UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE MATEMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM MATEMÁTICA APLICADA

CRIPTOGRAFIA DE CHAVE PÚBLICA

por

Ruy Carlos Miritz

Dissertação submetida como requisito parcial para a obtenção do grau de Mestre em Matemática Aplicada.

Prof. Dr. Vilmar Trevisan
Orientador

PORTO ALEGRE, Abril de 2000.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Miritz, Ruy Carlos

Criptografia de chave pública / Ruy Carlos Miritz. – Porto Alegre: PPGMAP da UFRGS, 2000.

93p.:il.

Dissertação(mestrado) – Universidade Federal do Rio Grande do Sul, Instituto de Matemática, Programa de Pós-Graduação em Matemática Aplicada, Porto Alegre, 2000.

Orientador: Vilmar Trevisan

Dissertação: Cripto-sistemas, Assinatura Digital.

AGRADECIMENTOS

Ao professor Vilmar, pela dedicação.

À Laura, pela paciência e carinho.

Aos colegas Cirlei e Roque pelo apoio.

Ao José, Luiz, Cristiane e Luciane, meus filhos, pelo incentivo.

RESUMO

A criptografia de chave pública tem sido usada intensamente para manter segura a troca de mensagens entre partes, nos dias atuais, em que há a necessidade de exaustiva permuta de dados, seja no mundo comercial, seja industrial ou acadêmico.

Neste trabalho discutem-se sistemas de criptografia de chave pública baseados em três paradigmas: o knapsack, o RSA e o logaritmo discreto, dando ênfase especial à matemática subjacente, necessária para a fundamentação dos métodos e aos algoritmos resultantes que possibilitam a implementação.

Também é discutida a assinatura digital, uma necessidade em sistemas públicos e como obtê-la com os atuais sistemas criptográficos. Apresentamos ainda o sistema PGP de criptografia, recentemente lançado, que é um sistema misto, com aparente aceitação na comunidade de usuários.

Os métodos são examinados levando em consideração não só a sua aplicabilidade, mas também a sua segurança.

ABSTRACT

The public key cryptography is important for communication with security. Cryptosystems, in special, based on the problem from knapsack, the RSA and the one based on discrete logarithm allow the key-exchange and the digital signature presents, as well as the underlying mathematics. The security and privacy provided on the PGP system are showed. Maple codes for the implementation of algorithms are presented.

SUMÁRIO

ALGORITMOS

FIGURAS

TABELAS

CRIPTOGRAFIA DE CHAVE PÚBLICA

1. INTRODUÇÃO

A palavra *cryptòs*, em grego, significa escondido, oculto. A criptografia consiste em maneiras para escrever em código uma mensagem de tal forma que somente pessoas autorizadas consigam decifrá-la. Por outro lado, decodificar uma mensagem sem possuir a autorização é denominado criptoanálise, ou seja, o criptoanalista procura um método para quebrar a codificação.

A idéia básica consiste em modificar uma mensagem para torná-la inintelegível para qualquer outra pessoa, menos para a destinatária.

Mensagens codificadas têm uma longa história. Os espartanos empregavam um interessante método para codificar e decodificar há aproximadamente 400 anos antes de Cristo, que consistia de um bastão em volta do qual era enrolado, em espiral, uma faixa de pergaminho contendo a mensagem. Quando a faixa era desenrolada, as letras da mensagem apareciam misturadas; entretanto, quando o pergaminho era enrolado em torno de outro bastão de tamanho similar, a mensagem original reaparecia [WEL 90].

O sistema de criptografia mais simples consiste em substituir uma letra do alfabeto pela letra seguinte. Julio César (101 aC - 44 aC) usava um sistema semelhante para as comunicações com as suas legiões: substituia a letra A por D, a letra B por E, e assim por diante. Somente quem conhecia a regra "mudar 3" conseguia decifrar as mensagens.

Cada sistema traz consigo a exigência de dois conhecimentos: um para a codificação de uma mensagem e outro para a decodificação da mensagem codificada.

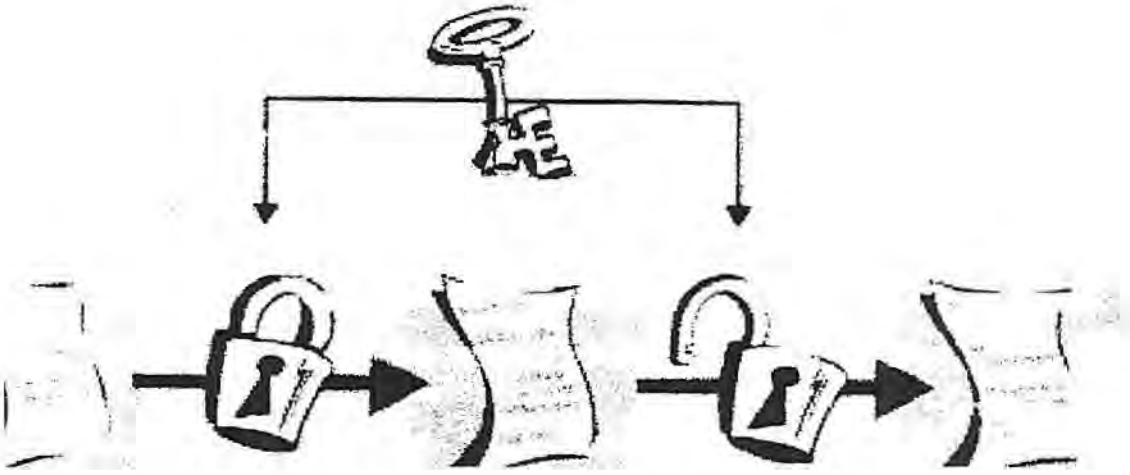
Assim, na figura abaixo, procura-se mostrar que um texto pleno ou mensagem, escrito na forma natural, é codificado através de um sistema de criptografia, resultando o texto codificado. Para a decodificação precisa-se ter o conhecimento suficiente que capacita transformar a mensagem ou texto codificado e retornar ao texto pleno.

Figura 1 - Codificação e decodificação (extraído de [ZIM98]).

Evidentemente os sistemas citados anteriormente e outros semelhantes não são seguros. Sistemas que envolvem a troca sistemática de uma letra por uma outra letra ou símbolo, tornam-se fáceis de quebrar, observando a freqüência com que determinada letra é usada. Sabendo, por exemplo, que a língua usada é a portuguesa, tem-se que a letra mais usada é a letra A, as vogais são mais usadas do que as consoantes, as consoantes mais usadas são M e S; torna-se possível, com certa facilidade, decodificar uma mensagem assim codificada. Foi com estas constatações que Champollion, em 1822, descobriu a chave para a decifração da pedra de Roseta.

Sendo a língua conhecida, o cripto-sistema, baseado na substituição de letras, hoje, é absolutamente inviável porque grande parte do trabalho de decodificação pode ser realizado por computadores. Foi exatamente para isto que alguns dos primeiros computadores foram construídos: para a decifração dos códigos secretos usados pelos alemães durante a segunda guerra mundial. Alan Turing (1912-1954) liderou uma equipe de criptoanalistas que desvendou o funcionamento da máquina "Enigma", usada pelos nazistas para codificar mensagens.

Para a codificação e decodificação das mensagens era usada a mesma chave. Interceptando uma mensagem, esta chave precisava ser conhecida para poder realizar a decodificação.



texto pleno codificação texto codificado decodificação texto pleno

Figura 2 - Codificação convencional (extraído de [ZIM98]).

Cripto-sistemas convencionais necessitam, visando uma melhor segurança, a freqüente troca de chaves, tornando o sistema caro e, mesmo assim, pouco seguro. Nos dias atuais, considerando o vertiginoso crescimento do uso da INTERNET, tais sistemas se tornaram impossíveis, porque permitem, com facilidade, decodificar uma mensagem.

O objetivo fundamental da criptografia é permitir e garantir a segura comunicação de mensagens entre duas pessoas, considerando a possibilidade de uma terceira pessoa interceptar a mensagem, o que poderá trazer prejuízos.

Como conseqüência, a grande preocupação de um cripto-sistema é com a segurança das comunicações diante das investidas de bisbilhoteiros (*eavesdropper*) ou *hacker*. Vários métodos foram desenvolvidos, visando oferecer uma melhor e maior segurança, baseados em problemas considerados de (muito) difícil resolução. Embora não haja um sistema absolutamente seguro, a adoção de chaves dificulta a ação de estranhos.

Hoje, problemas novos surgiram com a segurança nas comunicações, em especial com a INTERNET e o crescente comércio eletrônico envolvendo transações mediante o uso do cartão de crédito, e também porque é relativamente fácil interceptar mensagens enviadas por linha telefônica, novos problemas surgiram relacionados com a segurança nas comunicações. Para evitar, ou ao menos dificultar a ação de estranhos, a mensagem deve estar bem protegida por um bom sistema de criptografia - e mais - precisa estar assinada.

Para suprir essas dificuldades foram inventados novos sistemas que, mesmo com a ajuda de computador, tornam a tarefa da decodificação no mínimo muito difícil. Além disso, incluem

uma chave pública. Essa idéia foi introduzida em 1976 por W.Diffie e M.E.Hellman, da Universidade de Stanford e por R.C.Merkle, da Universidade da Califórnia, um independente do outro. Comparando: no sistema de Julio César, quem sabia codificar também sabia decodificar; num sistema de chave pública, saber codificar não implica em saber decodificar.

O objetivo principal desta dissertação consiste no estudo da criptografia de chave pública, alguns cripto-sistemas, enfatizando a matemática subjacente, em especial a teoria de números, e algoritmos para a resolução de problemas atinentes a teoremas e conceitos matemáticos usados em sistemas de criptografia.

Associado a este objetivo, tem-se o uso da criptografia de chave pública para a assinatura digital e recentemente o sistema PGP.

Um outro objetivo consiste em analisar e comparar os cripto-sistemas abordados, apresentando as suas vantagens e desvantagens.

Para atingir o objetivo o trabalho segue a seguinte estrutura: no capítulo segundo serão abordados os fundamentos da criptografia, os conceitos de criptografia, conceito de cripto-sistema de chave pública, conceitos da teoria dos números, necessário para o prosseguimento dos estudos e uma abordagem da teoria da complexidade.

No capítulo terceiro é abordado o cripto-sistema baseado no problema da mochila (*knapsack*), a fundamentação matemática e o sistema de criptografia.

No capítulo quarto é apresentado o sistema de criptografia RSA: a sua definição, a matemática em que se baseia, com destaque aos testes de primalidade probabilísticos, em especial o teste de Miller-Rabin; e como funciona o sistema RSA.

No capítulo quinto tem-se o sistema baseado no logaritmo discreto. Apresentam-se as definições, a matemática subjacente, onde o grupo cíclico é de suma importância. São apresentados diversos métodos de criptografia fundamentados no logaritmo discreto.

O capítulo sexto traz referências à assinatura digital, a sua importância quanto à autenticidade e integralidade nas informações. É explicado o funcionamento da assinatura digital, considerando sistemas de criptografia de chave pública.

PGP (Pretty Good Privacy) é abordado no capítulo sétimo. É uma das mais recentes novidades em criptografia, trazendo uma implementação (gratuita) de um sistema híbrido que usa criptografia convencional e pública.

No capítulo oitavo são apresentadas algumas conclusões obtidas a partir deste estudo. Finalmente apresentamos como apêndice códigos elaborados no Maple a partir de teoremas e definições, facilitando o trabalho e provando a sua veracidade, que implementam vários algoritmos para a criptografia.

2. FUNDAMENTOS

Com o objetivo de, numa forma sistemática, estudar sistemas de criptografia, tornam-se necessários alguns conceitos básicos e que serão apresentados a seguir. Trata-se de conceitos de criptografia, da teoria dos números e da teoria de complexidade.

2.1. CONCEITOS DE CRIPTOGRAFIA

Inicialmente serão apresentados conceitos fornecidos por estudiosos de criptografia e a seguir alguns exemplos de cripto-sistemas.

Criptografia é o estudo de métodos para enviar mensagens em forma disfarçada de modo que somente o receptor projetado pode remover o disfarce e ler a mensagem [KOB94].

Criptografia é a arte e a ciência de produzir mensagens que têm alguma combinação de ser privada, assinada, inalterada com não-repúdio [ZIM98].

Decodificar é o que um usuário legítimo do código faz quando recebe uma mensagem codificada e deseja lê-la. Já decifrar significa ler uma mensagem codificada sem ser um usuário legítimo [COU97].

Entender-se-á, por criptografia, ao longo desse trabalho, o uso de códigos para converter dados de maneira que apenas o destinatário seja capaz de decodificá-los, usando uma chave.

Tipicamente representa-se uma mensagem pela letra m e é formada por um conjunto finito de símbolos. Usa-se $c(m)$ para designar a codificação de m e d, para a decodificação. Tem-se assim a relação fundamental: $d(c(m)) = m$.

Ilustrando com exemplos os conceitos apresentados nesta introdução, far-se-á referência a dois cripto-sistemas bastante simples, o chamado de "simples substituição" e o baseado em aritmética modular. Posteriormente, e este é o que mais interessa neste estudo, será apresentado o cripto-sistema de chave pública.

O cripto-sistema de "simples substituição" consiste na troca das letras do alfabeto. Considerando as 23 letras do alfabeto português, faz-se uma correspondência unívoca entre as letras e que é a chave de codificação, podendo ser da seguinte forma:

a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*l*	*m*	*n*	*o*	*p*	*q*	*r*	*s*	*t*	*u*	*v*	*x*	*z*
X	*N*	*A*	*H*	*P*	*O*	*G*	*Z*	*Q*	*D*	*B*	*T*	*S*	*F*	*L*	*R*	*C*	*V*	*M*	*U*	*E*	*J*	*I*

Tabela 1 - Correspondência entre letras para codificar.

Assim, para codificar: $c(a) = X$, $c(b) = N$, ...

A decodificação é feita pela função inversa, isto é, escrevendo a segunda linha em ordem e fazendo a devida correspondência da primeira. Tem-se assim a chave de decodificação. Resulta o seguinte:

A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*L*	*M*	*N*	*O*	*P*	*Q*	*R*	*S*	*T*	*U*	*V*	*X*	*Z*
c	*l*	*r*	*j*	*v*	*o*	*g*	*d*	*z*	*x*	*p*	*t*	*b*	*f*	*e*	*i*	*q*	*n*	*m*	*u*	*s*	*a*	*h*

Tabela 2 - Correspondência entre letras para decodificar.

Codificando a mensagem m = *criptografia*, ter-se-á:

$c(criptografia) = ACQLMFGCXOQX$

ACQLMFGCXOQX será a mensagem enviada e assim recebida.

Para decodificar, usa-se a tabela 2.

$d(ACQLMFGCXOQX) = criptografia$

retornando a mensagem enviada.

Pelo critpo-sistema baseado em aritmética modular, faz-se a correspondência unívoca entre as letras do alfabeto *A-Z*, e os números *0-22*. Resulta a tabela seguinte:

A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*L*	*M*	*N*	*O*	*P*	*Q*	*R*	*S*	*T*	*U*	*V*	*X*	*Z*
00	*01*	*02*	*03*	*04*	*05*	*06*	*07*	*08*	*09*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*

Tabela 3 - Correspondência entre letras e números.

A função codificação é definida por $C \equiv P + b \bmod N$, onde $P \in \{0, 1, 2, ..., N-1\}$, ou seja P representa o número correspondente à letra a ser codificada, b é um número qualquer, segredo da codificação ou chave e N representa a quantidade de letras do alfabeto.

Considerando, como exemplo, $b = 3$, para codificar, soma-se sempre $3 \bmod 23$, ou seja

$C \equiv P + 3 \ mod \ 23$.

No caso particular em que $b = 3$ o cripto-sistema é chamado de Código Secreto de César, já mencionado anteriormente, porque foi usado na Roma antiga por Julio César, sendo-lhe atribuída a invenção deste sistema.

Para decodificar a mensagem usa-se como chave a função inversa $P \equiv C - b \ mod \ N$.

Desejando enviar a mensagem $m = COMPUTADOR$, com $b = 11$, a codificação, usando $C \equiv P + b \ mod \ N$, será $C(C) \equiv 2 + 11 \ mod \ 23 = 13$, isto é a letra C será codificada pela letra O, que é a letra correspondente ao número 13; $C(O) \equiv 13 + 11 \ mod \ 23 = 1$; a letra O será codificada por B que corresponde ao número 1; e assim por diante. Sendo a mensagem codificada, correspondente a

$COMPUTADOR \rightarrow 13\ 1\ 22\ 2\ 7\ 6\ 11\ 14\ 1\ 4 \rightarrow OBZCHGMPBE$

A mensagem transmitida será $OBZCHGMPBE$.

O receptor, para decodificar, usará a função $P \equiv C - b \ mod \ N$.

$P(O) \equiv 13 - 11 \ mod \ 23 = 2$ e ao número 2 corresponde a letra C.

$P(B) \equiv 1 - 11 \ mod \ 23 = 13$, ao número 13 corresponde a letra O

e assim por diante. Ocorrendo a correspondência

$OBZCHGMPBE \rightarrow 2\ 13\ 11\ 14\ 19\ 18\ 0\ 3\ 13\ 16 \rightarrow COMPUTADOR$

O sistema apresentado, baseado em aritmética modular, oferece pouca segurança. Em textos maiores, mediante a análise (criptoanálise) verifica-se qual foi a letra mais usada no texto codificado e sabendo qual é a letra mais usada no idioma em questão, basta fazer a correspondência, conseguindo assim quebrar a codificação.

Para suprir a falta de segurança foi criado o cripto-sistema de chave pública.

2.2. CRIPTO-SISTEMA DE CHAVE PÚBLICA

A idéia do sistema de chave pública evita a necessidade da comunicação secreta das chaves. Depende de uma função "alçapão". Todos os usuários usam o mesmo algoritmo c para codificar e o mesmo algoritmo d para decodificar. Cada usuário tem um par de chaves (K_i, L_i) tal

que, para uma mensagem m, exista a identidade $d(c(m, K_i), L_i) = m$, onde K_i, tornado público, é a chave pública e L_i, mantido secreto, é a chave privada.

Para a codificação e depois a decodificação são criadas duas chaves: uma chave pública e uma chave secreta (privada), sendo uma do emitente da mensagem e a outra do destinatário.

Da mensagem codificada resultará uma seqüência de números. Se a mensagem for muito grande ela pode ser dividida em blocos e assim ser codificada.

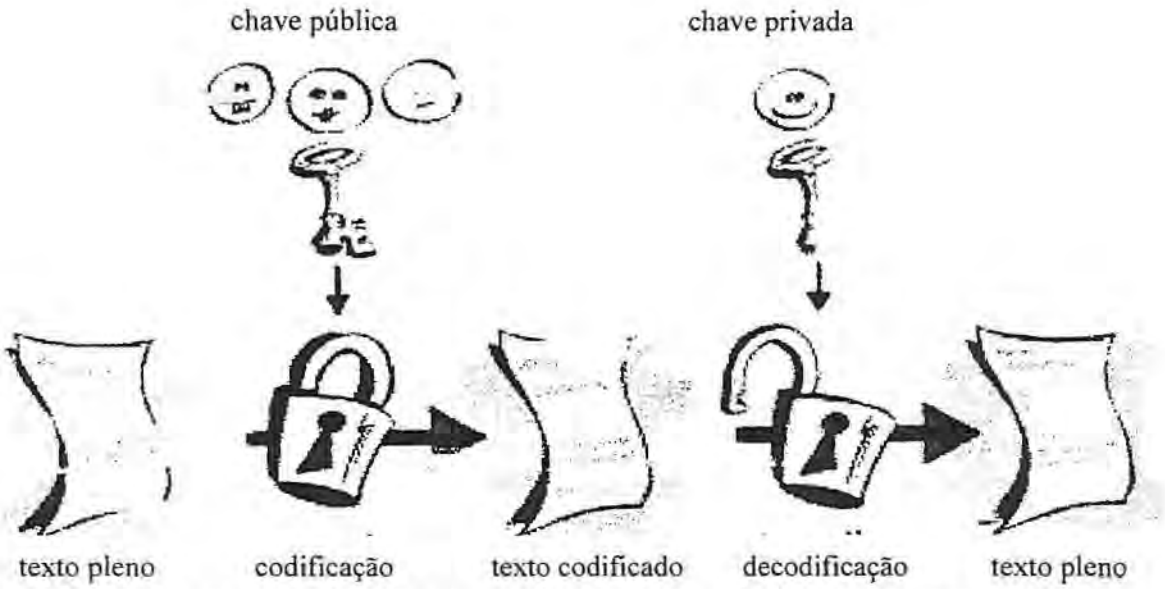


Figura 3 - Codificação com chave pública (extraído de [ZIM98]).

Uma função codificadora pública c_K pode ser fácil de calcular, mas a função inversa, a decodificadora, deve ser difícil para algum estranho. Esta propriedade de tornar fácil para calcular mas difícil para inverter é chamada de propriedade de *uma-via, injetiva ou unidirecional*. É recomendável que c_K seja uma função de uma-via.

Funções uma-via ocupam um lugar central em criptografia; são muito importantes na construção de cripto-sistemas de chave pública.

Informalmente, uma função uma-via é uma função $f: S \rightarrow T$, em que S e T são conjuntos tais que:

(i) para qualquer $x \in S$, $f(x)$ é 'fácil' para calcular,

(ii) dada a informação de que $f(x) = y$, não há um caminho 'possível' para obter (calcular) x para uma razoavelmente grande parte de y pertencente a T.

Um exemplo de função uma-via: supondo n como um número primo grande, e seja $f(x)$ um polinômio do corpo dos inteiros *mod n*. É relativamente fácil calcular $f(x)$ $(1 \leq x \leq n-1)$, mas geralmente difícil para encontrar a solução de $f(x) = y$. Um outro exemplo de função uma-via, que é usada no cripto-sistema RSA: dados dois números primos grandes p e q, é fácil calcular $n = pq$, mas é muito difícil, tendo n, determinar os fatores primos p e q.

Para a construção de um cripto-sistema não é suficiente encontrar uma função uma-via. É necessário, para que a mensagem possa ser decodificada seguramente, que seja enviada e recebida por um caminho eficiente e que se tenha um alçapão (*trapdoor*).

A idéia da função alçapão surgiu em 1976, através dos trabalhos de Whitfield Diffie e Martin Hellman, e consta de uma informação secreta que permite fácil inversão de c_K (=regra de codificação), isto é, alguém pode decodificar eficientemente porque tem algum conhecimento secreto extra sobre K (=chave).

Dos diversos cripto-sistemas de chave pública existentes, três merecerão uma atenção especial neste estudo: o baseado no problema da mochila, também conhecido por *knapsack*; o sistema *RSA*, e o problema do *logaritmo discreto*.

Atualmente, o cripto-sistema de chave pública *RSA* é o mais utilizado em aplicações comerciais. É usado no *Netscape*, o mais popular dos *softwares* de navegação da *Internet*. Como é relativamente fácil interceptar mensagens enviadas por linha telefônica, incluindo transações bancárias, comerciais e compras feitas com cartão de crédito, é necessário proteger a mensagem; precisa-se ter a segurança de que a mensagem recebida foi enviada por um usuário da empresa, ou seja, a mensagem precisa estar assinada, e isto é possível pela criptografia de chave pública.

Através da criptografia de chave pública, para uma maior segurança, pode ser realizada a troca de chaves quando isto se julgar necessário e pode ser usada em combinação com um sistema de criptografia convencional.

2.3. TEORIA DOS NÚMEROS

A criptografia vale-se de vários conceitos matemáticos, principalmente da Teoria dos Números. A seguir apresentaremos alguns conceitos básicos que serão úteis para a compreensão e obtenção de um cripto-sistema.

