Ana Filipa Pinheiro Sequeira

Medusa - Uma cifra inspirada na Bífida e uma sua implementação

Departamentos de Matemática Pura e Aplicada
Faculdade de Ciências da Universidade do Porto
Março / 2007

Ana Filipa Pinheiro Sequeira

Medusa - Uma cifra inspirada na Bífida e uma sua implementação

Tese submetida à Faculdade de Ciências da Universidade do Porto
para obtenção do grau de Mestre em Engenharia Matemática

Departamentos de Matemática Pura e Aplicada
Faculdade de Ciências da Universidade do Porto
Março / 2007

Resumo

Os ataques dirigidos à cifra Bífida que são conhecidos baseiam-se no facto de o período daquela cifra ser constante. A cifra Medusa é baseada na Bífida, pois conserva algumas das suas características. No entanto, a Medusa foi construída de forma a que o comprimento dos blocos seja variável e dependente do conteúdo da mensagem original com o objectivo de evitar a aplicação dos referidos ataques.

Foi desenvolvida uma implementação da cifra Medusa em C++. O método de construção da chave exige que se trabalhe com inteiros de grandes dimensões, maiores do que o permitem os registos dos CPUs actuais. Assim, para efectuar essa implementação foi necessário utilizar uma forma de representar inteiros de grandes dimensões (não-negativos) e de efectuar operações com eles, como a soma, subtracção, multiplicação e divisão de inteiros. Com esta aritmética de precisão arbitrária é possível obter a chave da Medusa a partir de um valor trocado pelo protocolo de Diffie-Hellman e cifrar e decifrar mensagens.

Abstract

The known attacks to Bifid cipher are based on the fact that the period of that cipher is constant. The Medusa cipher is based on the Bifid sharing some of its characteristics. However, the Medusa was built so that the length of the blocks of the message is variable, and dependent of the original message, with the purpose of avoiding the aplication of the mentioned attacks.

An implementation of the Medusa has been developed in C++. The method of construction of the key requires dealing with large integers, bigger than what is allowed by actual CPUs. Therefore, in order to implement the Medusa cipher, was necessary to use a way to represent large (non negatives) integers, and to perform operations on them, as addition, subtraction, multiplication and division of integers. With this long precision arithmetic it is possible to obtain the key of the Medusa from an integer value obtained by the Diffie-Hellman protocol, and to cipher and decipher messages.

"Rule 8: The development of fast algorithms is slow!"

Arnold Schönhage (1994)

Agradecimentos

Agradeço ao Professor António Machiavelo o ter sido sempre uma fonte inesgotável de optimismo, apoio e força e o muito que me ensina desde os meus primeiros tempos como aluna nesta faculdade.

Ao Professor Rogério Reis (co-orientador não oficial) agradeço a disponibilidade que demonstrou ao apoiar a elaboração desta tese e os contributos inestimáveis que conduziram o nosso trabalho a bom porto.

A todas as pessoas que partilharam esta caminhada comigo, seja colaborando activamente, apoiando-me nos momentos de fraqueza ou simplesmente estando do meu lado, o meu mais sincero obrigada. Sei que sabem que nunca me esquecerei de partilhar com elas este momento feliz.

Por fim, aos meus pais de quem sou e a quem tudo devo, obrigada.

Conteúdo

Introdução:

A característica de maior fragilidade da cifra Bífida prende-se com o facto de o período ser fixo [4]. Para ultrapassar esta fragilidade foi criada uma cifra que mantém algumas características da Bífida mas em que o comprimento de bloco é variável, a cifra Medusa. Esta adaptação, que implica um acréscimo de cálculos, torna-se possível porque a Medusa, ao contrário da Bífida, não é pensada para ser utilizada "à mão", se o fosse, a cifra e decifração com este método torna-se-iam demasiado fastidiosas para que a sua utilização fosse proveitosa.

Ao criar a Medusa pretendeu-se, por um lado, criar uma cifra segura e resistente aos ataques aplicados à Bifida e, por outro lado, que fosse uma cifra passível de ser implementada de forma eficiente em dispositivos com recursos de memória limitados, como por exemplo, telemóveis. A hipótese de criar um protocolo criptográfico que permitisse a troca de mensagens SMS de forma segura foi a ideia que motivou este estudo. O que se procurou desenvolver foi um sistema de cifra e decifração de mensagens escritas que, a ser aplicado, carece naturalmente de algumas adaptações.

No primeiro capítulo apresenta-se a cifra Medusa descrevendo as suas características em termos da chave utilizada, do cálculo do comprimento dos blocos e dos métodos de cifra e decifração. Começa-se por caracterizar a chave e apresentar as várias formas que esta pode assumir. De seguida expõe-se o método de cálculo do comprimento dos blocos e passa-se a explicitar os métodos de cifra e decifração das mensagens. Por fim, justifica-se detalhadamente a forma como a chave desta cifra é obtida a partir de uma chave partilhada, usando o Protocolo de Diffie-Hellman [1], entre as partes que pretendem comunicar.

No segundo capítulo expõe-se a implementação que foi elaborada da Medusa. Uma vez que os cálculos necessários à construção da chave envolvem números inteiros com um número de dígitos maior do que os actuais CPUs nos permitem foi necessário criar uma forma de representar esses números convenientemente e de construir as operações aritméticas necessárias à obtenção da chave. Estas operações são a adição, subtracção, multiplicação e divisão de números inteiros não-negativos. No caso da multiplicação e da divisão os algoritmos utilizados são apresentados detalhadamente e devidamente justificados. Depois de construídas estas operações expõe-se a forma de efectuar a troca da chave pelo Protocolo de Diffie-Hellman e, de seguida, apresenta-se a implementação dos métodos de construção da

chave da cifra. Por fim, apresenta-se a implementação da cifra e decifração de mensagens.

No terceiro capítulo apresenta-se os resultados de alguns dos estudos estatísticos efectuados a mensagens e respectivos criptogramas e as conclusões retiradas em relação à segurança da cifra Medusa. Estes estudos têm como ponto de referência os ataques à Bifída conhecidos até ao momento. Apresenta-se também algumas adaptações que poderiam ser feitas para reforçar a segurança e aplicabilidade prática da cifra.

Na *Conclusão e trabalho futuro*, refere-se as limitações desta implementação e aventa-se algumas possíveis melhorias.

No Anexo apresenta-se o código da implementação desenvolvida.

Capítulo 1

A Cifra Medusa

1.1 Descrição da Cifra Medusa

A Cifra Medusa é uma cifra de bloco, de chave simétrica. Esta cifra é inspirada na Cifra Bífida, descrita em [4], no entanto, foram introduzidas alterações no sentido de aumentar a sua segurança, nomeadamente no que diz respeito ao comprimento dos blocos. Ao contrário do que acontece na cifra Bífida, o comprimento de bloco na cifra Medusa é variável e tem a particularidade de variar mediante o conteúdo do texto simples, excepto no primeiro bloco, em que depende da chave, e nos dois últimos, em que depende do tamanho restante da mensagem.

O método utilizado para a determinação do comprimento de bloco implica que sejam efectuados alguns cálculos a partir de cada bloco a cifrar. É possível introduzir esta alteração, que induz um aumento de cálculos a efectuar, pois a Medusa, ao contrário da Bífida, não é pensada para ser utilizada à mão.

A chave da cifra Medusa, tal como acontece na Bífida, consiste num quadrado onde estão dispostos os símbolos do alfabeto e, por essa razão, dado um alfabeto de tamanho N, a chave consiste num quadrado de lado k onde $N = k^2$. Note-se que o alfabeto Σ é um conjunto *ordenado* de símbolos.

Dado o texto original, $\mathcal{O}$, a forma de o cifrar consiste em tomar sucessivos blocos, cujo tamanho depende do conteúdo do texto do bloco anterior (à excepção do primeiro que é determinado a partir da chave $\mathcal{K}$ e dos dois últimos), que depois de cifrados são concatenados para compôr o criptograma, $\mathcal{C}$.

A cada símbolo do alfabeto são atribuídas duas coordenadas que traduzem a posição do símbolo na chave, ou seja, a primeira é a linha em que o símbolo se encontra e a segunda é a coluna.

Seja Σ um alfabeto tal que $|\Sigma| = 9$. A seguir representa-se uma das possíveis chaves:

	0	1	2
0	σ_0	σ_1	σ_2
1	σ_3	σ_4	σ_5
2	σ_6	σ_7	σ_8

As coordenadas do símbolo σ_1 são $(x, y) = (0, 1)$, pois σ_1 encontra-se na linha 0 e na coluna 1.

A cifra de um dado bloco é feita atribuíndo a cada símbolo do texto as suas coordenadas e rearranjando-as, de uma determinada forma, em novos pares de coordenadas que originam os símbolos que constituem o criptograma. A título de exemplo, dado o bloco "$\sigma_0\sigma_1\sigma_8$", tem-se:

$$\begin{array}{ccc} \sigma_0 & \sigma_1 & \sigma_8 \\ 0 & 0 & 2 \\ 0 & 1 & 2 \end{array}$$

lendo os pares de coordenadas horizontalmente da esquerda para a direita como a figura seguinte sugere:

$$\begin{array}{cc} 1 & \longrightarrow \\ 2 & \longrightarrow \end{array}$$

e escrevendo-os da forma sugerida na figura seguinte:

$$\begin{array}{ccc} 1 & 2 & \ldots \\ \downarrow & \downarrow & \ldots \end{array}$$

obtêm-se novos pares de coordenadas que correspondem ao criptograma:

$$\begin{array}{ccc} 0 & 2 & 1 \\ 0 & 0 & 2 \\ \sigma_0 & \sigma_6 & \sigma_5 \end{array}$$

O criptograma obtido é "$\sigma_0\sigma_6\sigma_5$".

No sentido inverso, dado o criptograma, é efectuada a decifração do primeiro bloco (do qual é conhecido o comprimento a partir da chave $\mathcal{K}$) executando o processo inverso do que foi descrito acima. À custa deste primeiro bloco da mensagem original determina-se o comprimento do bloco seguinte e procedendo sucessivamente deste modo obtêm-se vários blocos decifrados que, ao serem concatenados, compõem a mensagem original.

A decifração de um bloco é feita atribuíndo a cada símbolo do criptograma as suas coordenadas e rearranjando-as, de uma determinada forma, em novos pares de coordenadas que originam os símbolos que constituem a mensagem original. Por exemplo, dado o bloco "$\sigma_0\sigma_6\sigma_5$", tem-se:

$$\begin{array}{ccc} \sigma_0 & \sigma_6 & \sigma_5 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{array}$$

lendo os pares de coordenadas verticalmente da esquerda para a direita como a figura seguinte sugere:

$$\begin{array}{ccc} 1 & 2 & \dots \\ \downarrow & \downarrow & \dots \end{array}$$

e escrevendo-os horizontalmente da esquerda para a direita da forma sugerida na figura seguinte:

$$\begin{array}{cc} 1 & \longrightarrow \\ 2 & \longrightarrow \end{array}$$

obtêm-se novos pares de coordenadas que correspondem à mensagem original:

$$\begin{array}{ccc} 0 & 0 & 2 \\ 0 & 1 & 2 \\ \sigma_0 & \sigma_1 & \sigma_8 \end{array}$$

Sendo obtida assim a mensagem original "$\sigma_0\sigma_1\sigma_8$".

Refira-se ainda que, como a Medusa é uma cifra simétrica, o protocolo de comunicação entre as duas partes pressupõe a partilha de chaves o que será feito usando o Protocolo de Diffie-Hellman [1].

1.2 A Chave

A chave da cifra Medusa pode ser vista como um quadrado onde são dispostos os símbolos do alfabeto. Seja Σ, o alfabeto no qual foi fixada uma ordenação dos símbolos, com $|\Sigma| = N$, onde $N = k^2$ para algum $k \in \mathbb{N}$. A tabela seguinte representa uma chave da cifra Medusa:

	0	1	$\cdots$	$k-2$	$k-1$
0	σ_0	σ_1	$\cdots$	σ_{k-2}	σ_{k-1}
1	σ_k	σ_{k+1}	$\cdots$	σ_{2k-2}	σ_{2k-1}
$\vdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$
$k-2$	$\sigma_{(k-2)\cdot k}$	$\sigma_{(k-2)\cdot k+1}$	$\cdots$	$\sigma_{(k-1)\cdot k-2}$	$\sigma_{(k-1)\cdot k-1}$
$k-1$	$\sigma_{(k-1)\cdot k}$	$\sigma_{(k-1)\cdot k+1}$	$\cdots$	σ_{k^2-2}	σ_{k^2-1}

Na tabela estão dispostos os k^2 símbolos do alfabeto e note-se que o que distingue as chaves entre si é precisamente esta forma como os símbolos estão organizados.

Convencione-se ler a tabela horizontalmente da esquerda para a a direita de acordo com a

figura seguinte:

	$0 \cdots k-1$
0	$\longrightarrow$
1	$\longrightarrow$
$\vdots$	$\vdots$
$k-1$	$\longrightarrow$

Assim, pode fazer-se corresponder à tabela a seguinte sequência $\mathcal{S}$ dos símbolos do alfabeto:

$$\mathcal{S} = [\sigma_0, \sigma_1, \cdots, \sigma_{k-1}, \sigma_k, \cdots, \sigma_{2k-1}, \sigma_{2k}, \cdots, \sigma_{k^2-1}],$$

em que $\mathcal{S}$ resulta de ter sido aplicada uma permutação ao conjunto ordenado dos símbolos do alfabeto Σ.

Resulta desta observação que uma outra representação da chave equivalente à tabela acima representada é a ordenação dos símbolos do alfabeto, uma vez que, conhecido o número que corresponde à posição de um dado símbolo em $\mathcal{S}$, o quociente da divisão desse número por k corresponde à linha em que ele se encontra na tabela e o resto dessa divisão corresponde à coluna. Ou seja, é possível passar da posição de um símbolo em $\mathcal{S}$ para as coordenadas desse símbolo na tabela e vice-versa.

Por exemplo, o símbolo σ_{2k-2} é o $(2k-2)$-ésimo símbolo na ordenação correspondente à tabela e assim posso concluir que as suas coordenadas em relação à tabela são dadas por $(\lfloor \frac{2k-2}{k} \rfloor, 2k-2 \mod k) = (1, k-2)$; o símbolo σ_{k^2-1} tem coordenadas $(k-1, k-1)$ logo na ordenação está na posição $k(k-1)+k-1$, ou seja, k^2-1.

Já se observou que considerar uma tabela que representa uma chave da Medusa é equivalente a considerar uma sequência $\mathcal{S}$ dos símbolos do alfabeto. Por outro lado, sendo o alfabeto Σ um conjunto ordenado, cada sequência dos símbolos do alfabeto corresponde a uma e uma só sequência dos N números de 0 a $N-1$. Assim, tem-se uma correspondência entre as chaves da Medusa e as sequências de N números.

Considere-se uma ordenação lexicográfica das sequências de N números, a cada uma das sequências fica associado um número que é a sua posição na lista em causa, a que se chamará *número de ordem* da sequência. Uma vez que o número destas sequências é $N!-1$, dada a sequência, $\mathcal{S}_\mathcal{K}$, o número $\mathcal{K}$ que representa a posição da sequência está no intervalo $[0, ..., N!-1]$ e corresponde ao número de sequências que a precedem, tendo em conta aquela ordenação. Estabelece-se então uma bijecção natural entre as sequências e os valores, que se ilustra na figura seguinte:

$\mathcal{S}_\mathcal{K}$		$\mathcal{K}$
$0123...(N-3)(N-2)(N-1)$		0
$0123...(N-3)(N-1)(N-2)$	$\longleftrightarrow$	1
$\cdots$		$\cdots$
$(N-1)(N-2)(N-3)...3201$		$N!-2$
$(N-1)(N-2)(N-3)...3210$		$N!-1$

Tendo em conta a bijecção estabelecida entre as chaves da cifra em forma de quadrado e os números do intervalo $[0, ..., N! - 1]$ fica claro que escolher uma chave dispondo os símbolos do alfabeto num quadrado ou escolher aleatoriamente um valor no intervalo $[0, ..., N! - 1]$ são maneiras equivalentes de cria uma chave.

Assim, a chave da cifra pode tomar várias formas, todas elas equivalentes, e sempre que não haja necessidade de distinção, a palavra *chave* será utilizada para referir, indiferentemente, o quadrado dos símblos, a sequência dos números de 0 a $N - 1$, a sequência aplicada ao alfabeto ou ainda o número de ordem da sequência. Uma vez que, notoriamente, é equivalente considerar a chave em cada uma das formas apresentadas, escolher-se-á a forma mais adequada e conveniente ao que se pretende em cada momento.

De facto, há chaves que são "equivalentes" uma vez que, considerando a chave como um quadrado, se se permutar as linhas e as colunas do mesmo modo (usando a mesma sequência) obtém-se uma chave que produz o mesmo criptograma. No entanto, nesta descrição vamos considerar cada sequência como uma chave diferente, pois não é fácil obter um conjunto das sequências que corresponda biunivocamente ao conjunto das chaves que dão codificações distintas e tal esforço é desnecessário pois cada codificação repetida aparece um mesmo número de vezes, correspondendo a $k!$ chaves distintas.

1.3 Génese da Chave

A partir do número trocado entre os elementos envolvidos na comunicação, utilizando o protocolo de Diffie-Hellman, é obtido o valor $\mathcal{K}$, que corresponde a uma sequência $\mathcal{S}_\mathcal{K}$, onde $\mathcal{K} \in [0, ..., N! - 1]$.

