e $\vee$.

```
? isprime(11%8) && !isprime(8\2)
% = 1
```

Aqui, `%, \` indicam, respectivamente, o resto da divisão inteira e o quociente da divisão inteira. Como atrás, `!` indica a negação e o resultado final refere a expressão inscrita como verdadeira.

Já indicámos atrás a forma de definirmos variáveis. Vejamos um exemplo simples de como se pode incrementar valores a uma variável:

```
? a++
% = 1001
? a+=2
% = 1003
? a=a+5
% = 1008
```

Pode ligar o "timer" para saber quanto tempo demora a executar cada instrução, fazendo `#` para ligar e desligar. Para apenas o fazer uma única vez, basta introduzir `##`.

Para definirmos uma função e calcular a imagem de um certo objecto, siga o exemplo seguinte:

```
? f=x^2+1
% = x^2 + 1
? type(f)
% = "t_POL"
? subst(f,'x,1)
% = 2
```

3 Funções definidas pelo utilizador

É possível definir novas funções no Pari/GP. A sintaxe é

```
nome(lista de variaveis formais) =
local(lista de variaveis locais); sequencia de comandos
```

que tem um aspecto mais simpático se for escrita como

```
nome(x0 , x1 , . . . ) =
{
  local(t0 , t1 , . . . );
  local(. . . );
  ...
}
```

Um exemplo simples:

```
? primo(p)=if(isprime(p),print(p," e' primo"),print(p," nao e' primo"));
? primo(123)
123 nao e' primo
? primo(11)
11 e' primo
```

Uma forma mais elaborada seria a de, num editor de texto, escrever a função

```
\\ linha comentada pois comeca por dupla backslash
\\
\\ uma funcao muito simples
\\ o texto entre /* e */  tambem esta comentado

primo()= /* esta funcao nao tem argumentos de entrada */
{
local (numero,opcao);  /* definicao das variaveis locais */

     print("Escreve um numero");
     numero=qualquercoisa;
     while(type(numero) != "t_INT", /* para nao se brincar em servico */
           numero=input(); /* input de numero pelo utilizador */
           );  /* fim do while */
     if(isprime(numero),
                       print(numero," e' primo")
                       , /* else */
                       print(numero, " nao e' primo. Factorizo-o? (s/n))";
                       opcao=input();
                       if(opcao==s,
                                  print(factor(numero))
                       )
      );
      print("Que os deuses te acompanhem")
}
```

Gravou-se num ficheiro com a extensão `.gp`. No Pari/GP, faremos `\r nomedoficheiro.gp`. No Windows, pode arrastar o ficheiro para a sessão do Pari/GP.

4 Contando os primos

A função $\pi(x)$ está definida no Pari/GP pelo comando `primepi`. Por exemplo,

```
? primepi(1000)
% = 168
```

Recorde que $\pi(x) \sim \frac{x}{\ln x}$ e que $\pi(x) \sim \int_2^x \frac{1}{\ln t} dt = li(x)$. À custa de cálculos desenvolvidos por X. Gourdon, sabe-se que $\pi(10^{21}) = 21127269486018731928$.

```
? li(x)={intnum(X=2,x,1/log(X))}
? li(10^21)
% = 21127269486616126181.287894555829594679
? 10^21/log(10^21)
```