Dados os inteiros a e b, diz-se que a divide b ou b é divisível por a e escreve-se $b|a$ se existe algum inteiro d tal que $b = ad$. Nesse caso diz-se que a é divisor de b. Todo número inteiro b maior do que 1 tem, no mínimo, dois divisores, 1 e b. Por *divisor próprio* de b entende-se um

divisor de b, não-igual a ele mesmo e por *divisor não-trivial* de b, o divisor de b não-igual a 1 e a b. Um número inteiro e positivo maior do que 1 é chamado *primo* quando não tem outros divisores além de 1 e *ele mesmo*; um número é chamado *composto* quando tem, no mínimo, um divisor não-trivial.

As propriedades da divisibilidade mais utilizadas são:

(1) Se $a|b$ e c é um inteiro, então $a|bc$.

(2) Se $a|b$ e $b|c$ então $a|c$.

(3) Se $a|b$ e $a|c$ então $a|b\pm c$.

(4) Se um número primo p divide ab, então ou $p|a$ ou $p|b$.

(5) Se $m|a$ e $n|a$, e se m e n não tem divisores comuns além de 1, então $mn|a$.

Se p é um número primo e d é um inteiro não-negativo, então a notação $p^d \| b$ significa que p^d é a maior potência de p que divide b, isto é, p^d|b e p^{d+1} ~|b .

Também é importante o Teorema Fundamental da Aritmética. Ele diz que qualquer número natural n pode ser escrito de uma única maneira como produto de seus fatores primos, exceto pela ordem de seus fatores. Por exemplo: $4725 = 3^3.5^2.7$.

Uma conseqüência da fatoração única de um número é que os fatores e suas potências permitem calcular todos os divisores deste número. Para encontrar todos os divisores de 4725, começa-se testando, em ordem, os números primos como possíveis divisores; como 4725 é ímpar, não tem o número 2 como divisor; testando o número 3, vê-se que 3 é divisor, dividindo 4725 por 3, tem-se o quociente 1575 que ainda é divisível por 3 e o quociente de 1575 por 3 que é 525 e ainda é divisível por 3, resultando o quociente 175; toma-se, por isso, o fator primo 3 com os expoentes 0, 1, 2 e 3, resultando os divisores 3^0 ,3^1 ,3^2 e 3^3 ou seja, a partir do 3 descobre-se os divisores 1, 3, 9 e 27. Passando para o divisor 5 que é o próximo número primo, tem-se 175 divisível por 5 e o quociente 35, ainda divisível por 5, fornecendo o quociente primo 7. Por isso, o fator 5 é tomado com os expoentes 0, 1 e 2, tem-se os divisores 5^0 , 5^1 e 5^2 ; mas $5^0 = 1$ e já se tem o divisor 1, surgindo assim o 5 e o 25 como novos divisores. Pela aplicação da propriedade 5, descobre-se que 5.3 = 15, 5.9 = 45, 5.27 = 135, 25.3 = 75, 25.9 = 225 e 25.27 = 675 também são divisores de 4725. Ordenando, tem-se o seguinte conjunto de divisores de 4725:{1, 3, 5, 9, 15, 25, 27, 45, 75, 135, 225, 675}, ainda sem levar em consideração o fator 7. Com o fator primo 7 e os expoentes 0 e 1, tem-se $7^0 = 1$ e $7^1 = 7$, sendo o novo divisor o 7, e novamente pela aplicação da propriedade 5, tem-se ainda como divisores de 4725, 7.3 = 21, 7.5 = 35, 7.9 = 63, 7.15 = 105, 7.25 = 175, 7.27 = 189, 7.45 = 325, 7. 75 = 525, 7.135 = 945, 7.225 = 1575 e 7.675 = 4725.

Assim, o conjunto de todos os divisores de 4725, obtido a partir de seus fatores primos e os correspondentes expoentes é

D(4725) = {1, 3, 5, 7, 9, 15, 21, 25, 27, 35, 45, 63, 75, 105, 135, 175, 189, 225, 325, 525, 675, 945, 1575, 4725}

O número de divisores é o produto dos expoentes dos fatores primos mais um, isto é, um número $n = p_1^{a_1} p_2^{a_2} \ldots p_r^{a_r}$ tem $(a_1 + 1)(a_2 + 1)\ldots(a_r + 1)$ divisores diferentes.

Por exemplo, o número 4725 utilizado acima, tem (3 + 1)(2 + 1)(1 + 1) = 24 divisores.

Apesar de ser um método determinístico que permite descobrir todos os divisores ou simplesmente a quantidade de divisores de um número, para números grandes é ineficiente porque é muito lento, pois exige a determinação de todos os fatores primos do número em questão.

Dados três números inteiros *a*, *b* e *m*, diz-se que "*a* é congruente a *b* em módulo *m*" e escreve-se $a \equiv b \bmod m$, se a diferença *a* - *b* é divisível por *m*. Denomina-se *m* de módulo de congruência.

As propriedades das congruências são:

(i) $a \equiv a \bmod m$ (reflexividade)

(ii) $a \equiv b \bmod m$ se e somente se $b \equiv a \bmod m$ (simetria)

(iii) se $a \equiv b \bmod m$ e $b \equiv c \bmod m$ então $a \equiv c \bmod m$ (transitividade)

Pelos itens (i)-(iii) tem-se que a congruência em módulo *m* é uma relação de equivalência.

(iv) Fixado *m*, cada classe de equivalência[1] com respeito ao módulo de congruência *m*, tem uma e somente uma representação entre *0* e *m - 1*. O conjunto das classes de equivalência, também chamada classe de resíduos, será representado por *Z/mZ*.

(v) Se $a \equiv b \bmod m$ e $c \equiv d \bmod m$, então $a \pm c \equiv b \pm d \bmod m$ e $ac \equiv bd \bmod m$.

(vi) Se $a \equiv b \bmod m$, então $a \equiv b \bmod d$ para qualquer divisor *d* de *m*.

(vii) Se $a \equiv b \bmod m$, $a \equiv b \bmod n$, e *m* e *n* são relativamente primos, então $a \equiv b \bmod mn$.

(viii) Se o *mdc(a,m) = 1*, então a potência negativa $a^{-n} \bmod m$ obtém-se pela n-ésima potência do inverso da classe de resíduo, isto é, alguma n-ésima potência de um inteiro *b* para o qual $ab \equiv 1 \bmod m$.

[1] Uma relação é de equivalência em um conjunto quando possui as propriedades reflexiva, simétrica e transitiva. Das subdivisões de um conjunto produzidas por uma relação de equivalência resultam as classes de equivalência.

A operação que determina o inverso em módulo n é essencial para vários métodos de criptografia e, por isso, detalhar-se-á mais esta operação, deduzindo inclusive um algoritmo.

Entretanto, antes de deduzir-se o método, julga-se interessante exemplificar: seja determinar $160^{-1} \bmod 841$; em outras palavras, o inverso de 160 em módulo 841.

Inicia-se dividindo o número 841 por 160. O importante na divisão não é o quociente mas sim, o resto; depois, e assim sucessivamente, divide-se o quociente pelo resto até encontrar resto 1. Tem-se

$$841 = 5.160 + 41 \qquad (1)$$
$$160 = 3.41 + 37 \qquad (2)$$
$$41 = 1.37 + 4 \qquad (3)$$
$$37 = 9.4 + 1 \qquad (4)$$

de onde se conclui que o mdc(841, 160) = 1, atendendo à propriedade (viii).

Retornando, através de substituições, obter-se-á

de (4) $1 = 37 - 9.4$

de (3) $1 = 37 - 9(41 - 1.37) = 10.37 - 9.41$

de (2) $1 = 10(160 - 3.41) - 9.41 = 10.160 - 39.41$

de (1) $1 = 10.160 - 39(841 - 5.160) = 205.160 - 39.841$

ou seja: $205.160 - 39.841 = 1$. Reescrevendo sob forma de módulo:

$205.160 \equiv 1 \bmod 841$ e retornando à questão de origem

$160^{-1} \bmod 841 = 205.$

O processo utilizado para determinar o inverso de um número em módulo é conhecido como algoritmo de Euclides estendido e que será apresentado a seguir.

O tão conhecido algoritmo de Euclides usado para calcular o máximo divisor comum de dois números inteiros pode ser estendido para calcular os inteiros u' e v' tais que

$$uu' + vv' = mdc(u,v)$$

ao mesmo tempo em que é calculado o $mdc(u,v)$.

Descrito, usando a notação vetorial e sob a forma de pseudo-algoritmo, o algoritmo de Euclides estendido passa a ser: "Dados dois números inteiros não-negativos u e v, o algoritmo determina um vetor (u_1, u_2, u_3) tal que $uu_1 + vu_2 = u_3 = \text{mdc}(u,v)$. Nos cálculos são usados os vetores auxiliares (v_1, v_2, v_3) e (t_1, t_2, t_3). Todos os vetores são manipulados de tal maneira que permaneçam através dos cálculos as seguintes relações:

$$ut_1 + vt_2 = t_3, \qquad uu_1 + vu_2 = u_3, \qquad uv_1 + vv_2 = v_3.$$

Formalmente tem-se:

Algoritmo 1 - Cálculo do inverso de v em módulo u.

mdceuinv(u, v)

1. Inicialização.

   Faça $(u_1, u_2, u_3) \leftarrow (1, 0, u)$

   Faça $(v_1, v_2, v_3) \leftarrow (0, 1, v)$

2. $v_3 = 0$?

   Se $v_3 = 0$, o algoritmo termina e RETURN (O mdc é u_3).

   Se $u_3 = 1$ RETURN (O inverso de v em módulo u é u_2).

3. Divide e subtrai.

   Faça $q \leftarrow \lfloor u_3 / v_3 \rfloor$ e então

   faça $(t_1, t_2, t_3) \leftarrow (u_1, u_2, u_3) - (v_1, v_2, v_3) q$

   $(u_1, u_2, u_3) \leftarrow (v_1, v_2, v_3)$

   $(v_1, v_2, v_3) \leftarrow (t_1, t_2, t_3)$

4. Retorne à etapa 2.

Como exemplo da aplicação do algoritmo de Euclides estendido, resulta a seguinte tabela:

Q	u1	u2	u3	v1	v2	v3	t1	t2	t3
-	1	0	841	0	1	160	1	-5	41
5	0	1	160	1	-5	41	-3	16	37
3	1	-5	41	-3	16	37	4	-21	4
1	-3	16	37	4	-21	4	-39	205	1
9	4	-21	4	-39	205	1	160	-841	0
4	-39	205	1	160	-841	0	-	-	-

Tabela 4 - Algoritmo de Euclides estendido

Interpretando: 205.160 - (-39).(-841)= 1

Como *mdc(841, 160) = 1*, tem-se que 160^{-1} *mod 841* é equivalente a $160.b \equiv 1 \bmod 841$, ou seja, $b = 205$, respondendo também à questão proposta:

$160^{-1} \bmod 841 \equiv 205$.

2.4. TEORIA DA COMPLEXIDADE

Também uma compreensão dos conceitos básicos da teoria da complexidade é importante em criptografia para que o custo computacional de cada método seja obtido e esses serão abordados a seguir.

A teoria da complexidade computacional está interessada em classes de problemas que podem ser resolvidos por algoritmos e tenta classificar os problemas de acordo com sua dificuldade computacional, medida como a quantidade de tempo para o cálculo e solução ou espaço, que a solução irá ocupar.

Quando é proposto um problema para ser resolvido, em geral tem-se vários algoritmos para essa tarefa. Dependendo da prioridade e equipamentos disponíveis, pode-se optar:

1) pelo que toma menos tempo;
2) pelo espaço que ele ocupa;
3) pelo que é mais fácil de implementar;
4) por desenvolver um algoritmo novo.

De um modo geral, a escolha de determinado algoritmo para resolver um problema depende da sua eficiência; levando-se em conta o número de operações que executa, ou o espaço necessário para implementá-lo.

Considerando, por exemplo, a multiplicação de dois números inteiros x e y, em que $x = x_1 \ldots x_n$ e $y = y_1 \ldots y_n$. O método para multiplicar, aprendido na escola elementar, que consiste em multiplicar cada dígito de x por $y_1 \ldots y_n$, mudando a posição e depois somando o resultado, realiza n operações para cada multiplicação de x por y_i, logo, para fazer a multiplicação de x e y, totaliza n^2 multiplicações. A questão é: para efetuar multiplicações existe um método que realize esta tarefa com menos operações ? Ou seja, com menos gasto computacional ?

Para responder este tipo de questões, são necessárias algumas definições referentes à **notação assintótica.**

Definição 1: Seja $f: N \to R^+$. $O(f)$ é o conjunto das funções $g: N \to R^+$ tal que $\exists\, c \in R^+$, $n_0 \in N$, $g(n) \leq cf(n)$, $\forall\, n \geq n_0$.

$O(f)$ são as funções que não crescem mais rapidamente do que f.

Equivalentemente, $g \in O(f)$ se $\lim_{n\to+\infty} \frac{g(n)}{f(n)} = c \geq 0$, caso o limite exista.

Definição 2: Seja $f: N \to R^+$. $\Omega(f)$ é o conjunto das funções $g: N \to R^+$ tal que $\exists c \in R^+$, $n_0 \in N$, $g(n) \geq cf(n)$, $\forall n \geq n_0$.

$\Omega(f)$ são as funções que crescem pelo menos tão rapidamente como f.

Equivalentemente $g \in \Omega(f)$ se $\lim_{n\to+\infty} \frac{g(n)}{f(n)} = c > 0$ ou quando o limite é igual ao infinito, caso o limite exista.

Definição 3: Seja $f: N \to R^+$. $\theta(f)$ é o conjunto das funções $g: N \to R^+$ tal que $\exists c \in R^+$, $n_0 \in N$, $g(n) = cf(n)$, $\forall n \geq n_0$. Ou seja, $\theta(f) = \Omega(f) \cap O(f)$.

$\theta(f)$ são as funções que crescem tão rapidamente como f.

Equivalentemente $g \in \theta(f)$ se $\lim_{n\to+\infty} \frac{g(n)}{f(n)} = c > 0$, caso o limite exista.

Usando um algoritmo, pode-se, pela sua análise ao resolver um problema, verificar se o tempo é de $O(f(n))$, para alguma função $f(n)$ que se deseja reduzir o máximo possível. Usando complexidade, por outro lado, tenta-se encontrar uma função $g(n)$ tão ampla quanto possível para a qual pode-se provar que algum algoritmo capaz de resolver o problema corretamente em todas as suas instâncias deve necessariamente tomar o tempo em $\Omega(g(n))$. Chama-se a função $g(n)$ de limite inferior de complexidade do problema. Quando $f(n) \in \theta(g(n))$ tem-se o algoritmo mais eficiente possível, exceto talvez pela troca da constante multiplicativa.

Para maiores detalhes recomenda-se [WEL90], onde, no capítulo 9, trata da complexidade computacional. Para exemplificar, deseja-se listar somente a complexidade de alguns algoritmos:

a) A multiplicação de dois números inteiros com n dígitos cada um
   - tradicional: $O(n^2)$;
   - por Karatsuba: $O(n^{\log_2 3}) \approx O(n^{1,59})$

b) 2^n pode ser calculado com $O(\log_2 n)$ multiplicações.

É interessante, para entender melhor, que seja dado um exemplo da complexidade de dois algoritmos para calcular potências: o tradicional, que faz as multiplicações em seqüência, e o baseado no divide-e-conquista. Como esta operação é essencial para a implementação dos algoritmos para a criptografia, vamos apresentá-los com mais detalhes.

Consideremos que o objetivo é calcular $x = a^n$.

Algoritmo 2 - Cálculo de potências em seqüência.

exposeq (a, n)

1. Faça $r \leftarrow a$
2. Para $i \leftarrow 1$ até n - 1 faça $r \leftarrow a \, . \, r$
3. RETURN r.

Este algoritmo é de um tempo em $\theta(n)$ porque a instrução $r \leftarrow a \, . \, r$ é executada exatamente n - 1 vezes, desde que as multiplicações são contadas como operações elementares e não se considerem operandos muito grandes. Assim para calcular a^{29} são necessárias 28 multiplicações.

Algoritmo 3 - Cálculo de potências baseado no divide-e-conquista.

expoDC (a, n)

1. Se $n = 1$, então RETURN a
2. Se n é par então RETURN $[expoDC(a, n/2)]^2$
3. RETURN a.expoDC (a, n - 1).

O tempo deste algoritmo está em $\theta(\log n)$, o que pode ser verificado pela relação de recorrência

$$N(n) = \begin{cases} 0 \ \ se \ \ n = 1 \\ N(n/2) + 1 \ \ se \ \ n \ \ e \ \ par \\ N(n-1) + 1 \ \ em \ \ caso \ \ contrario \end{cases}$$

que determina a complexidade do algoritmo.

Para calcular a^{29} é usado o seguinte:

$$a^{29} = a.a^{28} = a(a^{14})^2 = a((a^7)^2)^2 = \ldots = a((a(a.a^2)^2)^2)^2,$$

envolvendo somente sete multiplicações: três multiplicações simples e mais quatro, relativas às elevações ao quadrado, em vez de 28 multiplicações do algoritmo exposeq.

Para maiores detalhes recomenda-se [BRA96], páginas 243 a 247, onde também é feito o estudo considerando-se o tamanho do número a e ainda o tipo de algoritmo usado nas multiplicações.

Por definição, diz-se que um algoritmo A tem complexidade de tempo polinomial se existe algum polinômio $p(x)$ tal que $t_A(n) \leq p(n)$ para cada inteiro n, onde $t_A(n)$ representa o tempo máximo gasto pelo algoritmo para todos os *inputs* de tamanho n.

Muitos problemas podem ser formulados como *problemas de decisão*, isto é, problemas para os quais o *output* é simplesmente uma resposta *sim* ou *não* para cada *input*.

Considerando o problema da mochila (*knapsack*), em que a mochila tem uma capacidade C (um número inteiro positivo) e existem n objetos com pesos $s_1,\ldots,s_n$ e valores $p_1,\ldots,p_n$ ($s_1,\ldots,s_n$ e $p_1,\ldots,p_n$ são números inteiros positivos). A otimização do problema consiste em encontrar o maior valor total de algum subconjunto de objetos que cabem na mochila, ou seja, encontrar um subconjunto que atinge um valor máximo. O problema de decisão passa a ser: dado k, existe um subconjunto de objetos que cabem na mochila e que tem um valor total no mínimo k ?

Numa versão mais simples, tem-se o problema da soma dos elementos de um subconjunto: o *input* é um número inteiro positivo C e existem n objetos cujos pesos são os números inteiros positivos $s_1,\ldots,s_n$. O problema de otimização é: entre os subconjuntos de objetos (elementos) cuja soma é, no máximo, C, qual é a maior soma de elementos do subconjunto ? O problema de decisão é: existe algum subconjunto de objetos (elementos) cujo soma dos pesos é exatamente C ?

Problemas de decisão podem surgir quando não é natural a interpretação de "soluções" e "soluções propostas". Um problema de decisão é, abstratamente, alguma função de um conjunto de *inputs* para o conjunto {sim, não}.

Um problema pode ser resolvido em *tempo polinomial* se existe algum algoritmo que resolve o problema e que tem uma complexidade de tempo polinomial; nesse caso, diz-se que o problema pertence a *classe P*. *P* é definida somente para problemas de decisão, ou seja, *P* é a classe de problemas de decisão, limitada por tempo polinomial. São exemplos de problemas de classe P: a multiplicação de inteiros, cálculo de determinante, cálculo do máximo divisor comum.

Um algoritmo é considerado eficiente se existe um polinômio $p(n)$ tal que o algoritmo pode resolver qualquer caso de tamanho n em um tempo em $O(p(n))$. Tais algoritmos são ditos de tempo polinomial.

Dados dois algoritmos que requerem um tempo de $\theta(n^{lg\,lg\,n})$ e $\theta(n^{10})$, respectivamente, o primeiro é ineficiente conforme a definição, porque não é de tempo polinomial. A questão é distinguir que tipo de problema pode ser resolvido eficientemente dos que não podem.

Surge assim uma melhor definição: "P é a classe de problemas de decisão que podem ser resolvidos por um algoritmo de tempo polinomial". Por outro lado, existem problemas não pertencentes a P e são chamados intratáveis.

NP (*NP = nondeterministic polynomial time*) é a classe de problemas de decisão, para os quais uma dada solução proposta para um dado *input* pode ser verificado rapidamente (em tempo polinomial) para ver se ela é realmente uma solução, ou seja, se satisfaz todas as exigências do problema.

Uma definição formal de NP considera todos os problemas de decisão.

Um algoritmo não-determinístico tem duas fases:

1. A fase não-determinística. Um *string* de caracteres, *s*, totalmente arbitrário, é escrito começando em algum lugar indicado na memória. Cada vez que o algoritmo é executado, a escrita do *string* pode diferir.
2. A fase determinística. Um algoritmo determinístico inicia a execução. Além do *input* do problema de decisão, o algoritmo pode ler *s*, ou ele pode ignorar *s*. Eventualmente ele pára com um *output* de *sim* ou *não*, ou ele pode fazer uma infinidade de *loops* e nunca parar.

Normalmente, cada vez que se executa um algoritmo com o mesmo *input*, obtém-se o mesmo *output*. Isto não acontece com algoritmos não-determinísticos; para um *input x*, o *output* em uma execução pode diferir do *output* de outra porque pode depender de *s*.

NP é a classe de problemas de decisão para os quais há um algoritmo não-determinístico limitado por polinômio [BAA88].

NP-completo é o termo usado para descrever problemas de decisão que são os mais difíceis em NP no aspecto em que se há um algoritmo limitado por polinômio que resolve um problema NP-completo, então haverá um algoritmo limitado por polinômio resolvendo cada problema em NP. A definição formal de NP-completo usa reduções ou transformações de um problema para outro.

Qualquer problema que pode ser resolvido em tempo polinomial está automaticamente em NP e não se sabe como provar a existência de mesmo um único problema em NP que não pode ser resolvido em tempo polinomial, ainda assim conjectura-se que ele existe [BRA96].

Uma propriedade muito importante do NP é que ele contém 'propriedades mais rigorosas' tais que, se qualquer uma destas propriedades pode ser determinada em tempo polinomial, então toda propriedade em NP pode ser determinada em tempo polinomial [WEL90].

Para tornar isto mais preciso, diz-se que a propriedade π_1 é polinomialmente redutível a π_2 se existe alguma função f que está em P e que tem a propriedade. Dessa maneira x tem propriedade π_1 se e somente se $f(x)$ tem a propriedade π_2. Escreve-se como

$$\pi_1 \propto \pi_2.$$

Se $\pi_1 \propto \pi_2$ então a existência de um algoritmo com tempo polinomial para π_2 implica na existência de um algoritmo de tempo polinomial para π_1.

Provando: Supondo *A* como um algoritmo para π_2 que trabalha em um tempo *t(n)*. Se *x* é algum *input* para π_1 com $|x| = n$*(digamos)*, então transforma *x* para *f(x)* e aplica *A*. Depois a transformação trabalha em tempo polinomial, $|f(x)|$ é limitado em tamanho por algum polinômio *g(n)*. Então a transformação e *A* juntos trabalham em tempo limitado por algum polinômio.

Define-se um problema π como NP-difícil se existe alguma propriedade π_0 NP-completa tal que, se existe um algoritmo polinomial para π, então existe um algoritmo polinomial para π_0.

Conseqüências desta definição são as seguintes:

(1) Se existe algum algoritmo com tempo polinomial para algum problema NP-difícil, então NP = P.