O processo de construção da sequência $\mathcal{S}_\mathcal{K}$ compreende as seguintes etapas:

1. Escreve-se $\mathcal{K}$ na forma:
$$\mathcal{K} = \sum_{i=0}^{N-1} \alpha_{N-1-i} \cdot (N - 1 - i)!$$
onde os α_is são determinados por ordem decrescente dos índices e são os maiores possíveis, condições que garantem que esta escrita é única (como se verá mais à frente na secção 1.7.2);

2. O primeiro número da sequência é $k_0 = \alpha_{N-1}$;

3. Faz-se $j \leftarrow 1$;

4. Para determinar o j-ésimo número da sequência, k_j, supondo que já são conhecidos $k_0, k_1, ..., k_{j-1}$, determina-se o único número n, onde $n \notin [k_0, k_1, ..., k_{j-1}]$ e satisfaz a condição seguinte:
$$n = \alpha_{N-1-j} + |\{\ell \in [0, \ldots, j-1] : k_\ell < n\}|'' .$$

E então $k_j = n$;

5. Seja $j \leftarrow j + 1$, se $j \leq N - 1$ repita-se o passo 4.

Desta forma obtém-se a sequência $\mathcal{S}_\mathcal{K} = [k_0, k_1, ..., k_{N-1}]$.

1.4 O comprimento dos blocos

O comprimento de cada bloco é variável e, à excepção do primeiro e dos dois últimos, depende do conteúdo do bloco anterior, sendo à partida fixados um valor mínimo, m, e um valor máximo, M, para o comprimento dos blocos.

Seja $O = o_0 o_1 \cdots o_{n-1}$ a mensagem orignal de comprimento n. Seja $(O_{b_j})_j = O_0 O_1 \ldots O_{b_j - 1}$ o j-ésimo bloco a cifrar com $\left|O_{b_j}\right| = b_j$. O comprimento do primeiro bloco de mensagem a ser cifrado é calculado à custa do primeiro símbolo, k_0, da chave, (sendo a chave $\mathcal{S}_\mathcal{K} = [k_0, k_1, ..., k_{N-1}]$). Como a cifra Medusa é uma cifra simétrica, a chave é partilhada pelas duas partes, logo o comprimento do primeiro bloco está acessível a ambas. Tendo em conta os valores máximo e mínimo para os comprimentos dos blocos, o valor do comprimento do primeiro bloco, b_0, é um valor inteiro no intervalo $[m, M]$ dado por:

$$b_0 = \left\lfloor (M - m) \cdot \frac{k_0}{N - 1} \right\rfloor + m.$$

Nos blocos a seguir ao primeiro calcular-se-á o comprimento de cada bloco à custa do conteúdo da mensagem original, usando as posições de cada símbolo no alfabeto. Uma vez que o número de símbolos do alfabeto Σ é N, a posição de cada símbolo no alfabeto varia entre 0 e $N - 1$.

Dado $(O_{b_j})_j$, o j-ésimo bloco da mensagem original a ser cifrado, e S a soma das posições dos símbolos desse bloco, tem-se que $S \in [0, (N-1) \cdot b]$. Então o comprimento do $j+1$-ésimo bloco, b_{j+1} dado por:

$$S \mod (M - m + 1) + m,$$

é um valor inteiro no intervalo $[m, M]$.

Quando o número de símbolos da mensagem que falta cifrar, $|MRestante|$, for inferior a $2M$ o comprimento do bloco seguinte é dado por:

$$\left\lfloor \frac{|MRestante|}{2} \right\rfloor.$$

E, por fim, o último bloco a cifrar é constituído pela mensagem restante.

1.5 Cifra

A mensagem original é dividida em blocos de tamanho váriavel que depois de cifrados serão concatenados para originar o respectivo do criptograma.

Dado um bloco da mensagem original, a cada símbolo do bloco são atribuídas as suas duas coordenadas x, y (com $x, y \in \{0, 1, 2, ..., k-1\}$) que traduzem a posição do símbolo na chave. Depois de obtidas as coordenadas de todos os símbolos do bloco estas coordenadas são lidas da esquerda para a direita horizontalmente sendo assim rearranjadas em novos pares. A partir destes novos pares de coordenadas obtêm-se os símbolos que lhe correspondem e que constituem o criptograma.

Observe-se o seguinte exemplo, para um alfabeto Σ tal que $|\Sigma| = 25$, seja uma chave dada por:

	0	1	2	3	4
0	f	a	x	i	k
1	u	v	q	m	z
2	o	p	w	c	d
3	b	e	g	y	s
4	h	l	n	r	t

Para cifrar a frase "o segredo é a alma do negócio" como um bloco de mensagem, começa-se por convertê-la devidamente em "osegredoeaalmadonegocio" e depois faz-se corresponder a cada símbolo as suas coordenadas em relação à chave:

o	s	e	g	r	e	d	o	e	a	a	l	m	a	d	o	n	e	g	o	c	i	o
2	3	3	3	4	3	2	2	3	0	0	4	1	0	2	2	4	3	3	2	2	0	2
0	4	1	2	3	1	4	0	1	1	1	1	3	1	4	0	2	1	2	0	3	3	0

Depois leêm-se as coordenadas horizontalmente da esquerda para a direita e escrevendo-as verticalmente também da esquerda para a direita formam-se novos pares de coordenadas que dão origem aos símbolos do criptograma:

2	3	4	2	3	0	1	2	4	3	2	2	4	2	1	0	1	1	1	0	1	0	3
3	3	3	2	0	4	0	2	3	2	0	0	1	3	4	1	1	3	4	2	2	3	0
c	y	r	w	b	k	u	w	r	g	o	o	l	c	z	a	v	m	z	x	g	i	b

Obtendo-se assim o criptograma: "cyrwbkuwrgoolczavmzxgib".

De uma forma geral, o bloco $O = o_0 o_1 ... o_{\ell-1}$, de comprimento ℓ, dá origem ao bloco do criptograma $C = c_0 c_1 ... c_{\ell-1}$, também de comprimento ℓ, do seguinte modo:

No caso em que ℓ é ímpar,

o_0	o_1	$\cdots$	$o_{\ell-2}$	$o_{\ell-1}$
x_0	x_1	$\cdots$	$x_{\ell-2}$	$x_{\ell-1}$
y_0	y_1	$\cdots$	$y_{\ell-2}$	$y_{\ell-1}$

$\longmapsto$

c_0	$\cdots$	$c_{\frac{\ell-1}{2}-1}$	$c_{\frac{\ell-1}{2}}$	$c_{\frac{\ell-1}{2}+1}$	$\cdots$	$c_{\ell-1}$
x_0	$\cdots$	$x_{\ell-3}$	$x_{\ell-1}$	y_1	$\cdots$	$y_{\ell-2}$
x_1	$\cdots$	$x_{\ell-2}$	y_0	y_2	$\cdots$	$y_{\ell-1}$

Por outro lado, se o comprimento do bloco ℓ for par será:

o_0	o_1	$\cdots$	$o_{\ell-2}$	$o_{\ell-1}$
x_0	x_1	$\cdots$	$x_{\ell-2}$	$x_{\ell-1}$
y_0	y_1	$\cdots$	$y_{\ell-2}$	$y_{\ell-1}$

$\longmapsto$

c_0	c_1	$\cdots$	$c_{\frac{\ell}{2}-1}$	$c_{\frac{\ell}{2}}$	$c_{\frac{\ell}{2}+1}$	$\cdots$	$c_{\ell-1}$
x_0	x_2	$\cdots$	$x_{\ell-2}$	y_0	y_2	$\cdots$	$y_{\ell-2}$
x_1	x_3	$\cdots$	$x_{\ell-1}$	y_1	y_3	$\cdots$	$y_{\ell-1}$

1.6 Decifração

Dado o criptograma, C, e o comprimento do bloco inicial, b_0, determinado a partir da chave, decifra-se o primieiro bloco do criptograma obtendo o primeiro bloco da mensagem original. À custa do conteúdo deste determina-se o comprimento do bloco seguinte, b_1. Repete-se o procedimento até que o tamanho do criptograma que está ainda por decifrar seja menor ou igual a $2M$, ou seja, ao dobro do valor máximo do comprimento de bloco, e nesta situação o tamanho do bloco seguinte é dado por: $\left\lfloor \frac{|CRestante|}{2} \right\rfloor$ e o último é o restante do criptograma.

Para efectuar a decifração de cada um dos blocos do criptograma executa-se um processo inverso à cifração.

A cada símbolo do bloco do criptograma são atribuídas as duas coordenadas que lhe correspondem na chave. Estas coordenadas são lidas verticalmente da esquerda para a direita, obtendo-se assim uma sequência do tipo $x_0y_0x_1y_1 \ldots x_iy_i \ldots$ em que x_i, y_i são as coordenadas do i-ésimo símbolo do bloco. De seguida esta sequência é lida horiontalmente da esquerda para a direita formando-se desta forma novos pares de coordenadas. Com estes novos pares obtêm-se os símbolos da mensagem original.

Utilize-se a mesma chave do exemplo anterior:

	0	1	2	3	4
0	f	a	x	i	k
1	u	v	q	m	z
2	o	p	w	c	d
3	b	e	g	y	s
4	h	l	n	r	t

Para decifrar o bloco de criptograma "cyrwbkuwrgoolczavmzxgib" faz-se corresponder a cada símbolo as suas coordenadas em relação à chave:

c	y	r	w	b	k	u	w	r	g	o	o	l	c	z	a	v	m	z	x	g	i	b
2	3	4	2	3	0	1	2	4	3	2	2	4	2	1	0	1	1	1	0	1	0	3
3	3	3	2	0	4	0	2	3	2	0	0	1	3	4	1	1	3	4	2	2	3	0

Leêm-se as coordenadas verticalmente da esquerda para a direita e depois escrevem-se horizontalmente também da esquerda para a direita formando-se assim novos pares de coordenadas que dão origem aos símbolos da mensagem original:

2	3	3	3	4	3	2	2	3	0	0	4	1	0	2	2	4	3	3	2	2	0	2
0	4	1	2	3	1	4	0	1	1	1	1	3	1	4	0	2	1	2	0	3	3	0
o	s	e	g	r	e	d	o	e	a	a	l	m	a	d	o	n	e	g	o	c	i	o

Obtendo-se assim a mensagem original: "osegredoeaalmadonegocio".

A seguir apresenta-se de forma esquematizada a decifração de um bloco de comprimento ℓ ímpar:

c_0	$\cdots$	$c_{\frac{\ell-1}{2}-1}$	$c_{\frac{\ell-1}{2}}$	$c_{\frac{\ell-1}{2}+1}$	$\cdots$	$c_{\ell-1}$
x_0	$\cdots$	$x_{\frac{\ell-1}{2}-1}$	$x_{\frac{\ell-1}{2}}$	$x_{\frac{\ell-1}{2}+1}$	$\cdots$	$x_{\ell-1}$
y_0	$\cdots$	$y_{\frac{\ell-1}{2}-1}$	$y_{\frac{\ell-1}{2}}$	$y_{\frac{\ell-1}{2}+1}$	$\cdots$	$y_{\ell-1}$

$\downarrow$

$$x_0\ y_0\ x_1\ y_1\ \ldots\ x_{\frac{\ell-1}{2}-1}\ y_{\frac{\ell-1}{2}-1}\ x_{\frac{\ell-1}{2}}\ y_{\frac{\ell-1}{2}}\ \ldots\ x_{\ell-1}\ y_{\ell-1}$$

$\downarrow$

$$\begin{array}{cccccccc} x_0 & y_0 & x_1 & \ldots & x_{\frac{\ell-1}{2}-1} & y_{\frac{\ell-1}{2}-1} & x_{\frac{\ell-1}{2}} \\ y_{\frac{\ell-1}{2}} & x_{\frac{\ell-1}{2}+1} & y_{\frac{\ell-1}{2}+1} & \ldots & y_{\ell-2} & x_{\ell-1} & y_{\ell-1} \end{array}$$

$\downarrow$

o_0	o_1	$\cdots$	$o_{\ell-2}$	$o_{\ell-1}$
x_0	y_0	$\cdots$	$y_{\frac{\ell-1}{2}-1}$	$x_{\frac{\ell-1}{2}}$
$y_{\frac{\ell-1}{2}}$	$x_{\frac{\ell-1}{2}+1}$	$\cdots$	$x_{\ell-1}$	$y_{\ell-1}$

Por outro lado, se o comprimento do bloco ℓ for par, a decifração é feita do seguinte modo:

c_0	c_1	$\cdots$	$c_{\frac{\ell}{2}-1}$	$c_{\frac{\ell}{2}}$	$c_{\frac{\ell}{2}+1}$	$\cdots$	$c_{\ell-1}$
x_0	x_1	$\cdots$	$x_{\frac{\ell}{2}-1}$	$x_{\frac{\ell}{2}}$	$x_{\frac{\ell}{2}+1}$	$\cdots$	$x_{\ell-1}$
y_0	y_1	$\cdots$	$y_{\frac{\ell}{2}-1}$	$y_{\frac{\ell}{2}}$	$y_{\frac{\ell}{2}+1}$	$\cdots$	$y_{\ell-1}$

$$\downarrow$$

$$x_0\ y_0\ x_1\ y_1\ \ldots\ x_{\frac{\ell}{2}-1}\ y_{\frac{\ell}{2}-1}\ x_{\frac{\ell}{2}}\ y_{\frac{\ell}{2}}\ \ldots\ x_{\ell-1}\ y_{\ell-1}$$

$$\downarrow$$

$$\begin{array}{cccccc} x_0 & y_0 & x_1 & \ldots & y_{\frac{\ell}{2}-2} & x_{\frac{\ell}{2}-1} & y_{\frac{\ell}{2}-1} \\ x_{\frac{\ell}{2}} & y_{\frac{\ell}{2}} & x_{\frac{\ell}{2}+1} & \ldots & y_{\ell-2} & x_{\ell-1} & y_{\ell-1} \end{array}$$

$$\downarrow$$

o_0	o_1	$\cdots$	$o_{\ell-2}$	$o_{\ell-1}$
x_0	y_0	$\cdots$	$y_{\frac{\ell}{2}-1}$	$x_{\frac{\ell}{2}}$
$y_{\frac{\ell}{2}}$	$x_{\frac{\ell}{2}+1}$	$\cdots$	$x_{\ell-1}$	$y_{\ell-1}$

1.7 Justificação do método de obtenção da chave

Na secção 1.3 expôs-se uma forma de se obter a chave da cifra, na forma de uma sequência de N números, denotada por $\mathcal{S}_{\mathcal{K}}$, a partir de um número entre 0 e $N!-1$, denotado por $\mathcal{K}$.

Nesta secção pretende-se justificar o procedimento exposto explicitando uma bijecção entre as sequências de N números e os números do intervalo $[0, ..., N!-1]$. Esta correspondência biunívoca permite garantir que cada número corresponde a uma e uma só chave da cifra e inversamente a cada cifra corresponde um e um só número que é o seu número de ordem.

Seja Σ um alfabeto com N símbolos, o número de chaves da cifra é $N!$, pois corresponde ao número de sequências de N números.

Comece-se por escolher uma ordenação das sequências que consiste em considerá-las pela ordem lexicográfica. Por um lado, irá ser mostrada uma forma de, dada uma sequência, $\mathcal{S}_{\mathcal{K}}$, se obter o seu número de ordem $\mathcal{K}$ que corresponde à posição dessa sequência na ordenação. Por outro lado, mostrar-se-á uma forma de, dado um número $\mathcal{K}$, se obter a sequência que está na posição $\mathcal{K}$ nessa mesma ordenação, ou seja, cujo número de ordem é $\mathcal{K}$.

Dada uma sequência, $\mathcal{S}_{\mathcal{K}}$, o seu número de ordem $\mathcal{K}$, ou seja, a posição da sequência na ordenação corresponde ao número de sequências que a precedem nessa ordenação. Inversamente, dado um número de ordem $\mathcal{K}$, a sequência $\mathcal{S}_{\mathcal{K}}$, corresponde à sequência precedida

por $\mathcal{K}$ sequências na mesma ordenação, como se pode observar na figura seguinte.

$\mathcal{S}_\mathcal{K}$		$\mathcal{K}$
$0123...(N-3)(N-2)(N-1)$		0
$0123...(N-3)(N-1)(N-2)$		1
$0123...(N-2)(N-3)(N-1)$	$\longleftrightarrow$	2
$\cdots$		$\cdots$
$(N-1)(N-2)(N-3)...3120$		$N!-3$
$(N-1)(N-2)(N-3)...3201$		$N!-2$
$(N-1)(N-2)(N-3)...3210$		$N!-1$

A título de exemplo, o número de ordem $\mathcal{K} = N!-3$ corresponde à chave $\mathcal{S}_\mathcal{K} = (N-1)(N-2)(N-3)...3120$ pois há $N!-3$ sequências que a precedem na ordenação, ou seja, esta é a $(N!-2)$-ésima sequência.

1.7.1 Obtenção do número de ordem $\mathcal{K}$ da sequência $\mathcal{S}_\mathcal{K}$

Dada a sequência $\mathcal{S}_\mathcal{K} = [k_0, k_1, ..., k_i, ..., k_{N-2}, k_{N-1}]$, o respectivo número de ordem da sequência , $\mathcal{K}$, pode ser determinado usando a seguinte observação:

Ao construir uma sequência de N números tem-se N escolhas para o primeiro número, $N-1$ escolhas para o segundo assim sucessivamente até se ter duas escolhas para o $(N-1)$-ésimo número e por fim uma escolha para o último, ou seja, para o N-ésimo número da sequência. Assim, fixado o primeiro número de uma sequência existem $(N-1)!$ sequências que começam por esse número (pois resta escolher $N-1$ números e estes podem ser ordenados de $(N-1)!$ maneiras possíveis), fixado também o segundo existem $(N-2)!$ sequências que começam por esses dois números e sucessivamente até que, fixados os $N-2$ primeiros números existem $2!$ sequências, fixados os $N-1$ primeiros números existe $1!$ sequência, e, por fim, fixados os N números existe $0!$ sequência começadas, ou melhor, constituída por esses números.

Por exemplo, seja Σ um alfabeto tal que $N = |\Sigma| = 4$. Sendo o alfabeto constituído por quatro símbolos então o número de chaves é igual a $4!$, ou seja, as 24 chaves possíveis correspondem às 24 sequências de quatro números. Pelo que foi visto no início de 1.7 existe uma correspondência biunívoca entre as 24 sequências e os números no intervalo $[0, ..., 4!-1]$.