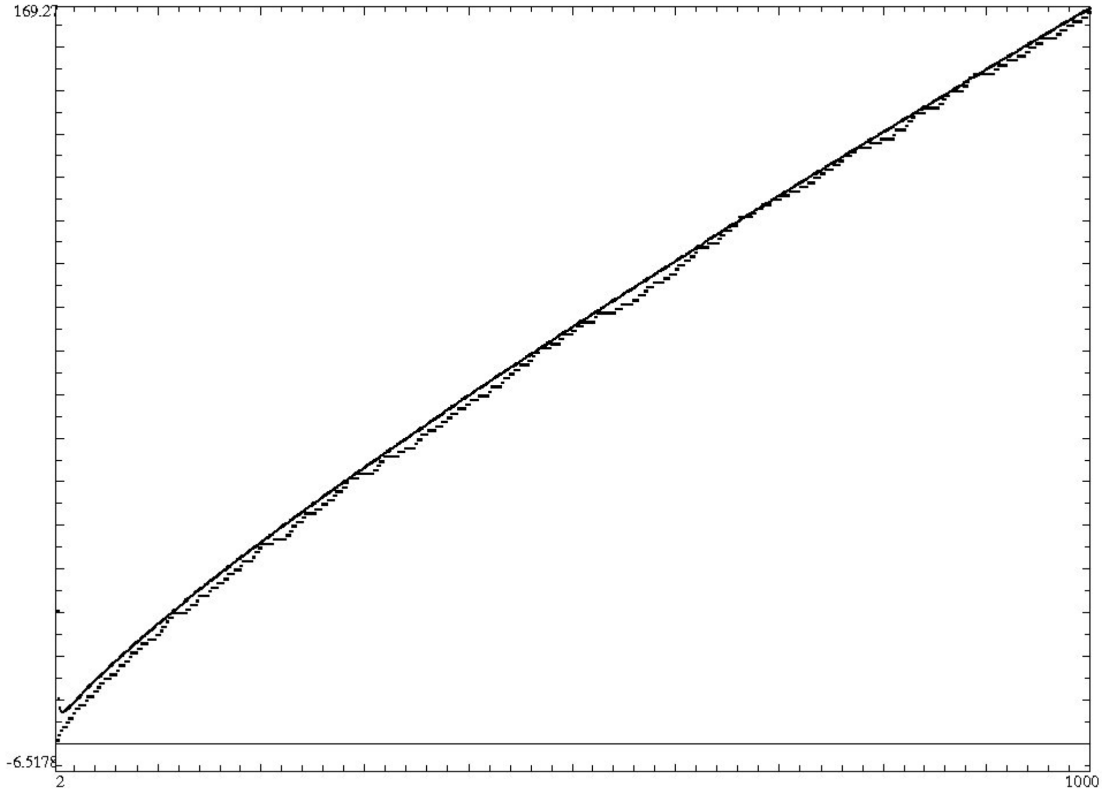


Figura 2: $\pi(x)$ e $\frac{x}{\ln(x)-1}$, com x entre 2 e 1000.

```
% = 20680689614440563221.482329472219289633
? ploth(X=2,1000,[primepi(X),li(X),X/log(X)],64)
% = [2.000000000000000000, 1000.0000000000000000, 0.E-307, 176.56449421003472367]
```

Recorde que $\pi(x) \sim \frac{x}{\ln x - a}$. Mostra-se que $a = 1$ é a melhor escolha para aproximação de $\pi(x)$. Vejamos graficamente:

```
? ploth(X=2,1000,[primepi(X),X/(log(X)-1)],64);
```

Para gravar o resultado num ficheiro *postscript* deverá usar o comando

```
? psploth(X=2,1000,[primepi(X),X/(log(X)-1)],64);
```

O resultado final está no ficheiro `pari.ps`.

5 Divisão trivial

A forma mais simples de verificar a primalidade de certo número pequeno é pela divisão trivial: testar a divisibilidade de n por todos os primos não superiores a $\lfloor\sqrt{n}\rfloor$.

```
\\ Pedro Patricio, 2009

\\ Teste de primalidade pela divisao trivial
\\ uso: divtrivial(n)
\\ resultado: 1 se e’ primo, 0 caso contrario

divtrivial(n)=
{       local(flag, k);

        flag=0;
        k=2;
        while(k<=floor(sqrt(n)) && flag==0,
                if(isprime(k) && n%k==0,
                flag=1,
                \\else
                k++
                )
        );
        return(!flag)
}
```

6 O crivo de Erastótenes

Recorde o crivo de Erastótenes: para encontrar os números primos inferiores a n, listam-se os primos inferiores a $\sqrt{n}$, e retiram-se os seus múltiplos (que não os próprios primos) da lista. Ou seja, se p é um primo inferior a $\sqrt{n}$, então são retirados da lista os números da forma kp, com $1 < k \leq \frac{\lfloor\sqrt{n}\rfloor}{p}$.

Na implementação que descrevemos de seguida, criou-se uma lista `Lista` onde se inseriram os inteiros `1..n` e onde as entradas correspondentes aos múltiplos dos primos são convertidas em `0`. Estas entradas com valores nulos servirão de bandeira para indicar quais os números que são retirados da lista.

```
\\ Crivo de Eratostenes
\\ 2009, Pedro Patricio

eratostenes(n)=
{
        local(p, k, Lista, i);

        Lista=listcreate(n);
        for(i=1,n,
                listput(Lista,i,i)
        );
        forprime(p=2,floor(sqrt(n)),
                k=2;
                while(k*p<=n,
                        listput(Lista,0,k*p);
                        k++)
                );
```

```
        Lista[1]=0;
        return(Lista)
}
```

7 O algoritmo estendido de Euclides

O algoritmo estendido de Euclides é uma variação do algoritmo de Euclides para cálculo do máximo divisor comum entre dois naturais $a \geq b$. Ao contrário da versão clássica que usa o princípio da casca da cebola para cálculo (os primeiros valores a serem encontrados são os últimos a ser usados), apenas são necessários os quocientes e restos dos dois passos anteriores. Definindo as sucessões $(s_k), (t_k)$ como

$$s_0 = 1, s_1 = 0, s_{i+1} = s_{i-1} + s_i q_i$$

$$t_0 = 0, t_1 = 1, t_{i+1} = t_{i-1} + t_i q_i$$

onde q_i indica o i-ésimo quociente no algoritmo de Euclides, provou-se nas aulas que

$$s_i a + t_i b = r_i$$

onde r_i indica o i-ésimo resto no algoritmo de Euclides. Em particular, se r_ℓ for o último resto não nulo (ou seja, iguala o máximo divisor comum entre a e b), então