(2) Se π_1 é NP-difícil e $\pi_1 \propto \pi_2$ então π_2 é NP-difícil.

(3) Qualquer propriedade NP-completa é NP-difícil.

3. O CRIPTO-SISTEMA BASEADO NO KNAPSACK

O cripto-sistema baseado no problema da mochila (knapsack) foi desenvolvido por Merkle e Hellman em 1978 [WEL90]. É um sistema de chave pública.

3.1. O PROBLEMA DO KNAPSACK

A forma como geralmente é apresentado o problema do *knapsack* consiste no seguinte: supondo que se tem uma mochila (*knapsack*) grande e nela deve-se acondicionar elementos para uma longa viagem numa região deserta, ocupando todo o espaço da mochila com elementos de valor máximo, ou seja, tem-se um número de itens, k, por exemplo, de volume v_i, $i = 0, ..., k-1$, para acomodar na mochila, que suporta um volume total V, e que se pode sempre acomodar itens sem desperdício de espaço. Ter-se-á a maior carga possivel formada por um subconjunto de k itens que preenchem exatamente a mochila [KOB94].

Em outras palavras: é dado um conjunto de n objetos e uma mochila. Cada objeto $i = 1, 2, ..., n$, tem um peso positivo w_i e um valor positivo v_i. Na mochila podem ser carregados objetos com um peso máximo W. O problema consiste em colocar na mochila objetos com valor máximo, considerando a capacidade de peso que suporta.

Há versões deste problema referentes à possibilidade de se quebrar (fracionar) ou não, os objetos. Admitir-se-á neste primeiro caso a hipótese de que os objetos podem ser quebrados, tomando-se, para encher a mochila com sua capacidade máxima de peso, pedaços de objetos. Desta maneira, pode-se levar uma fração x_i do objeto i, onde $0 \leq x_i \leq 1$, contribuindo com um total de peso $x_i w_i$ e um valor de $x_i v_i$ no total da carga.

Em símbolos matemáticos o problema consiste em

$$maximizar \sum_{i=1}^{n} x_i v_i \ \ sujeito\ a \ \sum_{i=1}^{n} x_i w_i \leq W$$

onde $v_i > 0$, $w_i > 0$ e $0 \leq x_i \leq 1$ para $1 \leq i \leq n$.

Existem diferentes táticas para resolver o problema da mochila mas somente uma dá certo. O objetivo é carregar o maior valor possível considerando o peso suportado pela mochila.

Verifica-se que, *sendo permitido tomar frações de objetos para encher a mochila, o melhor resultado é obtido quando se considera a razão entre o valor e o peso.* A demonstração encontra-se na página 204 de [BRA96].

A solução apresentada, desta maneira, para o problema da mochila, resulta em um algoritmo, chamado 'avarento'. A estratégia geral consiste em escolher o objeto em uma ordem adequada, colocando na mochila a maior fração possível, parando apenas quando a mochila está completamente cheia.

Na segunda hipótese do problema da mochila, não se admite quebrar o objeto, isto é, ou se carrega o objeto todo ou não se carrega o objeto. Então, supondo-se $i = 1, 2, \ldots, n$ e cada objeto i com um peso positivo w_i e um valor positivo v_i, o problema consiste em carregar na mochila (*knapsack*) um peso que não pode exceder a W mas com um máximo em valor; assim x_i será igual a 0 quando o objeto i não é carregado e 1 em caso contrário. Em simbologia matemática o problema pode ser traduzido da seguinte maneira:

$$maximizar \sum_{i=1}^{n} x_i v_i \quad sujeito\ a \quad \sum_{i=1}^{n} x_i w_i \leq W$$

onde $v_i > 0$, $w_i > 0$ e $x_i \in \{0, 1\}$ para $1 \leq i \leq n$.

Visando à demonstração da dificuldade deste problema, vamos considerar somente três objetos, o primeiro com peso 6 e valor 8 e os outros dois com peso 5 e valor 5 cada. Se o limite da mochila é 10 ($W = 10$), a solução ótima é carregar os dois objetos de menor peso mas totalizando o valor 10. Por outro lado, pelo algoritmo avarento, iniciar-se-ia escolhendo o objeto de peso 6, porque este é o de maior valor, considerando a razão valor/peso; mas assim só pode ser carregado um objeto, o de peso 6, cujo valor é 8. Observe-se que, por não poder fracionar o objeto, não é possível carregar mais nenhum dos outros dois objetos, porque, como o seu peso é 5, ultrapassaria o suportado pela mochila ($6 + 5 > 10$).

Para resolver o problema, constrói-se uma tabela $V[1...n, 0...W]$, com uma linha para cada objeto e uma coluna para cada peso de 0 a W. Na tabela, $V[i, j]$ é o valor máximo de objetos que podem ser carregados se o limite de peso é j, $0 \leq j \leq W$, e se são considerados somente objetos numerados de 1 a i, $1 \leq i \leq n$. Para encontrar a solução deve ser aplicado o princípio da otimização. Preenche-se linha por linha ou coluna por coluna da tabela. De um modo geral $V[i, j]$

é o maior valor entre $V[i - 1, j]$ e $V[i - 1, j - w_i] + v_i$. A primeira opção significa não acrescentar o objeto i e a segunda, escolher o objeto i, aumentando o valor em v_i e reduzindo o peso em w_i. De um modo geral usa-se a regra

$$V[i, j] = max(V[i - 1, j], V[i - 1, j - w_i] + v_i).$$

Como entrada para os valores fora dos limites da tabela define-se $V[0, j] = 0$ quando $j \geq 0$ e $V[i, j] = -\infty$ para todo i quando $j < 0$.

Como exemplo, vamos considerar 5 objetos com os pesos respectivamente 1, 2, 5, 6 e 7 e cujos valores são 1, 6, 18, 22 e 28. Conseqüentemente as razões valor/peso serão 1,00; 3,00; 3,60; 3,67 e 4,00. Supondo que o máximo de peso que pode ser carregado é 11, tem-se a tabela:

Peso limite:	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
$w_1 = 1, v_1 = 1$	*0*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*
$w_2 = 2, v_2 = 6$	*0*	*1*	*6*	*7*	*7*	*7*	*7*	*7*	*7*	*7*	*7*	*7*
$w_3 = 5, v_3 = 18$	*0*	*1*	*6*	*7*	*7*	*18*	*19*	*24*	*25*	*25*	*25*	*25*
$w_4 = 6, v_4 = 22$	*0*	*1*	*6*	*7*	*7*	*18*	*22*	*24*	*28*	*29*	*29*	*40*
$w_5 = 7, v_5 = 28$	*0*	*1*	*6*	*7*	*7*	*18*	*22*	*28*	*29*	*34*	*35*	*40*

Tabela 5 - Composição do problema da mochila.

A tabela mostra toda a composição do problema e ainda que o valor máximo está limitado em 40. Começando a examinar a tabela por $V[5, 11]$: como $V[5, 11] = V[4, 11]$ mas $V[5, 11] \neq V[4, 11 - w_5] + v_5$, para obter a otimização não pode ser incluído o objeto 5. Continuando, como $V[4, 11] \neq V[3, 11]$ *mas* $V[4, 11] = V[3, 11 - w_4] + v_4$, o objeto 4 deve ser incluído. Continuando, como $V[3, 5] \neq V[2, 5]$ mas $V[3, 5] = V[2, 5 - w_3] + v_3$, então o objeto 3 deve ser incluído. Continuando assim, vê-se que $V[2, 0] = V[1, 0]$ e $V[1, 0] = V[0, 0]$. Assim, para encontrar o resultado ótimo não podem ser incluídos nem o objeto 2 e nem o objeto 1. Logo, a única solução ótima consiste em carregar os objetos 3 e 4, cuja soma dos pesos completa 11 e soma dos valores é 40.

Analisando sob outra hipótese: se carregasse o objeto 5 cujo peso é 7, já não poderia mais carregar os objetos 4 ou 3, porque passaria do limite permitido que é 11; mas carregando ainda os objetos 2 e 1, o que é possível porque completaria uma carga de peso 10, o valor ficaria restrito a $28 + 6 + 1 = 35$.

Salienta-se que a técnica utilizada na resolução do problema anterior é a da *programação dinâmica*, isto é, uma vez resolvida uma subinstância do problema este resultado é guardado para uma possível reutilização futura, evitando assim a duplicidade de cálculos.

A análise do algoritmo de programação dinâmica é direta, encontrando-se um tempo em $\theta(nW)$ para construir a tabela e a composição da carga ótima pode ser determinada em um tempo em $O(n + W)$.

3.2. SISTEMA DE CRIPTOGRAFIA BASEADO NO KNAPSACK

Retornando ao cripto-sistema de chave pública baseado no problema da mochila propriamente dito, são necessárias algumas definições.

Definição 3.2.1.: O problema knapsack pode passar a ter o seguinte enunciado: "dado um conjunto $\{v_i\}$ de k inteiros positivos e um inteiro V, encontrar um inteiro $n = \varepsilon_{k-1}\varepsilon_{k-2}\ldots\varepsilon_1\varepsilon_0$ de k-bits, onde $\varepsilon_i \in \{0, 1\}$ é cada dígito binário de n, tal que $\sum_{i=0}^{k-1} \varepsilon_i v_i = V$, se um tal n existe".

Tal problema pode ter nenhuma, uma ou várias soluções, dependendo da k-upla e do inteiro V.

Como ilustração para verificar a sua dificuldade, vamos considerar o seguinte exemplo: o inteiro 516 não tem representação como soma de inteiros do conjunto

$\{14, 28, 56, 82, 90, 132, 197, 284, 341, 455\}$.

E o inteiro 515, por outro lado, tem 3 representações distintas:

$$515 = \begin{cases} 341 + 132 + 28 + 14 \\ 197 + 132 + 90 + 82 + 14 \\ 341 + 90 + 56 + 28 \end{cases}$$

O sistema de chave pública Merkle-Hellman é baseado na dificuldade deste problema, já que é possível mostrar que é um problema NP-completo. Não existe um algoritmo que resolve um problema *knapsack* arbitrário em tempo polinomial em k e $\log B$, onde B é um limite do tamanho de V e v_i [KOB94].

Definição 3.2.2.: Uma seqüência $v_1, \ldots, v_n$ é supercrescente se para cada k $(1 \leq k \leq n\text{-}1)$

$$v_{k+1} > \sum_{i=1}^{k} v_i$$

Para resolver o problema da mochila com uma seqüência supercrescente, inicia-se com o maior v_i até encontrar o primeiro que seja menor ou igual a V. Inclui-se este i no subconjunto I, onde $I \subset \{1,\ldots,k\}$ tal que $\sum_{i\in I} v_i = V$, isto é, $\varepsilon_i = 1$, substitui-se V por V - v_i e continua-se procurando na seqüência um elemento que seja menor ou igual à diferença. Seguindo assim obtém-se um subconjunto de $\{v_i\}$ cuja soma dos elementos é V ou esgotamos todos os elementos de $\{v_i\}$ sem que $V - \sum_{i\in I} v_i = 0$, e neste caso não há solução.

Para resolver o problema da mochila tem-se o seguinte pseudo-algoritmo, onde v_i é uma k-upla supercrescente e V um inteiro:

Algoritmo 4 - Resolução do problema knapsack

knapsack (k, V)

1. Faça $W = V$ e $j = k$.
2. Inicie com ε_{j-1} e diminuindo o índice de ε , escolha todos os ε_i iguais a 0 até encontrar o primeiro i - chamado i_0 - tal que $v_{i_0} \leq W$. Faça $\varepsilon_{i_0} = 1$. Se todos os ε_i forem iguais a 0, não há solução e o algoritmo termina.
3. Substitua W por $W - v_{i_0}$, faça $j = i_0$, e, se $W > 0$ retorne à etapa 2.
4. Se $W = 0$, está concluído. Se $W > 0$ e todos os restantes v_i são maiores do que W, então não há solução $n = (\varepsilon_{k-1}\ldots\varepsilon_0)_2$ para o problema.

Note-se que a solução, se existe alguma, é única, porque, ordenadamente, todos os elementos foram testados.

Exemplificando, vamos considerar a seqüência supercrescente, a 5-upla v_i = (2, 3, 7, 15, 31) e $V = 24$. Percorrendo a 5-upla da direita para a esquerda, obtém-se $\varepsilon_4 = 0$ porque $31 > 24$, $\varepsilon_3 = 1$ (por isso substitui-se 24 por 24 - 15 = 9), $\varepsilon_2 = 1$ (substitui-se 9 por 9 - 7 = 2), $\varepsilon_1 = 0$, $\varepsilon_0 = 1$. Então $n = (01101)_2 = 13$.

Usando somente as letras do nosso alfabeto, faz-se corresponder a cada letra um inteiro de 5-bits: de $A = 0 = (00000)_2$ até $Z = 22 = (10110)_2$. São necessários 5-bits porque a representação do número 22 na forma binária exige tantos bits. Depois, para esse caso, cada usuário escolhe uma k-upla $(v_1,\ldots,v_k)$ supercrescente, com $k = 5$, e um inteiro N que é maior

do que $\sum_{i=1}^{k} v_i$ e um inteiro *a* primo com *N*, $0 < a < N$. Isto pode ser feito de forma randômica, usando o código **relpri**, que se encontra no apêndice, junto com vários outros códigos construídos tendo por base o Maple. Por exemplo, pode-se escolher uma seqüência arbitrária de $k + 1$ inteiros positivos z_i, $i = 0, 1, ..., k$. Fazendo $v_0 = z_0$, $v_i = z_i + v_{i-1} + v_{i-2} + ... + v_0$ para $i = 1, ..., k - 1$ e $N = z_k + \sum_{i=0}^{k-1} v_i$. Então pode-se escolher randomicamente um positivo $a_0 < N$, tomando como *a* o primeiro inteiro maior ou igual a a_0 que é relativamente primo com N. Depois calcula-se $b \equiv a^{-1} \bmod N$, ou seja, *b* é o menor inteiro positivo tal que $ab \equiv 1 \bmod N$, e também calcula-se a k-upla (w_i) definida por $w_i = av_i \bmod N$, isto é, w_i é o menor resíduo positivo de av_i módulo *N*. O usuário mantém os números v_i, *N*, *a* e *b* todos secretos, mas publica a k-upla de w_i. Isto é, a chave de codificação é $K_E = \{w_0, ..., w_{k-1}\}$. A chave de decodificação é $K_D = (b, N)$, o que junto com a chave de codificação permite determinar $\{v_0, ..., v_{k-1}\}$.

Para enviar uma mensagem $P = (\varepsilon_{k-1}\varepsilon_{k-2}...\varepsilon_1\varepsilon_0)_2$ a um usuário com a chave de codificação (w_i), calcula-se $C = f(P) = \sum_{i=0}^{k-1} \varepsilon_i w_i$ e transmite-se esse inteiro. Para ler a mensagem, o usuário primeiro acha o menor resíduo positivo *V* de *bC módulo N*, uma vez que $bC \equiv \sum \varepsilon_i b w_i \equiv \sum \varepsilon_i v_i \bmod N$, porque $bw_i \equiv bav_i \equiv v_i \bmod N$, segue que $V = \sum \varepsilon_i v_i$. (Aqui usou-se o fato de que ambos V < N e $\sum \varepsilon_i v_i \leq \sum v_i < N$ para converter a congruência de módulo N em igualdade). É então possível usar o algoritmo acima em problemas de mochila supercrescente para encontrar a solução única $(\varepsilon_{k-1}...\varepsilon_0)_2 = P$ do problema determinando um subconjunto de (v_i) cuja soma é exatamente *V*. Nesse sentido recupera-se a mensagem *P*.

Note-se que um bisbilhoteiro que conhece somente (w_i) é acareado com o problema da mochila $C = \sum \varepsilon_i w_i$, que não é um problema supercrescente, porque a propriedade supercrescente da k-upla de v_i é destruída quando v_i é substituído pelo menor resíduo positivo de av_i módulo N. Assim o algoritmo acima não pode ser usado, e, à primeira vista, a pessoa não autorizada parece estar frente a um problema muito mais difícil.

Aplicando-se em um exemplo numérico, onde a 5-upla é a seqüência secreta supercrescente *(2, 3, 7, 15, 31), a chave privada* escolhida é $N = 61$ e $a = 17$. Convém lembrar que *N* deve ser maior do que a soma dos elementos da 5-upla e *a* deve ser relativamente primo com *N*.

Transformando-se em *chave pública* por $T(v_i) = av_i \bmod N$ $(1 \leq i \leq k)$, tem-se que, se

$v_i = (2, 3, 7, 15, 31)$, então $T(v_i) = (34, 51, 58, 11, 39)$.

Desejando-se enviar a mensagem $M = SOL$ (usando o alfabeto português com 23 letras), codifica-se a mensagem M, usando a correspondência entre as letras e os números, então ter-se-á

$A = 0 = (00000)_2$, $B = 1 = (00001)_2$, ..., $Z = 22 = (10110)_2$.

Logo $S = 17 = (10001)_2 \rightarrow 34 + 39 = 73$,

$O = 13 = (01101)_2 \rightarrow 51 + 58 + 39 = 148$,

$L = 10 = (01010)_2 \rightarrow 51 + 11 = 62$.

Na mensagem sob forma binária, o dígito 1 significa, na ordem da esquerda para a direita, o elemento da seqüência $T(v_i)$, e o 0, a sua ausência. Fica aqui caracterizado o problema da mochila.

A mensagem codificada transmitida é $C = (73, 148, 62)$.

Na decodificação, pode ser usado o algoritmo de Euclides estendido, **mdceuinv**, para calcular $b \equiv a^{-1} \bmod N$. Encontra-se $b = 18$.

Então, $73 \rightarrow 73*18 \bmod 61 = 33 = 2 + 31 = 10001$

$148 \rightarrow 148*18 \bmod 61 = 41 = 3 + 7 + 31 = 01101$

$62 \rightarrow 62*18 \bmod 61 = 18 = 3 + 15 = 01010$

Explicando: depois de calculado o módulo, este número é decomposto em soma dos elementos da 5-upla escolhida e os uns e zeros na decodificação correspondem, em seqüência, à presença ou não destes elementos somados na 5-upla.

Retornando os k-bits para a base decimal e a correspondente letra do alfabeto, tem-se

$(10001)_2 = 17 = S$,

$(01101)_2 = 13 = O$,

$(01010)_2 = 10 = L$.

Resumindo, os itens para a codificação são:

1. a mensagem a ser enviada é codificada na forma binária *m;*
2. o usuário mantém os números v_i, *a, N* e *b* secretos;
3. as chaves públicas são uma coleção de *k*-uplas $T(v_i)$ de inteiros positivos;
4. a chave de codificação é $K_E = \{T(v_i)\}$, onde $T(v_i) = av_i \bmod N$;
5. a mensagem binária *m* a ser enviada é quebrada em *n* blocos tais que $m = m_1 \ldots m_n$, onde cada m_i é uma *k*-upla de zeros e uns;
6. para cada *j* $(1 \leq j \leq n)$, $c_j = \sum_{i=1}^{k} M_i v_i$ onde $m_j = (M_1, \ldots, M_k)$;
7. transmite-se a seqüência $c_1, \ldots, c_t$ de inteiros positivos como a mensagem codificada.

Algumas considerações sobre a decodificação:

1. a chave de decodificação é $K_D = (b, N)$;
2. para decifrar a mensagem de $c_1, \ldots, c_t$ usando a chave pública $(v_1, \ldots, v_k)$, precisa-se resolver *t* diferentes problemas NP difíceis para cada c_i.

Concluindo, cabem alguns comentários sobre o sistema *knapsack.* Em 1982, Shamir produziu uma análise que mostrou a vulnerabilidade do sistema Merkle-Hellman. Ele provou que 'quase todos' os casos podem ser resolvidas em tempo polinomial. Isto foi seguido por Adleman (1983) que deu forte evidência de que a versão iterativa[2] do esquema Merkle-Hellman pode ser quebrada [KOB94 e WEL90].

[2] Iterativa: execução, repetição de uma ou mais instruções. As instruções executadas dessa forma são consideradas como estando em um *loop*.

4. O CRIPTO-SISTEMA RSA

O sistema de criptografia RSA surgiu em 1978 [COU97]. O nome RSA deve-se ao sobrenome dos três autores Ron **R**ivert, Adi **S**hamir e Leonard **A**dleman. A função uma-via usada pelo sistema consiste em que, dados dois números primos grandes, é fácil efetuar o produto entre eles, mas é muito difícil, tendo o produto, fatorá-lo para encontrar os dois números primos

4.1. DEFINIÇÃO DO SISTEMA

O método RSA é um sistema de chave pública. Tem a sua segurança baseada na convicção de que não é fácil fatorar um número que é o produto de dois números primos grandes. Dados dois números primos grandes p e q, multiplicá-los para encontrar n é muito mais fácil do que o contrário: dado n, encontrar os dois números primos p e q. Está assim presente a propriedade "*alçapão*" ou "*uma-via*".

A dificuldade maior deste sistema está em gerar números primos grandes, com mais de cem dígitos. Será estudado logo a seguir.

4.2. A MATEMÁTICA DO RSA - PRIMALIDADE

No sistema RSA, números primos e testes de primalidade são conceitos básicos. Apesar de já ter definido e citado propriedades na página 10, retornar-se-á a elas, mas principalmente à questão da primalidade de um número.

Os chineses, na antiguidade, já sabiam que se p é primo então p divide $2^p - 2$.

Números da forma $M(n) = 2^n - 1$ são chamados números de Mersenne (Marin Mersenne foi um frade e matemático amador do século XVII que foi correspondente de muitos matemáticos famosos da época, entre eles Fermat, Descartes e Pascal). Determinar quais dos números de Mersenne são primos é uma questão herdada da Grécia e teve grande repercussão por muito tempo. Segundo Mersenne, quando $n = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127$ e 257; $M(n) = 2^n - 1$ será um número primo. Isto não é bem verdade. Em anos posteriores matemáticos descobriram que M(61), M(89) e M(107) também são primos, mas que M(67) e M(257) são compostos.

Alguns dos maiores números primos conhecidos são números de Mersenne, por exemplo M(1398269), um número com 420921 dígitos, é primo.

Segundo a Revista do Professor de Matemática de número 41, do 3° quadrimestre de 1999, página 36, nota da redação: "Acredita-se que existem infinitos primos de Mersenne, mas, até agora, só foram encontrados 38 deles. O 38° é $2^{6972593} - 1$, descoberto em junho/99. O tamanho desse número desafia a imaginação: tem 2 098 960 dígitos e é o maior número primo conhecido. Primos de Mersenne constituem uma área de pesquisa matemática ainda em franca atividade. Desde 1995 há um endereço na Internet *(http:www.mersenne.org/prime.htm)* com informações sobre esses primos e *software* específico para a busca de novos primos de Mersenne. Dezenas de especialistas e milhares de amadores têm participado desse esforço conjunto que levou à descoberta dos quatro últimos primos de Mersenne".

Dos testes de primalidade que serão mencionados a seguir, o Crivo de Eratóstenes e o Teorema de Wilson são exemplos de testes *determinísticos*, porque através de sua aplicação determina-se os números primos. Os demais, o teste baseado no teorema de Fermat, o Btest, o teste de Miller-Rabin, são testes *probabilísticos*; respondem com alguma probabilidade de que o número considerado é primo. Interessa, neste estudo, que a probabilidade de o número ser primo, deve ser muito grande, quase certeza. Se a resposta de um teste probabilístico é de que o número é composto, então é certo de que o número é composto, mas quando a resposta é de que o número é primo, surge ainda a dúvida: o número pode ser composto ou primo.