Esta correspondência está ilustrada na figura:

$\mathcal{S}_{\mathcal{K}}$		$\mathcal{K}$
0123		0
0132		1
0213	$\longleftrightarrow$	2
$\cdots$		$\cdots$
3120		$4! - 3 = 21$
3201		$4! - 2 = 22$
3210		$4! - 1 = 23$

Ao construir uma sequência há quatro opções para o primeiro número, para o segundo temos três opções, para o terceiro temos duas opções e o quarto fica determinado pelos anteriores. Como se explicita na figura seguinte, uma vez fixado o primeiro símbolo, existem três opções para o símbolo seguinte:

0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3	3
1	1	2	2	3	3	0	0	2	2	3	3	0	0	1	1	3	3	0	0	1	1	2	2
$\underbrace{\qquad\qquad}_{3}$						$\underbrace{\qquad\qquad}_{3}$						$\underbrace{\qquad\qquad}_{3}$						$\underbrace{\qquad\qquad}_{3}$					

Da mesma forma, depois de fixados os dois primeiros símbolos tem-se duas opções para o terceiro como se exemplifica na imagem seguinte:

0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3	3
1	1	2	2	3	3	0	0	2	2	3	3	0	0	1	1	3	3	0	0	1	1	2	2
2	3	1	3	1	2	2	3	0	3	0	2	1	3	0	3	0	1	1	2	0	2	0	1
$\underbrace{\qquad}_{2}$		$\underbrace{\qquad}_{2}$		$\underbrace{\qquad}_{2}$		$\underbrace{\qquad}_{2}$		$\underbrace{\qquad}_{2}$		$\underbrace{\qquad}_{2}$		$\underbrace{\qquad}_{2}$		$\underbrace{\qquad}_{2}$		$\underbrace{\qquad}_{2}$		$\underbrace{\qquad}_{2}$		$\underbrace{\qquad}_{2}$		$\underbrace{\qquad}_{2}$	

Por fim, para o quarto símbolo há apenas uma opção.

0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3	3
1	1	2	2	3	3	0	0	2	2	3	3	0	0	1	1	3	3	0	0	1	1	2	2
2	3	1	3	1	2	2	3	0	3	0	2	1	3	0	3	0	1	1	2	0	2	0	1
3	2	3	1	2	1	3	2	3	0	2	0	3	1	3	0	1	0	2	1	2	0	1	0

Note-se que para cada um dos números escolhidos para a primeira posição tem-se 3! sequências possíveis, para cada número escolhido na segunda posição tem-se 2! sequências possíveis, para cada número na terceira posição tem-se 1! sequência possível e, por fim, escolhidos todos os números há 0! sequência possível.

Pode representar-se as sequências, ordenadas lexicograficamente da esquerda para a direita, da forma seguinte:

0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3	3
1	1	2	2	3	3	0	0	2	2	3	3	0	0	1	1	3	3	0	0	1	1	2	2
2	3	1	3	1	2	2	3	0	3	0	2	1	3	0	3	0	1	1	2	0	2	0	1
3	2	3	1	2	1	3	2	3	0	2	0	3	1	3	0	1	0	2	1	2	0	1	0
$\underbrace{}_{1!}$						$\underbrace{}_{1!}$						$\underbrace{}_{1!}$						$\underbrace{}_{1!}$					
$\underbrace{}_{2!}$						$\underbrace{}_{2!}$						$\underbrace{}_{2!}$						$\underbrace{}_{2!}$					
$\underbrace{}_{3!}$						$\underbrace{}_{3!}$						$\underbrace{}_{3!}$						$\underbrace{}_{3!}$					

Voltando ao caso geral, note-se que, para determinar o número de sequências que estão antes da sequência $\mathcal{S}_\mathcal{K}$, dada por $[k_0, k_1, ..., k_i, ..., k_{N-2}, k_{N-1}]$, é necessário contabilizar quantas sequências precedem as sequências começadas por k_0, quantas sequências começadas por k_0 precedem as sequências em que k_0 é seguido por k_1, quantas sequências começadas por $[k_0, k_1]$ precedem as que têm k_2 a seguir, e assim sucessivamente.

Desta forma, considerando a ordenação lexicográfica das sequências, resulta que determinar o número de sequências que precedem uma dada sequência consiste em contabilizar os "blocos" de tamanho $(N-1)!$, $(N-2)!$, ..., $2!$, $1!$ e $0!$, que a precedem na ordenação. Observa-se assim que o número de ordem de uma sequência é dado por uma soma do tipo:

$$\alpha_{N-1} \cdot (N-1)! + ... + \alpha_{N-1-i} \cdot (N-1-i)! + ... + \alpha_0 \cdot 0!.$$

Em face disto, a questão que se coloca é *como determinar os coeficientes desta combinação linear de factoriais decrescentes?* Os coeficientes da combinação linear dependem dos elementos da sequência. Dado o elemento da sequência k_i é necessário contabilizar o número de "blocos de tamanho $(N-1-i)!$" que estão antes da sequência em questão para determinar o respectivo coeficiente, α_{N-1-i}.

O primeiro elemento da sequência, k_0, vai contribuir para $\mathcal{K}$ com uma parcela do tipo $k_0 \cdot (N-1)!$ pois na ordenação estão, antes de $\mathcal{S}_\mathcal{K}$, todas as chaves começadas pelos números inferiores a k_0 e há $(N-1)!$ sequências possíveis para cada uma dessas opções. Assim:

$$\mathcal{K} = \sum_{i=0}^{N-1} \alpha_{N-1-i} \cdot (N-1-i) = k_0 \cdot (N-1)! + \ldots$$

Resultando então que $\alpha_{N-1} = k_0$.

Para determinar os coeficientes seguintes é necessário notar que, como uma sequência $\mathcal{S}_\mathcal{K}$ é uma sequência dos N números de 0 a $N-1$ em que não se verificam repetições, se um dado número, menor do que k_i, surgiu numa posição anterior a i esse número não é uma opção

válida para a escolha do k_i e assim a anteceder esta sequência irá haver menos um bloco de tamanho $(N-1-i)!$.

Voltando ao exemplo de um alfabeto com quatro símbolos, considere-se a sequência $\mathcal{S}_\mathcal{K} = [1, 3, ...]$. Para determinar o valor de $\mathcal{K}$ é necessário contabilizar 1 bloco de tamanho 3!, o que dá origem à parcela $1 \cdot 3!$. Para o cálculo da parcela seguinte é necessário ter em conta que $k_1 = 3$ é maior do que $k_0 = 1$ logo nos símbolos possíveis para a segunda posição não se inclui o 1 e assim é necessário "descontar" um bloco. Assim, tem-se $3-1$ blocos de tamanho 2!, o que dá origem à parcela $2 \cdot 2!$. A figura seguinte ilustra esta situação:

0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3	3
1	1	2	2	3	3	0	0	2	2	3	3	0	0	1	1	3	3	0	0	1	1	2	2
2	3	1	3	1	2	2	3	0	3	0	2	1	3	0	3	0	1	1	2	0	2	0	1
3	2	3	1	2	1	3	2	3	0	2	0	3	1	3	0	1	0	2	1	2	0	1	0

$1 \cdot 3!$ $\quad$ $(3-1) \cdot 2!$

Em geral, se a sequência for $\mathcal{S}_\mathcal{K} = [n, m, ...]$ com $m > n$ vem que, para calcular $\mathcal{K}$, há que contabilizar n blocos de tamanho $(N-1)!$, dando origem a uma parcela do tipo $n \cdot (N-1)!$, e $m-1$ blocos de tamanho $(N-2)!$, dando origem a uma parcela do tipo $(m-1) \cdot (N-2)!$ como se ilustra na figura:

0	$\vdots$	n	$\vdots$	$N-1$
$1 \ldots (N-1)$	$\cdots$	$0 \ \ldots \ (n-1)(n+1) \ \ldots \ m \ \ldots \ (N-1)$	$\cdots$	$0 \ldots (N-2)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

$n \cdot (N-1)!$ $\quad$ $(m-1) \cdot (N-2)!$

Este caso particular mostra que ao determinar α_{N-2}, à custa de k_1, é necessário ter em conta se k_0 é menor do que k_1 e, em caso afirmativo, descontar este valor no número de sequências a contabilizar. Em geral, esta situação verifica-se na determinação dos números das posições seguintes logo é necessário contabilizar os números da permutacão, nas posições anteriores, que são menores do que aquele que se está a considerar, para obter o coeficiente da combinação linear dos factoriais.

Suponha-se que já foram determinados os coeficientes $\alpha_{N-1}, \alpha_{N-2}, \ldots, \alpha_{N-i}$ à custa dos primeiros i elementos da sequência $k_0, k_1, \ldots, k_{i-1}$. Então, as primeiras i parcelas do número de ordem da sequência estão encontradas e o valor da sua soma:

$$\sum_{i=0}^{N-1} \alpha_{N-1-i}(N-1-i)!,$$

corresponde ao número de sequências que precedem as começadas por $k_0, k_1, \ldots, k_{i-1}$. Pretende-se agora determinar α_{N-1-i}, ou seja, o coeficiente da parcela seguinte, o que deve ser feito tendo em conta não só k_i como também os ks anteriores.

Por tudo o que foi afirmado até agora, é claro que o valor de α_{N-1-i} é dado pelo número de valores menores do que k_i e diferentes dos valores da sequência anteriores a k_i, ou seja:

$$\alpha_{N-1-i} = |\{j : j < k_i \wedge (j < k_\ell,\ \forall \ell < i)\}| .$$

O que é o mesmo que dizer que:

$$\alpha_{N-1-i} = k_i - |\{\ell : \ell < i \wedge k_\ell < k_i\}| .$$

Assim, dada a sequência $\mathcal{S}_\mathcal{K} = [k_0, k_1, ..., k_i, ..., k_{N-2}, k_{N-1}]$ o seu valor de ordem $\mathcal{K}$ é dado por:

$$\mathcal{K} = \sum_{i=0}^{N-1} \alpha_{N-1-i} \cdot (N-1-i)!, \tag{1.1}$$

onde

$$\alpha_{N-1-i} = k_i - |\{\ell \in [0, \ldots, i-1] : k_\ell < k_i\}| . \tag{1.2}$$

Algoritmo 1.7.1 (Obtenção do número de ordem $\mathcal{K}$ de uma sequência $\mathcal{S}_\mathcal{K}$)
Entrada: sequência de N números $\mathcal{S}_\mathcal{K} = [k_0, k_1, ..., k_{N-1}]$.
Saída: $\mathcal{K}$ valor no intervalo $[0, ..., N! - 1]$.

1. $j \leftarrow 0$, $\alpha_{N-1-j} \leftarrow k_j$, $\mathcal{K} = \alpha_{N-1-j} \cdot (N-1-j)!$;
2.
   - $j \leftarrow j + 1$,
   - $\alpha_{N-1-j} \leftarrow k_j - |\ell : \ell < j \wedge k_\ell < k_j|$,
   - $\mathcal{K} = \mathcal{K} + \alpha_{N-1-j} \cdot (N-1-j)!$;
3. *se* $j < N-1$ *ir para o passo 2;*
4. *retornar* $\mathcal{K}$.

Exemplo para $|\Sigma| = 4$: Obtenção de $\mathcal{K}$ a partir da sequência $\mathcal{S}_\mathcal{K}$

Dada uma sequência, o número de sequências que a antecedem na ordenação é obtido por uma soma do tipo:

$$\alpha_3 \cdot 3! + \alpha_2 \cdot 2! + \alpha_1 \cdot 1! + \alpha_0 \cdot 0!$$

onde os coeficientes α_i dependem dos símbolos da sequência pois, como já se viu no caso geral, nas sequências não há repetições de números logo ao contabilizar as sequências que

precedem uma dada sequência é necessário ter em conta os números da sequência que já sairam.

Em geral, dada uma sequência $\mathcal{S}_\mathcal{K} = [k_0, k_1, k_2, k_3]$, o cálculo de $\mathcal{K}$ é dado por:

$$\mathcal{K} = \alpha_3 \cdot 3! + \alpha_2 \cdot 2! + \alpha_1 \cdot 1! + \alpha_0 \cdot 0!$$

onde α_{N-1-i} é dado pela quantidade de números inferiores a k_i que ainda não apareceram na sequência.

Considere-se a sequência $\mathcal{S}_\mathcal{K} = [0, 2, 1, 3]$ e observe-se a tabela da ordenação lexicográfica das sequências de 4 números. Esta sequência tem exactamente duas sequências a antecedê-la logo o seu número de ordem é $\mathcal{K} = 2$.

O primeiro número de $\mathcal{S}_\mathcal{K}$ é $k_i = 0$ e na tabela vê-se que não há nenhum bloco de tamanho 3! a anteceder esta sequência, assim é fácil concluir que $\alpha_3 = k_0$, isto é, $\alpha_3 = 0$.

O número seguinte da sequência é $k_1 = 2$ mas na escolha do segundo símbolo o zero não é uma opção viável porque já aparece na primeira posição, então as sequências começadas por zero que antecedem $\mathcal{S}_\mathcal{K}$ só podem começar por um. De facto, para determinar o coeficiente da combinação linear deve subtrair-se uma unidade a k_1 (correspondente ao valor que é inferior a ele, k_0), assim $\alpha_1 = 1$, ou seja, há um bloco de tamanho 1! a anteceder $\mathcal{S}_\mathcal{K}$.

De seguida, tem-se $k_2 = 1$ e, mais uma vez, há nos números anteriores da sequência um valor inferior a este o que se traduz em haver menos um bloco de tamanho 1! a anteceder esta sequência, então $\alpha_1 = k_2 - 1$, ou seja, $\alpha_1 = 0$.

Por fim, $k_3 = 3$ e como é o último número só há uma escolha, logo há apenas um bloco de tamanho 0! a anteceder a sequência. tendo em conta que os k_is anteriores são todos inferiores a k_3 fazendo $\alpha_3 = k_3 - 3$, ou seja, $\alpha_3 = 0$ conduz à mesma conclusão.

Assim obteve-se os valores dos coeficientes α_is:

$$\alpha_i \quad = \quad \begin{cases} 0 & , i = 3 \\ 1 & , i = 2 \\ 0 & , i = 1 \\ 0 & , i = 0 \end{cases}$$

E então $\mathcal{K} = 0 \cdot 3! + 1 \cdot 2! + 0 \cdot 1! + 0 \cdot 0!$, ou seja, $\mathcal{K} = 2$.

1.7.2 Obtenção da sequência $\mathcal{S}_\mathcal{K}$ a partir de $\mathcal{K}$

Pelo que for visto na secção anterior, o número de ordem $\mathcal{K}$ corresponde ao número de sequências que precedem $\mathcal{S}_\mathcal{K}$. Este valor é obtido somando o número de blocos de sequências

de tamanho $(N-1)!$ (que correspondem às sequências que precedem as começadas pelo primeiro elemento) com o número de blocos de tamanho $(N-2)!$ (que, por usa vez, correspondem às que precedem as sequências começadas pelos dois números da sequência) e assim sucessivamente até somar o número de blocos de tamanho 0!.

Deve começar-se por escrever $\mathcal{K}$ na forma 1.1, como uma combinação linear dos factoriais de $(N-1)!$ até 0!, em que os coeficientes são os maiores possiveis, condição que garante a unicidade da escrita nesta forma, para cada valor $\mathcal{K}$, como se observa de seguida:

O valor de α_{N-1} é o maior número tal que $\alpha_{N-1} \cdot (N-1)! \leq \mathcal{K}$.

Determinados $\alpha_{N-1}, \alpha_{N-2}, \ldots, \alpha_{N-1-(i-1)}$, o valor de α_{N-1-i} é dado por:

$$\max \left\{ \alpha_{N-1-i} : \alpha_{N-1-i} \cdot (N-1-i)! \leq \mathcal{K} - \sum_{j=0}^{i-1} \alpha_{N-1-j} \cdot (N-1-j)! \right\}.$$

Este procedimento determina uma sequência de valores $\alpha_{N-1}, \ldots, \alpha_0$ única.

Depois de calculados os coeficientes da combinação linear de factoriais, $\alpha_{N-1}, \ldots, \alpha_0$, pretende obter-se os símbolos da chave, k_is, a partir dos coeficientes, α_is, usando o processo inverso ao da obtenção do valor $\mathcal{K}$ a partir da sequência $\mathcal{S}_{\mathcal{K}}$. O primeiro símbolo da chave coincide com o primeiro coeficiente, ou seja, $k_{N-1} = \alpha_{N-1}$. Para determinar os seguintes é necessário ter em conta, não só os coeficientes, mas também os símbolos da chave já recuperados. Supondo já conhecidos $k_0, k_1, ..., k_{i-1}$, ou seja, os primeiros i números da sequência, o k_i é o único número n, tal que $n \notin [k_0, k_1, ..., k_{i-1}]$ e satisfaz a condição seguinte:

$$n = \alpha_{N-1-i} + \left| \{ \ell : \ell < i \wedge k_\ell < n \} \right|,$$

e então $k_i = n$.

Sejam os coeficientes α_is tais que:

$$\mathcal{K} = \sum_{i=0}^{N-1} \alpha_{N-1-i} \cdot (N-1-i)!.$$

Os números da sequência $\mathcal{S}_{\mathcal{K}}$ pode ser obtidos da seguinte forma:

$$k_i = \alpha_{N-1-i} + \left| \{ \ell \in [0, \ldots, i-1] : k_\ell < k_i \} \right|. \tag{1.3}$$

Esta construção segue um processo inverso ao da secção 1.7.2. De facto, a condição 1.3 que permite determinar k_i, acabada de apresentar, é inversa da condição 1.2 que permite determinar α_{N-1-i} a partir de k_i.

Exemplo para $|\Sigma| = 4$: obtenção da sequência $\mathcal{S}_{\mathcal{K}}$ a partir de $\mathcal{K}$.