$$s_\ell a + t_\ell b = (a, b).$$

```
/* Algoritmo estendido de Euclides
Uso: exteuclides(a,b)
onde a,b sao naturais
Output: [s,t,d]
onde d=(a,b) e as+bt=d */

\\ Pedro Patricio, 2009


exteuclides(a,b)=

{
        local( \\variaveis locais
        aorig, borig, aaux, qult,
        qpenult, quoc, spenult, sult, tpenult, tult,
        flag
        );

        aorig=a; borig=b; \\ guardar o input original para calcular o mdc
        if(a%b==0 || b%a==0, \\ caso estranho
                if(a<b,
                        return([1,0,a]), \\ else
                        return([0,1,b])),
        /* else */
        flag=0; \\ flag de troca de valores a<b
        if(a<b,
```

```
                aaux=a;
                a=b;
                b=aaux;
                aorig=a; borig=b; flag=1
        );
        spenult=1; sult=0;
        tpenult=0; tult=1;
        qpenult=a\b;
        aaux=a; a=b; b=aaux%b;
        qult=a\b;
        aaux=a; a=b; b=aaux%b;
        if(b==0,
                if(flag==0,
                return([-qpenult,1,-qpenult*aorig+borig]),
                /*else*/ return([1,-qpenult,-qpenult*aorig+borig])),
        \\else
        quoc=a\b;
        resto=a%b;
        while(resto!=0,
                s=spenult-qpenult*sult;
                t=tpenult-qpenult*tult;
                qpenult=qult;
                qult= quoc;
                spenult=sult;
                tpenult=tult;
                sult=s;  tult=t;
                aaux=a; a=b; b=aaux%b;
                quoc=a\b;
                resto=a%b
        );
        s=spenult-qpenult*sult;
        t=tpenult-qpenult*tult;
        spenult=sult;
        tpenult=tult;
        sult=s; tult=t;
        qpenult=qult;
        s=spenult-qpenult*sult;
        t=tpenult-qpenult*tult;
        if(flag==0,
                return([s,t,aorig*s+borig*t]),
                /*else*/ return([t,s,aorig*s+borig*t])
        )
        ))
}
```

8 Factorização de Fermat

Recorde que a factorização de Fermat subentende a possibilidade de n dado, que se pretende factorizar, se pode escrever como a diferença de dois quadrados. Tal como foi mostrado nas aulas, tal nem sempre é possível. Suponhamos, no entanto, que existem $s, t \in \mathbb{N}$ para os quais $n = t^2 - s^2$. Ou seja, $t^2 - n = s^2$, isto é, $t^2 - n$ é um quadrado perfeito. Existindo solução (t, s) para $n = t^2 - s^2$, facilmente se obtem

$$n = (t - s)(t + s),$$

ou seja, encontrou-se uma factorização de n, eventualmente trivial. Sabendo que $n = ab$, com $a \geq b$, então necessariamente $a \geq \sqrt{n}$.

Para iniciarmos o algoritmo, tomamos $t = \lceil \sqrt{n} \rceil$, ou seja, t é o menor inteiro não inferior a $\sqrt{n}$. Pretendemos, depois, averiguar se $t^2 - n$ é um quadrado perfeito. Vejamos um exemplo: seja $n = 275$.

```
? n=275;
? t=ceil(sqrt(n))
 = 17
? issquare(t^2-n)
 = 0
```

$t^2 - n = 14$ não é um quadrado perfeito. Incrementamos[3] uma unidade de t e repetimos o raciocínio:

```
 ? t++
 = 18
? issquare(t^2-n)
 = 1
```

O critério de paragem $t^2 - n = s^2$ para algum natural s foi satisfeito. Ou seja, encontraram-se soluções de $t^2 - s^2 = n$, a saber $t = 18$ e $s = \sqrt{t^2 - n}$.

```
 ? s=sqrt(t^2-n)
 = 7.000000000000000000000000000000000000
? s=floor(s)
 = 7
```

Um factor de n será $a = t + s$ e outro será $b = t - s$.

```
? a=t+s
 = 25
? b=t-s
 = 11
? a*b
 = 275
```

Uma forma simples de implementarmos o algoritmo é à custa da construção de uma função. Use o seu editor favorito[4].

```
\\ Funcao que implementa a factorizacao de Fermat
\\ Pedro Patricio, 2008
```

[3] Recorde que tal pode ser feito com o comando `t++`.

[4] Não nos cansamos de sugerir o `VI`.

```
\\ sintaxe: fermat(n,ntent)
\\ onde n é o numero que se pretende factorizar e ntent e' o numero de tentativas


fermat(n,ntent=30)=
{
        local(t,s);

        t=ceil(sqrt(n));
        while(!issquare(t^2-n) && t-ceil(sqrt(n))<ntent,
                print("t = ",t,"; t^2-n = ", t^2-n);
                t++);
        if(issquare(t^2-n),
                s=sqrt(t^2-n);
                print("t^2-n = ",t^2-n," quadrado perfeito :-)");
                return([floor(t+s),floor(t-s)]),
                /*else*/
                print("Numero de tentativas excedidas :-("));
                break
}
```

9 Teorema chinês dos restos

Nesta secção vamos descrever uma aplicação simples do Teorema Chinês dos restos numa máquina com capacidades *exageradamente limitadas*, e que mesmo assim permite operar com números *grandes*. De facto, usaremos o facto de $\mathbb{Z}_{\prod_{i=1}^k n_i} \cong \mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \cdots \times \mathbb{Z}_{n_k}$, se $(n_i, n_j) = 1$ para $i \neq j$. O isomorfismo é dado pela aplicação definida por $\psi([x]_{n_1 n_2 \cdots n_k}) = ([x]_{n_1}, [x]_{n_2}, \ldots, [x]_{n_k})$.

Suponha que se pretende calcular $x + y$, com $x = 123684$ e $y = 413456$, numa máquina que admite números não superiores a 99. Repare que 99, 98, 97 e 95 são primos relativos dois a dois. Usaremos o isomorfismo $\mathbb{Z}_{89403930} \cong_\psi \mathbb{Z}_{99} \times \mathbb{Z}_{98} \times \mathbb{Z}_{97} \times \mathbb{Z}_{95}$.