Iniciemos o estudo dos testes de primalidade pelo conhecido como "Crivo de Eratóstenes". É o mais antigo método para encontrar números primos de que se tem notícia. Foi desenvolvido pelo sábio Eratóstenes, matemático grego, da escola de Alexandria, que nasceu 284 anos antes de Cristo. Consiste em escrever os números inteiros positivos em ordem crescente a começar com o 2 (porque o número 1 não é primo nem composto) até n, formando uma tabela. A seguir eliminam-se, com exceção dele mesmo, 2, todos os múltiplos de 2; depois os múltiplos de 3; os múltiplos de 4 já foram eliminados quando se eliminaram os múltiplos de 2; depois os múltiplos de 5; e assim por diante, restando assim somente os números primos: {2, 3, 5, 7, 11, 13, ...}. Como o objetivo é encontrar números primos grandes, este método não serve porque é muito lento.

O teorema de Wilson estabelece que "para todo inteiro $n > 1$, n divide $(n - 1)! + 1$ se e somente se n é primo". Mesmo para testar se um número é primo ou composto, a aplicação do teorema de Wilson é de um custo computacional bastante alto porque precisa calcular um fatorial. Implementou-se, baseado neste teorema o código **Wilson** para, aleatoriamente, gerar um número primo. Apesar de introduzir alguma melhoria, como eliminar do teste quando o número é par, o

algoritmo é muito lento, mesmo para números de poucos dígitos. O código, em Maple, encontra-se no apêndice.

Para a demonstração do teorema de Wilson, considerar-se-á um enunciado correspondente ao anterior, mas nos seguintes termos: "se n é primo então $(n - 1)! \equiv -1 \bmod n$". A recíproca deste teorema é verdadeira isto é, $(n - 1)! \equiv -1 \bmod n$ implica que n seja primo, porque, se não for assim, $(n - 1)!$ tem um fator comum com n. Portanto o teorema dá uma condição necessária e suficiente para que n seja primo.

Na demonstração do teorema, tomar-se-á o conjunto de números

$$1, 2, 3, \ldots, n - 2, n - 1$$

onde o correspondente de 2 é o número a_2 do conjunto, para os quais $2\,a_2 \equiv 1 \bmod n$. Sabe-se que existe tal número porque 2 é primo com n. Reciprocamente o correspondente de a_2 é 2. Depois associa-se 3 com seu correspondente, e assim sucessivamente. Se um número b for seu próprio correspondente ter-se-á que $b^2 \equiv 1 \bmod n$; isto é, $(b - 1)(b + 1) \equiv 0 \bmod n$, o que implica que b é igual a 1 ou a $n - 1$. Omitindo esses dois números, ter-se-á

$$(n - 2)! \equiv (2a_2)\,(..)\,(..)\ldots(..) \pmod n$$

onde cada parênteses contém dois números cujo produto é congruente com $1 \bmod n$.

Ter-se-á, portanto,

$$(n - 1)! \equiv 1\,(1)\,(1)\ldots(1)\,(n - 1) \equiv -1 \bmod n. \qquad \text{c.q.d.}$$

Um exemplo pode ajudar na compreensão da demonstração. Seja $n = 11$. Então $2x \equiv 1 \bmod 11$ tem como solução 6 e, conseqüentemente, 2 e 6 formam um par. O correspondente de 3 é 4 e assim por diante. Pode-se então escrever

$$10 ! \equiv 1\,(2*6)\,(3*4)\,(5*9)\,(7*8)\,(10) \equiv 1*1*1*1*1*(-1) \bmod 11.$$

Na procura de um teste mais rápido, vamos examinar o "Pequeno Teorema de Fermat" (Pierre de Fermat: 1601-1658) cujo enunciado é: - **Seja p um número primo e a um número inteiro, então**

(i) p divide $a^p - a$ (generalização do conhecimento chinês antigo)

ou usando módulo

(ii) $a^p \equiv a \bmod p$ **(que pode ser escrito** $a^{p-1} \equiv 1 \bmod p$ **)** [KOB94].

No apêndice encontra-se o código **Fermat,** elaborado a partir do pequeno teorema de Fermat, visando testar a primalidade de um número p considerando um a aleatório.

Para a demonstração do teorema de Fermat usar-se-á um lema, que será mostrado primeiro.

Lema 4.1.: Seja p um número primo e a e b inteiros, então $(a+b)^p \equiv a^p + b^p \pmod{p}$.

Prova: Do desenvolvimento do binômio de Newton , tem-se

$$(a+b)^p = a^p + b^p + \sum_{i=1}^{p-1} \binom{p}{i} a^{p-i} b^i .$$

Basta provar que

$$\sum_{i=1}^{p-1} \binom{p}{i} a^{p-i} b^i \equiv 0 \bmod p.$$

Considerando o número

$$\binom{p}{i} = \frac{p(p-1)\ldots(p-i+1)}{i!}$$

que se sabe ser inteiro, mas para tanto o denominador deve ser totalmente simplificado com termos do numerador. Supondo $1 \leq i \leq p-1$ então o denominador não pode ter como algum de seus fatores primos o número p. Logo o fator p que aparece no numerador não pode ser cancelado porque é primo. Então $\binom{p}{i}$ é múltiplo de p e $\sum_{i=1}^{p-1} \binom{p}{i} a^{p-i} b^i \equiv 0 \bmod p.$

A prova do teorema de Fermat será feita por indução finita.

Tem-se como proposição $P(n)$, $n^p \equiv n \bmod p$

então $P(1)$ será $1^p \equiv 1 \bmod p$ o que é verdadeiro.

Usando o Lema, tem-se que $(n+1)^p \equiv n^p + 1^p \equiv (n^p + 1) \bmod p$

Pela hipótese de indução $n^p \equiv n \bmod p$ então $(n+1)^p \equiv n^p + 1 \equiv (n+1) \bmod p$

Supondo o a negativo, tem-se $-a$ positivo e substituindo $(-a)^p \equiv -a \bmod p$

Sendo p ímpar $(-a)^p \equiv -a^p$, substituindo na congruência anterior $-a^p \equiv -a \bmod p$ ou

$a^p \equiv a \bmod p$ c.q.d.

É importante ressaltar que a recíproca deste teorema de Fermat não é verdadeira. Analisando para $a = 4$ e $p = 15$, tem-se então o menor contra-exemplo não-trivial que é $4^{14} \equiv 1 \bmod 15$. Apesar de 15 ser composto, pela aplicação de Fermat, encontrou-se também resultado 1. Note que pelo teorema de Fermat está-se supondo p como número primo.

Entretanto o objetivo é testar algum número p sobre a sua primalidade. Logo, pelo teorema de Fermat, se $a^{p-1} \neq 1 \bmod p$, então o número p é composto, mas em caso contrário o número p pode ser primo ou composto, porque o número a é randômico e há a possibilidade de sortear para a um número chamado "falsa testemunha" porque ele testemunha falsamente a primalidade de p.

Números que são falsa testemunha são raros. Embora somente 5 dentre os 332 números compostos ímpares menores do que 1000 não tenham testemunha falsa, mais da metade destes têm somente duas falsas testemunhas e menos do que 16% destes têm mais do que 15. Além disso, há somente 4490 falsas testemunhas para todos os números compostos tomados ao mesmo tempo. Comparando com 172878, que é o total de números candidatos a testemunha que existem para o mesmo conjunto de números, a probabilidade média de erro do teste de Fermat para números ímpares compostos menores do que 1000 é menor de 3,3% [BRA96].

Um inteiro a, tal que $2 \leq a \leq p - 2$ e $a^{p-1} \equiv 1 \bmod p$ é chamado falsa testemunha da primalidade de p se p é, na verdade, composto. [BRA96]

Como exemplo: $a = 158; p = 289;$

$$158^{288} = 1 \bmod 289$$

Sabe-se que 289 é composto porque $289 = 17.17$. Logo o número 158 é falsa testemunha da primalidade de 289.

Fazendo $2 \leq a \leq 287$, verifica-se que o número 289 tem 14 falsas testemunhas que são: $38, 40, 65, 75, 110, 131, 134, 155, 158, 179, 214, 224, 249$ e 251.

Este exemplo demonstra a incerteza ao se usar o Pequeno Teorema de Fermat para testar a primalidade de um número. Mas existem casos ainda piores. Mais um número que tem muitas falsas testemunhas: 561 admite 318 falsas testemunhas.

Números compostos ímpares que satisfazem $a^p \equiv a \bmod p$, para todo a tal que $1 < a < p - 1$, são chamados números de Carmichael. O número 561 é o menor número de Carmichael. Outros números de Carmichael são: 1105, 1729, 2465, 2821, 6601, 9341, 17081 278545,...

Em geral, um algoritmo baseado no teorema de Fermat tem boa probabilidade de funcionar corretamente. O problema é que ele pode falhar em alguns casos.

Um caso convincente acontece com o número de 15 dígitos, 651693055693681. O algoritmo informa ser número primo com uma probabilidade maior de 99,9965%, apesar do fato de ele ser composto.[BRA96]

Examinando o número de 15 dígitos citado anteriormente, verificou-se que não se trata de número de Fermat e nem de Mersenne. Tem 3 fatores primos: 87517, 102103 e 72931. Logo tem 6 divisores não-triviais. Apoiado no Teorema de Korselt [COU97] que diz: "Um inteiro positivo ímpar n é um número de Carmichael se, e somente se, cada fator primo p de n satisfaz as duas condições seguintes:

(1) p^2 não divide n;

(2) $p - 1$ divide $n - 1$," [3]

descobriu-se que é um número de Carmichael.

No apêndice encontra-se o código **Carmi1** elaborado em Maple, baseado no Teorema de Korselt, que testa se determinado número é ou não de Carmichael. O código tem como entrada o número a ser testado e como saída, a informação se o número é ou não é de Carmichael. O código fatora o número, usando a primitiva (comando) *ifactor* do Maple e depois examina cada fator primo dentro das duas condições estabelecidas pelo teorema de Korselt.

Conclusão: não é seguro utilizar, dessa maneira, o pequeno teorema de Fermat para testar a primalidade de um número. Quando retorna *composto*, o número é realmente *composto*, mas quando retorna *primo*, o número pode ser *primo ou composto*, devido à *falsa testemunha*.

Um exemplo, tido como anedótico, foi dado pelo próprio Fermat ao formular a conjectura de que todos os números da forma $F_n = 2^{2^n} + 1$ são primos para todo n. Estes números são chamados números de Fermat. De fato, $F_0 = 3$, $F_1 = 5$, $F_2 = 17$, $F_3 = 257$ e $F_4 = 65537$ são primos, mas $F_5 = 4294967297$ é composto. Que ironia ! O menor fator de 4294967297 é 641.

Como a grande dificuldade do bom funcionamento do pequeno teorema de Fermat são as falsas testemunhas, o teste de primalidade de Miller-Rabin resolve, em grande parte, esse problema. Modificando levemente o Teorema de Fermat, o teste determina se a base escolhida é ou não é falsa testemunha, com grande probabilidade de acerto. As bases que, mesmo após o teste de primalidade de Miller-Rabin, atestam que um número composto é primo, são chamadas falsas testemunhas fortes. Felizmente as falsas testemunhas fortes são muito poucas.

Eis o teste:

---

[3] A demonstração do Teorema de Korselt pode ser encontrada na página 109 de [COU 97].

Seja n um número ímpar maior do que 4, e sejam s e t dois inteiros tais que $n-1 \equiv 2^s t$ onde t é ímpar. Seja $B(n)$ o conjunto de inteiros definido da seguinte maneira: a $\in$ B(n) se e somente se $2 \leq a \leq n-2$ e

- $a^t \equiv 1 \bmod n$ **ou**
- **existe algum inteiro i, $0 \leq i < s$, tal que** $a^{2^i t} \bmod n \equiv n-1$.

Como n é ímpar, $n-1$ é par. Fatora-se o expoente $n-1$ na maior potência de 2 possível.

O teste consiste em calcular a seqüência de potências módulo n:

$$a^t, a^{2t}, \ldots, a^{2^{s-1}t}, a^{2^s t}.$$

Se n for primo, então pelo menos uma das potências é congruente a 1. De fato, pelo Teorema de Fermat sabe-se que

$$a^{2^s t} \equiv a^{n-1} \equiv 1 \bmod n.$$

Sendo j o menor expoente tal que $a^{2^j t} \equiv 1 \bmod n$, $0 \leq j \leq s$, então se $j \geq 1$ pode-se decompor em produtos notáveis:

$$a^{2^j t} - 1 = (a^{2^{j-1} t} - 1)(a^{2^{j-1} t} + 1).$$

Como n é primo e divide $a^{2^j t} - 1$, então ou n divide $a^{2^{j-1} t} - 1$ ou n divide $a^{2^{j-1} t} + 1$.

Mas j é o menor expoente tal que $a^{2^j t} - 1$ é divisível por n. Logo $a^{2^{j-1} t} - 1$ não é divisível por n, restando $a^{2^{j-1} t} + 1$, ou seja $a^{2^{j-1} t} \equiv -1 \bmod n$.

Conclui-se que se n é primo, então uma das potências da seqüência

$$a^t, a^{2t}, \ldots, a^{2^{s-1} t}$$

tem que ser congruente a $-1 \bmod n$.

Está-se supondo $j \geq 1$. Se $j = 0$ então $a^t \equiv 1 \bmod n$. A esta congruência não se pode aplicar produto notável porque t é ímpar.

Conclusão: se n é primo então uma das potências da seqüência é congruente a

- -1 *ou*
- $a^t \equiv 1 \bmod n$.

A cada chamada do teste de Miller-Rabin, de uma base a para um número n, tem-se como resposta *verdadeiro* se e somente se $a \in B(n)$. Ou seja, sendo n composto a é falsa testemunha da primalidade de n. E, claro, se a é falsa testemunha da primalidade de n, deve-se procurar outro a que não seja falsa testemunha.

Essa extensão do teorema de Fermat mostra que $a \in B(n)$ para todo $2 \leq a \leq n-2$ quando n é primo.

Se n é composto e $a \in B(n)$, então se diz que n é pseudoprimo forte na base a ou que a é falsa testemunha forte da primalidade de n.

Sob forma de pseudo-algoritmo, o teste de Miller-Rabin, baseado na extensão do teorema de Fermat, pode ser o seguinte, sendo a a testemunha e n o número inteiro ímpar:

Algoritmo 5 - Teste de primalidade

MilRab (a, n)

1. Inicialização.

   $s \leftarrow 0; t \leftarrow n - 1$

2. Repete

   $s \leftarrow s + 1; t \leftarrow t : 2$

   até $t \, mod 2 = 1$

3. Faça

   $x \leftarrow a^t \bmod n$

4. Se $x = 1$ ou $x = n - 1$ retorna (verdadeiro)
5. Para i = 1 : s - 1 faça

   $x \leftarrow x^2 \bmod n$

6. Se $x = n - 1$ retorna (verdadeiro)
7. Se nada do anterior acontecer retorna (falso).

Como foi visto no exemplo anterior, 158 é falsa testemunha da primalidade de 289 e é uma falsa testemunha forte. Falsas testemunhas fortes são muito mais raras do que falsas testemunhas. Usando Fermat, por exemplo, 4 é falsa testemunha da primalidade de 15, porque $4^{14} \bmod 15 \equiv 1$; mas não é forte falsa testemunha porque $4^7 \bmod 15 \equiv 4$ (15 = 2*7+1; então s = 1 e t = 7). O número 561, mesmo admitindo 318 falsas testemunhas, somente 8 destas são fortes falsas testemunhas e são os números: 50, 101, 103, 256, 305, 458, 460 e 511.

O código que determina se $a \in B(n)$ implementado no Maple é o **Btest**. Encontra-se no apêndice.

Surge assim a idéia de que repetindo várias vezes o teste de primalidade de Miler-Rabin, ter-se-á praticamente a certeza de que o número n examinado é ou não primo, já que o a é escolhido aleatoriamente. Mas quantas vezes deve ser repetido o teste ? Neste sentido vem o Teorema de Rabin (1978) que diz:

Se $n > 4$ e é ímpar e

(i) **n é primo, então $a \in B(n)$ para $2 \leq a \leq n - 2$**

(ii) n é composto, então $|B(n)| \leq (n-1)/4$

Apresentar-se-á somente uma idéia da prova do Teorema de Rabin. Para maiores detalhes ver [ROS93].

Conforme ficou determinado no teste anterior, para n ser pseudoprimo forte na base a,

- $a^t \equiv 1 \bmod n$ ou
- existe algum inteiro i, $0 \leq i < s$ tal que $a^{2^i t} \bmod n \equiv n-1$.

Em qualquer caso tem-se: $a^{n-1} \equiv 1 \bmod n$.

Considerando a decomposição de n em fatores primos como:

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \ldots p_k^{\alpha_k},$$

onde p é um número positivo ímpar e α é um inteiro positivo, tem-se que

$$\text{mdc}(n-1, p_i^{\alpha_i}(p_i - 1)) = \text{mdc}(n-1, p_i - 1)$$

representa a quantidade de soluções incongruentes para

$$x^{n-1} \equiv 1 \bmod p_i^{\alpha_i}, \quad i = 1, 2, \ldots, k.$$

Pelo teorema Chinês do resto, sabe-se que, para a congruência

$$x^{n-1} \equiv 1 \bmod n,$$

existem exatamente

$$\prod_{i=1}^{k} \text{mdc}(n-1, p_i - 1)$$

soluções incongruentes.

Para provar o teorema de Rabin, há três casos a considerar:

(i) a forma de decomposição de n contém uma potência prima $p_r^{\alpha_r}$ com expoente α_r maior ou igual a 2;

(ii) $n = pq$, com p e q primos ímpares distintos;

(iii) $n = p_1 p_2 \ldots p_k$, com $p_1, p_2, \ldots, p_k$ primos ímpares distintos.

O segundo caso pode ser incluído no terceiro e, como a demonstração é muito extensa e complicada, sem interesse para este estudo, considerar-se-á apenas o primeiro caso

$$\frac{p_r - 1}{p_r^{\alpha_r}} = \frac{1}{p_r^{\alpha_r - 1}} - \frac{1}{p_r^{\alpha_r}} \leq \frac{2}{9}.$$

Então

$$\prod_{i=1}^{k} \text{mdc}(n-1, p_i - 1) \leq \prod_{i=1}^{k} (p_i - 1)$$

$$\leq \left(\prod_{i=1, i\neq r}^{k} p_i\right)\left(\frac{2}{9} p_r^{\alpha_r}\right)$$

$$\leq \frac{2}{9} n$$

$$\leq \frac{n-1}{4} \quad \text{para } n \geq 9$$

Interpretando: há, no máximo, *(n-1)/4* números inteiros *a*, com $2 \leq a \leq n-2$, para os quais *n* é pseudoprimo forte na base *a*.

$|B(n)|$ representa a quantidade de falsas testemunhas e que é no máximo 25% de todas as bases *a* com $0 < a < n$.

Com base no teorema de Miller-Rabin, construiu-se o código **MilRab** que se encontra no Apêndice.

Se após um certo número de laços (loops), o algoritmo baseado em MillerRabin, retorna "composto", *n* é com certeza, um número composto. Se retorna "primo", ainda se tem uma incerteza. No pior caso, que acontece com os números de Carmichael, a probabilidade de erro é menor do que

$$\left(\frac{1}{4}\right)^k = \frac{1}{4^k} = 2^{-2k}.$$

Quer dizer que, após 10 repetições (k = 10), a probabilidade de erro é, no máximo, $1/2^{20}$, ou seja um erro em cada 1048576.

4.3. CRIPTO-SISTEMA RSA

Todo este estudo a respeito de números primos e testes de primalidade é necessário, porque o sistema de criptografia RSA tem a sua base em dois números primos grandes *p* e *q*. Vamos, agora, aos procedimentos do RSA:

1. Geram-se dois números primos grandes *p* e *q* e define-se *n* por $n = pq$. Os números p e *q* devem ser mantidos em segredo.
2. Define-se $\varphi(n) = (p-1)(q-1)$. Procura-se, randomicamente, um inteiro grande *d*, relativamente primo com $\varphi(n)$, tal que $1 < d < \varphi(n)$. O número *d* é a chave privada.
3. Calcula-se o inteiro *e* tal que $1 \leq e \leq \varphi(n)$ pela fórmula $ed \equiv 1 \bmod \varphi(n)$.

4. Torna-se conhecida a chave pública P, que consiste no par de inteiros (e, n).

5. Representando por M a mensagem a ser transmitida, como um inteiro no limite

$\{1,..., n\}$; quebra-se M em blocos se é grande demais.

6. Codifica-se M no criptograma C, pela regra $C \equiv M^e \bmod n$.

7. Decodifica-se C usando a chave privada d pela fórmula $D \equiv C^d \bmod n$.

A demonstração de que a mensagem recebida é realmente a mensagem enviada, ou seja, que $D \equiv M$, encontra-se logo a seguir, nas páginas 41 e 42. Entretanto já é interessante analisar um exemplo para perceber a seqüência dos passos, utilizando para p e q números primos pequenos para facilitar os cálculos.

Suponhamos que os números primos sejam $p = 7$ e $q = 13$.

Então $n = 7 * 13 = 91$.

Logo $\varphi(n) = (7 - 1)(13 - 1) = 72$.

Vamos usar $d = 25$ que satisfaz as condições de ser relativamente primo com $\varphi(n)$ e se encontra entre 1 e 72.

O número e deve ser calculado, já que $25e \equiv 1 \bmod 72$, ou seja $e \equiv 25^{-1} \bmod 72 = 49$.

Publica $P = (49, 91)$.

Usando a correspondência entre letras e números da Tabela 3, página 6, e desejando enviar uma mensagem qualquer, digamos, $M = oi$, ter-se-á $M = 1308$.

Codificando M no criptograma $C \equiv 1308^{49} \bmod 91$, o que calculado resultará em $C = 34$.

Decodificando com a chave privada $d = 25$, encontrar-se-á $D = 34^{25} \bmod 91 = 34$. Não deu certo, a mensagem $M = 1308$ e não 34. A razão é simples: não se atendeu à exigência de que a mensagem M deva estar no intervalo de 0 a $n - 1$, e, no caso $M > n$ $(1308 > 91)$.

Refazendo, visando acertar, dividir-se-á a mensagem em blocos, cuidando para que cada bloco seja menor do que N.

Assim, digamos, $M_1 = \text{o} = 13$ e $M_2 = \text{i} = 08$.

Codificando, ter-se-á $C_1 = 13^{49} \bmod 91 = 13$ e $C_2 = 8^{49} \bmod 91 = 8$.

Assim, a mensagem codificada será enviada em dois blocos e decodificando cada bloco usando a chave privada, ter-se-á $D_1 = 13^{25} \bmod 91 = 13$ e $D_2 = 8^{25} \bmod 91 = 8$, o que, juntando os blocos, retorna a mensagem enviada, ainda codificada.

A mensagem M enviada, encontra-se pela correspondência entre os números e as letras de acordo com a Tabela 3, página 6.

Para implementar o sistema RSA, elaborou-se procedimentos utilizando o MAPLE.

Explicar-se-á cada um destes códigos, iniciando pelo *expomod*. Trata-se de um código muito importante pela freqüência com que é usado nos sistemas de criptografia. Tem como entrada a base, o expoente e o módulo desejado e o código devolve a potência em módulo. Apresentar-se-á, a seguir, o código *expomod* na forma de pseudo-algoritmo.