Observe-se novamente o exemplo de um alfabeto Σ tal que $|\Sigma| = 4$.

Dado o número de ordem $\mathcal{K} = 2$ pretende-se obter a sequência $\mathcal{S}_{\mathcal{K}}$ que é precedida por duas permutaçõs na ordenação lexicográfica.

Na tabela da ordenação das chaves já apresentada observa-se que a terceira sequência não é precedida por nenhum bloco de sequências de tamanho 3!, no entanto é precedida por um bloco de tamanho 2! e por fim não é precedida por nenhum bloco de tamanho 1!, nem de tamanho 0!. Com esta observação estão encontrados os coeficientes da combinação linear.

A análise da tabela conduz ao mesmo resultado que seria obtido ao escrever $\mathcal{K}$ como combinação linear dos factoriais 3!, 2!, 1! e 0! exigindo que os coeficientes sejam os maiores possíveis, o que já foi visto que garante que essa escrita é única.

Então:

$$\alpha_i \quad = \quad \begin{cases} 0, & i = 3 \\ 1, & i = 2 \\ 0, & i = 1 \\ 0, & i = 0 \end{cases}$$

Agora pretende-se obter os números da sequência à custa dos α_is.

Se $\alpha_3 = 0$ a sequência não é antecedida por nenhum bloco de tamanho 3!, logo o primeiro número deve ser o primeiro possível (que é o zero) então $k_0 = 0$. A seguir tem-se $\alpha_2 = 1$ e o primeiro número da sequência é $k_0 = 0$. Sabe-se que α_2 foi determinado subtraíndo a k_1 os números da sequência que eram inferiores a este, ou seja, $\alpha_2 = k_1 - |\{\ell : \ell < i \wedge k_\ell < k_1\}|$. Então inversamente ter-se-á $k_1 = \alpha_2 + |\{\ell : \ell < 1 \wedge k_\ell < k_1\}|$, com a condição de k_1 ter de ser diferente de k_0, pois em sequências não há repetições de números. Por tentativas, conclui-se que o número dois verifica a condição logo $k_1 = 2$, e de facto, no processo inverso, obteve-se $\alpha_2 = 1$ subtraíndo a $k_1 = 2$ uma unidade correspondente ao facto de $k_0 = 0$ ser inferior a k_1.

Da mesma forma, $\alpha_1 = 0$ conduz a $k_2 = 1$ pois o um é o número (diferente de zero e dois) que verifica $0 = k_2 - |\{\ell : \ell < 2 \wedge k_\ell < k_2\}|$.

Por fim, resta a possibilidde de $k_3 = 3$ o que condiz com o facto de o três ser o número que verifica a condição $0 = k_3 - |\{\ell : \ell < 3 \wedge k_\ell < k_3\}|$.

Concluiu-se assim que:

$$k_i \quad = \quad \begin{cases} 0 & , i = 0 \\ 2 & , i = 1 \\ 1 & , i = 2 \\ 3 & , i = 3 \end{cases}$$

Algoritmo 1.7.2 (Obtenção da sequência $\mathcal{S}_\mathcal{K}$ a partir do valor $\mathcal{K}$)
Entrada: $\mathcal{K} \in [0, ..., N! - 1]$ *e o tamanho do alfabeto,* N.
Saída: sequência $\mathcal{S}_\mathcal{K}$ *de* N *números.*

1. *Cálculo dos coeficientes* $\alpha_i s$*:*
   (a) $f \leftarrow (N-1)!$, $k \leftarrow \mathcal{K}$, $j \leftarrow N-1$;
   (b) $\alpha_j \leftarrow \left\lfloor \frac{k}{f} \right\rfloor$, $k \leftarrow k \bmod f$, $f \leftarrow \left\lfloor \frac{f}{j} \right\rfloor$;
   (c) $j \leftarrow j-1$ *e ir para o passo 1b se* $j > 0$;
   (d) $\alpha_j \leftarrow \left\lfloor \frac{k}{f} \right\rfloor$;
2. *Determinação dos elementos da sequência a partir dos* $\alpha_i s$*:*
   (a) $c = c_0 c_1 ... c_{N-1}$ *com* $c_i = 0, \forall i$;
   (b) $\ell \leftarrow 0$, $k_\ell \leftarrow \alpha_{N-1-\ell}$;
   (c) $\ell \leftarrow \ell + 1$;
   (d) $f \leftarrow 0$, $t \leftarrow \alpha_{N-1-\ell}$, $n \leftarrow t$;
   (e) *enquanto*$(f == 0)$
      i. $m \leftarrow |\{i : i < j \wedge k_i < n\}|$;
      ii. *se* $(\ n - t - m == 0$ *&&* $c_n == 0\ )$ *então* $f \leftarrow 1$;
      iii. *se não* $n \leftarrow n + 1$;
   (f) $k_\ell \leftarrow n$ *e* $c_n \leftarrow 1$;
   (g) *se* $\ell < N - 1$ *ir para o passo 2c;*
3. *retornar* $k_0, k_1, ..., k_{N-2}, k_{N-1}$.

1.7.2.1 Exemplo para $|\Sigma| = 9$

Seja agora Σ um alfabeto com 9 símbolos e $\mathcal{S}_\mathcal{K}$ a sequência $\mathcal{S}_\mathcal{K} = [4, 3, 0, 2, 6, 7, 8, 1, 5]$.

Pelo que foi visto na secção 1.7 esta sequência corresponde a um número de ordem $\mathcal{K}$, que é o número de sequências que a precedem na ordenação lexicográfica. Esse número é obtido contabilizando o número de sequências que precedem as começadas por $k_0 = 4$ (e que são blocos de tamanho 8! pois para cada primeiro número há 8! sequências possíveis), o número de sequências que precedem as começadas por k_0 seguido de k_1 e que correspondem a blocos de tamanho 7! e assim sucessivamente até aos blocos de tamanho 0!. Assim, o número $\mathcal{K}$ é dado por uma soma do tipo:

$$\mathcal{K} = \alpha_8 \cdot 8! + \alpha_7 \cdot 7! + ... + \alpha_1 \cdot 1! + \alpha_0 \cdot 0!, \tag{1.4}$$

onde

$$\alpha_{N-1-i} = k_i - \left|\{\ell \in [0, \ldots, i-1] : k_\ell < k_i\}\right|.$$

Deste modo tem-se:

$$\mathcal{K} = 4 \cdot 8! + 3 \cdot 7! + 0 \cdot 6! + 1 \cdot 5! + 2 \cdot 4! + 2 \cdot 3! + 2 \cdot 2! + 0 \cdot 1! + 0 \cdot 0!.$$

Ficando assim determinado o número de ordem da sequência $\mathcal{S}_\mathcal{K}$ que é 176584.

Inversamente, pretende-se agora obter os símbolos da chave á custa do valor $\mathcal{K}$. Depois de escrever este valor na forma 1.4 obtêm-se os coeficientes α_is e é a partir destes que se determinarão os números da sequência.

Assim:

$$\alpha_{N-1-i} = \begin{cases} 4, & i = 0 \\ 3, & i = 1 \\ 0, & i = 2 \\ 1, & i = 3 \\ 2, & i = 4 \\ 2, & i = 5 \\ 2, & i = 6 \\ 0, & i = 7 \\ 0, & i = 8 \end{cases}$$

Como $\alpha_8 = 4$ vem que $k_0 = 4$. Para os seguintes procura-se k_i tal que $k_i \neq k_j \ \forall l < i$ que satisfaz a condição seguinte:

$$k_i = \alpha_i + \left|\{\ell : \ell < i \wedge k_\ell < k_i\}\right|.$$

Recuperando-se assim a sequência $\mathcal{S}_\mathcal{K}$ inicial:

$$k_i = \begin{cases} 4, & i = 0 \\ 3, & i = 1 \\ 0, & i = 2 \\ 2, & i = 3 \\ 6, & i = 4 \\ 7, & i = 5 \\ 8, & i = 6 \\ 1, & i = 7 \\ 5, & i = 8 \end{cases}$$

Capítulo 2

A Implementação

2.1 A aritmética de precisão arbitrária

Tanto na partilha de chave, usando o protocolo de Diffie-Hellman [1], como na construção da sequência de números, que conduz à chave da cifra Medusa, surge a necessidade de representar inteiros "grandes". Se, por exemplo, se pretender usar um alfabeto com 256 símbolos, a chave da cifra será obtida a partir de um valor (partilhado usando o protocolo DH) que se situa no intervalo $[0, ..., 256! - 1]$, ou seja, corresponde a um número que tem 1684 bits; se o alfabeto tiver 81 símbolos o "valor" que corresponde à chave pode ter 402 bits.

A implementação da Medusa foi efectuada em C++ [6, 8], usando o MSVC [9] como IDE. Existem limites dos CPUs actuais em relação ao tamanho dos inteiros que se podem utilizar, pois só é possível trabalhar com inteiros positivos com, no máximo, 32 bits. Para superar essa limitação foi criada uma estrutura que pudesse representar inteiros de tamanho arbitrariamente grande (tanto quanto a memória o permita) e foram implementadas as operações aritméticas necessárias. Este é o trabalho que se apresenta neste capítulo.

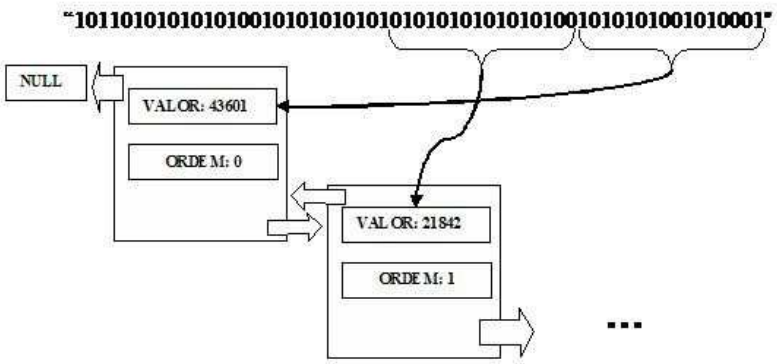
2.1.1 Representação de inteiros não-negativos na base 2^{16}

Perante a necessidade de utilizar números inteiros cujo tamanho pode ir até algumas centenas de bits e sendo 32 bits o tamanho máximo de um *unsigned long int*, foi usada uma forma de representar números inteiros não-negativos de grandes dimensões ao convertê-los para uma dada base e guardando os coeficientes da sua representação nessa base.

Na implementação optou-se pela representação em base 2^{16} e definiu-se como variáveis globais $NBITS = 16$ e $NNBITS = 2^{NBITS} - 1$, sendo que $NNBITS$ representa o maior valor possível para os coeficientes da representação em base 2^{NBITS}. Esta opção foi tomada de forma a utilizar o mais possível a artitmética natural dos CPUs.

A representação de um número na base 2^{NBITS} é constituída por uma sucessão de "cartões" em que cada um possui um *valor*, isto é, o coeficiente, uma *ordem*, ou seja, a posição do coeficiente na representação, um apontador para o "cartão" anterior e um apontador para o "cartão" seguinte. Estes cartões, em conjunto e ordenados convenientemente, constituem a representação do número na base referida.

Refira-se ainda que os valores, ou seja, os dígitos do número representado na base NBITS, estão por ordem crescente de significância, isto é, no primeiro cartão, de ordem 0, está representado o dígito menos significativo e no último cartão está o dígito mais significativo.



Mais concretamente, cada "cartão" é uma estrutura do tipo *struct intpiece* formada por:

- *val*, o coeficiente, armazenado numa variável do tipo *unsigned long int*;
- *ord*, a ordem, armazenada numa variável do tipo *unsigned short int*;
- **prev*, a ligação ao anterior, um apontador do tipo *struct intpiece **;
- **next*, a ligação ao seguinte, um apontador do tipo *struct intpiece **.

A seguir apresenta-se a definição da estrutura:

```
typedef struct intpiece{
 unsigned long val;
 unsigned short ord;
 struct intpiece *prev;
 struct intpiece *next;
} IntPiece;
```

A variável *val*, por ser do tipo *unsigned long int*, como já foi referido, pode tomar valores até 32 bits mas vai ser utilizada de forma a armazenar um número inteiro positivo com no máximo $NNBITS = 2^{NBITS} - 1$, ou seja, metade do tamanho possível. A variável *ord*

representa um inteiro positivo, até 16 bits, que representa o lugar que o "cartão" ocupa na estrutura. Os apontadores **prev* e **next*, estabelecem ligações entre os "cartões" e permitem percorrer a estrutura tanto no sentido ascedente como descendente.

Daqui em diante, para simplificação de linguagem, será utilizado o termo *IntPiece* para designar a representação de um inteiro em base 2^{16}, que poderá ser constituída por um conjunto de estruturas do tipo *IntPiece*.

Representação de um inteiro (com no máximo NBITS bits) em base 2^{16}:

A função ***IntPiece *smallint2IntP(int i)*** representa um inteiro i, com no máximo $NNBITS$ bits, num IntPiece.

```
IntPiece *smallint2IntP(int i){
  IntPiece *pt = new IntPiece;
  pt->val = i;
  pt->ord = 0;
  pt->next = NULL;
  pt->prev = NULL;
  return pt;
}
```

Representação de uma string binária em base 2^{16}:

Dada uma string binária S a função ***IntPiece *Str2IntP(string S)*** representa-a em base 2^{16} usando a estrutura IntPiece.

```
IntPiece *Str2IntP(string S)
{
  IntPiece *l=new IntPiece, *li = l;
  int c_S=S.length();
  int i=0;
  int base=NBITS;
  li->prev=NULL;
  while(c_S>base){
    string sub_S = S.substr(c_S-base,base);
    li->val = ConvertBinaryStringToInteger(sub_S);
    li->ord = i;
    li->next = new IntPiece;
    (li->next)->prev=li;
    li = li->next;
```

```
    S = S.erase(c_S-base,base);
    c_S = S.length();
    i++;
  }
  li->ord = i;
  li->val = ConvertBinaryStringToInteger(S);
  li->next = NULL;
  return l;
}
```

A função auxiliar *unsigned long int ConvertBinaryStringToInteger(string S)*, que é usada na função apresentada anteriormente, converte uma string binária, com um número de bits menor ou igual a NBITS, num inteiro escrito em base decimal.

Transformação de um número representado em base 2^{NBITS} numa string decimal:

Dado um número em base 2^{NBITS}, ou seja, representado usando a estrutura IntPiece, a função ***string IntP2DecimalStr(IntPiece *li)*** converte-o numa string decimal, ou seja, numa string que representa esse número em base decimal.

```
string IntP2DecimalStr(IntPiece *li){
  IntPiece *pt1 = li, *pt2 = new IntPiece, *pt3 = new IntPiece;
  IntPiece *pt = new IntPiece, *ptd = new IntPiece;
  string DC = "";
  pt2 = smallint2IntP(10);
  pt = smallint2IntP(0);
  ptd = DuplicateIntP(pt);
  pt3 = ptd;
  while( LessEqualBiggerIntP( pt1 , pt2 ) > 0 ){
    EuclDivIntPbysmallint(  pt1, pt2 , &pt1, &pt);
    ptd->val = pt->val;
    ptd->next = new IntPiece;
    (ptd->next)->prev = ptd;
    (ptd->next)->next = NULL;
    (ptd->next)->ord = ptd->ord + 1;
    ptd = ptd->next;
  }
  ptd->val = pt1->val;
  ptd->next = NULL;
  while(ptd){
    int temp = ptd->val;
```

```
    DC.append(1, temp + 48);
    ptd = ptd -> prev;
  }
  return DC;
}
```

2.1.2 Soma de dois inteiros não-negativos

Uma vez que acabou de se definir uma representação de inteiros não-negativos de grandes dimensões, maiores do que o que é suportado pelos CPUs, é necessário agora implementar as operações básicas para efectuar cálculos com estes números. Comece-se pela adição.

Dados dois inteiros não negativos, a e b, escritos na base β, com comprimentos m e n, respectivamente: $a = \sum_{i=0}^{m-1} a_i\beta^i$ e $b = \sum_{i=0}^{n-1} b_i\beta^i$, irá apresentar-se um algoritmo que permite calcular a sua soma.
Seja $m \geq n$, sem perda de generalidade, então pode escrever-se $b = \sum_{i=0}^{m-1} b_i\beta^i$, sendo que $b_n = b_{n+1} = ... = b_{m-1} = 0$.
Assim, a soma de a e b é dada por: $c = \sum_{i=0}^{m-1} (a_i + b_i)\beta^i$.

Algoritmo 2.1.1 (Adição de inteiros não-negativos)
Entrada: a, b *inteiros não-negativos tais que* $a = \sum_{i=0}^{m-1} a_i\beta^i$ *e* $b = \sum_{i=0}^{n-1} b_i\beta^i$.
Saída: $c = a + b = \sum_{i=0}^{m} c_i\beta^i$.

1. $r \leftarrow 0$, $i \leftarrow 0$;
2. $s \leftarrow a_i + b_i + r$;
3. *se* $s < \beta$ *então* $c_i \leftarrow s$ *e* $r \leftarrow 0$;
4. *caso contrário* $c_i \leftarrow s - \beta$ *e* $r \leftarrow 1$;
5. $i \leftarrow i + 1$ *e se* $i < m$ *ir para o passo 2;*
6. *se* $r > 0$ *então* $c_m = 1$;
7. *se não então* $c_m = 0$.
8. *retornar* $c = c_0...c_m$.

A implementação do algoritmo 2.1.1 [5] da soma de inteiros não-negativos consiste na função ***IntPiece *SumIntP(IntPiece *li, IntPiece *lj)*** que será apresentada a seguir.