Em primeiro lugar, calculamos a imagem de $[x]_{89403930}$ por ψ:

```
? x=123684
 = 123684
? m1=99; m2=98; m3=97; m4=95;
? x1=Mod(x,m1)
 = Mod(33, 99)
? x2=Mod(x,m2)
 = Mod(8, 98)
? x3=Mod(x,m3)
 = Mod(9, 97)
? x4=Mod(x,m4)
 = Mod(89, 95)
```

Isto é, $\psi(x) = ([33]_{99}, [8]_{98}, [9]_{97}, [89]_{95})$. Para y,

```
? y=413456
 = 413456
? y1=Mod(y,m1); y2=Mod(y,m2); y3=Mod(y,m3); y4=Mod(y,m4);
```

Finalmente, calculamos a soma em cada classe de equivalência e fazemos uso do Teorema Chinês dos restos para calcular o menor representante da classe de equivalência módulo 89403930:

```
? s1=x1+y1
 = Mod(65, 99)
? s2=x2+y2; s3=x3+y3; s4=x4+y4;
? chinese(s1,chinese(s2,chinese(s3,s4)))
 = Mod(537140, 89403930)
```

Este método pode ser usado em máquinas com limitações bem mais próximas das reais. Para tal, basta que recorde que $2^n - 1$ e $2^m - 1$ são primos relativos se e só se n e m o forem.

10 Factorização $\rho-$Pollard

Nesta secção, faremos uso da uma sequência pseudo-aleatória dada por $x_{k+1} \equiv f(x_k) \mod n$, com $f(x) = x^2 + 1$ e $x_0 = 2$. Suponha que se pretende encontrar uma factorização de $n = 8051$, recorrendo ao algoritmo $\rho-$Pollard. Recorde que, sucessivamente, calcula-se $d = (x_{2k} - x_k, n)$, com $k = 1, 2, \ldots$ até que $d \neq 1$ ou que se ultrapasse um certo número máximo de tentativas previamente definido. Heuristicamente, k que satisfaz o critério de paragem está proximo de p, factor não trivial desconhecido de n.

```
? n=8051
 = 8051
? f(x)=(x^2+1)%n
? x0=2;
? x1=f(x0)
 = 5
? x2=f(x1)
 = 26
? gcd(x2-x1,n)
 = 1
```

Uma forma de calcular os valores de x_k pode ser à custa da implementação de uma função recursiva. Para casos pequenos, pode ser útil.

```
? rhox(k)={ local(i,res); i=k; if(i<>0, return((rhox(i-1))^2+1),return(2))}
```

Seguimos os passos de algoritmo até que $(x_{2k} - x_k, n) \neq 1$:

```
? gcd(rhox(4)-rhox(2),n)
 = 1
? gcd(rhox(6)-rhox(3),n)
 = 97
```

97 é um factor não trivial de n.

Para terminar, deixamos aqui uma possível implementação do algoritmo no Pari/GP.

```
\\ Pedro Patricio, 2007
\\ rho_pollard
\\ para factorizar um numero n
\\ f(x)=x^1+1 mod n ; x_0=2
```

```
rho(n)=
{local(fact,a,b);

  if(n%2==0,
    print("O numero "n" e’ par!");
    fact=2;
    return(fact)
  , \\ else

  a=2; b=2;
  \\ gerando sequencia pseudoaleatoria
  a=(((a^2+1)%n)^2+1)%n; b=(b^2+1)%n;
  while((a-b)%n !=0 && gcd(a-b,n)==1,
    a=(((a^2+1)%n)^2+1)%n;
    b=(b^2+1)%n;
    print("mcd="gcd(a-b,n));
  );
  ); \\ fim do else
  if((a-b)%n ==0,
    print("Nao consigo factorizar :-(");
    break,
  /*else*/ fact=gcd(a-b,n);
    print("Um factor de "n" e’ "fact);
  return(fact))
}
```

11 Primos de Wilson

Um número primo p diz-se primo de Wilson se

$$(p-1)! \equiv -1 \mod p^2.$$

O cálculo do factorial é, neste caso, um problema, pelo que vamos implementar o factorial modular.

```
? factmod(n,p)={if(n==1,return(Mod(1,p^2)), Mod(n,p^2)*factmod(n-1,p))}
? factmod(562,563) \\ teste
% = Mod(316968, 316969)
? forprime(p=3,100000,if(factmod(p-1,p)==Mod(-1,p^2), print(p) ) )
5
13
563
 ***   deep recursion: if(n==1,return(Mod(1
                       ^--------------------
```

Encontrámos os 3 primos de Wilson conhecidos até à data. Um quarto primo de Wilson, se existir, será maior que 500000000. A nossa busca ficou muito aquém desse número.