Algoritmo 6 - Cálculo de potências em módulo

expomod (a, n, z)

1. Inicialização.

   $i \leftarrow n;\ r \leftarrow 1;\ x \leftarrow a \bmod z$

2. Enquanto $i > 0$ faça

   se i é ímpar então $r \leftarrow r.x \bmod z$

   senão $x \leftarrow x^2 \bmod z$

   $i \leftarrow solo\ (i / 2)$

3. Retorna (r).

Aparentemente o algoritmo realiza um número de multiplicações em módulo em $\theta(\ln n)$ para calcular $a^n \bmod z$. Uma análise mais precisa revela que o número de multiplicações em módulo é igual ao número de bits da expansão binária de n, mais o número destes bits que é igual a 1, ficando, então, aproximadamente igual a $\frac{3}{2}\log_2 n$, onde n representa o expoente.

O código *Btest* é construído a partir da extensão do teorema de Fermat. Testa para uma determina base a a primalidade de um número n. Tem como entrada a base a que deve ser entre 2 e $n-2$ e o número n e a saída é verdadeiro ou falso. Repetindo cinco vezes o Btest é o suficiente para decidir deterministicamente a primalidade de um número inteiro até 10^{13} [BRA96].

Baseado no teorema de Miller-Rabin tem-se o código *MilRab*, que tem como entrada um número n, cuja primalidade está sendo testada e k que representa a quantidade de repetições (loops) que se deseja para assegurar um máximo de fidelidade. A saída é a informação se o número n é primo ou composto. É um teste de primalidade que responde corretamente com uma

probabilidade de $1-4^{-k}$. O tempo requerido para cada repetição do MilRab é controlado pelo número de multiplicações em módulo, isto é, de *O(ln n)*. Se as multiplicações são realizadas pelo algoritmo clássico, cada uma toma um tempo em $O(\ln^2 n)$ porque reduz-se a módulo *n* depois de cada multiplicação, onde *n* representa o número a ser testado.

Finalmente foi possível elaborar o código ***sorpri*** com a finalidade de gerar números primos com *n* dígitos, apoiado no MilRab com 10 repetições. Tem como entrada a quantidade *n* de dígitos do número primo desejado e a saída é este número primo. A probabilidade do número ser primo é de 99,99990463%. Como o código está apoiado unicamente no MilRab, tem requerido o mesmo tempo deste.

Para calcular o número *d*, que é chave privada e deve ser relativamente primo com $\varphi(n)$, pode ser usado o código **mdc** ou melhor, o **relprium** que se encontram no Apêndice.

O mdc foi construído tendo por base o método de Euclides para calcular o máximo divisor comum entre dois números inteiros. Tem como entrada os dois números *x* e *y*, dos quais se pretende calcular o mdc e a saída é o mdc. O tempo do código mdc está em $O(\ln^3 x)$, considerando $x > y$.

Necessita-se, também no RSA, um número randômico que seja relativamente primo a outro número. Para suprir essa necessidade tem-se o código *relprium* que tem como entrada o número *n*, para o qual está-se procurando um número relativamente primo e *d* que representa a quantidade de figuras do número que se está procurando. A saída é um número relativamente primo a *n* com *d* dígitos. Como testa a primalidade pelo código mdc, é do mesmo tempo daquele e foi citado no parágrafo anterior.

Para determinar o inverso de um número em módulo, tem-se o código **mdceuinv,** baseado no algoritmo de Euclides estendido. Tem como entrada os números inteiros *a* e *b* e como saída o inverso de *b* módulo *a*, quando o *mdc(a, b) = 1*, ou seja, *a* e *b* devem ser primos entre si. Quando *a* e *b* não são primos entre si retorna ERRO e a justificativa do erro. Tem o tempo em $O(\ln^3 a)$ quando $a > b$.

O número *e*, chave pública, encontra-se pela aplicação do código *mdceuinv*.

Finalmente, de posse dos códigos citados antes, foi possível elaborar o código **RSAemi** em que um usuário do sistema de criptografia RSA, o emitente, de posse do código *RSAemi*, pode enviar uma mensagem *M* codificada. A entrada é a quantidade de dígitos que se deseja para o menor dos números primos (o outro número primo terá dois dígitos a mais); a mensagem será

digitada entre aspas simples, podendo-se quebrá-la em blocos quando muito grande. Permitiu-se, para que na saída fosse mostrada a chave pública *(e, n)*, a chave privada *d* e a mensagem codificada *Cod*. Na implementação optou-se por utilizar o código ASCII, que necessita atribuir um número de até três dígitos para cada letra, número ou outro símbolo. Assim o *RSAemi* inicia verificando o comprimento da mensagem M, ou seja de quantos dígitos ela é formada, lembrando que espaço vazio também conta como um dígito. Usa depois o código *sorpri* para determinar os números primos, e no código *RSAemi* seguem-se os passos do RSA já descritos anteriormente. Cabe ainda ressaltar que além do *sorpri*, os códigos *mdc*, *mdceuinv* e *expomod* estão declaradamente dentro do *RSAemi* e os códigos *Btest* e *MilRab*, embora não sejam citados, se encontram dentro do *sorpri*. O cálculo de $\varphi(n)$ a partir de p e q envolve $O(\ln n)$ bit operações e pode-se calcular p e q de n e $\varphi(n)$ em $O(\ln^3 n)$ bit operações.

O outro usuário do sistema RSA, o destinatário, de posse do algoritmo de decodificação **RSAdes**, pode decodificar a mensagem recebida. A entrada do código é o número n, que é público, a chave privada d do destinatário e a mensagem codificada recebida *Cod*. A saída é a mensagem decodificada. Como consta no procedimento 7 da descrição do funcionamento do RSA: "Decodifica-se usando a chave privada d pela fórmula $D \equiv C^d \bmod n$." Usou-se, por isso, mais uma vez o *expomod*, mas o resultado, evidentemente, é um número que precisa ser convertido tendo como base o código ASCII. Para tanto, mediu-se o comprimento deste número e dividiu-se em blocos de 3 dígitos, convertendo e juntando sucessivamente para formar, finalmente, a mensagem enviada.

Com o objetivo de destacar qual é a matemática subjacente, especialmente a Teoria dos Números, vamos reexaminar como funciona o sistema RSA.

Na tarefa de codificar-decodificar, com uma dada chave pública *(e, n)* e a chave privada *d*, e para uma mensagem *M* representada por inteiros do intervalo de *0* a *n - 1*, tem-se que, para codificar, pode ser usado o algoritmo **expomod**

$$(1) \qquad C = Cod(M) = M^e \bmod n$$

e para decodificar, o destinatário, o único que conhece d, também pode usar **expomod**

$$(2) \qquad Deco(C) = C^d \bmod n = M$$

A questão é verificar se a mensagem decodificada é igual a mensagem enviada, ou seja se

$$(3) \qquad Deco(Cod(M)) = M, \quad \text{para } 0 \le M < n$$

Para tanto vamos substituir (1) em (3):

(4) $Deco(Cod(M)) = Deco(M^e \bmod n)$.

Usando a idéia dada por (2), tem-se que

(5) $Deco(Cod(M)) = (M^e \bmod n)^d \bmod n = M^{ed} \bmod n$

Por definição, $ed \equiv 1 \bmod \varphi(n)$. E portanto $ed = 1 + t(p - 1)(q - 1)$,

(6) $Deco(Cod(M)) = M^{t(p-1)(q-1)+1} \bmod n$

para um t inteiro não-negativo. Ou ainda, $ed \bmod (p - 1) = 1$.

O teorema de Fermat: "Se p é um número primo, m um inteiro não múltiplo de p, então, para todo natural r, $m^r \bmod p = m^{r \bmod (p-1)} \bmod p$", vem em nosso socorro.

Lembrando que p é primo por definição, e sendo M não múltiplo de p:

$$M^{t(p-1)(q-1)+1} \bmod p = M^{ed \bmod (p-1)} \bmod p = M^{ed} \bmod p = M \bmod p$$

Se M for múltiplo de p então M^{ed} também será múltiplo de p.

Portanto, para qualquer que seja M,

$$M^{ed} = M \bmod p$$

ou seja, para qualquer s inteiro,

$$M^{ed} = M + sp.$$

De forma análoga,

$$M^{ed} = M + tq,$$

para qualquer inteiro t.

Portanto,

$$M^{ed} - M = sp = tq.$$

Mas p e q são primos distintos e, como p divide sp e p também divide tq, logo p divide t.

Ou seja, para algum inteiro w,

$$M^{ed} - M = wpq.$$

Como $pq = n$,

$$M^{ed} = M + wn$$

$$M^{ed} \bmod n = M \bmod n = M.$$

Conforme (5) e (3), era o que queríamos demonstrar.

4.4. QUEBRANDO O RSA

Uma questão importante que se impõe é se e como o sistema RSA pode ser quebrado.

Para quebrar o sistema, admitimos que se conhece o seguinte:

1. o par de inteiros *(e, n)*, porque são a chave pública;
2. algumas mensagens M que correspondem aos criptogramas C. O criptograma C, pode ter sido captado por grampo telefônico, por exemplo. Em cada par *(M, C)*, M e C são números inteiros de *1* a $n - 1$.

Os caminhos para quebrar o sistema são:

(a) por radiciação:

Calcular M, conhecendo C, a mensagem codificada, e a chave pública *(e, n)*.

Tem-se que $C \equiv M^e \bmod n$.

A questão a ser respondida é "qual é o número que, elevado ao expoente e em módulo n, resulta C ?" ou "qual é a raiz e-ésima de C em módulo n ?".

Ainda existe a condição de $M \leq n$.

Visando resolver esta questão, ainda que parcialmente, elaborou-se o código **queRSA** que tem como entrada os três elementos conhecidos: a chave pública *(e, n)* e a mensagem codificada C. A saida é a raiz e-ésima. Este número representa a mensagem codificada, mas tendo sido usado um código conhecido para codificar, tipo ASCII, usando o mesmo código para decodificar, fica evidentemente muito fácil transformar a mensagem de números para letras.

O código *queRSA* trabalha por tentativas, começa atribuindo ao número M o valor de *1* até n, quando encontra o que satisfaz a condição $C \equiv M^e \bmod n$, pára a execução e responde à pergunta. Quando os números são grandes passa a ser impraticável porque trabalha por tentativas.

Em outro sentido, trata-se de resolver a equação:

$$irem\left(\frac{x^e}{n}\right) - C = 0$$, o que não se sabe fazer.

(b) fatorando n:

obviamente, conseguindo fatorar n, obtém-se os primos p e q. A partir destes, calcula-se $\varphi(n)$ e como e é conhecido, pode-se calcular d, que é a chave privada. Este cálculo pode ser feito por $ed = 1 + k\varphi(n)$ ou melhor, por $d = e^{-1} \bmod n$.

(c) inventando um algoritmo que conseguisse calcular *d* diretamente a partir de *(e, n)*:

como $ed \equiv 1 \bmod \varphi(n)$, isto implica conhecer um múltiplo de $\varphi(n)$. Isto também é suficiente para fatorar *n*. Maiores detalhes no livro de Rivest, Shamir e Adleman 1978, parágrafo IX.C [COU97].

Acredita-se que quebrar o RSA e fatorar *n* sejam problemas equivalentes, embora até agora isto não tenha sido demonstrado.

Cabem algumas sugestões que tornam o sistema RSA ainda mais difícil para ser quebrado com algumas medidas de prevenção e precaução.

1. O que torna o sistema RSA um problema difícil é a fatoração e para tornar razoavelmente seguro, a escolha dos números primos *p* e *q* deve ser da ordem de, no mínimo, 200 dígitos.
2. Os números primos *p* e *q* devem diferir em comprimento, evitando assim ficarem muito próximos de $\sqrt{n}$.
3. Encontrar o número *d*, relativamente primo a *(p - 1)(q - 1)*, também pode ser feito tomando *d* como um número primo maior do que *max {p, q}*, de forma randômica.
4. Assegurar que *p - 1* e *q - 1* tenham fatores primos grandes. Isso reduz o risco de *n* ser fatorado.
5. O mdc*(p - 1, q - 1)* deve ser pequeno.

Há uma implementação promovendo o uso do produto de três primos grandes, $n = p_1 p_2 p_3$ onde cada p_i é aproximadamente $n^{1/3}$. Alguns dos cálculos podem ser feitos em $mod\ p_i$ e o resultado (mod n) deduzido via teorema Chinês do resto. É mais seguro usar três primos em vez de dois. Por outro lado, a segurança do sistema pode ser comprometida porque *n*, tendo fatores primos menores, pode ser mais fácil de ser fatorado do que no caso de dois primos [BRE99].

5. O CRIPTO-SISTEMA BASEADO NO LOGARITMO DISCRETO

Neste capítulo abordar-se-á o cripto-sistema de chave pública baseado no logaritmo discreto. Depois dos conceitos necessários, serão apresentadas cinco idéias: a de Diffie-Hellman, para gerar as chaves; a de ElGamal e a de Massey-Omura, para enviar mensagens; a de Shanks e a de Silver-Pohlig-Hellman, para calcular o logaritmo discreto, quebrando o sistema.

Além desses, existem outros trabalhos baseados no logaritmo discreto: método "*index-calculus*" que apresenta considerável semelhança com vários dos melhores algoritmos de fatoração e o método "*canguru*" de Pollard, que podem ser usados para calcular o logaritmo discreto. Recentemente, as "*anômalas*" curvas elípticas tem-se mostrado extremamente eficientes em algoritmos de logaritmo discreto.

5.1. INTRODUÇÃO

Relacionando o cripto-sistema do logaritmo discreto com o cripto-sistema RSA, tem-se que o RSA está baseado no fato de que encontrar dois números primos grandes p e q e multiplicá-los para determinar n é muito mais fácil do que realizar a operação inversa, isto é, tendo n encontrar os dois primos p e q. Além da maneira como é usada no RSA, existem outros sistemas que têm as propriedades do "alçapão" ou "uma-via". Um dos mais importantes é elevar a uma potência em um corpo finito grande.

Operando com números reais, a exponenciação (encontrar b^x com uma exatidão estabelecida) não é significativamente mais fácil do que a operação inversa, a logaritmação (encontrar $\log_b x$ com uma exatidão estabelecida). Operando em um grupo finito com logaritmo discreto o logaritmo fica bem definido.

Em 1976, Diffie-Hellman criou o primeiro algoritmo de chave pública, usando o logaritmo discreto em um corpo finito.

Para fundamentar o sistema de criptografia de chave pública baseado no logaritmo discreto são necessárias algumas definições que serão estabelecidas inicialmente.

Um gerador g de um corpo finito F_q é um elemento de ordem $q - 1$; equivalentemente, as potências de g esgotam todos os elementos de F_q^*. Os elementos não-nulos de um corpo finito formam um grupo cíclico, isto é, são todos potências de um único elemento. Todos os corpos finitos têm um gerador.

Considerando um corpo finito F_q e lembrando que o grupo multiplicativo F_q é cíclico, um gerador de F_q é chamado **elemento primitivo**.

Dado um y, como calcular o valor do expoente x de b^x para um b fixado, ou seja calcular $x = \log_b y$? Esta questão é chamada de **problema do logaritmo discreto**. Discreto para distinguir o grupo finito do caso clássico, ou seja, da forma como comumente é empregado.

Se G é um grupo finito, b um elemento de G, e y um elemento de G que é potência de b, então o **logaritmo discreto** de y com base b é algum inteiro x tal que $b^x = y$ [KOB94].

Um exemplo: se $G = F_{19}^* = (Z / 19Z)^*$ e $b = 2$ o gerador, então o logaritmo discreto de 7 na base 2 é 6, porque $7 \equiv 2^6 \ mod\ 19$ (pode ser calculado por *expomod(2,6,19) = 7).*

Para algum inteiro p, o conjunto Z_p^* de inteiros módulo p, que são relativamente primos com p, formam um grupo multiplicativo módulo p.

A ordem deste grupo é representada por $\varphi(p)$ e é chamada função **totient** ou função **phi de Euler.**

Para qualquer x $(1 \leq x < p)$ que é relativamente primo com p, tem-se, pelo pequeno teorema de Fermat, que

$$x^{\varphi(p)} = 1 \bmod p .$$

No caso especial em que p é um primo n, cada inteiro menor que n é relativamente primo com n. Assim

$$\varphi(n) = n - 1.$$

Então tem-se

$$x^{n-1} \equiv 1 \bmod n \ (x = 1, \ldots, n - 1).$$

Considerando algum inteiro a no intervalo $1 \leq a \leq p$, se $mdc\ (a, p) = 1$ e tendo a a propriedade adicional de que $a^d \neq 1 \ mod\ p$ para todo e qualquer d, $1 \leq d < \varphi(p)$, diz-se que a é a **raiz primitiva** de p.

Por exemplo, 2 é uma raiz primitiva de 5 porque,

$2^1 = 2 \neq 1 \bmod 5$,

$2^2 = 4 \neq 1 \bmod 5$,

$2^3 = 8 = 3 \bmod 5 \neq 1 \bmod 5$,

lembrando que $\varphi(5) = 4$ e o expoente d deve ser menor do que $\varphi(p)$.

Tomando-se qualquer inteiro n que tem uma raiz primitiva, para qualquer x $(0 \leq x < \varphi(n))$, se $y = a^x \bmod n$, então x é chamado o **logaritmo discreto** de y para a base a *módulo* n, e é escrito $x = \log_a y \pmod n$. A importância de a ser um elemento primitivo é que ele garante que, para algum $y \in Z_p^*$, existe um único x tal que o logaritmo discreto é bem definido [WEL90].

A função exponencial $y = a^x \bmod n$ é uma função uma-via. Em outras palavras: a exponenciação é 'fácil' de calcular, mas encontrar logaritmos é 'difícil'.

Exponenciação pode ser feita em $\lceil \log_2 n \rceil$ multiplicações módulo n onde n representa o expoente. Por exemplo

$$2^{18} = (((2^2)^2)^2)^2 2^2$$

pode ser obtido com exatamente cinco multiplicações.

Calculado no Maple com ceil(evalf(log[2] (18))) = 5.

Para calcular o logaritmo discreto elaborou-se o código **logdis,** baseado na definição de logaritmo discreto, usando o Maple. Tem como entrada a base b, o número n e o módulo m; e como saída o logaritmo discreto. O código testa números inteiros de 1 em diante, até encontrar aquele que satisfaz a definição, ou seja, a base b elevada ao número testado, em módulo m, apresenta como resultado n. Como poderá testar diversos números, torna-se muito lento. O código encontra-se no Apêndice.

Não é conhecido um algoritmo com tempo polinomial para calcular logaritmos discretos. Para prevenir ataques, p deve ter no mínimo 150 dígitos, e $p - 1$ deve ter no mínimo um fator primo grande [STI95].

A dificuldade desses cálculos é comumente considerada igual à da fatoração. O logaritmo discreto é considerado um problema de 'difícil' inversão.

Cabe ainda mencionar que cripto-sistemas baseados no logaritmo discreto podem ser usados para a troca de chaves e a assinatura digital com segurança.

5.2. CRIPTO-SISTEMAS BASEADOS NO LOGARITMO DISCRETO

Apoiados na idéia do logaritmo discreto, surgiram vários trabalhos ligados à criptografia. Uns, preocupados com a elaboração e troca de chave pública, outros, com o envio de mensagens codificadas e ainda outros, com o cálculo do logaritmo discreto e conseqüente quebra do cripto-sistema. Dentro dos vários trabalhos existentes, serão apresentados cinco, todos apoiados no logaritmo discreto.

5.2.1. DIFFIE-HELLMAN

O primeiro sistema de chave pública, devido a W.Diffie e M.E.Hellman, era baseado no problema do logaritmo discreto e no fato de que é praticamente impossível calcular g^{ab}, conhecendo apenas g^a e g^b.

Nesta subseção será descrita a maneira como são geradas as chaves, tanto a chave privada como a chave pública e o funcionamento do esquema de troca da chave secreta. Para o envio de mensagens é utilizado algum tipo de criptografia tradicional.

Inicialmente será visto o método para gerar um elemento randômico de um corpo finito grande Z_q. Supõe-se que q é de conhecimento público e que g é determinado elemento de Z_q que também não é mantido em segredo.

Suponha que dois usuários concordem sobre uma chave - um elemento randômico de Z_q^* - o qual será usado para codificar suas mensagens. O usuário A escolhe um inteiro randômico a entre 1 e $q - 1$, que mantém em segredo, e calcula $g^a \in Z_q$, que é público. O usuário B procede de forma idêntica: escolhe um randômico b e torna g^b público. A chave secreta usada por eles é então g^{ab}. Repetindo: A conhece g^b que é público e a que é sua chave secreta. Um terceiro usuário conhece as chaves públicas g^a e g^b.

O sistema parte da hipótese de que é difícil calcular logaritmos discretos em um grupo, ou seja, se o logaritmo discreto pode ser calculado, então o Diffie-Hellman falha. Ainda está em aberto o caminho que permite passar de g^a e g^b para g^{ab} sem primeiro determinar a e b; mas conjectura-se que tal caminho deva existir [KOB94].

Exemplificando, vamos considerar $q = 53$ e $g = 2$ (que é gerador de Z_{53}). Supondo-se que o usuário A tenha escolhido randomicamente $a = 29$ e o usuário B tenha tornado público $2^b = 12 \ \ (12 \in Z_{53})$. A então sabe que a chave de codificação é $12^{29} = 21 \in Z_{53}$, isto é, b = 21, *(expomod(*12, 29, 53) = 21*)*, porque sendo $g^b = 12$ então $g^{ab} = (g^b)^a = 12^{29}$. Entretanto A tornou público $2^{29} = 45$ *(expomod(*2, 29, 53); = 45*)*, e assim B pode também encontrar a chave b = 21 elevando 45 a b-ésima potência (expoente secreto de B é b = 19), (*expomod(*45, 19, 53) = 21). De fato, não é seguro trabalhar com um corpo pequeno; um estranho pode facilmente achar o logaritmo discreto de base 2 de 12 ou 45 módulo 53. Pelo código *logdis* elaborado no Maple e que se encontra no Apêndice, entrando com logdis(2, 12, 53); tem-se a saída: o logaritmo discreto de 12, na base 2, em módulo 53 é 19. E logdis(2, 45, 53); devolve: o logaritmo discreto de 45, na base 2, em módulo 53 é 29. Esse exemplo ilustra o mecanismo do sistema de **permutação de chave** de Diffie-Hellman.

Para quebrar o sistema tem-se algumas possibilidades:

1) encontrar um algoritmo que permita calcular g^{ab}, conhecendo g^a e g^b, mas sem conhecer a e b;
2) calcular o logaritmo discreto. Neste caso, obviamente, o método de Diffie-Hellman falha.

5.2.2. ELGAMAL

Em 1985, Taher Elgamal anunciou um sistema baseado na aparente intratabilidade do logaritmo discreto, para a codificação e decodificação de mensagens.

O sistema consiste no seguinte: inicialmente é fixado um corpo finito grande Z_q e um elemento $g \in Z_q^*$. Cada usuário *A* escolhe randomicamente um inteiro $a = a_A$ do intervalo $0 < a < q - 1$. Este inteiro *a* é a chave secreta de decodificação. A chave pública para a codificação é o elemento $g^a \in Z_q$.

Para enviar uma mensagem *M* para o usuário *A*, escolhe-se aleatoriamente um inteiro *k* e envia-se para *A* o par de elementos de Z_q:

$$(g^k, Mg^{ak})$$

Note que é possível calcular g^{ak} sem conhecer *a*: basta elevar g^a ao expoente *k*. O usuário *A*, que conhece *a*, pode decodificar a mensagem *M* pela elevação do primeiro elemento do par ao expoente *a* e dividindo o segundo elemento por este resultado.