Refira-se que nesta implementação os apontadores li e lj apontam para os *IntPiece* que armazenam os dois números a somar e os valores armazenados na variável *val* de cada um dos cartões correspondem aos digítos da representação dos números na base 2^{NBITS}. Tendo o apontador li num dado cartão, $li \rightarrow val$ e $li \rightarrow ord$ correspondem, respectivamente, ao valor e à ordem do cartão. E da mesma forma, $li \rightarrow next$ e $li \rightarrow prev$ correspondem, respectivamente, aos cartões seguinte e anterior na estrutura.

```
IntPiece *SumIntP(IntPiece *li, IntPiece *lj){
  int o=0;
  unsigned long carry=0, sum=0;
  IntPiece *pt1=new IntPiece, *pt = pt1;
  while(li || lj || carry){
    if(o){
      pt->next = new IntPiece;
      (pt->next)->prev = pt;
      pt = pt->next;
    }
    else pt->prev = NULL;
    sum = (li?li->val:0) + (lj?lj->val:0) + carry;
    pt->val = sum & NNBITS;
    carry = sum>>NBITS;
    pt->ord = o++;
    li = li?li->next:NULL;
    lj = lj?lj->next:NULL;
  }
  pt->next = NULL;
  return pt1;
}
```

2.1.3 Subtracção

Pretende-se agora definir a subtracção de dois inteiros de grandes dimensões.

Dados dois inteiros não negativos, a e b, escritos na base β, com comprimentos m e n, respectivamente: $a = \sum_{i=0}^{m-1} a_i\beta^i$ e $b = \sum_{i=0}^{n-1} b_i\beta^i$, apresentar-se-á de seguida um algoritmo para calcular a diferença $a - b$ sendo $a \geq b$.

Algoritmo 2.1.2 (Subtracção de inteiros não-negativos escritos na base β)

Entrada: a, b *inteiros não-negativos tais que* $a = \sum_{i=0}^{m-1} a_i\beta^i$ *e* $b = \sum_{i=0}^{n-1} b_i\beta^i$, *com* $a \geq b$.

Saída: $c = a - b = \sum_{i=0}^{m-1} c_i\beta^i$.

1. $r \leftarrow 0$, $i \leftarrow 0$;
2. $s \leftarrow a_i - b_i - r$;
3. *se* $s < 0$ *então* $c_i \leftarrow (s + \beta - 1)$ *e* $r \leftarrow 1$;
4. *caso contrário* $c_i \leftarrow s$ *e* $r \leftarrow 0$;
5. $i \leftarrow i + 1$ *e se* $i < m$ *ir para o passo 2*;
6. *retornar* $c = c_0...c_{m-1}$.

Refira-se que nesta implementação os apontadores li e lj apontam para os *IntPiece* que armazenam os dois números a operar e os valores armazenados na variável *val* de cada um dos cartões correspondem aos digítos da representação dos números na base 2^{NBITS}. Tendo o apontador li num dado cartão, $li \rightarrow val$ e $li \rightarrow ord$ correspondem, respectivamente, ao valor e à ordem do cartão. E da mesma forma, $li \rightarrow next$ e $li \rightarrow prev$ correspondem, respectivamente, aos cartões seguinte e anterior na estrutura.

A implementação do algoritmo 2.1.2 [5] de subtracção de números não-negativos, ou seja, a função ***IntPiece *SubIntP(IntPiece *li, IntPiece *lj)*** é apresentada a seguir:

```
IntPiece *SubIntP(IntPiece *li,IntPiece *lj){
  int o=0;
  unsigned long carry=0, sub=0, a, b;
  IntPiece *pt1=new IntPiece, *pt = pt1, *pt2;
  while(li || lj || carry){
    if(o){
      pt->next = new IntPiece;
      (pt->next)->prev = pt;
      pt = pt->next;
    } else
      pt->prev = NULL;
    a = (li?li->val:0);
    b = (lj?lj->val:0);
    if (a<(b+carry)){
      sub = NNBITS+1+a-b-carry;
      carry = 1;
    } else {
```

```
      sub = a-b-carry;
      carry = 0;
    }
    pt->val = sub;
    pt->ord = o++;
    li = li?li->next:NULL;
    lj = lj?lj->next:NULL;
  }
  pt->next = NULL;
  while((!pt->val) && pt->prev){
    (pt->prev)->next = NULL;
    pt2 = pt;
    pt = pt->prev;
    delete pt2;
  }
  return pt1;
}
```

2.1.4 Multiplicação

Depois de definidas a adição e a subtração de inteiros não-negativos de grandes dimensões representados usando a esctrutura IntPiece pretende-se agora definir a multiplicação.

Sejam a e b, dois inteiros não negativos, escritos na base β, com comprimentos u e v, respectivamente: $a = \sum_{i=0}^{u-1} a_i\beta^i$ e $b = \sum_{i=0}^{v-1} b_i\beta^i$. Então o produto $c = a \cdot b$, é dado por $c = \sum_{k=0}^{u+v-1} \sum_{i+j=k} (a_i \cdot b_j)\beta^{i+j}$.

O algoritmo utilizado é o Algoritmo de Karatsuba [2]. Na bibliografia este algoritmo é apresentado na sua versão para polinómios a qual foi adaptada para números representados numa base β qualquer. A ideia essencial deste algoritmo é diminuir o número de multiplicações intermédias necessárias ao cálculo de determinado produto, usando a ideia que se apresenta a seguir.

Para calcular o produto $(ax + b)(cx + d)$ usando:

$$(ax + b)(cx + d) = acx^2 + (ad + bc)x + bd,$$

é necessário efectuar quatro multiplicações e uma adição. No entanto, se se calcular ac, bd, $u = (a + b)(c + d)$ e $ad + bc = u - ac - bd$ determina-se o mesmo produto efectuando apenas três multiplicações e quatro adições e subtracções:

$$(ax + b)(cx + d) = acx^2 + [(a + b)(c + d) - ac - bd)]\, x + bd.$$

Esta ideia pode ser aplicada de forma análoga a inteiros não negativos, a e b, escritos em base β: sejam $a = A_1 \cdot \beta + A_0$ e $b = B_1 \cdot \beta + B_0$, com $A_0, A_1, B_0, B_1 < \beta$. Então:

$$a \cdot b = A_1 B_1 \cdot \beta^2 + ((A_0 + A_1)(B_0 + B_1) - A_0 B_0 - A_1 B_1) \cdot \beta + A_0 B_0.$$

Este método, apresentado para o caso mais simples, mostra o seu poder quando aplicado a números maiores em que o problema do cálculo do seu produto é reduzido sucessivamente ao problema do cálculo do produto de números menores (e também a algumas adições e subtracções).

Observe-se como este algoritmo oferece sérias vantagens quando aplicado, por exemplo, recursivamente. Dados os inteiros não-negativos a e b, escritos na base β, com $a \geq b$ e a tem comprimento $m = 2^s$ (para algum s), para calcular o produto $a \cdot b$ começa-se por escrever os números na seguinte forma: $a = A_1 \cdot \beta^m + A_0$ e $b = B_1 \cdot \beta^m + B_0$, com $A_0, A_1, B_0, B_1 < \beta^m$. Ao reescrever o produto de a por b como

$$a \cdot b = A_1 B_1 \cdot \beta^{2m} + ((A_0 + A_1)(B_0 + B_1) - A_0 B_0 - A_1 B_1) \cdot \beta^m + A_0 B_0.$$

resulta que este requer apenas três multiplicações de números menores que β^m e algumas adições e subtracções.

Para efectuar o cálculo de $A_1 \cdot B_1$, $A_0 \cdot B_0$ e $(A_0 + A_1) \cdot (B_0 + B_1)$ é aplicado de novo o algoritmo, sendo que agora os números em causa serão reescritos como uma expressão do tipo $X_0 \cdot \beta^{\frac{m}{2}} + X_1$, com $X_0, X_1 < \beta^{\frac{m}{2}}$. Este procedimento é aplicado recursivamente até se obter apenas produtos de números de comprimento um.

Tem-se então:

Algoritmo 2.1.3 (Algoritmo de Karatsuba para a multiplicação de inteiros)
Entrada: a, b *inteiros não-negativos menores ou iguais a* β^n*, onde* n *é uma potência de* 2*.*
Saída: $a \cdot b$*.*

1. *se* $n = 1$ *retornar* $a \cdot b \in N$*.*
2. *seja* $a = A_1 \beta^{\frac{n}{2}} + A_0$ *e* $b = B_1 \beta^{\frac{n}{2}} + B_0$*, com* A_0, A_1, B_0, B_1 *menores que* $\beta^{\frac{n}{2}}$*,*
3. *calcular* $A_0 B_0, A_1 B_1$ *e* $(A_0 + B_0)(A_1 + B_1)$ *recursivamente,*
4. *retornar* $A_1 B_1 \cdot \beta^n + ((A_0 + A_1)(B_0 + B_1) - A_0 B_0 - A_1 B_1) \cdot \beta^{\frac{n}{2}} + A_0 B_0$*.*

Note-se que no algoritmo apresentado se supõe que n é uma potência de 2. Mas, se isto não acontecer o algoritmo é aplicado da mesma forma desde que se divida os números em dois blocos de "aproximadamente" metade do tamanho do maior.

Se, por exemplo, $a \geq b$ e n é o comprimento de a, então, no passo 2 do algoritmo faz-se:

$$a = A_1\beta^{\frac{n+1}{2}-1} + A_0 \text{ e } b = B_1\beta^{\frac{n+1}{2}-1} + B_0.$$

Uma vez que $\frac{n+1}{2} - 1 = \frac{n}{2}$ se n é par e $\frac{n+1}{2} - 1 = \left\lfloor \frac{n}{2} \right\rfloor + 1$ se n é ímpar obtêm-se, respectivamente, a divisão de a exactamente ao meio: $a = A_1\beta^{\frac{n}{2}} + A_0$ (com $A_0, A_1 < \beta^{\frac{n}{2}}$) e uma divisão em que $A_1 > A_0$: $a = A_1\beta^{\left\lfloor \frac{n}{2} \right\rfloor + 1} + A_0$ (com $A_0, A_1 < \beta^{\left\lfloor \frac{n}{2} \right\rfloor + 1}$). O resto do cálculo processa-se de forma análoga à apresentada no algoritmo apenas tendo em conta que, em cada iteração, o valor de n é dado por $n = \frac{n+1}{2} - 1$.

A função ***IntPiece *KaratR(IntPiece *li, IntPiece *lj)*** implementa o algoritmo 2.1.3 na sua forma recursiva. Esta função tem como input dois IntPieces *li e *lj e devolve um IntPiece que é o produto de *li e *lj.

Refira-se ainda que nesta implementação cada apontador do tipo *IntPiece ** aponta para um *IntPiece* (que armazena um número inteiro) e os valores armazenados na variável *val* de cada um dos cartões correspondem aos digítos da representação do número na base 2^{NBITS}. Tendo o apontador li num dado cartão, $li \rightarrow val$ e $li \rightarrow ord$ correspondem, respectivamente, ao valor e à ordem do cartão. E da mesma forma, $li \rightarrow next$ e $li \rightarrow prev$ correspondem, respectivamente, aos cartões seguinte e anterior na estrutura.

A função *KaratR* usa, entre outras, as funções auxiliares que a seguir se referem e das quais se faz uma descrição breve.

- int LenIntP(IntPiece *pt): determina o comprimento do IntPiece apontado por pt, ou seja, o seu número de "cartões";
- IntPiece *SplitIntP(IntPiece *li, int lastord): devolve (um apontador para?) um IntPiece constituído pelos cartões de li de ordem superior a lastord e transforma o IntPiece li num IntPiece formado pelos cartões até lastord (inclusivé);
- IntPiece *MulIntP(IntPiece *li,IntPiece *lj): devolve (um apontador para?) um IntPiece com, no máximo comprimento dois, que é o produto de dois IntPieces de comprimento um apontados por li e lj;
- IntPiece *DuplicateIntP(IntPiece *i): devolve (um apontador para?) um IntPiece que é um duplicado do IntPiece apontado por i;
- IntPiece *ShiftIntP(int i, IntPiece *li): desloca os valores do IntPiece apontado por li um número de ordens para a direita igual a i e coloca o valor nulo nos i cartões iniciais;

A implementação do algoritmo na sua forma recursiva é apresentada a seguir:

```
IntPiece *KaratR(IntPiece *li, IntPiece *lj){
  int ti, tj, n;
  ti = LenIntP(li);
```

```
  tj = LenIntP(lj);
  n = max(ti,tj);
  if(n > 1){
    IntPiece *li1, *lj1, *li0, *lj0, *lie, *lje, *kl0, *kl1, *r,
      *t1, *t2, *t3, *t4, *t5, *t6;
    n = (n+1)/2-1;
    li0 = DuplicateIntP(li);
    lj0 = DuplicateIntP(lj);
    li1 = SplitIntP(li0,n);
    lj1 = SplitIntP(lj0,n);
    lie = SumIntP(li0,li1);
    lje = SumIntP(lj0,lj1);
    kl0 = KaratR(li0,lj0);
    DeleteIntP(li0);
    DeleteIntP(lj0);
    kl1 = KaratR(li1,lj1);
    DeleteIntP(li1);
    DeleteIntP(lj1);
    t4  = ShiftIntP(2*n+2,kl1);
    t3 = SumIntP(kl0,kl1);
    t5 = KaratR(lie,lje);
    DeleteIntP(lie);
    DeleteIntP(lje);
    t2 = SubIntP(t5,t3);
    DeleteIntP(t3);
    DeleteIntP(t5);
    t1 = ShiftIntP(n+1,t2);
    t6 = SumIntP(t1,t4);
    DeleteIntP(t4);
    DeleteIntP(t1);
    r = SumIntP(kl0,t6);
    DeleteIntP(kl0);
    DeleteIntP(t6);
    return r;
  } else
    return MulIntP(li,lj);
}
```

2.1.5 Divisão Euclidiana

Por fim, pretende-se definir a divisão euclidiana de dois inteiros não-negativos representados usando a estrutura IntPiece.

Dados dois números inteiros positivos, a e b, existem um único inteiro positivo q e um único inteiro não negativo r tais que:

$$a = b \cdot q + r,$$

onde $0 \leq r < b$.

O número q é o quociente da divisão de a por b e será aqui denotado por $Quo(a, b)$ e o número r é o resto dessa divisão e será aqui denotado por $Res(a, b)$.

Notação 1 *Seja x um número inteiro não-negativo cuja representação na base β tem n dígitos. A expressão $x = (x_0...x_{n-1})_\beta$ denota a sua representação na base β sendo x_0 o dígito mais significativo.*

Dados os números a e b cujas representações numa base β qualquer, são, respectivamente, $a = (a_0...a_{m+n-1})_\beta$ e $b = (b_0...b_{n-1})_\beta$, com $b_0 \neq 0$, o quociente é dado por $Quo(a, b) = \left\lfloor \frac{a}{b} \right\rfloor = (q_0...q_m)_\beta$ e o resto é $Res(a, b) = a \ mod \ b = (r_0...r_{n-1})_\beta$.

Tal como o que se verifica no algoritmo usual de divisão, o algoritmo que vai ser utilizado tem em conta o facto de ao dividir um número cuja representação na base β tem tamanho $m + n$ por um outro cuja representação na base β tem tamanho n ser possível desdobrar o problema em passos intermédios nos quais o dividendo, a, tem tamanho $n + 1$ e o divisor, b, tem tamanho n, com $0 \leq \left\lfloor \frac{a}{b} \right\rfloor < \beta$ e, dado que em cada passo o resto é menor do que o divisor, pode usar-se a quantidade $rb+$*"posição seguinte do dividendo"* como dividendo no passo seguinte [3].

Tendo em conta o que acabou de ser referido então a procura de um algoritmo de divisão reduz-se à resolução da questão seguinte:

Dados dois inteiros $a = (a_0...a_n)_\beta$ e $b = (b_0...b_{n-1})_\beta$, não-negativos escritos em base β, com $a < b\beta$, encontre-se um algoritmo que determine $Quo(a, b) = \left\lfloor \frac{a}{b} \right\rfloor$.

Antes de avançar note-se que no algoritmo usual da divisão, em cada passo, o dividendo pode ter o mesmo número de dígitos do dividendo ou mais um. No entanto, no algoritmo que será apresentado acontece que, em cada passo, a excede b em um dígito sendo verificada a condição $a < b\beta$. Será visto mais adiante como se garante que esta condição é verificada.

Considere-se a base decimal, ou seja, $\beta = 10$. Na divisão de $(3142)_\beta$ por $(47)_\beta$, no primeiro passo, tomar-se-á como dividendo $(314)_\beta$ obtendo $(6)_\beta$ como quociente e $(032)_\beta$ como resto, no passo seguinte o dividendo é $(322)_\beta$ obtendo o quociente $(6)_\beta$ e $(040)_\beta$ como resto. Por fim, o quociente da divisão é 66 e o resto é 40. Observe-se que a condição em causa foi sendo verificada em cada passo. No entanto, se se estivesse a dividir 5142 por 47 qual seria o

dividendo no primeiro passo? De acordo com as nossas condições o divisor deve ter mais um dígito do que o dividendo mas não pode ser 514 pois não é verdade que $\frac{514}{47} < \beta$. O mesmo problema se colocaria se estivéssemos a dividir apenas 514 por 47. Voltar-se-á a esta questão mais adiante.

2.1.5.1 Divisão Euclidiana de inteiros não-negativos $a = (a_0...a_{n-1})_\beta$ e $b = (b_0)_\beta$, com $n \geq 1$.

Comece-se por apresentar o algoritmo da divisão de a, com n dígitos na base β, por b composto por apenas um dígito na base β.

A ideia do algoritmo que se segue consiste em tomar para dividendo o primeiro dígito de a, a_0, dividí-lo por b, obtendo o primeiro dígito do quociente, e de seguida obter o resto calculando $a_0 \mod b$. Nos passos seguintes, o novo dividendo é obtido somando o resto ao dígito seguinte de a multiplicado por β e repete-se o procedimento anterior para calcular o novo valor do resto. No algoritmo seguinte, a variável r representa o valor do resto e a variável j representa as posições dos dígitos de a.