12 Factorização $(p-1)-$Pollard

Como exemplo, iremos aplicar o algoritmo $(p-1)-$Pollard para encontrar um factor não trivial de $n = 540143$. Recorde que será necessário calcular $2^{k!} \mod n$, e que esse cálculo pode ser efectuado iterativamente à custa da sequência $r_k = r_{k-1}^k \mod n$, se $k \leq 2$, e partindo de $r_1 = 2$. Quando $d = (r_k - 1, n) \neq 1$, obtemos um factor d não trivial de n. A estratégia será de calcular cada termo da suquência r_n se tal for estritamente necessário. Ou seja, desde que o critério de paragem $(r_k - 1, n) \neq 1$ não seja satisfeito.

```
? n=540143;
? r=2; i=2; while(gcd(r-1,n)==1,  r=Mod(r^i,n); print("passo "i" : r="r"; mdc =
"gcd(r-1,n)); i++)

passo 2 : Mod(4, 540143) mdc = Mod(1, 540143)
passo 3 : Mod(64, 540143) mdc = Mod(1, 540143)
passo 4 : Mod(32783, 540143) mdc = Mod(1, 540143)
passo 5 : Mod(54805, 540143) mdc = Mod(1, 540143)
passo 6 : Mod(518077, 540143) mdc = Mod(1, 540143)
passo 7 : Mod(167138, 540143) mdc = Mod(421, 540143)
```

Portanto 421 é um factor não trivial de n.

```
\\ Pedro Patricio, 2007
\\ (p-1)-Pollard para factorizar numeros
\\ n inteiro a factorizar, t e' o # tentativas (default=10)

pmenosum(n,t=10)=

{       local(r,i);

        r=2;  i=0;
        while(gcd(r-1,n)==1 && i<t,
                i++;
                r=lift(Mod(r^i,n));
                print("passo "i" : r="r"; mdc = "gcd(r-1,n));
        );
        if(i==t, print("nao consigo factorizar de forma nao trivial :-("),
        /*else*/
                fact=gcd(r-1,n);
                print("um factor de "n" e' "fact);
                return(fact)
        );
}
```

13 Pseudo-primos fracos

Dado $b \in \mathbb{N}$, dizemos que $n \in \mathbb{N}$ composto é um *pseudo-primo (fraco) na base* b se $b^n \equiv b \mod n$. O menor pseudo-primo fraco na base 2 foi encontrado por Sarrus, em 1919. Mostrou que o composto 341 satisfaz $2^{340} \equiv 1 \mod 341$.

```
? for(n=2,341, if(Mod(2^n,n)==Mod(2,n)&& !isprime(n), print(n)) )
341
```

Podemos calcular, usando o Pari/gp, o número de pseudo-primos na base 2 menores do que, digamos, 10^5:

```
? contagem=0; for(n=2,10^5, if(Mod(2^n,n)==Mod(2,n)&& !isprime(n), contagem++) );
             print(contagem)
78
? primepi(10^5)
%2 = 9592
```

Podemos considerar simultanemante os pseudo-primos de várias bases.

```
? contagem=0; for(n=2,10^5,
                if(Mod(2,n)^n==Mod(2,n)&& Mod(3,n)^n==Mod(3,n)&&!isprime(n),
                        contagem++) );
        print(contagem)
25
? contagem=0; for(n=2,10^5,
                if(Mod(2,n)^n==Mod(2,n)&&Mod(3,n)^n==Mod(3,n)&&
                                        Mod(5,n)^n==Mod(5,n)&&!isprime(n),
                        contagem++) );
        print(contagem)
16
```

Só existem 16 pseudo-primos fracos de bases 2, 3 e 5 inferiores a 10^5. São eles 561 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841. 29341, 41041, 46657, 52633. 62745, 63973 e o 75361.

14 Teste de primalidade de Miller-Rabin

Recorde que n composto se diz um *pseudoprimo (fraco) na base* b se $b^n \equiv b \mod n$. Por exemplo, $n = 1387$ é um pseudoprimo na base 2:

```
? isprime(1387)
 = 0
? Mod(2^1387,1387)
 = Mod(2, 1387)
```

No entanto, não é um pseudoprimo forte na base 2. Escrevendo $1387 = n = 2^s t+1$, para algum natural s e t ímpar, verifiquemos se $2^t \equiv 1 \mod n$ ou se existirá algum j, com $0 \leq j \leq s-1$, para o qual $2^{2^j t} \equiv -1 \mod n$.

```
? n=1387
%3 = 1387
? (n-1)\2
%4 = 693
```

Portanto, $n-1 = 2^1 693$, e $s = 1, t = 693$. Vejamos se $2^t \equiv 1 \mod n$:

```
? t=693
 = 693
? Mod(2^t,n)
 = Mod(512, 1387)
```

Repare que $s = 1$, o que leva a $j = 0$ como única escolha, e que para esse caso $2^t \not\equiv -1 \mod n$. Portanto, 1387 não passa o teste de Miller e consequentemente é um número composto.

Vejamos outro exemplo. Considere o composto $n = 1373653$.

```
? n=1373653
 = 1373653
? isprime(n)
 = 0
? (n-1)\2
 = 686826
? (n-1)\2^2
 = 343413
```

Escrevamos $n-1=2^2\cdot 343413$. Ou seja, tomamos $s=2, t=343413$ na expressão $n=2^st+1$. Vamos aferir se n é um pseudoprimo de base 2, ou seja, se passa o teste de Miller de base 2 sendo ele composto. Em primeiro lugar, verificamos a validade da congruência $2^t \equiv q \mod n$.

```
? t=343413
 = 343413
? Mod(2^t,n)
 = Mod(890592, 1373653)
```

A primeira condição da disjunção não é satisfeita, pelo que será necessário verificar a segunda. Temos duas escolhas possíveis para j, a saber $j=0,1$. Para $j=0$ obtemos $2^t \not\equiv -1 \mod n$, como acabámos de ver no Pari/GP. Para $j=1$,

```
? Mod(2^(2*t),n)
 = Mod(1373652, 1373653)
```

ou seja, $2^{2^1t} \equiv -1 \mod n$. Portanto, n passa o teste do Miller, isto apesar de ser um composto.