Exemplificando, vamos supor que $q = 71$ e $g = 7$. Também vamos supor que o usuário *A* tenha escolhido como chave secreta $a_A = 26$, logo a sua chave pública será

$$g^a = 7^{26} \, mod \, 71 = 3, \quad \text{calculada por } expomod(7, 26, 71);$$

tomando-se $k = 2$, então $g^k = 7^2 \bmod 71 = 49$. Pretendendo-se enviar a mensagem M = 30 para A, codifica-se a mensagem $M (= Mg^{ak})$:

$$Mg^{ak} = 30 * 3^2 \bmod 71 = 57 \quad (\text{mod } p(30 * 3^2, 71);)$$

e envia-se o par (49, 57).

Para decodificar, o usuário *A* de posse de sua chave privada $a_A = 26$, calcula

$$g^{ak} = 49^{26} \bmod 71 = 9 \quad \text{(expomod(49, 26, 71);)}.$$

Finalmente encontra *M* por *modp*(57/9, 71) = 30.

Conseguindo resolver o problema do logaritmo discreto em Z_q, quebra-se o criptosistema, pois se acha a chave secreta de decodificação *a* a partir da chave pública g^a. Teoricamente não existe uma maneira para calcular g^{ab}, conhecendo-se g^a e g^b.

5.2.3. MASSEY-OMURA

Nesta sub-seção será apresentado o cripto-sistema de Massey-Omura para a transmissão de mensagens, baseado no problema do logaritmo discreto.

O sistema considera fixado e de conhecimento público um corpo finito Z_q. Cada usuário do sistema seleciona secretamente e randomicamente um inteiro *e* tal que $0 < e < q - 1$ e $mdc(e, q - 1) = 1$, ou seja, *e* e $q - 1$ devem ser relativamente primos entre si, podendo-se usar o algoritmo de Euclides. Calcula-se o seu inverso $d = e^{-1} \bmod (q - 1)$, isto é, $ed \equiv 1 \bmod (q - 1)$.

O usuário A, desejando enviar uma mensagem *M* para B, envia primeiro M^{e_A}. Isso nada significa para B, porque não conhece d_A e nem e_A, não podendo descobrir *M*. Determina então o seu e_B e envia de volta para A, $M^{e_A e_B}$. A seguir cabe a A, calcular a d_A-ésima potência de $M^{e_A e_B}$, porque $M^{d_A e_A} = M$, isso significa que A deve devolver M^{e_B} para B, que pode então ler a mensagem pela elevação a d_B-ésima potência.

Para segurança do sistema, é absolutamente necessário usar um bom esquema de assinaturas, senão um personagem C, que não deve conhecer a mensagem *M*, pretendendo ser B, devolve para A, $M^{e_A e_C}$; não sabendo que alguém estranho introduziu e_C, A pode aplicar d_A, permitindo assim a C ler a mensagem. Por isto, a mensagem $M^{e_A e_B}$ de B para A, deve estar acompanhada de alguma autenticação, ou seja um esquema de assinatura que somente B possui.

No próximo capítulo far-se-á a descrição de métodos para se obter a assinatura digital.

É ainda importante ressaltar que os usuários B e C, após decodificar várias mensagens *M*, e para tanto conhecendo vários pares $\left(M, M^{e_A}\right)$, não podem usar este conhecimento para calcular e_A, isto é, supondo que B pode resolver o problema do logaritmo discreto em Z_q^*, determina assim de M e M^{e_A} qual é o valor de e_A. Nesse caso pode facilmente calcular $d_A = e_A^{-1} \bmod(q - 1)$ e então interceptar e ler mensagens de A.

Vamos apresentar um exemplo: supondo que A deseja enviar a mensagem $M = oi$ para B, usando a seguinte correspondência entre as 23 letras e os números: $a = 00$, $b = 01$, ..., $i = 08$, ..., $o = 13$, ..., $z = 22$. Então a mensagem $M = 1308$.

A seguir o usuário A escolhe um número primo p maior do que M, podendo usar o código **sorpri**. Consideremos $p = 7487$, o que é tornado público. O número e_A escolhido por A, que deve ser relativamente primo com p, foi $e_A = 17$ e calcula d_A, o que pode ser feito pelo código **mdceuiv**, resultando $d_A = 6165$.

Envia, então, para B, $M^{e_A} = 5507$, calculado pelo código expomod(1308, 17, 7487).

O usuário B, tendo escolhido $e_B = 5$, relativamente primo com $p = 7487$ e calculado $d_B = 5989$, calcula também $M^{e_A e_B} = 5307$ (por expomod (5507, 5, 7487)) enviando de volta para A este número.

Como o usuário A sabe que $M^{e_A} = 5507$, calcula, a partir de $M^{e_A e_B} = 5307$, $\left(M^{e_A e_B}\right)^{d_A} = 3158$ *(expomod(5307, 6165, 7487))*, que é M^{e_B}, porque $M^{d_A e_A} = M$.

O usuário A envia então este número para B, que usando $d_B = 5989$, decodifica a mensagem M porque $M^{e_B d_B} = M$, por expomod(3158, 5989, 7487) = 1308, retornando a tabela de conversão de letra em número encontra M = oi.

5.2.4. SHANKS

Nesta subseção será visto o algoritmo de Shanks para calcular o logaritmo discreto.

Considerando p como número primo e α como elemento primitivo módulo p, o problema do logaritmo discreto pode ser apresentado da seguinte forma: "dado $\beta \in Z_p^*$ encontre o único expoente a, $0 \leq a \leq p-2$, tal que $\alpha^a \equiv \beta \bmod p$".

O algoritmo de Shanks é o seguinte:

(i) chamando $m = \lceil \sqrt{p-1} \rceil$

(ii) calcula-se $\alpha^{mj} \bmod p$, $0 \leq j \leq m-1$

(iii) classifica-se os m pares ordenados $(j, \alpha^{mj} \bmod p)$ com relação às suas segundas coordenadas, obtendo uma lista L_1

(iv) calcula-se $\beta\alpha^{-i} \bmod p$, $0 \leq i \leq m-1$

(v) classifica-se os m pares ordenados $(i, \beta\alpha^{-i} \bmod p)$ com relação às suas segundas coordenadas, obtendo uma lista L_2

(vi) encontra-se um par $(j, y) \in L_1$ e um *par* $(i, y) \in L_2$ (as segundas coordenadas devem ser iguais)

(vii) define $\log_\alpha \beta = mj + i \bmod (p-1)$.

Observe que as etapas (i) e (ii) podem ser pré-calculadas e ainda que, se $(j, y) \in L_1$ e $(i, y) \in L_2$, então $\alpha^{mj} = y = \beta\alpha^{-i}$, assim $\alpha^{mj+i} = \beta$. Inversamente, para qualquer β, pode-se escrever

$$\log_\alpha \beta = mj + i \text{ onde } 0 \leq j, i \leq m-1.$$

Exemplo:

O número primo p pode ser obtido pelo código *sorpri*. Digamos que p = 809 e que se deseja encontrar $\log_3 525$. Tem-se $\alpha = 3$, $\beta = 525$ e $m = \lceil \sqrt{808} \rceil = 29$.

Então por (i) $\alpha^{mj} \bmod p$, *expomod*(3, 29, 809) = 99, e calculando os pares ordenados (j, 99^j mod 809) para $0 \leq j \leq 28$, obtém-se a lista L_1.

(No Maple: *for j from* 0 *to* 28 *do expomod*(99, j, 809) *od;)*

(0, 1)	(1, 99)	(2, 93)	(3, 308)	(4, 559)
(5, 329)	(6, 211)	(7, 664)	(8, 207)	(9, 268)
(10, 644)	(11, 654)	(12, 26)	(13, 147)	(14, 800)
(15, 727)	(16, 781)	(17, 464)	(18, 632)	(19, 275)
(20, 528)	(21, 496)	(22, 564)	(23, 15)	(24, 676)
(25, 586)	(26, 575)	(27, 295)	(28, 81)	

A segunda lista contém os pares ordenados *(i,* $525*(3^i)^{-1}$ *mod* 809*)*, $0 \leq i \leq 28$.

(No Maple: *for i from* 0 *to* 28 *do modp*(525*(3^i)^(-1), 809) *od;)*

obtendo a lista L_2

(0, 525)	(1, 175)	(2, 328)	(3, 379)	(4, 396)
(5, 132)	(6, 44)	(7, 554)	(8, 724)	(9, 511)
(10, 440)	(11, 686)	(12, 768)	(13, 256)	(14, 355)
(15, 388)	(16, 399)	(17, 133)	(18, 314)	(19, 644)
(20, 754)	(21, 521)	(22, 713)	(23, 777)	(24, 259)
(25, 356)	(26, 658)	(27, 489)	(28, 163)	

Percorrendo simultaneamente as duas listas à procura de segundos elementos iguais, encontra-se (10, 644) em L_1 e (19, 644) em L_2. Pode-se então calcular

$\log_3 525 = 29*10+19 = 309.$

Verificando, tem-se: $3^{309} \equiv 525 \ mod\ 809$

obtido por *expomod*(3, 309, 809) = 525.

Ou calculando pelo código *logdis*(3, 525, 809, tem-se a resposta:

O logaritmo discreto de 525, *na base* 3, *em módulo* 809 *é* 309.

A complexidade do algoritmo de Shanks é de um tempo em *O(m)*.

5.2.4. SHANKS

Nesta subseção será visto o algoritmo de Shanks para calcular o logaritmo discreto.

Considerando p como número primo e α como elemento primitivo módulo p, o problema do logaritmo discreto pode ser apresentado da seguinte forma: "dado $\beta \in Z_p^*$ encontre o único expoente a, $0 \leq a \leq p-2$, tal que $\alpha^a \equiv \beta \bmod p$".

O algoritmo de Shanks é o seguinte:

(i) chamando $m = \lceil \sqrt{p-1} \rceil$

(ii) calcula-se $\alpha^{mj} \bmod p$, $0 \leq j \leq m-1$

(iii) classifica-se os m pares ordenados $(j, \alpha^{mj} \bmod p)$ com relação às suas segundas coordenadas, obtendo uma lista L_1

(iv) calcula-se $\beta\alpha^{-i} \bmod p$, $0 \leq i \leq m-1$

(v) classifica-se os m pares ordenados $(i, \beta\alpha^{-i} \bmod p)$ com relação às suas segundas coordenadas, obtendo uma lista L_2

(vi) encontra-se um par $(j, y) \in L_1$ e um *par* $(i, y) \in L_2$ (as segundas coordenadas devem ser iguais)

(vii) define $\log_\alpha \beta = mj + i \bmod (p-1)$.

Observe que as etapas (i) e (ii) podem ser pré-calculadas e ainda que, se $(j, y) \in L_1$ e $(i, y) \in L_2$, então $\alpha^{mj} = y = \beta\alpha^{-i}$, assim $\alpha^{mj+i} = \beta$. Inversamente, para qualquer β, pode-se escrever

$$\log_\alpha \beta = mj + i \text{ onde } 0 \leq j, i \leq m-1.$$

Exemplo:

O número primo p pode ser obtido pelo código *sorpri*. Digamos que p = 809 e que se deseja encontrar $\log_3 525$. Tem-se $\alpha = 3$, $\beta = 525$ e $m = \lceil \sqrt{808} \rceil = 29$.

Então por (i) $\alpha^{mj} \bmod p$, *expomod*(3, 29, 809) = 99, e calculando os pares ordenados (j, 99^j mod 809) para $0 \leq j \leq 28$, obtém-se a lista L_1.

(No Maple: *for j from* 0 *to* 28 *do expomod*(99, j, 809) *od;)*

(0, 1)	(1, 99)	(2, 93)	(3, 308)	(4, 559)
(5, 329)	(6, 211)	(7, 664)	(8, 207)	(9, 268)
(10, 644)	(11, 654)	(12, 26)	(13, 147)	(14, 800)
(15, 727)	(16, 781)	(17, 464)	(18, 632)	(19, 275)
(20, 528)	(21, 496)	(22, 564)	(23, 15)	(24, 676)
(25, 586)	(26, 575)	(27, 295)	(28, 81)	

A segunda lista contém os pares ordenados *(i,* $525*(3^i)^{-1}$ *mod* 809*),* $0 \leq i \leq 28$.

(No Maple: *for i from* 0 *to* 28 *do modp*(525*(3^i)^(-1), 809) *od;)*

obtendo a lista L_2

(0, 525)	(1, 175)	(2, 328)	(3, 379)	(4, 396)
(5, 132)	(6, 44)	(7, 554)	(8, 724)	(9, 511)
(10, 440)	(11, 686)	(12, 768)	(13, 256)	(14, 355)
(15, 388)	(16, 399)	(17, 133)	(18, 314)	(19, 644)
(20, 754)	(21, 521)	(22, 713)	(23, 777)	(24, 259)
(25, 356)	(26, 658)	(27, 489)	(28, 163)	

Percorrendo simultaneamente as duas listas à procura de segundos elementos iguais, encontra-se (10, 644) em L_1 e (19, 644) em L_2. Pode-se então calcular

$\log_3 525 = 29*10+19 = 309.$

Verificando, tem-se: $3^{309} \equiv 525 \ mod\ 809$

obtido por *expomod*(3, 309, 809) = 525.

Ou calculando pelo código *logdis*(3, 525, 809, tem-se a resposta:

O logaritmo discreto de 525, *na base* 3, *em módulo* 809 *é* 309.

A complexidade do algoritmo de Shanks é de um tempo em *O(m)*.

5.2.5. SILVER-POHLIG-HELLMAN

O algoritmo que será estudado a seguir é o de Silver-Pohlig-Hellman. Através dele é possível calcular o logaritmo discreto de um número.

Supõe-se que em

$$p-1=\prod_{i=1}^{k} p_i^{c_i} \quad ,$$

p_i são números primos distintos. O valor $a = log_{\alpha}\beta$ é determinado unicamente em módulo p - 1. Podendo-se calcular $a \bmod p_i^{c_i}$ para cada i, $1 \leq i \leq k$, então também pode ser calculado a *mod* $(p$ - $1)$ pela aplicação do Teorema Chinês do Resto. Assim, supondo-se que q é primo,

$$p-1 \equiv 0 \bmod q^{c}$$

e $$p-1 \neq 0 \bmod q^{c+1} \ .$$

Quer-se mostrar como calcular o valor de

$$x = a \ mod q^{c} \qquad \text{onde} \quad 0 \leq x \leq q^{c}-1$$

Pode-se expressar x em função de *raiz* q como sendo

$$x=\sum_{i=0}^{c-1} a_i q^{i} \ , \qquad \text{onde} \quad 0 \leq a_i \leq q-1 \quad \text{para} \quad 0 \leq i \leq c-1$$

Observa-se também que a pode ser expresso por

$$a = x + q^{c} + s, \qquad \text{para } s \text{ inteiro.}$$

A primeira etapa do algoritmo é calcular a_0. A principal observação é que

$$\beta^{(p-1)/q} \equiv \alpha^{(p-1)a_0/q} \bmod p \ .$$

Para ver isto, deve-se notar que

$$\beta^{(p-1)/q} \equiv \alpha^{(p-1)(x+q^{c}s)/q} \bmod p \ .$$

Assim, basta mostrar que

$$\alpha^{(p-1)(x+q^{c}s)/q} \bmod p \equiv \alpha^{(p-1)a_0/q} \bmod p \ .$$

Isso é verdadeiro se e somente se

$$\frac{(p-1)(x+q^{c}s)}{q} \equiv \frac{(p-1)a_0}{q} \bmod (p-1) \ .$$

Tem-se

$$\frac{(p-1)(x+q^c s)}{q} - \frac{(p-1)a_0}{q} = \frac{p-1}{q}(x+q^c s-a_0)$$

$$= \frac{p-1}{q}\left(\sum_{i=0}^{c-1} a_i q^i + q^c s - a_0\right)$$

$$= \frac{p-1}{q}\left(\sum_{i=1}^{c-1} a_i q^i + q^c s\right)$$

$$= (p-1)\left(\sum_{i=1}^{c-1} a_i q^{i-1} + q^{c-1} s\right)$$

$$\equiv 0 \ mod(p-1),$$

que era o que se queria provar.

Portanto, começou-se calculando $\beta^{(p-1)/q} \bmod p$.

Se $\beta^{(p-1)/q} \equiv 1 \bmod p$, então $a_0 = 0$.

Em caso contrário, calcula-se sucessivamente

$$\gamma = \alpha^{(p-1)/q} \bmod p, \gamma^2 \bmod p, \ldots$$

até

$$\gamma^i \equiv \beta^{(p-1)/q} \bmod p \quad \text{para todo } i.$$

Se $c = 1$, está concluído. Se $c > 1$, determina-se a_1. Para tanto define-se

$$\beta_1 = \beta\alpha^{-a_0},$$

e indica-se

$$x_1 = \log_\alpha \beta_1 \bmod q^c.$$

Não é difícil ver que

$$x_1 = \sum_{i=1}^{c-1} a_i q^i.$$

Daqui segue que

$$\beta_1^{(p-1)/q^2} \equiv \alpha^{(p-1)a_1/q} \bmod p$$

Assim, deve-se calcular $\beta_1^{(p-1)/q^2} \bmod p$ e então encontrar i tal que

$$\gamma^i \equiv \beta_1^{p-1)/q^2} \bmod p.$$

Então tem-se $a_1 = i$.

Se $c = 2$ terminou-se agora; em caso contrário, repete-se o processo $(c - 2)$ vezes mais, obtendo $a_2,\ldots,a_{c-1}$.

Exemplo:

Supondo $p = 29$; então $n = p - 1 = 28 = 2^2 * 7^1$.

Supondo ainda $\alpha = 2$ e $\beta = 18$ e que se deseja determinar $a = \log_2 18$. Primeiro calcula-se $a \bmod 4$ e $a \bmod 7$.

Inicia-se com $q = 2$ e $c = 2$. Primeiro,

$$\gamma_0 = 1$$

e

$$\gamma_1 = \alpha^{28/2} \bmod 29 = 2^{14} \bmod 29 = 28.$$

Posteriormente,

$$\delta = \beta^{28/2} \bmod 29 = 18^{14} \bmod 29 = 28.$$

Daqui, $a_0 = 1$. Seguindo, calcula-se

$$\beta_1 = \beta_0 \alpha^{-1} \bmod 29 = 9$$

e

$$\beta_1^{28/4} \bmod 29 = 9^7 \bmod 29 = 28.$$

Conseqüentemente

$$\gamma_1 \equiv 28 \ mod \ 29,$$

tendo-se $a_1 = 1$. Daqui, $a \equiv 3 \ mod \ 4$.

Prosseguindo, fazendo $q = 7$ e $c = 1$, tem-se

$$\beta^{28/7} \bmod 29 = 18^4 \bmod 29 = 25$$

e

$$\gamma_1 = \alpha^{28/7} md 29 = 2^4 \bmod 29 = 16.$$

Então pode-se calcular

$$\gamma_2 = 24$$

$$\gamma_3 = 7$$

$$\gamma_4 = 25.$$

Então $a_0 = 4$ e $a \equiv 4 \ mod \ 7$.

Finalmente, resolvendo o sistema

$$a \equiv 3 \ mod\ 4$$

$$a \equiv 4 \ mod\ 7$$

usando o Teorema Chinês do Resto, obtém-se

$$a \equiv 11 \ mod\ 28.$$

Isto é, calculou-se

$$\log_2 18 \text{ em } Z_{29} \text{ que é } 11.$$

O resultado pode ser confirmado pelo código **logdis** feito no Maple:

logdis(2, 18, 29); que nos fornece a resposta:

O logaritmo discreto de 18, na base 2, em módulo 29, é 11.

6. ASSINATURA DIGITAL

O maior benefício da criptografia de chave pública é que ele proporciona um método empregando assinaturas digitais [ZIM98]. Neste capítulo serão apresentados dois esquemas de assinatura digital: o DSS e o do RSA. Esquemas tais como o de DiffieLamport, o probabilístico de Rabin, o ElGamal, do qual mediante modificações surgiu o DSS, o Bos-Chaum, o Chaum-van Antwerpen Undeniable Signature Scheme, o van Heyst and Pedersen Fail-stop, não serão abordados.

6.1. INTRODUÇÃO

Assinaturas digitais capacitam o receptor da informação verificar a sua autenticidade e integralidade. O uso de um esquema seguro de assinatura prevenirá a possibilidade de falsificação.

Figura 4 - Assinatura digital (extraído de [ZIM98]).

Em uma assinatura convencional, a pessoa que assina o documento encontra-se presente (fisicamente) e a verificação da autenticidade da assinatura é feita por comparação, o que pode não ser muito seguro porque com alguma facilidade alguém pode imitar a assinatura e falsificá-la.

A assinatura digital acontece sem a presença física da pessoa que assina um documento ou uma mensagem. É usado um algoritmo para relacionar a assinatura com a mensagem. A autenticação da assinatura é feita através de um algoritmo de conhecimento público, assim qualquer pessoa pode verificar a assinatura digital.

Um esquema de assinaturas compõe-se de dois elementos básicos: um algoritmo de assinatura e um algoritmo de autenticação.

Supondo que um usuário ao assinar uma mensagem M use um algoritmo secreto de assinatura *sig*. A assinatura resultante *sig(M)* pode ser verificada usando um algoritmo público de autenticação *aut*, retornando *verdadeiro* ou *falso*, dependendo da autenticidade da assinatura.

Reforçando, *aut* é uma função pública e *sig* é secreta.

Dado M, somente uma pessoa é capaz de calcular a assinatura N tal que *aut(M,N) = verdadeiro*. Mas um esquema de assinaturas não é incondicionalmente seguro; dispondo de tempo suficiente e usando o algoritmo público *aut*, alguém pode testar todas as assinaturas possíveis N até encontrar a correta, podendo então falsificar a assinatura. Como opção para eliminar falsificações, podem ser usadas funções que confundem (*hash*) juntamente com um esquema de assinaturas.

Vamos exemplificar supondo que um usuário A deseja enviar uma mensagem codificada assinada M. Primeiro o usuário A calcula sua assinatura $N = sig_A(M)$ e então codifica M e N usando a função codificadora de B, representada por c_B, obtendo $P = c_B(M, N)$. A mensagem codificada P será transmitida para B. Quando B recebe a mensagem P, primeiro a decodifica usando a sua função decodificadora d_B descobrindo *(M, N)*, depois usando a função pública de autenticação de A, verifica se $aut_A(M, N) = verdadeiro$..

6.2. DIGITAL SIGNATURE STANDARD - DSS

Em 1991 o National Institute of Standards and Technology (NIST) do governo dos Estados Unidos propôs a Assinatura Digital Padrão (Digital Signature Standard - DSS), cujo objetivo é elaborar um método padrão de assinatura digital para uso do governo e de organizações comerciais. NIST escolheu por base de seu esquema de assinaturas o problema do logaritmo discreto em um corpo finito primo.

Para montar o esquema capaz de assinar mensagens, o DSS procede para cada usuário como segue:

(1) escolhe um número primo q de cerca de 160 bits (para fazer isto, ele usa um gerador randômico de números e um teste de primalidade);

(2) depois escolhe um segundo primo p que é $\equiv 1 \bmod q$ e tenha cerca de 512 bits;

(3) escolhe um gerador do único subgrupo cíclico F_p^* de ordem q (calculando $g_0^{(p-1)/q} \bmod p$ para um inteiro randômico g_0; se este número é diferente de um, ele será um gerador);

(4) toma um inteiro randômico x no intervalo $0 < x < q$ como sua chave secreta, e põe sua chave pública igual a $y = g^x \bmod p$.

Os procedimentos de A para assinar uma mensagem são os seguintes: primeiro aplica uma função "mistura" (hash) para seu texto pleno, obtendo um inteiro h nos limites $0 < h < q$. Depois seleciona um inteiro randômico k no mesmo intervalo, calculando $g^k \bmod p$, considera r igual ao menor resíduo não-negativo *módulo q* do último número (isto é, g^k é primeiro calculado em *módulo p*, e o resultado é então reduzido *módulo o menor primo q*). Finalmente, A encontra um inteiro s tal que $sk \equiv h + xr \bmod q$. Sua assinatura é então o *par (r, s)* de inteiros *módulo q*.