Algoritmo 2.1.4 (Divisão Euclidiana de $a = (a_0...a_{n-1})_\beta$ por $b = (b_0)_\beta$)
Entrada: a, b inteiros não-negativos, tais que $a = (a_0...a_n)_\beta$ e $b = (b_0)_\beta$.
Saída: $Quo(a,b) = \left\lfloor \frac{a}{b} \right\rfloor$ e $Res(a,b) = a \mod b$.

1. $r \leftarrow 0$, $j \leftarrow 0$;
2. $q_j \leftarrow \left\lfloor \frac{r \cdot \beta + a_j}{b} \right\rfloor$, $r \leftarrow (r \cdot \beta + a_j) \mod b$;
3. $j \leftarrow j + 1$ *e ir para o passo 2 se* $j \leq n - 1$.
4. *retornar* $Quo(a,b) = (q_0...q_{n-1})_\beta$ *e* $Res(a,b) = r = (r_0)_\beta$.

A implementação do algoritmo 2.1.4 consiste na função ***void EuclDivIntPbysmallint(IntPiece *li, IntPiece *lj, IntPiece **quo, IntPiece **res)*** que é apresentada a seguir:

```
void EuclDivIntPbysmallint(IntPiece *li,IntPiece *lj,
                           IntPiece **quo,IntPiece **res){
  IntPiece *pt1 = li, *pt2 = lj, *pt3 = new IntPiece;
  int j=0;
  unsigned long int w, r=0;
  int n=LenIntP(pt1);
  int nn = LessEqualBiggerIntP(pt1,pt2);
  pt1 = GoToLastOrd(pt1);
```

```
  if (! nn){
    *quo=smallint2IntP(0);
    *res=DuplicateIntP(pt1);
  }
  if (nn == 1){
    *quo=smallint2IntP(1);
    *res=smallint2IntP(0);
  }
  if (nn == 2){
    pt3 = ZeroIntP(0);
    while(pt1 && j<n){
      w = r * (NNBITS+1) + (pt1->val);
      w = w / (pt2->val);
      pt3 -> val = w;
      r = r * (NNBITS+1) + pt1->val;
      r = r % (pt2->val);
      j++;
      pt1 = pt1->prev;
      if (pt1) pt3=ShiftIntP(1,pt3);
    }
    *quo=DuplicateIntP(pt3);
    *res=smallint2IntP(r);
  }
}
```

2.1.5.2 Divisão Euclidiana de inteiros $a = (a_0...a_n)_\beta$ e $b = (b_0...b_{n-1})_\beta$, tais que $0 < a < b\beta$.

Sejam a e b dois números inteiros não-negativos, tais que $0 < a < b\beta$, cujas representações na base β são $a = (a_0...a_n)_\beta$ e $b = (b_0...b_{n-1})_\beta$, respectivamente.

Pretende-se expôr um algoritmo que determine o quociente e o resto da divisão de a por b, ou seja, $Quo(a, b) = \left\lfloor \frac{a}{b} \right\rfloor$ e $Res(a, b) = a \ mod \ b$.

Seja $\hat{q}$ o valor candidato para o quociente da divisão de a por b. A abordagem mais óbvia para obter este valor é basear o "palpite" nos dígitos mais significativos de a e b. Tal como se faz no algoritmo que se utiliza usualmente para a divisão, considera-se o número formado pelos dois dígitos mais significativos de a, ou seja, $a_0\beta + a_1$, e vê-se "quantas vezes b_0 cabe em $a_0\beta + a_1$". A fórmula seguinte dá uma aproximação que é bastante razoável:

$$\hat{q} = min\left(\left\lfloor \frac{a_0 \cdot \beta + a_1}{b_0} \right\rfloor, \beta - 1\right), \tag{2.1}$$

como se mostrará no resultado apresentado mais à frente.

Observe-se um exemplo em base decimal, ou seja, $\beta = 10$. Sejam $a = (314)_\beta$ e $b = (47)_\beta$. Usando a fórmula anterior, vem que o valor candidato a quociente é dado por:

$$\hat{q} = min\left(\left\lfloor \frac{3 \cdot 10 + 1}{4} \right\rfloor, 10 - 1\right),$$

ou seja,

$$\hat{q} = 7.$$

O valor de $\hat{q}$ assim obtido representa uma boa aproximação, uma vez que o valor correcto é 6.

Teorema 2.1.5.1 *Seja $Quo(a,b) = \left\lfloor \frac{a}{b} \right\rfloor$ o quociente da divisão de a por b onde $a = (a_0...a_n)_\beta$ e $b = (b_0...b_{n-1})_\beta$ e $0 < a < b\beta$. Seja $\hat{q}$ o valor calculado pela fórmula 2.1. Então:*

1. *$\hat{q} \geq Quo(a,b)$;*
2. *se $b_0 \geq \left\lfloor \frac{\beta}{2} \right\rfloor$ então $\hat{q} - 2 \leq Quo(a,b)$.*

Prova.

1. Se $\hat{q} = \beta - 1$ então, como $a < b\beta \Rightarrow Quo(a,b) \leq \beta - 1$ resulta que $\hat{q} \geq Quo(a,b)$. Caso contrário, tem-se $\hat{q} = \left\lfloor \frac{a_0\beta + a_1}{b_0} \right\rfloor$ e como:

$$\hat{q} > \frac{a_0\beta + a_1}{b_0} - 1 \Longrightarrow \hat{q}b_0 > a_0\beta + a_1 - b_0,$$

vem $\hat{q} \cdot b_0 \geq a_0\beta + a_1 - b_0 + 1$. Resultando assim que:

$$a - \hat{q}b \leq a - \hat{q}b_0\beta^{n-1} \leq a_0\beta^n + ... + a_n - (a_0\beta^n + a_1\beta^{n-1} - b_0\beta^{n-1} + \beta^{n-1}) =$$

$$= a_2\beta^{n-2} + ... + a_n - \beta^{n-1} + b_0\beta^{n-1} < b_0\beta^{n-1} \leq b.$$

Portanto $a - \hat{q}b < b$, de onde se conclui que $\hat{q} \geq Quo(a,b)$.

2. Suponha-se que $\hat{q} - 2 > Quo(a,b)$, o que é equivalente a $\hat{q} \geq Quo(a,b) + 3$. Como $\hat{q} \leq \left\lfloor \frac{a_0 \cdot \beta + a_1}{b_0} \right\rfloor$ e

$$\left\lfloor \frac{a_0 \cdot \beta + a_1}{b_0} \right\rfloor \leq \frac{a_0 \cdot \beta + a_1}{b_0} = \frac{a_0 \cdot \beta^n + a_1 \cdot \beta^{n-1}}{b_0 \cdot \beta^{n-1}} \leq \frac{a}{b_0 \cdot \beta^{n-1}} < \frac{a}{b - \beta^{n-1}},$$

vem que:

$$\hat{q} < \frac{a}{b - \beta^{n-1}}. \tag{2.2}$$

Por outro lado,

$$Quo(a,b) = \left\lfloor \frac{a}{b} \right\rfloor \geq \frac{a}{b} - 1. \tag{2.3}$$

Da hipótese e por (2.2) e (2.3) vem que:

$$3 \leq \hat{q} - Quo(a,b) < \frac{a}{b - \beta^{n-1}} - \frac{a}{b} + 1 =$$

$$= \frac{a}{b}\left(\frac{1}{1 - \frac{\beta^{n-1}}{b}} - 1\right) + 1 = \frac{a}{b}\left(\frac{\beta^{n-1}}{b - \beta^{n-1}}\right) + 1.$$

Logo,

$$\frac{a}{b} > 2\left(\frac{b - \beta^{n-1}}{\beta^{n-1}}\right) = 2\left(\frac{b}{\beta^{n-1}} - 1\right) \geq 2(b_0 - 1).$$

Portanto,

$$\frac{a}{b} > 2(b_0 - 1). \tag{2.4}$$

Como $\beta - 1 \geq \hat{q} \Leftrightarrow \beta - 4 \geq \hat{q} - 3$, por (2.4) resulta que:

$$\beta - 4 \geq \hat{q} - 3 \geq \left\lfloor \frac{a}{b} \right\rfloor \geq 2(b_0 - 1).$$

E assim,

$$\beta - 4 \geq 2(b_0 - 1) \Leftrightarrow \frac{\beta}{2} - 1 \geq b_0 \Leftrightarrow \left\lfloor \frac{\beta}{2} \right\rfloor > b_0.$$

Provou-se que $\hat{q} - 2 > Quo(a,b) \Rightarrow b_0 < \left\lfloor \frac{\beta}{2} \right\rfloor$, o que é o mesmo que provar que:

$$b_0 \geq \left\lfloor \frac{\beta}{2} \right\rfloor \Rightarrow \hat{q} - 2 \leq Quo(a,b).$$

□

Note-se que na segunda parte do Teorema 2.1.5.1 é imposta a condição $b_0 \geq \left\lfloor \frac{\beta}{2} \right\rfloor$. Uma forma de garantir que esta condição se verifica é multiplicar os valores iniciais a e b pela quantidade $\left\lfloor \frac{\beta}{b_0+1} \right\rfloor$, denotada por d, obtendo assim novos valores para o dividendo e divisor que estão nas condições pretendidas. Quando se fala na divisão de $a = (a_0...a_n)_\beta$ por $b = (b_0...b_{n-1})_\beta$, com $0 < a < b\beta$, estes valores foram obtidos a partir de $a = (a_0...a_{n-1})_\beta$ e $b = (b_0...b_{n-1})_\beta$ multiplicando-os por $d = \left\lfloor \frac{\beta}{b_0+1} \right\rfloor$.

Este procedimento, isto é, a multiplicação do dividendo e do divisor pela mesma quantidade d é executado num passo prévio ao algoritmo de divisão que é apresentado mais à frente. Será provado no Teorema 2.1.5.2, que ao executá-lo podemos garantir, que o valor de $Quo(a,b)$ não se altera, que a representação de b na base β não aumenta em número de dígitos, que

o "novo" valor de b_0 verifica a condição $b_0 \geq \lfloor \frac{\beta}{2} \rfloor$ e por fim, que os novos valores de a e b verificam $a < b\beta$.

Refira-se ainda que ao efectuar esta "transformação" dos valores do divisor e do dividendo resolve-se a questão de decidir se se toma, para o dividendo, o mesmo número de dígitos do divisor ou mais um, problema que no algoritmo usual se resolve por tentativa e erro.

Teorema 2.1.5.2 *Sejam a e b dois inteiros não-negativos tais que $a = (a_0...a_{n-1})_\beta$ e $b = (b_0...b_{n-1})_\beta$, na base β e $a > b$. Seja $d = \left\lfloor \frac{\beta}{b_0+1} \right\rfloor$, $(d \geq 1)$. Então:*

1. *$Quo(a, b) = Quo(a \cdot d, b \cdot d)$;*
2. *sendo $\widetilde{b} = b \cdot d$, tem-se $\widetilde{b} < \beta^n$;*
3. *sendo $\widetilde{b} = \widetilde{b}_0\beta^{n-1} + ... + \widetilde{b}_{n-1}$, tem-se $\widetilde{b}_0 \geq \lfloor \frac{\beta}{2} \rfloor$;*
4. *sendo $\widetilde{a} = \widetilde{a}_0\beta^{n-1} + ... + \widetilde{a}_{n-1}$, tem-se $\widetilde{a} < \widetilde{b}\beta$.*

Prova.

1. Resulta imediatamente de:
$$Quo(a \cdot d, b \cdot d) = \left\lfloor \frac{a \cdot d}{b \cdot d} \right\rfloor = \left\lfloor \frac{a}{b} \right\rfloor = Quo(a, b).$$

2. Seja $b = b_0\beta^{n-1} + b_1\beta^{n-2} + \cdots + b_{n-1}$. Então:
$$b \leq b_0\beta^{n-1} + \beta^{n-1} - 1 = (b_0 + 1)\beta^{n-1} - 1 < (b_0 + 1)\beta^{n-1}.$$
Ou seja,
$$b < (b_0 + 1)\beta^{n-1}. \tag{2.5}$$
Por outro lado,
$$d \leq \frac{\beta}{b_0 + 1}. \tag{2.6}$$
De (2.5) e (2.6) resulta que:
$$\widetilde{b} = b \cdot d < \beta^n.$$

3. Seja $\widetilde{b} = b \cdot d = \widetilde{b}_0\beta^{n-1} + ... + \widetilde{b}_{n-1}$. Como, pela segunda parte deste teorema, o número de dígitos de $\widetilde{b}$ é igual ao de b, tem-se que $\widetilde{b}_0 \geq b_0 d$.
   - Se $b_0 \geq \lfloor \frac{\beta}{2} \rfloor$, como $d \geq 1$ então $\widetilde{b}_0 \geq \lfloor \frac{\beta}{2} \rfloor$;

- Se $1 \leq b_0 < \lfloor \frac{\beta}{2} \rfloor$, por um lado $b_0 \left\lfloor \frac{\beta}{b_0+1} \right\rfloor > b_0 \left(\frac{\beta}{b_0+1} - 1 \right)$ e, por outro lado, tem-se que $b_0 \left(\frac{\beta}{b_0+1} - 1 \right) \geq \left(\frac{\beta}{2} - 1 \right)$, porque $b_0 \left(\frac{\beta}{b_0+1} - 1 \right) - \left(\frac{\beta}{2} - 1 \right) = \frac{(\frac{\beta}{2} - b_0 - 1)(b_0 - 1)}{b_0+1} \geq 0$. Assim, vem que:

$$\widetilde{b}_0 > b_0 \left(\frac{\beta}{b_0+1} - 1 \right) \geq \frac{\beta}{2} - 1. \tag{2.7}$$

  Como $\frac{\beta}{2} - 1 \geq \lfloor \frac{\beta}{2} \rfloor - 1$ de (2.7) resulta que $\widetilde{b}_0 > \lfloor \frac{\beta}{2} \rfloor - 1$ e portanto $\widetilde{b}_0 \geq \lfloor \frac{\beta}{2} \rfloor$.

4. Como a só tem um dígito a mais do que b, tem-se que $a < b\beta$ e assim:

$$\widetilde{a} = ad < bd\beta = \widetilde{b}\beta.$$

□

Analise-se o exemplo da divisão de 51 por 47. Se for efectuada a multiplicação de a e b pelo valor $d = \left\lfloor \frac{\beta}{b_0+1} \right\rfloor = 2$ obter-se-ão os novos valores $a \leftarrow a \cdot d = (102)_\beta$ e $b \leftarrow b \cdot d = (94)_\beta$. O dividendo a considerar será $(102)_\beta$ e desta forma a condição $\frac{(102)_\beta}{(94)_\beta} < \beta$ é verificada. Assim, obtém-se o quociente $(1)_\beta$ e o resto $(008)_\beta$. Mas, será que a multiplicação pelo valor d resulta sempre num novo digíto não-nulo para a?

Observe-se a divisão de 60 por 52. O valor de d é dado por $d = \lfloor \frac{10}{5+1} \rfloor = 1$. Assim, os novos valores de são $a \leftarrow a \cdot d = (060)_\beta$ e $b \leftarrow b \cdot d = (52)_\beta$. Naturalmente o facto de o valor de d ser 1 equivale a acrescentar um dígito nulo à esquerda no dividendo.

Antes de apresentar o algoritmo que será usado para efectuar a divisão de $a = (a_0 \ldots a_n)$ por $b = (b_0 \ldots b_{n-1})$ refira-se que é possível usar uma outra condição que melhora a escolha de $\hat{q}$ e dessa forma pode garantir-se que o valor candidato a $Quo(a,b)$ obtido é $Quo(a,b)$ ou $Quo(a,b) - 1$. A introdução da condição de teste $b_1 \cdot \hat{q} > (a_0 \cdot \beta + a_1 - \hat{q} \cdot b_0) \cdot \beta + a_2$, o que é feito nos passos 2 e 3 do algoritmo, permite, de uma forma muito rápida (mais rápida do que os passos 4 e 5) eliminar a maioria dos casos em que o valor candidato $\hat{q}$ excede $Quo(a,b)$ em uma unidade e eliminar todos os casos e que o valor candidato $\hat{q}$ excede $Quo(a,b)$ em duas unidades. O que acabou de ser referido é provado no teorema seguinte:

Teorema 2.1.5.3 *Na notação dos Teoremas anteriores e tendo $b_0 > 0$:*

1. *se $b_1 \cdot \hat{q} > (a_0 \cdot \beta + a_1 - \hat{q} \cdot b_0) \cdot \beta + a_2$ então $Quo(a,b) < \hat{q}$;*

2. *se $b_1 \cdot \hat{q} \leq (a_0 \cdot \beta + a_1 - \hat{q} \cdot b_0) \cdot \beta + a_2$ então $Quo(a,b) = \hat{q}$ ou $Quo(a,b) = \hat{q} - 1$.*

Prova.

1. Denote-se por $\hat{r}$ a quantidade $(a_0 \cdot \beta + a_1 - \hat{q} \cdot b_0)$.

$$a - \hat{q}b \leq a - \hat{q}b_0\beta^{n-1} - \hat{q}b_1\beta^{n-2} = a_2\beta^{n-1} + \cdots + a_n + \hat{r}\beta^{n-1} - \hat{q}b_1\beta^{n-2} <$$

$$< \beta^{n-1}(a_2 + 1 + \hat{r}\beta - \hat{q}b_1) \leq 0$$

   Portanto $a - \hat{q}b < 0$ logo $Quo(a, b) < \hat{q}$.

2. Suponha-se, por absurdo, que $Quo(a, b) \leq \hat{q} - 2$.

$$a < (\hat{q} - 1)b < \hat{q}(b_0\beta^{n-1} + (b_1 + 1)\beta^{n-2} - b < \hat{q}b_0\beta^{n-1} + \hat{q}b_1\beta^{n-2} + \beta^{n-1} - b \leq$$

$$\leq \hat{q}b_0\beta^{n-1} + (\beta\hat{r} + a_2)\beta^{n-2} + \beta^{n-1} - b = a_0\beta^n + a_1\beta^{n-1} + a_2\beta^{n-2} + \beta^{n-1} - b \leq$$

$$\leq a_0\beta^n + a_1\beta^{n-1} + a_2\beta^{n-2} \leq a$$

   Portanto $a < a$. Conclui-se assim, por redução ao absurdo, que $Quo(a, b) = \hat{q} - 1$.