Terminamos esta secção com uma possível implementação no Pari/GP.

```
\\ Pedro Patricio
\\ verifica se n passa o teste de Miller de base 2
\\

/* a escrita de n-1=(2^s)t */


decomp(n)=
{       local(s,t);

        n=n-1; \\ queremos n-1=(2^s)*t
        s=1; t=n\2;
        while( t%2!=1,
                s++;
                t=t\2
        );
        return([s,t])
}

/* o teste de Miller base 2 */

Miller2(n)=
{       local(primo,s,t,j);
```

```
        primo=false;
        s=decomp(n)[1];
        t=decomp(n)[2];
        if(Mod(2,n)^t==Mod(1,n),
                primo=true,
        \\else
                j=0;
                while(Mod(2,n)^(2^j*t)!=Mod(-1,n) & j<s,
                        j++);
                if(j<s, primo=true)
        \\fim-else
        );
        return(primo)
}
```

Para $n = 2^s t + 1$, com n e t ímpares, o teste de Miller define as chamadas sequências–B como $(b^t, b^{2t}, b^{2^2 t}, \ldots, b^{2^{s-1}t}, b^{2^s t})$, onde as entradas deste $s+1$–uplo são tomadas módulo n.

```
? n= 1373653;
? s=2;
? t=1373653;
? for(i=0,s, print(Mod(2,n)^(2^i*t)))
Mod(890592, 1373653)
Mod(1373652, 1373653)
Mod(1, 1373653)
```

O inteiro $n = 1373653$ passa o teste de Miller na base 2, apesar de ser um composto (é um pseudoprimo forte na base 2). Recorde que os ímpares que passam o teste de Miller têm a sua sequência–B necessariamente de duas formas:

$$(?, ?, ?, \ldots, ?, -1, 1, \ldots, 1)$$

$$(1, 1, 1, 1, \ldots, 1, 1)$$

Se tomarmos $n = 1905$, e portanto $s = 4, t = 119$, facilmente comprovamos que n é um pseudoprimo fraco na base 2, mas não passa o teste de Miller na base 2. A sequência–B é

$$(128, 1144, 1, 1, 1)$$

```
? for(i=0,s, print(Mod(2,n)^(2^i*t)))
Mod(128, 1905)
Mod(1144, 1905)
Mod(1, 1905)
Mod(1, 1905)
Mod(1, 1905)
```

Ou seja, existe uma raiz quadrada não trivial de 1 módulo n, e portanto n é composto. De facto $1144^2 \equiv 1 \mod 1905$.

15 Primos de Mersenne

Os números de Mersenne não naturais da forma $M_n = 2^n - 1$. Um *primo de Mersenne* é um número de Mersenne primo. Sabendo que se $d \mid n$ então $(2^d - 1) \mid (2^n - 1)$, na procura de primos de Mersenne M_n temos necessariamente n é primo.

O algoritmo de Lucas-Lehmer é determinista na caracterização dos primos de Mersenne. Para p primo e $M_p = 2^p - 1$, define-se a $(p-1)$-sequência $\{r_k\}$ como

$$r_1 = 4,\ r_k \equiv r_{k-1}^2 - 1 \mod M_p\,,\ \text{para } 1 < k \leq p-1.$$

M_p é primo de Mersenne se e só se

$$r_{p-1} \equiv 0 \mod M_p.$$

```
\\ Lucas-Lehmer para primos de Mersenne
\\ Pedro Patricio, 2009

mersenne(p)={

        local(i,M,r);

        M=2^p-1;
        i=1;
        r=Mod(4,M);
        while(i<p-1,
                r=r^2-2;
                i++
        );
        if(r==Mod(0,M),
                return(1),
                \\ else
                return(0)
        )
}
```

Podemos, agora, procurar os primos de Mersenne M_p com $p \leq 5000$.

```
 ? forprime(p=2,5000, if(mersenne(p), print(p)))
3
5
7
13
17
19
31
61
89
107
127
521
607
```

```
1279
2203
2281
3217
4253
4423
```

Repare que para $p = 4423$ o número M_p é primo de Mersenne.

```
? 2^(4423)-1
 = 285542542228279613901563566102164008326164238644702889199247456602284400390600653875954571505539843239754513915896150297878399377056071435169747221107988791198200988477531339214282772016059009904586686254989084815735422480409022344297588352526004383890632616124076317387416881148592486188361873904175783145696016919574390765598280188599035578448591077683677175520434074287726578006266759615970759521327828555662781678385691581844436448125115624281367424904593632128101802760960881114010033775703635457251209240736469215767971461993876192965603026802617901181329250123230464444386223088779246093737730124816816724244936744744885377701557830068808526481615130671448147902883666640622572746652757871273746492310963750011709018907862633246195787957314256938050730561196775803380843333819875009029688319359130952698213111413223933564901784887289822881562826008138312961436638459454311440437538215428712777456064478585641592133284435802064227146949130917627164470416896780700967735904298089096167504529272580008435003448316282970899027286499819943876472345742762637296948483047509171741861811306885187927486226122933413689280566343844666463265724761672756608391056505289757138993202111214957953114279462545533053870678210676017687509778661004600146021384084480212250536890547937420030957220967329547507217181155318713102310579026085806 07
```