Para verificar a assinatura, o receptor B calcula $u_1 = s^{-1}h \bmod q$ e $u_2 = s^{-1}r \bmod q$. Depois calcula $g^{u_1} y^{u_2} \bmod p$. Se o resultado está de acordo com *módulo q com r*, ele está satisfeito. Note que

$$g^{u_1} y^{u_2} = g^{s^{-1}(h+xr)} = g^k \bmod p .$$

Esse esquema de assinatura tem a vantagem de que as assinaturas são razoavelmente curtas, consistindo de dois números de 160 bits (a magnitude de q). Por outro lado, a segurança do sistema parece depender da intratabilidade do problema do logaritmo discreto em um corpo

F_p. Embora para quebrar o sistema seja suficiente encontrar logaritmos discretos no menor subgrupo gerado por g, na prática isso parece não ser mais fácil do que achar arbitrariamente logaritmos discretos em F_p^*. Assim, o DSS parecer ter alcançado um nível razoavelmente alto de segurança, sem sacrificar o armazenamento de pequenas assinaturas e tempo de implementação [KOB94].

6.3. ASSINATURA DIGITAL NO RSA

A seguir será visto como funciona o esquema de assinaturas no cripto-sistema de chave pública RSA.

Chamando a chave pública do usuário I por (e_I, n_I) e a chave privada de d_I, então o procedimento de codificação de A que deseja remeter para B uma mensagem assinada M é a seguinte: A, primeiro calcula $S = M^{d_A} \bmod n_A$ e então codifica S. Chamando a codificação de C, onde

$$C = S^{e_B} \bmod n_B .$$

Ao receber a mensagem C, o usuário B decodifica C usando

$$S = C^{d_B} \bmod n_B \quad \text{e} \quad M = S^{e_A} \bmod n_A .$$

O usuário B então tem a mensagem assinada *(M, S)* em sua posse.

Há vários pontos a considerar. Ao lado do fato de que - para que o esquema funcione - $e(d(x)) = x$ deve manter-se, deve também ser verdadeiro que a assinatura S calculada por A está dentro do alcance do procedimento de codificação.

Esta última condição certamente pode não manter-se quando o sistema usado é o RSA; a assinatura S pode ser um inteiro maior que a chave pública n_B. Pode-se sempre evitar isso pelo rebloqueamento: em outras palavras, reajustando o tamanho da mensagem em blocos de tal maneira que estejam dentro do alcance requerido. Entretanto, uma solução mais elegante proposta por Rivest, Shamir e Adleman (1978) é a seguinte:

Um valor inicial h é adotado para o sistema de chave pública (por exemplo $h \sim 10^{199}$). Cada usuário então mantém dois pares de chave pública, um para codificação e outro para verificação de assinatura. Representando essas por (e_I, n_I) e (f_I, m_I) respectivamente, quando I

é o usuário em questão. A regra adotada é que o módulo de codificação n_I e o módulo de assinatura m_I de cada usuário I devem satisfazer

$$m_I < h < n_I .$$

Então, se g_A é a chave privada de A, correspondente ao seu par de assinatura (f_A, m_A), o protocolo modificado no exemplo será obtido calculando S por

$$S = M_A^g \bmod m_A .$$

Então como antes, $C = S_B^e \bmod n_B$, $S = C_B^d \bmod n_B$

mas B é redescoberto por $M = S_A^f \bmod m_A$.

É fácil verificar que, para esse sistema trabalhar e permitir a assinatura e autenticação de mensagens para todos os usuários do sistema, todos necessitam de que o conjunto das mensagens M satisfaça a condição

$$0 \le M \le min\ m_I ,$$

onde o mínimo do lado direito é global para os usuários I do sistema.

Na prática há inconvenientes para este sistema de assinaturas. Enumerando três:

(a) O emitente A pode, deliberadamente, `perder` sua chave privada, assim que, a menos que ela esteja depositada em um `banco de chave privada` antes do início do sistema, as mensagens enviadas por ele podem tornar-se inverificáveis.

(b) O emitente A pode, deliberadamente, exibir sua chave privada d_A e, assim fazendo, permitir que todas as assinaturas digitais atribuídas a ele tornarem-se questionáveis.

(c) O tempo envolvido em codificação, assinatura, decodificação e verificação pode ser excessivo, tornando as informações vulneráveis e o esquema de assinatura não tão seguro como se necessita.

7. PGP (Pretty Good Privacy = Privacidade bastante boa)

Como o PGP é a mais recente espécie de criptografia, justifica-se a sua inclusão neste trabalho.

O PGP é uma aplicação e protocolo para a segurança de e-mail e codificação de arquivo desenvolvido por Philip R. Zimmermann. Originalmente publicado como produto livre (*Freeware*), o código fonte tem sido sempre acessível para exame público minucioso. PGP usa uma variedade de algoritmos, entre eles o RSA, para a codificação, autenticação, integridade de mensagem, e controle de chave. PGP está baseado no modelo "Web-of-Trust" e tem distribuição mundial [ZIM98].

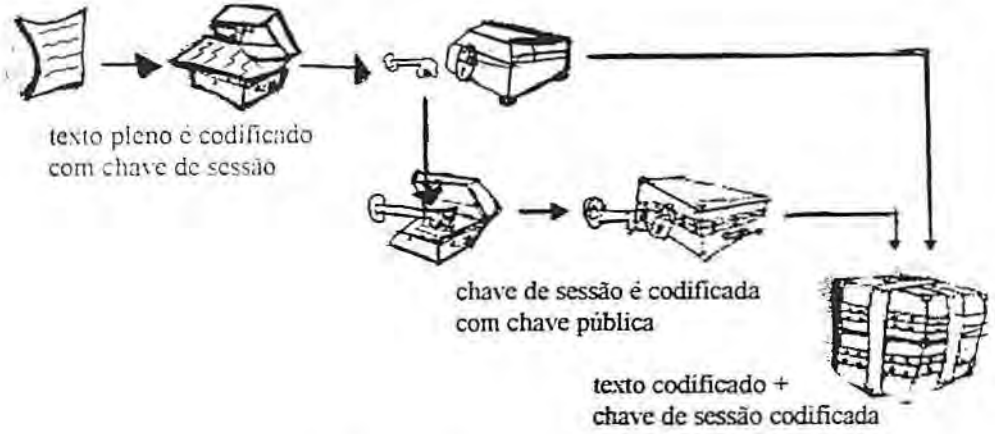


Figura 5 - Como o PGP codifica (extraído de [ZIM98])..

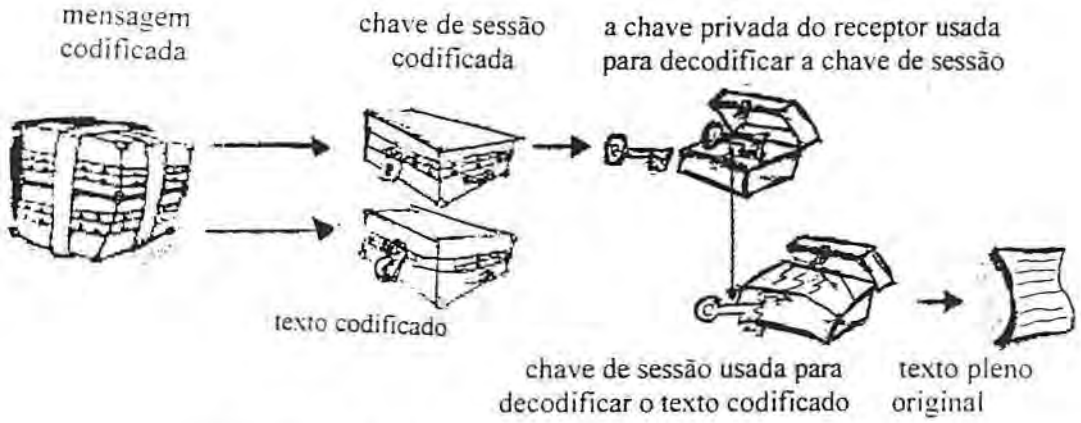


Figura 6 - Como o PGP decodifica (extraído de [ZIM98]).

Para maiores detalhes recomendamos a leitura do capítulo 2 de [ZIM98], escrito pelo próprio autor, onde ele apresenta uma introdução e informações básicas sobre a criptografia PGP. Justifica porque criou o sistema PGP: a partir de uma lei de 1991 do Congresso dos EUA que determinava que fabricantes ligados a equipamentos de comunicação introduzissem em seus equipamentos alçapões (*trap doors*) para que o governo pudesse ler quaisquer mensagens codificadas. Relata as dificuldades e conflitos gerados entre as indústrias e civis. Faz referencia à telefonia digital em 1994, que instalara uma conexão secreta para interceptar mensagens telefônicas; cita o envolvimento do FBI. Relata que em abril de 1993, durante o governo de Clinton, a iniciativa desenvolvida por NSA (National Security Agency), chamada *Clipper chip*, o governo encorajou as indústrias privadas a desenvolver em seus produtos uma comunicação segura, onde afirma que "para tornar o *Clipper* completamente efetivo, a próxima etapa lógica seria declarar ilegal outras formas de criptografia". Cita a administração Bush. Refere-se a um documento de relato de fatos intitulado "*Encryption: The Threat, Applications and Potential Solutions*", enviado para o Conselho Nacional de Segurança em fevereiro de 1993, o FBI, NSA e o Departamento de Justiça (DOJ). Zimmermann conclui a sua introdução de "*Why I wrote PGP*" afirmando que "Por isso, usar PGP é bom para a preservação da democracia". E "PGP dá poderes a pessoas para tomar sua privacidade em suas próprias mãos. Isso é um desenvolvimento social necessário para ela. É por isso que eu o criei."

Cabem, por isso mesmo, algumas considerações a respeito do PGP.

O PGP é um cripto-sistema. Está baseado no conceito de algoritmo de domínio público com chave dupla, de alta segurança. Isso significa que o algoritmo de criptografia por si só não permite a decodificação. É um cripto-sistema híbrido que combina algoritmos de chaves públicas com algoritmos convencionais, com a vantagem de utilizar a velocidade da criptografia convencional e a segurança da criptografia por chaves públicas.

O PGP é um programa que fornece privacidade à correspondência eletrônica. Ele criptografa a correspondência de tal modo que somente a pessoa pretendida pode lê-la. O PGP também pode ser utilizado para aplicar uma assinatura digital a uma mensagem sem criptografá-la. Uma vez criada uma assinatura digital, é impossível para qualquer um modificar tanto a mensagem quanto a assinatura sem a modificação ser detectada pelo PGP.

Quando um usuário codifica um texto pleno com PGP, PGP primeiro comprime o texto pleno. A compressão de dados economiza tempo de transmissão por *modem* e espaço no disco e - mais importante - fortalece a segurança da criptografia. A maioria das técnicas de criptoanálise utilizam modelos de texto pleno para quebrar o código. A compressão reduz esses modelos no texto pleno, aumentando intensamente a resistência para a criptoanálise.

PGP comprime o texto pleno antes de codificá-lo, porque é tarde demais para comprimir o texto pleno após ter sido codificado; dados codificados não são mais passíveis de compressão. Arquivos que são pequenos demais para serem comprimidos não são comprimidos pelo PGP, mas o programa reconhece arquivos produzidos por programas mais populares de compressão, tais como o PKZIP, e não tenta comprimir um arquivo que já foi comprimido.

PGP cria uma chave de sessão (*session key*), que é uma chave secreta de um tempo somente (*one-time-only*). Esta chave temporária é um número randômico gerado pelo movimento aleatório do *mouse* e o toque a uma tecla. Para cada sessão de comunicação de dados é usada uma chave de sessão diferente. Esta chave de sessão trabalha com muita segurança, apoiada em algoritmo convencional de codificação para codificar textos plenos; o resultado é um texto codificado.

Todos os dados são codificados, a chave de sessão é então também codificada para a chave pública do receptor. Essa chave pública codificada que contém a chave de sessão é transmitida junto com o texto codificado para o receptor.

A decodificação trabalha no sentido contrário. A cópia do receptor de PGP usa sua chave privada para recuperar a chave de sessão temporária, que o PGP então usa para decodificar o texto codificado pela codificação convencional.

Chaves são armazenadas sob forma codificada. PGP armazena as chaves em dois arquivos no disco rígido: um para a chave pública e outro para a chave privada. Esses arquivos são chamados chaveiros (*keyrings*). Quando se usa PGP, basicamente se junta a chave pública do receptor com a chave pública do emitente. Cada chave privada fica armazenada em seu chaveiro. Ao se perder o chaveiro privado passa-se a ser incapaz de decodificar qualquer informação codificada por chaves deste chaveiro.

Resumindo, o PGP trabalha da seguinte forma:

Para codificar:

1. o texto pleno é codificado com uma chave de sessão;
2. a chave de sessão é codificada com a chave pública do destinatário;
3. remete-se o texto codificado juntamente com a chave de sessão codificada.

Para decodificar:

1. a mensagem codificada recebida é separada em chave de sessão codificada e texto codificado;
2. o receptor usa a chave privada para decodificar a chave de sessão;
3. usa a chave de sessão para decodificar o texto codificado, retornando assim ao texto pleno original.

Mesmo considerando toda a segurança fornecida pelo PGP, convém citar a frase que consta na página 28 de [ZIM98]: "Diz-se que um segredo não é um segredo se ele é conhecido por mais de uma pessoa".

8. CONCLUSÕES

Nas suas aplicações, além do processo de codificar-decodificar com integridade, qualquer cripto-sistema precisa oferecer segurança. A isso várias referências foram feitas ao longo deste trabalho. Se, por um lado, a criptografia não elimina a necessidade da confiança em certas pessoas em última análise, por outro lado, permite reduzir o conhecimento privilegiado de informações e o universo de pessoas com esse conhecimento privilegiado.

Disse uma vez, Benjamin Franklin, o inventor do pára-raios, que '*três pessoas só conseguem guardar um segredo, se duas delas já estiverem mortas*' [COU97].

No que diz respeito a implementações práticas, a confidência dos usuários é essencial; portanto, à luz dos resultados de Adleman-Shamir, nenhum sistema baseado no problema *knapsack* parece ter, no futuro, uma viabilidade comercial ou prática. Entretanto, por razões históricas, e por sua função em dar compreensão no que exatamente é necessário em um sistema de chave pública ele tem sido estudado.

Por outro lado, conforme notícias divulgadas pela imprensa, os *hackers* trabalham exaustivamente. No programa Fantástico, da Rede Globo, do dia 12 de março de 2000, foi apresentada a atividade de um *hacker*: "invadiu" as informações do Ministério de Saúde do Brasil, mas optou por deixar somente um OI na tela principal, saindo sorrateiramente sem deixar qualquer vestígio. Evidentemente este *hacker* não quis provocar grandes prejuízos. Pensando em minimizar este tipo de ação, empresas que atuam no ramo da segurança das informações contratam, segundo a imprensa, *ex-hackers* para trabalhar em projetos de segurança. Até que ponto são confiáveis ?

Cabe citar que os cripto-sistemas apresentados: baseado no knapsack, RSA e logaritmo discreto, surgiram no final da década de setenta. A partir daí, cripto-sistemas de chave pública tiveram um grande desenvolvimento.

Entretanto, paralelamente ao desenvolvimento dos cripto-sistemas, no intuito de oferecer cada vez uma maior segurança apelando para números grandes, surgem os desafios visando quebrar as questões ou funções tidas de muito difícil solução, entre as quais está a fatoração.

Fatorar números grandes tem sido um desafio: uma questão famosa foi RSA-129, isto é, fatorar um número de 129 dígitos proposta em agosto de 1977. Pelos conhecimentos e tecnologia existentes na época, imaginavam que a solução demoraria muito tempo (20.000 anos), mas a questão (já) foi solucionada em abril de 1994, demorando, mesmo assim, perto de 17 anos. Para

maiores detalhes sobre esta interessante questão, sugerimos a leitura das páginas 90-99, volume 3 de [CIP96].

Em 1995 foi fatorado um número de 309 dígitos, resultante do número de Fermat $F_{10} = 2^{2^{10}} + 1$ em que um dos fatores primos tem 40 dígitos e o outro 252 dígitos. Em setembro de 1998 chegou a vez do número de Mersenne 2^{677} - 1 ser fatorado. Em fevereiro de 1999 foi resolvido o problema RSA140, isto é, o número de 140 dígitos foi fatorado resultando dois números primos de 70 dígitos cada. Em agosto de 1999 foi resolvido o RSA155 [BRE99].

O problema RSA129, mencionado neste trabalho, nos dá algumas lições relativas à segurança. Em 1977, quando foi lançado, certamente não se imaginava que os computadores evoluíssem tão rapidamente, com destaque à queda no custo, o que permitiu uma grande proliferação dos computadores; a velocidade cada vez mais crescente com que realiza operações, tidas há pouco tempo, como impossíveis; a exatidão das respostas quando possíveis e, em especial, o surgimento da INTERNET. Graças a esta evolução, o problema base do sistema de criptografia RSA, previsto para ser resolvido em vinte mil anos, foi resolvido em dezessete anos. Volta-se à questão: até que ponto o cripto-sistema RSA é seguro ? Uma alternativa que pode reduzir a possibilidade de fatorar um número que é o produto de dois números primos, além do tamanho dos números e dos cuidados já mencionados, é a troca freqüente dos fatores primos.

Considere-se que, apesar de existirem métodos eficazes para a fatoração, eles perdem sua eficiência quando os números são grandes. Não se tem, até o momento, o conhecimento de um algoritmo capaz de fatorar rapidamente um número grande; mas isto não significa que não possa ser descoberto. Não foi provado que tal algoritmo não pode existir. Evidentemente que, havendo um algoritmo para a rápida fatoração de um número grande, o cripto-sistema RSA perde a sua função *trapdoor* e deixa de ter aplicações práticas.

Uma boa medida da segurança do RSA é dada pelo governo americano ao classificar a tecnologia de criptografia como "equipamento militar". Isto impõe barreiras severas à sua exportação [COU97].

O primeiro cripto-sistema de chave pública que foi publicado, o algoritmo Diffie-Hellman de troca de chave, era baseado no conceito de que calcular o logaritmo discreto é difícil. Esta hipótese de intratabilidade é também a base para a suposta segurança de uma variedade de outros esquemas de chave pública. A par do substancial avanço nos algoritmos de logaritmo discreto nas últimas duas décadas, em geral o logaritmo discreto ainda parece ser difícil, especialmente para alguns grupos, como os de curvas elípticas. Infelizmente não há provas disponíveis da dificuldade nesta área; assim, é necessário confiar na experiência e intuição no julgamento dos parâmetros usados no cripto-sistema [ODL99].

Uma ameaça que paira sobre o cripto-sistema do logaritmo discreto e RSA está relacionada com o aumento na quantidade e velocidade dos computadores que estão sendo construídos, permitindo uma explosão na investigação, regularmente sugerida, sobre a fatoração de inteiros e logaritmos discretos, aproveitando o potencial fornecido pelo tempo ocioso de computadores na INTERNET, que facilmente pode ser aproveitada.

Os *experts* concordam que ainda demorará muitos anos para que os modernos sistemas de chave pública sejam realmente ameaçados, entretanto estes avanços deverão servir de aviso acerca das necessidades de desenvolver e dispor de sistemas alternativos [ODL99].

Técnicas desenvolvidas por Pollard e Shamir têm-se mostrado as melhores possíveis. No método "*index calculus*" tem havido progressos consistentes [ODL 99].

Mesmo assim, projetistas de sistemas sugerem que o tamanho das chaves para o cripto-sistema RSA e corpo finito do logaritmo discreto forneça uma generosa margem extra para a segurança, compensando uma eventual antecipação provocada pela potência de cálculos dos computadores ou no desenvolvimento de algoritmos.

A principal razão para proporcionar abundantes margens de segurança é que uma inesperada nova perspicácia matemática é a maior ameaça em potencial para ambos, logaritmo discreto e fatoração de inteiros. Um outro ponto é notar que o número de pessoas que têm trabalhado seriamente na fatoração de inteiros e logaritmo discreto não é tão elevado. Grande parte das idéias realmente novas (tais como o *rho* e método $p - 1$, o número básico do corpo "*sieve*", e "*lattice sieving*") têm surgido de um único indivíduo, John Pollard. Isto sugere que a área simplesmente não tem sido explorada completamente e nem minuciosamente como é muitas vezes alegado, e isto pode ser ainda um depósito de surpresas para nós [ODL99].

Considerando os avanços tenológicos e comparando um corpo finito de inteiros módulo um primo p, ou usando um inteiro composto n de mesmo tamanho, quebrar o logaritmo discreto módulo p mostra-se um pouco mais difícil do que fatorar o inteiro n. Além disso, o cripto-sistema da curva elíptica fornece a possibilidade de usar uma chave com tamanho muito menor do que requerido pelo RSA com uma segurança comparável [ODL99].

Com o vertiginoso crescimento do comércio eletrônico, via INTERNET, a segurança das informações adquire uma importância vital. Cripto-sistemas de chave pública são instrumentos valiosos de segurança. Propiciam essencialmente o único caminho para fornecer assinaturas digitais e é muitas vezes o método preferido para a autenticação ou distribuição de chaves.

O PGP, programa que fornece privacidade à correspondência eletrônica, criptografa a correspondência de tal modo que somente a pessoa autorizada possa ler. O PGP pode também ser utilizado para aplicar uma assinatura digital a uma mensagem sem criptografá-la. Uma vez criada uma assinatura digital, é impossível para qualquer um, modificar tanto a mensagem quanto a assinatura sem a modificação ser detectada pelo PGP. Como o PGP é relativamente novo (ver [ZIM98]), as versões do PGP ainda não se encontram muito difundidas e, conseqüentemente, são pouco usadas.

Considerando os cripto-sistemas estudados, chegou-se às seguintes conclusões básicas:

(i) o cripto-sistema baseado no knapsack, embora interessante como estudo, não oferece mais a segurança exigida em troca de informações;

(ii) o cripto-sistema RSA tem-se mostrado seguro. Sugere-se, mesmo assim, utilizar números primos grandes, de 200 dígitos. Evidentemente, se for descoberto um eficiente algoritmo para a fatoração de inteiros, o RSA estará quebrado;

(iii) o cripto-sistema baseado no logaritmo discreto apresenta-se seguro. É ainda usado como base para a pesquisa de novos métodos de criptografia.

Comparando a questão das chaves, tem-se, no sistema RSA, que o usuário A, tendo gerado a chave, a transmite para B, usando a sua chave pública. Um espião pode obrigar B a lhe revelar sua chave privada e então decodificar a mensagem enviada de A para B. Se A e B, usassem o sistema de Diffie-Hellman baseado no logaritmo discreto para gerar as chaves, destruindo-as depois de encerrada a sessão, e não guardando sua comunicação, ninguém seria capaz de descobrir qual a mensagem trocada.

A novidade fornecida pelo cripto-sistema PGP, combinando o sistema convencional com a criptografia de chave pública, criando assim um sistema híbrido, de fato fornece uma segurança e privacidade excepcional. É a última descoberta em criptografia.

Pode-se prever que, a par do desenvolvimento dos computadores, também em quantidade, a segurança dos cripto-sistemas está relacionada com o tamanho dos números, ou seja, a quantidade de dígitos nos sistemas RSA e baseada no logaritmo discreto. Uma ameaça é a descoberta de algum algoritmo eficiente que realize a fatoração ou o cálculo do logaritmo discreto.