□

Depois de tudo o que foi visto nesta secção pode apresentar-se o algoritmo de divisão.

Em primeiro lugar, dados $a = (a_0...a_{n-1})_\beta$ e $b = (b_0...b_{n-1})_\beta$, com $a \geq b$ e $n > 1$, efectue-se a multiplicação de a e b por $d = \left\lfloor \frac{\beta}{b_0+1} \right\rfloor$:

Algoritmo 2.1.5 (Multiplicação de $a = (a_0...a_{n-1})_\beta$ e $b = (b_0...b_{n-1})_\beta$ por $d = \left\lfloor \frac{\beta}{b_0+1} \right\rfloor$)

Entrada: $a = (a_0...a_{n-1})_\beta$ *e* $b = (b_0...b_{n-1})_\beta$, *com* $a \geq b$ *e* $n > 1$.

Saída: $a = (a_0...a_n)_\beta$ *e* $b = (b_0...b_{n-1})_\beta$ *tais que* $0 < a < b\beta$.

1. $d \leftarrow \left\lfloor \frac{\beta}{b_0+1} \right\rfloor$;
2. $a \leftarrow d \cdot a$, *ou seja,* $a = (a_0...a_n)_\beta \leftarrow (a_0...a_{n-1})_\beta \cdot d$;
3. $b \leftarrow d \cdot b$, *ou seja,* $b = (b_0...b_{n-1})_\beta \leftarrow (b_0...b_{n-1})_\beta \cdot d$.

Após este procedimento os valores de a e b obtidos verificam as condições $0 < a < b\beta$ e $b_0 \geq \left\lfloor \frac{\beta}{2} \right\rfloor$ e estão em condições de ser input do algoritmo de divisão que se apresenta a seguir.

Algoritmo 2.1.6 (Algoritmo da Divisão Euclidiana de $a = (a_0...a_n)_\beta$ por $b = (b_0...b_{n-1})_\beta$)

Entrada: $a = (a_0...a_n)_\beta$ *e* $b = (b_0...b_{n-1})_\beta$, *tais que* $0 < a < b\beta$, $n > 1$ *e* $b_0 \neq 0$.

Saída: $Quo(a, b) = \left\lfloor \frac{a}{b} \right\rfloor$ *e* $Res(a, b) = a \ mod \ b$.

1. *se* $a_0 = b_0$ *então* $\hat{q} \leftarrow \beta - 1$, *se não então* $\hat{q} \leftarrow \left\lfloor \frac{a_0 \cdot \beta + a_1}{b_0} \right\rfloor$;

2. *se* $b_1 \cdot \hat{q} > (a_0 \cdot \beta + a_1 - \hat{q} \cdot b_0) \cdot \beta + a_2$ *então* $\hat{q} \leftarrow \hat{q} - 1$;

3. *repetir o passo 2;*

4. $(a_0...a_n)_\beta \leftarrow (a_0...a_n)_\beta - \hat{q} \cdot (b_0...b_{n-1})_\beta$;

5. *se* $(a_0...a_n)_\beta < 0$ *então:*

   (a) $\hat{q} \leftarrow \hat{q} - 1$;

   (b) $(a_0...a_n)_\beta \leftarrow (a_0...a_n)_\beta + (b_0...b_{n-1})_\beta$

6. *retornar* $Quo(a, b) = \hat{q}$ *e* $Res(a, b) = \frac{(a_0...a_n)_\beta}{d}$.

2.1.5.3 Divisão Euclidiana de inteiros $a = (a_0...a_{m+n-1})_\beta$ e $b = (b_0..b_n)_\beta$, tais que $b_0 \neq 0$ e $n > 1$.

Analise-se o exemplo da divisão de 514 por 47 referido no início desta secção. Já foi obtida resposta à questão: *"qual seria o dividendo no primeiro passo?"*. Se for efectuada a multiplicação de a e b pelo valor $d = \left\lfloor \frac{\beta}{b_0+1} \right\rfloor = 2$ obter-se-ão os novos valores $a \leftarrow a \cdot d = (1028)_\beta$ e $b \leftarrow b \cdot d = (94)_\beta$ e o primeiro dividendo a considerar será $(102)_\beta$. Desta forma a condição $\frac{(102)_\beta}{(94)_\beta} < \beta$ verifica-se. Da divisão de $(102)_\beta$ por $(94)_\beta$ obtém-se o quociente $(1)_\beta$ e o resto $(008)_\beta$. Este resto dará origem ao dividendo do passo seguinte juntando-se-lhe na posição menos significativa o dígito seguinte do dividendo: $(088)_\beta$. Desta forma a condição $\frac{(088)_\beta}{(94)_\beta} < \beta$ é verificada. No passo seguinte vem $(0)_\beta$ como quociente e $(088)_\beta$ como resto e este valor originará o novo dividendo $(884)_\beta$, que, mais uma vez, verifica a condição, ou seja, $\frac{(884)_\beta}{(94)_\beta} < \beta$, e, por fim, vem o quociente $(9)_\beta$ e o resto $(038)_\beta$. Em conclusão, obteve-se o quociente da divisão $(109)_\beta$ e o resto $(38)_\beta$. Foi possível observar neste exemplo que os valores do dividendo e do divisor obtidos pela multiplicação pelo número d verificam que $(a_0 \dots a_n)_\beta < b\beta$ e, por outro lado, que nos passos seguintes os divisores obtidos juntando ao resto anterior a posição seguinte de a continuam a verificar esta condição.

No teorema seguinte provar-se-á uma condição análoga ao ponto 4 do teorema 2.1.5.2 para um valor de a nas condições desta secção, ou seja, tal que $a = (a_0 \dots a_{n+m-1})$, com $m > 0$.

Notação 2 *Seja a um inteiro não-negativo cuja representação numa base β é dada por $a = (a_0...a_m)_\beta$. Então $a_{[0,n-1]}$ denota os n dígitos mais significativos, ou seja $a_{[0,n-1]} = (a_0 \dots a_{n-1})_\beta$.*

Teorema 2.1.5.4 *Sejam a e b dois inteiros não-negativos tais que $a = (a_0...a_{n+m-1})_\beta$ e $b = (b_0...b_{n-1})_\beta$ com $a > b$. Seja $d = \left\lfloor \frac{\beta}{b_0+1} \right\rfloor$, $(d \geq 1)$. Sendo $\widetilde{a} = a \cdot d = \widetilde{a}_0\beta^{n-1} + ... + \widetilde{a}_{n+m}$, então $\widetilde{a}_0\beta^{n-1} + ... + \widetilde{a}_n < \widetilde{b}\beta$, ou seja, $a_{[0,n]} < b\beta$.*

Prova. Como $\widetilde{b} = b \cdot d = \widetilde{b}_0\beta^{n-1} + ... + \widetilde{b}_{n-1}$, pelo ponto 2 do Teorema 2.1.5.2, vem que

$$\widetilde{b} \cdot \beta = (\widetilde{b}_0\beta^n + ... + \widetilde{b}_{n-1}\beta) = (\widetilde{b}_0 \dots \widetilde{b}_{n-1}0)_\beta.$$

De onde resulta que:

$$\widetilde{a}_0\beta^{n-1} + ... + \widetilde{a}_n < \widetilde{b}\beta \Leftrightarrow \widetilde{a_0} < \widetilde{b_0},$$

pois se o número de dígitos é igual então o dígito mais significativo do menor tem de ser menor do que o dígito mais significativo do maior. E aquela condição é equivalente, pelo ponto 3 do Teorema 2.1.5.2, à condição $\widetilde{a_0} \leq \left\lfloor \frac{\beta}{2} \right\rfloor - 1$.

Note-se que o maior valor possível de $\widetilde{a}$ é obtido quando todos os seus dígitos forem iguais a $\beta - 1$ e quando $d = \left\lfloor \frac{\beta}{2} \right\rfloor$, pois $d \leq \left\lfloor \frac{\beta}{2} \right\rfloor$.

Ou seja, $\widetilde{a} \cdot d \leq [(\beta-1)\beta^{n+m-1}+...+(\beta-1)] \cdot \left\lfloor \frac{\beta}{2} \right\rfloor \leq ((\left\lfloor \frac{\beta}{2} \right\rfloor - 1)_0(\beta-1)_1 \dots (\beta-1)_{n+m-1}(\frac{\beta}{2})_{n+m})_\beta$.

Neste caso, resulta que o maior valor possível para $\widetilde{a_0}$ (o dígito mais significativo de $\widetilde{a}$ que vai resultar dos transportes dos dígitos anteriores) é $\left\lfloor \frac{\beta}{2} \right\rfloor - 1$.

Portanto $\widetilde{a_0} \leq \left\lfloor \frac{\beta}{2} \right\rfloor - 1$ o que prova que $\widetilde{a}_0\beta^{n-1} + ... + \widetilde{a}_n < \widetilde{b}\beta$. □

Resta ainda referir que se se verifica a condição $a_{[0,n]} < b\beta$ no primeiro passo, então no passo seguinte esta condição também se verifica pois o novo divisor é obtido juntando a posição seguinte do dividendo à quantidade $r\beta$, sendo r o resto do passo anterior, como se prova a seguir:

Facto 3 *Sejam $a = (a_0 \dots a_{n+m})_\beta$ e $b = (b_0...b_{n-1})_\beta$ dois inteiros não-negativos que verificam $0 < a_{[0,n]} < b\beta$ e seja r o resto da divisão de $(a_0 \dots a_n)_\beta$ por b. Então $r\beta + a_{n+1} < b\beta$.*

Prova. Por definição de r, tem-se $r < b$. E $r < b \Rightarrow r \leq b - 1 \Rightarrow r\beta + a_{n+1} \leq b\beta + a_{n+1} - \beta$. Uma vez que $a_{n+1} - \beta < 0$ vem que $r\beta + a_{n+1} < b\beta$. □

Algoritmo 2.1.7 (Divisão Euclidiana de inteiros $a = (a_0 \dots a_{n+m-1})_\beta$ e $b = (b_0...b_{n-1})_\beta$)

Entrada: a, b inteiros não-negativos, tais que $a = (a_0...a_{n+m-1})_\beta$ e $b = (b_0...b_{n-1})_\beta$, com $n > 1$ e $b_0 \neq 0$.
Saída: $Quo(a, b) = \left\lfloor \frac{a}{b} \right\rfloor = (q_0...q_m)_\beta$ e $Res(a, b) = a \ mod \ b = (r_0...r_{n-1})_\beta$.

1. *(a)* $d \leftarrow \left\lfloor \frac{\beta}{b_0+1} \right\rfloor$;

   (b) $a \leftarrow d \cdot a$, *ou seja,* $a = (a_0...a_{n+m})_\beta \leftarrow (a_0...a_{n+m-1})_\beta \cdot d$;

   (c) $b \leftarrow d \cdot b$, *ou seja,* $b = (b_0...b_{n-1})_\beta \leftarrow (b_0...b_{n-1})_\beta \cdot d$.

2. $j \leftarrow 0$;

3. *se* $a_j = b_0$ *então* $\hat{q} \leftarrow \beta - 1$, *se não então* $\hat{q} \leftarrow \left\lfloor \frac{a_j \cdot \beta + a_{j+1}}{b} \right\rfloor$;

4. *se* $b_1 \cdot \hat{q} > (a_j \cdot \beta + a_{j+1} - \hat{q} \cdot b_0) \cdot \beta + a_{j+2}$ *então* $\hat{q} \leftarrow \hat{q} - 1$;

5. *repetir o passo 4;*

6. $(a_j...a_{j+n})_\beta \leftarrow (a_j...a_{j+n})_\beta - \hat{q} \cdot (b_0...b_{n-1})_\beta$;

7. *se* $(a_j...a_{j+n})_\beta < 0$ *então:*

   (a) $\hat{q} \leftarrow \hat{q} - 1$;

   (b) $(a_j...a_{j+n})_\beta \leftarrow (a_j...a_{j+n})_\beta + (b_0...b_{n-1})_\beta$.

8. $q_j \leftarrow \hat{q}$;

9. $j \leftarrow j + 1$ *e se* $j \leq m$ *ir para o passo 3;*

10. *retornar* $Quo(a, b) = (q_0...q_m)_\beta$ *e* $Res(a, b) = \frac{(a_{m+1}...a_{m+n})_\beta}{d} = (r_0...r_{n-1})_\beta$.

A implementação do algoritmo 2.1.7 consiste na função ***void EuclDivIntP(IntPiece *li, IntPiece *lj, IntPiece **quo, IntPiece **res)*** que é apresentada a seguir:

```
void EuclDivIntP(IntPiece *li,IntPiece *lj,IntPiece **quo,IntPiece **res){
  IntPiece *pt1 , *pt2 , *pt3 = new IntPiece ,
      *ptd = new IntPiece, *pt= new IntPiece ;
  pt1 = DuplicateIntP(li);
  pt2 = DuplicateIntP(lj);
  pt3 = smallint2IntP(0);
  int t = LessEqualBiggerIntP(pt1,pt2);
  if (! t){
    *quo=smallint2IntP(0);
*res=DuplicateIntP(pt1);
  }
  if ( t == 1){
  *quo=smallint2IntP(1);
*res=smallint2IntP(0);
  }
```

```
if ( t == 2){
  int n = LenIntP(pt2);
  if(n<=1) EuclDivIntPbysmallint( pt1, pt2, quo, res);
  else{
    unsigned long int temp, w, d, uj0, uj1, uj2, v0, v1;
    IntPiece *ptP = new IntPiece, *ptlast = new IntPiece;
    pt2 = GoToLastOrd(pt2);
    v0 = pt2->val;
    pt2 = GoToGivenOrd(pt2,0);
    d = (NNBITS+1) / ( (v0) + 1);
    ptd = smallint2IntP(d);
    pt1 = KaratR( pt1 , ptd );
    pt2 = KaratR( pt2 , ptd );
    pt2 = GoToLastOrd(pt2);
    v0 = pt2->val;
    v1 = (pt2->prev)->val;
    pt2 = GoToGivenOrd(pt2,0);
    int N = LenIntP(pt1), m = N - n , j=0;
    int lastord = N - n - 1;
    pt = SplitIntP(pt1, lastord);
    t = LessEqualBiggerIntP(pt,pt2);
    if (t == 0 ){
      CutLastOrdPasteToFirstOrd( &pt1 , &pt );
      lastord = lastord - 1;
      m = m - 1;
    }
    else AddValInLastOrdIntP(pt,0);
    while( j <= m){
      if( j ){
        pt3 = ShiftIntP( 1 , pt3 );
        CutLastOrdPasteToFirstOrd( &pt1 , &pt );
        lastord = lastord - 1;
      }
      int b = n - LenIntP( pt );
      int k = 0;
      while ( k < b+1){
        pt = AddValInLastOrdIntP(pt,0);
        k++;
      }
      ptlast = GoToLastOrd(pt);
      uj0 = ptlast->val;
      uj1 = (ptlast->prev)->val;
```

```
        uj2 = ((ptlast->prev)->prev)->val;
        if ( ( uj0 ) == (v0) ){
          w = NNBITS;
          temp = ( ( uj0 ) * (NNBITS + 1)  + ( uj1 ) ) ;
        }
        else{
          temp = ( ( uj0 ) * (NNBITS + 1)  + ( uj1 ) ) ;
          w = temp / v0;
        }
        if ( ( ( temp - v0*w ) <= NNBITS  ) ){
          if ( ( v1*w ) > ( ( temp - v0*w ) * (NNBITS + 1) + uj2 ) ){
            w = w - 1;
          }
        }
        if ( ( ( temp - v0*w ) <= NNBITS  ) ){
          if ( ( v1*w ) > ( ( temp - v0*w ) * (NNBITS + 1) + uj2 ) ){
            w = w - 1;
          }
        }
        ptP = KaratR( smallint2IntP(w) , pt2  );
        t = LessEqualBiggerIntP( pt , ptP  );
        if ( t  ==  0 ){
          w = w - 1;
          ptP = SubIntP( ptP , pt2 );
        }
        t = LessEqualBiggerIntP( pt , ptP  );
        pt = SubIntP (  pt , ptP );
        pt3 ->val = w;
        j++;
      }
      *quo = DuplicateIntP(pt3);
      *res = smallint2IntP(0);
      IntPiece *ptresto = new IntPiece;
      ptresto = smallint2IntP(0);
      EuclDivIntPbysmallint( pt , ptd , res, &ptresto );
    }
  }
}
```

2.1.6 Outras operações

Foram implementadas outras operações que são utilizadas nas funções que dizem respeito à cosntrução da chave.

A função ***IntPiece *FactorialIntP(IntPiece *pt)*** calcula, recursivamente, o factorial do número representado pelo IntPiece *pt.

```
 IntPiece *FactorialIntP( IntPiece *pt )
{
   IntPiece *l, *pt1 = pt;
   if ((!pt1->ord) && (!pt1->val)){
      l=smallint2IntP(1);
   }
   if ( ( pt1->val > 0 )){
      l = KaratR( pt1, FactorialIntP(SubIntP(pt1, smallint2IntP(1))));
   }
   return l;
}
```

A função ***IntPiece *ModExpIntP(IntPiece *li, IntPiece *lj, IntPiece *lk)*** calcula a potência de ordem lj do número representado por li módulo lk, ou seja, $l_i^{\,l_j} \mod l_k$.

```
IntPiece *ModExpIntP( IntPiece *li, IntPiece *lj , IntPiece *lk ){
  IntPiece *pt1 = li, *pt2 = lj, *pt3 = lk ,  *pt4 = new IntPiece , *pt5, *pt6;
  int cont = 0;
  pt6 = smallint2IntP(1);
  IntPiece *ptemp = smallint2IntP(0);

  while ( LessEqualBiggerIntP( pt2 , smallint2IntP(1) ) > 0 ){
    pt4 = QUOEuclDivIntP( pt2 , smallint2IntP(2) );
    pt5 = SubIntP( pt2 , KaratR( pt4 , smallint2IntP(2)));
    pt2 = DuplicateIntP( pt4 );
    if ( LessEqualBiggerIntP( pt5 , smallint2IntP(1) ) == 1  ){
      pt6 = KaratR( pt6 , pt1 );
      pt6 = RESEuclDivIntP( pt6 , pt3 );
    }
    pt1 = RESEuclDivIntP( KaratR( pt1 , pt1 ) , pt3 );
    cont++;
  }
  return pt6;
}
```

2.2 Troca de chave.

A cifra Medusa é uma cifra de chave simétrica o que implica a utilização da mesma chave por ambas as partes intervenientes na comunicação. Este facto cria a necessidade de haver uma partilha de chave sem comprometer a segurança da cifra.