Este facto é verificável de uma forma rápida, tendo em conta a ordem de grandeza do número:

```
? mersenne(4423)
= 1
? ##
  ***   last result computed in 156 ms.
```

Para finalizar, apresentamos uma variação sobre o mesmo tema.

```
\\ Pedro Patricio
\\ Teste de Lucas-Lehmer para testar primos de Mersenne

lucaslehmer(n)=
{       local(r, M);

        if(!isprime(n),
                print(n" nao e’ primo");
                n=nextprime(n);
                print("vou considerar p="n)
        );
        M=2^n-1;
        print("O numero de Mersenne e’ M="M);
        r=4;
        for(k=2,n-1,
```

```
                r=lift(Mod(r^2-2,M));
        );
        if(r==0,
                print("2^"n"-1 e' primo de Mersenne");
                return(true);
        /*else*/
                ,
                print("2^"n"-1 nao e' primo de Mersenne");
                return(false)
        );
}
```

16 RSA

Sejam p, q primos ímpares distintos e $n = pq$. Sejam $m = \phi(n) = (p-1)(q-1)$ e $e \in \mathbb{Z}_m^*$. Como $(e, m) = 1$ então existe $d = e^{-1}$ em $\mathbb{Z}_m^*$. Torna-se público o par ordenado (n, e) e secreto p. Recordamos a equivalÊncia entre o cálculo de $\phi(n)$ e a factorização de n.

```
\\ Pedro Patricio, 2007
/*
-- breve how-to --
Antes de mais, é necessário criar uma chave. Para tal, basta fazer
> gerachave(n)
onde o n indica o numero de bits da chave; por exemplo,
> chave=gerachave(1024)
gera uma chave com 1024 bits */

{
tamanho(n)=floor(log(n)/log(2))+1
}

{
procuraprimo(nbits)=
        primo=2;
        while(tamanho(primo)!= nbits,
                primo=nextprime(random(2^nbits))
        );
        return(primo)
}

{
gerachave(n)=
        /* parte I: encontrar os primos p e q */
        bitprimo=round(n/2);
        p=2;
        q=2;
        while(tamanho(p*q) != n,
```

```
                print("encontrando um p com  ", bitprimo, " bits...");
                p = procuraprimo(bitprimo);
                print("p tem ", tamanho(p), " bits.");
                print("encontrando um q com  ", n-tamanho(p)," bits...");
                q = procuraprimo(n - tamanho(p));
                print("q tem ", tamanho(q), " bits.");
                if(tamanho(p*q) != n,
                        print("p*q tem "tamanho(p*q)" bits... vou procurar outros...")
                );
        );
        /* parte II: encontrar e primo relativo com phi(p*q) */
        phi=(p-1)*(q-1);
        print("gerando a chave publica ...");
        e=p-1;
        while( gcd(e,phi)!=1,
                e=random(phi)
        );
        /* parte III: encontrar o inverso de e mod phi */
        print("gerando a chave privada...");
        d=lift(Mod(e^(-1),phi));
        return([p*q,e,d])
}
```

Vamos supor que se pretende criar uma chave RSA com 256 bits:

```
? chave=gerachave(256)
encontrando um p com  128 bits...
p tem 128 bits.
encontrando um q com  128 bits...
q tem 128 bits.
gerando a chave publica ...
gerando a chave privada...
  = [99674460025163334770283829822712881683959339993526775830958718185586858455043,
21777040170807238460926506297647669586134172133184735282362041210284578653877,
87862190169716416623245537218831764166294852997176815108978808098199948935373]
```

No terno ordenado, a primeira componente indica $n = pq$, a segunda indica e e a terceira é o inverso de e módulo $\phi(n)$. Esta terceira componente é mantida secreta (é a chave privada), enquanto que (n, e) são tornados públicos.

Para cifrar uma mensagem x calcula-se $x^e \mod n$. A decifração da mensagem recebida y é feita como $y^d \mod n$.

```
/*
Para cifrar um texto, usa-se a funcao
> cifrar("TEXTO", chave)

Existem duas questoes: #1 o "TEXTO" nao pode ter muitos caracteres em relacao ao
numero de bits da chave. #2 APENAS se admitem MAIUSCULAS no texto. Por exemplo
> texto=cifrar("OLA",chave)
```

```
se tudo correu bem,
> decifrar(texto,chave)
mensagem numerica decifrada 797665
passando para alfanumerico...
 = "OLA"
*/

{
cifrar(texto,vectorchave)=
	lista=Vecsmall(texto);
	tamanholista=length(lista);
	mensagem=0;
	for(j=0,tamanholista-1,
		mensagem=mensagem+10^(2*j)*lista[tamanholista-j]
	);
	if(mensagem> vectorchave[1],
		error("ooops... o texto e’ demasiado grande para a chave :-(")
	);
	print("mensagem numerica ... "mensagem);
	print("usando o expoente de encriptacao "vectorchave[2]," mod "vectorchave[1]);
	cifrado=lift(Mod(mensagem,vectorchave[1])^vectorchave[2]);
	print("a mensagem cifrada e’ " cifrado);
	return(cifrado);
}

{
decifrar(mensagem,vectorchave)=

	local(comp,decifrado,alfanum,caract, mensdesc);

	decifrado=lift(Mod(mensagem,vectorchave[1])^(vectorchave[3]));
	print("mensagem numerica decifrada "decifrado);
	print("passando para alfanumerico...");
	comp=floor(log(decifrado)/log(10))+1;
	alfanum=listcreate(comp/2);
	for (j=1,comp/2,
		caract=decifrado%10^(2); ;
		decifrado=decifrado\10^(2);
		listput(alfanum,Strchr(caract),j);
	);
	\\ passar a lista para palavra
	kill(mensdec); mensdec="";

	forstep(j=comp/2,1,-1,
		mensdec=concat(mensdec,alfanum[j]);
	);
	return(mensdec);
}
```