Embora as pesquisas estejam ligadas aos sistemas já existentes, num sentido ao logaritmo discreto e noutro ao PGP, novos cripto-sistemas poderão ser descobertos. Afinal, os que ainda estão em uso já têm mais de vinte anos.

APÊNDICE

Neste apêndice estão, em Maple, todos os programas elaborados para resolver questões referentes ao estudo feito sobre Criptografia, com alguns exemplos da utilização.

1 - O código **mdc** calcula o máximo divisor comum entre dois números pelo método de Euclides.

```
mdc:=proc(x, y)
# Cálculo do máximo divisor comum entre dois
# números pelo método de Euclides.
# INPUT: x --> um número
#          y --> o outro número
# OUTPUT: o mdc.
local w, z, r;
w := x;
r := 1;
z := y;
        while r > 0 do
        r := irem(w, z);
        w := z;
        z := r;
                if z = 0 then break
                fi;
        od;
w;
end:
```

Exemplos:

1.1. mdc(98, 59);

1

1.2. mdc(123456789, 987654321018);

9

2 - Para encontrar dois números relativamente primos, elaborou-se o código **relpri**. Dentro deste programa encontra-se o código mdc que é preciso estar na memória do computador antes de chamar o relpri. Em caso contrário o relpri não atende.

```
relpri:=proc(n)
# Para calcular dois números relativamente
# primos.
# Cada um com n dígitos.
# Antes precisa carregar o código mdc.
# INPUT: n --> a quantidade de dígitos
# OUTPUT: N e w --> números relativamente primos.
local N, w;
N := 2; w := 4;
        while mdc(N, w) <> 1 do
        N := rand(10^(n-1)...10^n); N := N();
        w := rand(10^(n-1)...10^n); w :=  w();
                if mdc(N, w) = 1 then print(`Números relativamente
        primos:`, N, w);
                 fi;
        od;
end:
```

Exemplos:

2.1. relpri (2);

Números relativamente primos:, 98, 59

2.2. relpri (3);

Números relativamente primos:, 599, 155

3 - Para encontrar um número relativamente primo a um número dado, com uma quantidade dada de dígitos construiu-se o código **relprium**. Dentro dele encontra-se o código mdc que, por isso, é preciso chamar antes, senão o relprium não atende.

```
relprium:=proc(n, d)
# Para calcular um número relativamente
# primo a um número dado.
# Carregar antes o código mdc.
# Com n dígitos.
# INPUT: n --> o número dado
#          d --> a quantidade de dígitos
# OUTPUT: w --> número relativamente primo a n,
#                  com d figuras.
local w;
w := 2;
        while mdc(n, w) <> 1 do
        w := rand(10^(d-1)...10^d); w := w();
                if mdc(n, w)=1 then print(`Número relativamente
        primo:`, w);
                fi;
        od;
end:
```

Exemplos:

3.1. relprium(192, 2);

Número relativamente primo:, 65

3.2. relprium(87654, 4);

Número relativamente primo:, 8821

Para testar o exemplo 3.2.

mdc(87654, 8821);

1

4 - O código **mdceu** calcula o máximo divisor comum entre dois números pelo método de Euclides estendido e calcula o inverso de um número *b* em módulo *a* quando o *mdc(a, b)=1*. Não pode ser usado dentro de outro código devido a forma como fornece as respostas.

```
mdceu:=proc(a, b)
# Cálculo do máximo divisor comum pelo método de
# Euclides estendido e do inverso de um número b
# em módulo a quando o mdc(a, b) = 1.
# Não pode ser usado dentro de outro código
# devido à forma como fornece as respostas.
# INPUT: a --> um número
#          b --> o outro número
# OUTPUT: o máximo divisor entre a e b.
#              O inverso de b em módulo a quando possível.
local q, t, u, v;
u := [1, 0, a];
v := [0, 1, b];
        while v[3] > 0 do
        q := floor(u[3]/v[3]);
        t := u - v*q;
        u := v;
        v := t;
        od;
                if u[2] < 0 then u[2] := a + u[2]
                fi;
print(`O mdc é` ,u[3]);
        if u[3] = 1 then
        print(`O inverso de `, b, `em módulo`, a, `é`, u[2]);
        fi;
end:
```

Exemplos:

4.1. mdceu(61, 17);

O mdc é, 1

O inverso de, 17, em módulo, 61, é, 18

4.2. mdceu(199800, 119119);

O mdc é, 1

O inverso de , 119119, em módulo, 199800, é, 61279

4.3. mdceu (54, 36);

O mdc é, 18

5 - O código **mdceuinv** calcula, pelo método de Euclides estendido o inverso de *b* em módulo *a* quando *mdc(a, b) = 1*. Em caso contrário, informa o erro. Este algoritmo pode ser usado dentro de outro, porque fornece somente o resultado numérico.

```
mdceuinv:=proc(a, b)
# Cálculo do inverso de b em módulo a, quando
# a e b são números primos entre si.(mdc(a, b) = 1).
# Pelo método de Euclides estendido.
# INPUT: a --> um número
#        b --> o outro número
# OUTPUT: o inverso de b em módulo a
local q, t, u, v;
u := [1, 0, a];
v := [0, 1, b];
        while v[3] > 0 do
        q := floor(u[3]/v[3]);
        t := u - v*q;
        u := v;
        v := t;
        od;
                if u[3] = 1 then
                        if u[2] < 0 then u[2] := a + u[2]
                        fi;
                u[2];
                else print (`ERRO. Os números não são primos entre
                si.`);
                 fi;
end:
```

Exemplos:

5.1. mdceuinv(841, 160);

205

5.2. mdceuinv(54, 36);

ERRO. Os números não são primos entre si.

6 - Para calcular potências em módulo, elaborou-se o código **expomod.** É o código mais usado, tendo aplicações em todos os cripto-sistemas estudados.

```
expomod:=proc(a, n, m)
# Cálculo de potências em módulo.
# INPUT: a --> base
#          n --> expoente
#          m --> módulo
# OUTPUT: a potência em módulo.
local i, e, x;
i := n;
e := 1;
x := a;
        while i > 0 do
                if irem(i, 2) = 1 then e := e*x; e := irem(e, m);
                fi;
        x := x^2; x := irem(x, m);
        i := floor(i/2);
        od;
e;
end:
```

Exemplo:

6.1. expomod(153, 9, 10);

3

6.2. expomod(5, 3, 6);

5

6.3. a := expomod(1234567899, 55, 240);

a := 99

7 - Considerando o pequeno teorema de Fermat: "Seja n um número primo. Então $a^{n-1} \bmod n = 1$ para qualquer inteiro a tal que $1 \leq a \leq n - 1$", tem-se o código **Fermat**. Para que este código funcione é necessário antes chamar o código expomod, porque ele se encontra dentro do Fermat.

```
Fermat:=proc(n)
# Pequeno teorema de Fermat.
# Pierre de Fermat (1640).
# Se n é primo, então a^(n-1)mod n = 1
# para qualquer inteiro a, 1<= a <= n-1.
# Página 344 - Fundamentals of algorithmics de
# Billes Brassard e Paul Bratley.
# Antes carregar o código expomod.
# INPUT: n --> um número qualquer.
# OUTPUT: se o número é primo ou composto.
local a;
a := rand(1..(n-1)): a();
        if expomod(a(), n-1, n) = 1 then RETURN (primo);
        fi;
RETURN (composto);
end:
```

Exemplos:

7.1. Fermat(1999);

primo

7.2. Fermat(561);

primo

7.3. Fermat(651693055693681);

primo

7.4. Fermat(5693681);

composto

8 - Para comprovar que o número citado de 15 dígitos é composto e que no exemplo 7.3. Fermat(651693055693681) respondeu como primo, elaborou-se o código **fatora** com a idéia trivial da fatoração, para fornecer o menor fator primo de um número.

```
fatora:=proc(n)
# Fatora o número ímpar n pelo método trivial.
# INPUT: n --> número a ser fatorado.
# OUTPUT: o menor fator.
local l, k;
l := floor(sqrt(n))+1;
        for k from 3 by 2 to l do
                if irem(n, k) = 0 then print(`O menor fator é`, k); break
                fi;
        od;
end:
```

Exemplo:

8.1. fatora(651693055693681);

O menor fator é, 72931

9 - O código que determina se $a \in B(n)$ implementado no Maple é o **Btest**. Isto é se o número a escolhido é ou não é falsa testemunha da primalidade de n. Exige que seja chamado o código expomod que se encontra dentro do Btest.

```
Btest:=proc(a, n)
# Fermat modificado.
# Antes carregar o código expomod.
# INPUT: a --> 2<= a <= n - 2
#          n --> número cuja primalidade é testada
# OUTPUT: verdadeiro --> a é falsa testemunha
#              falso --> a não é falsa testemunha
# O a escolhido pode ser falsa testemunha.
# Páginas 345 e 346 de Fundamentals of
# Algorithmics de Gilles Brassard e Paul Bratley.
local s, t, x, i;
s := 0;
t := n - 1;
        while modp(t, 2) <> 1 do s := s + 1; t := iquo(t, 2);
        od;
x := expomod(a, t, n);
        if x = 1 then RETURN (verdadeiro);
        fi;
        if x = n - 1 then RETURN (verdadeiro);
        fi;
        for i from 1 to (s - 1) do x := modp(x^2, n);
                if x = n - 1 then RETURN (verdadeiro);
                fi;
        od;
RETURN (falso);
end:
```

Exemplos:

9.1. Btest(158, 289);

verdadeiro

9.2. Btest(7, 561);

falso

10 - Baseado no teorema de Miller-Rabin; construi-se o código **MilRab** com um número variável *k* de laços (loops) para possibilitar um máximo de fidelidade no resultado. Exige que antes se carregue o código Btest porque ele se encontra dentro do MilRab.

```
MilRab:=proc(n, k)
# Teste de primalidade de Miller-Rabin.
# Antes precisa carregar o código Btest( e expomod).
# INPUT: n --> número a ser testado.
#        k --> quantidade de repetições.
# OUTPUT: Se o número n é primo ou composto.
# Página 347 de Fundamentals of Algorithmics de
# Gilles Brassard e Paul Bratley.
local i, a;
        for i from 1 to k do a := rand(2..(n-2)); a();
                if Btest(a(), n) = falso then RETURN (composto);
                else RETURN (primo);
                fi;
        od;
end:
```

Exemplos:

10.1. MilRab(561, 20);

composto

10.2. MilRab(651693055693681, 5);

primo

10.3. MilRab(65537, 5);

primo

10.4. MilRab(4294967297, 5);

composto

11 - O código **sorpri** foi construído para gerar números primos com *n* dígitos, apoiado no MilRab com 10 loops, que por sua vez está apoiado no Btest e este no código expomod, devendo todos estes serem carregados antes de chamar o sorpri. Para confirmar, nos exemplos, os resultados usou-se o comando isprime do Maple que responde se o número considerado é primo ou não.

```
sorpri:=proc(n)
# Sorteia um número primo com n dígitos.
# Antes carregar MilRab(Btest(expomod))).
# INPUT: n --> a quantidade de dígitos desejada
# OUTPUT: o número primo.
local a;
a := 10^(n-1);
        while MilRab(a(), 10) = composto  do
        a := rand(10^(n-1)..10^n); a := a();
        od;
a();
end:
```

Exemplos:

11.1. p := sorpri(10);

p := 8580037327

isprime(p);

true

11.2. q := sorpri(6);

q := 909401

Testando isprime(q);

true

12 - O código **RSAemi** está baseado no sistema de criptografia RSA. Executa os passos que são dados pelo usuário que pretende remeter (emitir) uma mensagem. Como dentro dele está o sorpri, exige que todos os seguintes programas sejam carregados: expomod, Btest e MilRab, além do mdc e mdceuinv.

```
RSAemi:=proc(k, M::string)
# Determina os números primos p e q.
# Calcula o seu produto: n = pq.
# Calcula fi(n) = fiden = (p - 1)(q -1).
# Calcula a chave privada d e a pública e.
# Antes carregar sorpri(MilRab(Btest(expomod))),
# mdc e mdceuinv.
# INPUT: k --> a quantidade de dígitos de um
#                 número primo.
#          M --> a mensagem (entre aspas simples).
# OUTPUT: n --> o produto de p por q.
#              d --> chave privada.
#              e --> chave pública.
#              Cod --> a mensagem M codificada.
local l, p, q, n, fiden, d, e, V, y, m, j, men, Cod;
l := length(`M`);
p := sorpri(k);
q := sorpri(k + 2);
n := p*q; print(`n :=`, n);
        if length(n) < 3*l then print (`A mensagem deve ser dividida em
blocos.`); break
        fi;
fiden := (p-1)*(q-1);
d := 10^(l-1):
   while mdc(fiden(), d()) <> 1 do
   d := rand(2..fiden()); d := d();
   od:
print(`Privada d := `,d);
e := mdceuinv(fiden(), d()); print(`e :=`, e);
V := convert(M,bytes); y := nops(V); m := 0: j := 0:
   while y > 0 do V[y] := V[y]*1000^j:
   m := V[y] + m: y := y - 1: j := j + 1:
   od:
men := m;
Cod := expomod(men, e, n): print(`Cod := `, Cod);
end:
```

Exemplos:

12.1. RSAemi(10, Ruy);

n:=, 4465711375215138428483
Privada d:=, 233044684618708720309 1
e:=, 2854418702169516714011
Cod:= , 3148123474221157883934

12.2. RSAemi(5, `Hoje cedo eu vou trabalhar`);

n:=, 346906840337
A mensagem deve ser dividida em blocos.

12.3. RSAemi(60, `Dia 5 de abril, irei a Porto Alegre`);

n:=, 7468395739175166880171907599496950926682477313155929402845945 4\
769123009468760148827799430301097802367863503829906652266 69

Privada d:=, 528148033227222836932281109760708069449113254938503083\
806431988816935908160243138782398734304724444821033086822491 74\
19657

e:=, 642596187119012959021233768136732470833363157246807887229855 22\
965987602852225233377337643844873840277804246967755325788 97

Cod:= , 222015140161694349971983146916005798612172537917210279190 09\
740600000168357308719306495640512513102258473145740344536904 43

13 - O código **RSAdes** se destina ao usuário do sistema de criptografia RSA que vai receber a mensagem codificada. É o destinatário da mensagem e precisa decodificá-la. Ao receber a mensagem codificada (*Cod*), tendo a sua chave secreta (*d*) e de posse de *n* que é público, entra com estes dados no RSAdes, nesta ordem, e o programa decodificará a mensagem. Para que este programa funcione é preciso chamar antes somente o expomod.

```
RSAdes:=proc(Cod,d,n)
# Decodifica a mensagem recebida.
# INPUT: Cod --> a mensagem codificada recebida
#          d --> chave privada
#          n --> chave pública.
# OUTPUT: a mensagem decodificada.
local Deco, po, x, u, b, c;
Deco := expomod(Cod, d, n): po := ` `:
x := floor((length(`Deco`)/3)) + 1:
        for u from x by -1 to 1 do
        b[u] := iquo(Deco, 1000^(u-1)):
        Deco := Deco - b[u]*1000^(u-1):
        c[u] := convert([b[u]], `bytes`):
        po := cat(` `.po.(c[u])):
        od:
po;
end:
```

Exemplos, verificando as mensagens emitidas pelo código RSAemi:

13.1.

RSAdes(368058473595725726035,495178122839282637067,864328212584663331559);

Ruy

13.2.

RSAdes(222015140161694349971983146916005798612172537917210279190097406000001683573087193064956405125131022584731457403445369044 3,

528148033227222836932281109760708069449113254938503083806431988816935908160243138782398734304724444821033086822491741965 7,

746839573917516688017190759949695092668247731315592940284594547691230094687601488277994303010978023678635038299066522666 9);

Dia 5 de abril, irei a Porto Alegre

14 - Baseado na definição de logaritmo discreto, elaborou-se o código **logdis.** Como ele testa diversos números, em seqüência, até encontrar o resultado, funciona bem para números pequenos mas é demorado para números grandes.

```
logdis:=proc(b,n,m)
# Calcula logaritmo discreto em módulo.
# Por tentativas.
# INPUT: b --> a base
#        n --> o número
#        m --> o módulo
# OUTPUT: o logaritmo discreto de base b,
#      do número n em módulo m.
local i;
        for i from 1 to m-1 do
                if modp(b^i, m) = n then print(`O logaritmo discreto
        de`,n, `na base`, b, `em módulo`,m, `é`, i); break
                fi;
        od;
end:
```

Exemplos:

14.1. logdis(2, 7, 19);

O logaritmo discreto de, 7, na base, 2, em módulo, 19, é, 6

15 - Para calcular potências, iterativamente, elaborou-se o código **expoiter**:

```
expoiter:=proc(a,n)
# Cálculo de potências iterativamente.
# INPUT: a --> base
#          n --> expoente
# OUTPUT: a potência
local i, r, x;
i := n;
r := 1;
x := a;
        while i > 0 do
                if irem(i, 2) = 1 then r := r*x;
                fi;
        x := x^2;
        i := floor(i/2);
        od;
r;
end:
```

Exemplos:

15.1. expoiter(2, 64);

18446744073709551616

15.2. expoiter(5467, 34);

```
1211904818758929590782886071426746682815013417396201612648860215341
348495292464253460157808227245785495703461258581557677058759 29
```

15.3. d:=expoiter(9134567, 925): length(`d`);

6439

16 - Embora não seja prático para números maiores, adaptou-se o teorema de Wilson para determinar números primos, surgindo o código **Wilson.** É demorado porque além de escolher os números a serem testados aleatoriamente, precisa calcular um fatorial. Eliminou-se uma parte do trabalho pela rejeição de números pares na escolha.

```
Wilson:=proc(n)
# Determina um número primo aleatório
# baseado no Teorema de Wilson.
# É um algoritmo determinístico.
# INPUT: n--> a quantidade de dígitos do número
#          desejado.
# OUTPUT: o número primo.
local a, b;
a := 10^(n-1);
        if is(a, even) = true then a := 10^(n-1);
        fi;
b := irem((a - 1)! + 1, a);
        while b <> 0 do
        a := rand(10^(n-1)..10^n); a := a();
                if is(a, even) = true then a := 10^(n-1);
                fi;
        b := irem((a - 1)! + 1, a); b := b();
                if b = 0 then print(`Primo:`, a);break
                fi;
        od;
end:
```

Exemplos:

16.1. Wilson(4);

```
Primo:, 5039
```

Confirmando pelo isprime(5039);

```
true
```

16.2. Wilson(3);

```
Primo:, 839
```

17 - Para testar se determinado número é número de Carmichael, tem-se o código **Carmi1**, baseado no teorema de Korselt.

```
Carmi1:=proc(n)
# Testa se n é um número de Carmichael
# baseado no Teorema de Korselt.
# INPUT: n --> o número a ser testado.
# OUTPUT: se o número é ou não é de Carmichael.
local a, b, c, d, e, f;
a := ifactor(n):
b := nops(a):
for c from 1 to b do
        d[c] := op(1, op(c, a));
        e := irem(n, d[c]^2);
        f := irem(n-1, d[c]-1);
                if e = 0
                and f <> 0 then RETURN (Não); break;
                fi;
                if e <> 0
                and f <> 0 then RETURN (Não); break;
                fi;
                if e = 0
                and f = 0 then RETURN (Não); break;
                fi;
od;
RETURN (É);
end:
```

Exemplos:

17.1. Carmi1(561);

É

17.2. Carmi1(35);

Não

17.3. Carmi1(1105);

É

17.4. Carmi1(651693055693681);

É

17.5. Carmi1(349407515342287435050603204719587201);

É

18 - Eis o código **queRSA** e a seguir um pequeno exemplo, com os mesmos dados numéricos de um exemplo utilizado neste trabalho. O objetivo deste código é quebrar o RSA calculando a raiz e-ésima. Mas, como trabalha por tentativas, testando vários números até encontrar o que satisfaz a condição, torna-se impraticável para números grandes. Tem como entrada o índice da raiz, o módulo e o radicando e como saída a raiz em módulo.

```
queRSA:=proc(e,n,C)
# Calcula a raiz e-ésima de um número C, em
# módulo n.
# Por tentativas.
# INPUT: e --> o índice
#        n --> o módulo
#        C --> o radicando
# OUTPUT: a raiz.
local i;
for i from 1 to n do
        if modp(i^e, n) = C then print(`A raiz (mensagem) é`, i); break
        fi;
od;
end:
```

Exemplo:

18.1. queRSA(29, 221, 117);

A raiz (mensagem) é, 104

REFERÊNCIAS BIBLIOGRÁFICAS

[BAA88] BAASE, Sara. *Computer Algorithms: Introduction to Design and Analysis*. 2ª ed. San Diego, Addison-Wesley Publishing Company, 1988.

[BRA96] BRASSARD, Gilles; BRATLEY, Paul. *Fundamental of Algorithmics*. New Jersey, Prentice-Hall, 1996.

[BRE99] BRENT, Richard P. *Some parallel Algorithms for Integer Factorisation*. http://www/springer.de/comp/lncs/index.html, 1999.

[CIP96] CIPRA, Barry. *The Secret Life of Large Numbers*. In What's happening in the Matematical Sciences, American Mathematial Society, volume 3, 1995-1996, pp. 90-99.

[CIP99] CIPRA, Barry. *Tales from the Cryptosystem*. In What's happening in the Matematical Sciences, American Mathematial Society, volume 4, 1998-1999, pp. 97-102.

[COU97] COUTINHO, S.C. *Números inteiros e criptografia RSA*. Rio de Janeiro, IMPA/SBM, 1997.

[HEF94] HEFEZ, Abramo. *Teoria dos códigos*. XIII Escola de Álgebra, UNICAMP, 1994.

[HOH98] HOHL, Fritz. *Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts*. In Lecture Notes in Computer Science (1419), Springer, 1998, pp. 92-113.

[RPM] Revista do Professor de Matemática. Sociedade Brasileira de Matemática, São Paulo.

LEITE, Paulo Ferreira. *Números de Fermat.* N° 7 – 2° semestre de 1985, pp. 23-25.

FREIRE, Benedito Tadeu V. *Números primos. Os argumentos de Euclides e Aplicações.* N° 11 – 2° semestre de 1987, pp. 5-8.

TERADA, Routo. *Criptografia e a importância das suas aplicações.* N° 12 – 1° semestre de 1988, pp. 1-7.

ÁVILA, Geraldo. *A distribuição dos números primos.* N° 19 – 2° semestre de 1991, pp. 19-28.

FREIRE, Benedito Tadeu Vasconcelos. *Congruência, divisibilidade e adivinhações.* N° 22 – 3° quadrimestre de 1992, pp. 4-10.

WATANABE, Renate. *Uma fórmula para os números primos.* N° 37 – 2° Quadrimestre de 1998, 00. 19-21.

Dispositivo prático para expressar o mdc de dois números como combinação linear deles. N° 37 – 2° quadrimestre de 1998, pp. 27-33.

GARBI, Gilberto. *Outro belo teorema de Fermat.* N° 38 – 3° quadrimestre de 1998, pp. 1-9.

OCHOVIET, Cristina. *2x3=0 ?* N° 41 – 3° quadrimestre de 1999, pp. 30-33.

FONSECA, Rubens Vilhena. *Números perfeitos, amigos e sociáveis.* N° 41 – 3° Quadrimestre de 1999, pp. 34-37.

Você sabia ? N° 41 – 3° quadrimestre de 1999, p. 37.

Lista de sites interessantes para os professores de matemática. N° 41 – 3° quadrimestre de 1999, p. 39.

Endereços eletrônicos:

http://www.mersenne.org/

http://www.perfsci.com

http://www.research.att.com/~amo