Por exemplo, para cifrar a frase `OLA MUNDO`, faz-se:

```
? y=cifrar("OLA MUNDO", chave)
mensagem numerica ... 797665327785786879
usando o expoente de encriptacao
21777040170807238460926506297647669586134172133184735282362041210284578653877 mod
99674460025163334770283829822712881683959339993526775830958718185586858455043
a mensagem cifrada e'
45006577619069030979170826177370718008201479840643870395757936043878351172371
```

Para decifrar a mensagem recebida

$$y = 13891630208401250726566239828359124569775313308592426662416560068917691906286$$

basta calcula $y^d \mod n$:

```
? decifrar(y,chave)
mensagem numerica decifrada 7932836971826968793269838465327865327765838365
passando para alfanumerico...
 = "O SEGREDO ESTA NA MASSA"
```

Alertamos para os cálculos que o RSA requer, e do problema de *over-flow* que poderá ocorrer se não se tomarem as precauções devidas. No caso apresetado, tivemos o cuidado de efectuar as exponenciais modulares. O algoritmo que apresentamos de seguida, e que não usámos, é denominado *Square & Multiply*. Permite efectuar estas exponenciais modulares por forma a evitar erros de *over-flow*.

```
\\ Pedro Patricio
\\ square and multiply mod n
\\ argumento de entrada (x,e,n)
\\ calculo de x^e mod n

fmodexp(x,e,n)=
{       local(bin, comp, c);

        bin=binary(e);
        comp=length(bin);
        c=1;
        for(i=1,comp,
                c=Mod(c^2,n);
                if(bin[i]==1,
                        c=Mod(x*c,n)
                )
        );
        return(c);
}
```

Para terminar esta secção, apresentamos um algoritmo que transforma um certo número num vector cujas entradas são os dígitos do número.

```
\\ converte um numero numa lista com os seu algarismos
\\ Pedro Patricio
```

```
num2array(n)=
{       local(tamanho,lista,indice);

        tamanho=floor(log(n)/log(10))+1;
        if(tamanho==11,
                lista=listcreate(tamanho-1);
                for(indice=1,tamanho-1,
                        listput(lista,0,indice)
                );
                listput(lista,10,10);
                n=n\100;
                forstep(indice=tamanho-2,1,-1,
                resto=n%10;
                listput(lista,resto,indice);
                n=n\10;
                ), \\else
                lista=listcreate(tamanho);
                for(indice=1,tamanho,
                        listput(lista,0,indice)
                );
                forstep(indice=tamanho,1,-1,
                resto=n%10;
                listput(lista,resto,indice);
                n=n\10;
                );
        );
        return(lista);
}
```

Por exemplo,

```
? num2array(12335)
 = List([1, 2, 3, 3, 5])
```

17 Solovay-Strassen

O algoritmo Solovay-Strassen é um algoritmo probabilístico de primalidade (tal como o de Miller-Rabin). Historicamente, este algoritmo está associado à cifra RSA por garantir uma efectiva aplicação deste tipo de chave pública.

Dizemos que um ímpar n passa o teste de Solovay-Strassen de base b, com $1 \leq b < n$, se

$$b^{\frac{n-1}{2}} \equiv \left(\frac{b}{n}\right) \mod n.$$

Se a congruência não for válida para um certo b então n é composto, e a b chamamos a testemunha. Na prática, escolhem-se aleatoriamente k bases e efectua-se o teste para cada uma dessas bases. Se um deles falhar, então n é garantidamente composto. No caso contrário, a probabilidade de n ser primo é superior a $\frac{1}{2^k}$.

```
solovay(numero,nbases=1)=
{
        local(i,primo);

        i=1;
        primo=1;

        /*salvaguarda do caso em que numero e' par */
        if(numero==2, print("2 e' primo"), /*else*/
        if(numero%2==0, print(numero" e' garantidamente composto"),

        /*aqui comeca a parte nao trivial*/

        while(i<=nbases && primo,
                b=random(numero-1)+1;
                if(Mod(b,numero)^((numero-1)/2)!=Mod(kronecker(b,numero),numero),
                        print(numero" e' garantidamente composto; testemunha: "b);
                        primo=0;
                );
        i++);
        if(primo==1,
                print(numero" e' provavelmente primo");
        ,/*else*/
                primo=0);
        return(primo);

        ); /* fim dos if's do inicio */
        );
}
```

Recorde que 2047 é o mais pequeno pseudoprimo forte na base 2.

```
? solovay(2047)
2047 e' garantidamente composto; testemunha: 178
```

Tente, por exemplo, com um número de Mersenne que não seja primo. Por exemplo,

```
? n=2^(6531)-1;
? solovay(n,3)
[...] e' garantidamente composto; testemunha:
[ numero muito grande ]
```

Referências

[1] C. Batut, K. Belabas, D. Bernardi, H. Cohen, M. Olivier, *User's Guide to PARI/GP*, The PARI Group, 2006.

[2] Karim Belabas, *PARI-GP Reference Card*, disponível online.