O protocolo que se apresenta para ser utilizado na partilha da chave é o *Protocolo de Diffie-Hellman* [1], apresentado por Whitfield Diffie e Martin Hellman, em 1976, no seu artigo "New directions in Cryptography", como um procedimento seguro de troca de chave para ser utilizado em canais de comunicação públicos e continua a ser, até hoje, largamente utilizado. O protocolo DH de troca de chave permite a dois utilizadores partilharem uma chave secreta (*shared secret key*) num canal de comunicação público.

Protocolo de Diffie-Hellman para a troca de chave:

A Alice ($\mathcal{A}$) e o Bruno ($\mathcal{B}$) começam por concordar num número primo grande p e num número g, com $2 \leq g \leq p-2$, que seja um gerador de um *subgrupo cíclico de* $\mathbb{Z}_p{}^*$ de ordem elevada. Estes valores p e g são fixados à priori entre as partes envolvidas na comunicação e utilizados várias vezes.

Uma vez que p e g condicionam, de várias formas, a segurança do protocolo a escolha destes números deve ser feita criteriosamente obedecendo a algumas condições (ver [7] para todos os detalhes).

Depois de escolhidos os valores p e g, o protocolo desenrola-se da forma seguinte:

1. $\mathcal{A}$ e $\mathcal{B}$ escolhem aleatoriamente[1] um número x e um número y, respectivamente, no intervalo $\{1, ..., p-2\}$;
2. $\mathcal{A}$ envia g^x a $\mathcal{B}$ e $\mathcal{B}$ envia g^y a $\mathcal{A}$;
3. $\mathcal{A}$ conhece x e g^y logo determina g^{xy} que constitui a chave partilhada, calculando $g^{xy} = (g^y)^x$. $\mathcal{B}$ conhece y e g^x e assim determina g^{xy}, a chave partilhada, calculando $g^{xy} = (g^y)^x$.

Um atacante que tenha acesso às mensagens trocadas, ou seja, do tipo observador (*eavesdropper*), não é capaz de descobrir a chave obtida pela Alice e pelo Bruno porque isso implicaria o cálculo de logaritmos discretos para os quais não é conhecido um método eficaz de cálculo em tempo útil. Se fossem interceptadas as mensagens trocadas, ou seja, os valores g^x e g^y, com o intuito de descobrir a chave, g^{xy}, seria necessário calcular, por exemplo, x conhecendo g^x e assim obter a chave fazendo $g^{xy} = (g^y)^x$.

[1] A escolha dos números x e y também condiciona a segurança do protocolo e deve obedecer a certas condições, como pode ser visto em [7].

De acordo com [7], há várias razões para não utilizar a chave tal como se obtém pelo protocolo DH. Por exemplo, o facto de apesar de alguns bits da chave serem provavelmente seguros mas não se conhecer a segurança da maioria dos bits da chave obtida e também o facto de poder acontecer que a chave obtida *(shared secret key)* não tenha o tamanho necessário, ou seja, seja maior do que o tamanho pretendido para a *session key*, entre outras. Assim, antes de utilizar a chave deve ser efectuada uma derivação da chave (*key-derivation*) através da qual se obterá uma chave segura e adequada à cifra a ser utilizada.

2.3 A construção da Chave

A chave da cifra Medusa é determinada a partir da chave trocada pelo Protocolo de Diffie-Hellman, como já foi referido. Através de uma *derivação da chave DH* conveniente obtém-se um valor, denotado por $\mathcal{K}$, na forma de string binária, no intervalo $[0, ..., N! - 1]$, sendo N o tamanho do alfabeto escolhido. O valor $\mathcal{K}$ corresponde a uma sequência dos números entre 0 e $N! - 1$ que, quando aplicada ao alfabeto Σ, constitui uma chave da cifra, pois existe uma correspondência biunívoca entre os números $\mathcal{K}$ tais que $\mathcal{K} \in [0, ..., N! - 1]$ e as chaves da cifra. O método de construção da chave foi apresentado detalhadamente na secção 1.7.

A função ***vector <int> KeyConstr(IntPiece *li)*** tem como input o valor $\mathcal{K}$ (como IntPiece) e devolve a chave (como um vector). Esta função utiliza as duas funções: *Val2Coef* e *Coef2Key*, que serão apresentadas a seguir.

```
vector <int> KeyConstr(IntPiece *li ){
  IntPiece *pt1 = li, *pt2 = new IntPiece , *pt3 = new IntPiece;
  vector <int> key;
  key = Coef2Key( Val2Coef( li ) );
  return key;
}
```

2.3.1 Escrita de $\mathcal{K}$ como combinação linear de factoriais

O primeiro passo da construção da chave consiste em escrever o valor $\mathcal{K}$ partilhado pelo protocolo de Diffie-Hellman uma combinação linear de factoriais única num certo sentido. Assim, dado o valor $\mathcal{K}$ tal que $\mathcal{K} \in [0, ..., N! - 1]$, pretende-se obter os coeficientes $\alpha_0, \alpha_1, ..., \alpha_{N-1}$ tais que cada α_i é o maior possível e se tem:

$$\mathcal{K} = \alpha_{N-1} \cdot (N-1)! + \alpha_{N-2} \cdot (N-2)! + \cdots + \alpha_1 \cdot 1! + \alpha_0 \cdot 0!,$$

como foi visto detalhadamente na secção 1.7.

Algoritmo 2.3.1 (Escrita de $\mathcal{K}$ como combinação linear de factoriais)
Entrada: $\mathcal{K}$ *valor binário no intervalo* $[0, ..., N! - 1]$.
Saída: $\alpha_0, \alpha_1, ..., \alpha_{N-1}$ *tais que* $\alpha_i \in [0, ..., N-1]$ *e* $\mathcal{K} = \sum_{i=0}^{N-1} \alpha_{N-1-i} \cdot (N-1-i)!$.

1. $f \leftarrow (N-1)!$, $k \leftarrow \mathcal{K}$, $j \leftarrow N-1$;
2. $\alpha_j \leftarrow \left\lfloor \frac{k}{f} \right\rfloor$, $k \leftarrow k \ mod \ f$, $f \leftarrow \left\lfloor \frac{f}{j} \right\rfloor$;
3. $j \leftarrow j-1$ *e ir para o passo 2 se* $j > 0$;
4. $\alpha_j \leftarrow \left\lfloor \frac{k}{f} \right\rfloor$;
5. *retornar* $\alpha_0, \alpha_1, ..., \alpha_{N-1}$.

A função ***vector <int> Val2Coef(IntPiece *li)*** constitui a implementação do algoritmo 2.3.1 que recebe o valor $\mathcal{K}$ e devolve os valores dos coeficientes da combinação linear na forma de um vector.

Na função seguinte, a variável $NSIMB$ corresponde ao número de símbolos do alfabeto e é definida globalmente; os apontadores $pt1$, $pt2$ e $pt3$ vão apontar cada um para um dado IntPiece que representa um determinado número e é composto por "cartões" que armazenam os digítos da representação desse número na base 2^{NBITS} e, em particular, o apontador $pt1$ vai percorrer os vários "cartões" do input, isto é, do IntPiece li, que são os dígitos do valor $\mathcal{K}$ nessa base.

```
vector <int> Val2Coef( IntPiece *li){
  IntPiece *pt1 = li, *pt2 = new IntPiece , *pt3 = new IntPiece;
  int c;
  vector <int> coef;
  pt2 = FactorialIntP( smallint2IntP( NSIMB - 1 ) );
  for(int i=0;i<NSIMB;i++){
    if ( i == NSIMB-1 ) pt3 = QUOEuclDivIntP( pt1 , pt2 );
    else{
      EuclDivIntP( pt1 , pt2 , &pt3 , &pt1 );
      pt2 = QUOEuclDivIntP( pt2 , smallint2IntP( NSIMB - 1 - i) );
    }
    c = pt3->val;
    coef.push_back(c);
  }
  return coef;
}
```

2.3.2 Determinação dos números da chave

O segundo passo da cosntrução da chave consiste em cosntruir a sequência dos N números de 0 a $N-1$ a partir da combinação linear de factoriais obtida no primeiro passo. Assim, dados os coeficientes α_i obtidos da forma anteriormente descrita pretende-se determinar a chave $\mathcal{S}_\mathcal{K} = [k_0, k_1, ..., k_{N-2}, k_{N-1}]$ que corresponde ao valor $\mathcal{K}$.

Algoritmo 2.3.2 (Construção da sequência)
Entrada: $\alpha_0, \alpha_1, ..., \alpha_{N-1}$ *tais que* $\alpha_i \in [0, ..., N-1]$.

Saída: $\mathcal{S}_\mathcal{K} = [k_0, k_1, ..., k_{N-2}, k_{N-1}]$, *com* $k_i \in [0, ..., N-1]$ *e* $k_i \neq k_j, \forall i \neq j$.

1. $c = c_0 c_1 ... c_{N-1}$ *com* $c_i = 0, \forall i$;
2. $j \leftarrow 0$, $k_j \leftarrow \alpha_{N-1-j}$;
3. $j \leftarrow j + 1$;
4. $f \leftarrow 0$, $t \leftarrow \alpha_{N-1-j}$, $n \leftarrow t$;
5. *enquanto*$(f == 0)$
   (a) $m \leftarrow \#\ \{i : i < j \wedge k_i < n\}$;
   (b) *se* $(\ n - t - m == 0\ \&\&\ c_n == 0\ )$ *então* $f \leftarrow 1$;
   (c) *se não* $n \leftarrow n + 1$;
6. $k_j \leftarrow n$ *e* $c_n \leftarrow 1$;
7. *se* $j < N - 1$ *ir para o passo 3;*
8. *retornar* $k_0, k_1, ..., k_{N-2}, k_{N-1}$.

A implementação deste algoritmo constitui a função ***vector <int> Coef2Key(vector <int> coef)*** que recebe os coeficientes da combinação linear na forma de vector e devolve a sequência como um vector.

Novamente, na função seguinte, a variável $NSIMB$ corresponde ao número de símbolos do alfabeto e é definida globalmente; o apontador $pt1$ vai percorrer os vários "cartões" do input, isto é do IntPiece li, que são os dígitos do valor $\mathcal{K}$ escrito na base 2^{NBITS}.

```
vector <int> Coef2Key( vector <int> coef ){
  vector <int> key;
  map<int, int> Contr;
  for (int i=0;i<NSIMB;i++) Contr[i] = 0;
```

```
  key.push_back(coef[0]);
  for( i=1;i<NSIMB;i++){
    int c , temp = coef[i] , n = temp;
    int FLAG = 0;
    while (!FLAG){
      c = Count_if_MI( key , 0 , i-1 , n );
      if( ((n - temp - c) == 0) && (Contr[n] == 0) ) FLAG = 1;
      else n +=1;
    }
    key.push_back(n);
    Contr[n]=1;
  }
  return key;
}
```

2.4 Cifra e Decifração com a Cifra Medusa

2.4.1 Cifra

A função ***string CifraM(string MO , vector <int> key)*** permite efectuar a cifra do texto simples. Esta função tem como input a mensagem original como string e a chave como um vector e devolve uma string com a mensagem cifrada.

```
string CifraM(string MO , vector <int> key){
  int m = 0, M = 0 ;//minimo e Maximo para os comprimentos de bloco
  LenMAXmin( &M , &m );
  int b=LenIniBl(key);//comp. bloco inicial
  vector <int> coord_l(NSIMB), coord_c(NSIMB);
  for(int j=0;j<NSIMB;j++){
    coord_l[key[j]]=j/K;
    coord_c[key[j]]=j%K;
  }
  string MC;
  basic_string <char>::size_type pos;
  int a = 0; //posicao inicial
  int len_MO = MO.length();
  if( b > len_MO ) b = len_MO;
  int res = len_MO; //res = len_MO-1-b+1
  if ( res < (2*M) ) b = res/2;
  while (res > 0){
```

```
    res = res  - b; //res = len_MO-1-b+1
    vector <int> coord(2*b);
    int sum = 0;
    for(j=0; j<b;j++){
      pos = ALFABETO.find(MO[a+j],0);
      coord[j] = coord_l[pos];
      coord[j+b] = coord_c[pos];
      sum+=pos;
    }
    if ( res < (2*M) ) sum = res/2;
    if ( res < M ) sum = res;
    else sum = sum \% ( M-m+1 ) + m;
    for(j=0; j<b;j++){
      MC.append(1, ALFABETO[key[K*coord[2*j]+coord[2*j+1]]]);
    }
    a+=b;
    b = sum;
  }
  return MC;
}
```

A função usada para efectuar a cifra de uma mensagem utiliza como auxiliares duas funções: ***int LenIniBl(vector <int> key)*** que determina o comprimento do primeiro bloco a cifrar à custa da chave e ***void LenMAXmin(int *M , int *m)*** que dá os valores mínimo e máximo para o comprimento de bloco.

2.4.2 Decifração

A função ***string DeCifraM(string MC , vector <int> key)*** que permite efectuar a decifração do criptograma tem como input a mensagem cifrada (como string) e a chave (como um vector) e devolve uma string com a mensagem original.

```
string DeCifraM(string MC , vector <int> key){
  int m = 0 , M =0 ;
  LenMAXmin( &M , &m );
  int b=LenIniBl(key);
  vector <int> coord_l(NSIMB), coord_c(NSIMB);
  for(int j=0;j<NSIMB;j++){
    coord_l[key[j]]=j/K;
    coord_c[key[j]]=j%K;
  }
```

```
  string MO;
  basic_string <char>::size_type pos;
  int a = 0;
  int len_MC = MC.length();
  if( b > len_MC ) b = len_MC;
  int res = len_MC;
  if ( res < (2*M) ) b = res/2;
  while (res > 0){
    res = res- b; //res = len_MC-1-b+1
    vector <int> coord(2*b);
    int sum = 0;
    for(j=0; j<b;j++){
      pos = ALFABETO.find(MC[a+j],0);
      coord[2*j] = coord_l[pos];
      coord[2*j+1] = coord_c[pos];
    }
    for(j=0; j<b;j++){
      char t = ALFABETO[key[K*coord[j]+coord[j+b]]];
      MO.append(1, t);
      pos = ALFABETO.find(t,0);
      sum+=pos;
    }
    if ( res < (2*M) ) sum = res/2;
    if ( res < M ) sum = res;
    else sum = sum % ( M-m+1 ) + m;
    a = a + b;
    b = sum;
  }
  return MO;
}
```

A função que permite decifrar a mensagem utiliza, como auxiliares, as duas funções seguintes: ***int LenIniBl(vector <int> key)*** que determina o comprimento do primeiro bloco a cifrar à custa da chave e ***void LenMAXmin(int *M , int *m)*** que dá os valores mínimo e máximo para o comprimento de bloco.

Capítulo 3

Segurança da Medusa

Os ataques dirigidos à Bífida em [4] baseiam-se na determinação do período da cifra que, como já foi referido, é fixo. Para determinar o período é contado o número de letras iguais que aparecem a uma dada distância. Na maioria dos casos, a função que a cada distância possível faz corresponder o número de letras repetidas que surgem a essa distância no criptograma tem como gráfico uma sinusóide cujo período é o período da Bífida usada para cifrar essa mensagem. Nos casos em que o gráfico não é uma sinusóide o período pode ser obtido usando o desvio-padrão. Depois de conhecido o período são utilizadas três tipo de pistas que permitem ir construindo a chave da Bífida.

Em relação à Medusa foram efectuados alguns testes na tentativa de obter informação a partir do criptograma quer no que diz respeito à distribuição das letras do alfabeto quer à distância a que se encontram letras repetidas. Nos testes efectuados utilizaram-se uma mensagem $\mathcal{M}$, de tamanho 65001, constituída por um texto em língua portuguesa e uma mensagem $\mathcal{M}_{\mathcal{A}}$, do mesmo tamanho de $\mathcal{M}$, constituída por um texto construído aleatoriamente. O alfabeto Σ utilizado é constituído pelo conjunto de vinte e cinco letras do alfabeto excluíndo a letra j, ou seja,

$$\Sigma = \{a, b, c, d, e, f, g, h, i, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}.$$

3.1 Estatísticas dos símbolos do alfabeto em mensagens originais e respectivos criptogramas

Sejam $\mathcal{M}$ e $\mathcal{M}_{\mathcal{A}}$ mensagens escritas com o alfabeto Σ tais que a primeira, $\mathcal{M}$, tem tamanho 65001 e foi escrita em língua portuguesa e a segunda, $\mathcal{M}_{\mathcal{A}}$, foi obtida aleatoriamente e tem o mesmo tamanho de $\mathcal{M}$. Sejam $\mathcal{C}$ e $\mathcal{C}_{\mathcal{A}}$ os criptogramas obtidos a partir das mensagens $\mathcal{M}$ e $\mathcal{M}_{\mathcal{A}}$, respectivamente.

Analise-se o caso da mensagem original $\mathcal{M}$. No primeiro gráfico pode observar-se a distribuição dos símbolos da mensagem original $\mathcal{M}$:

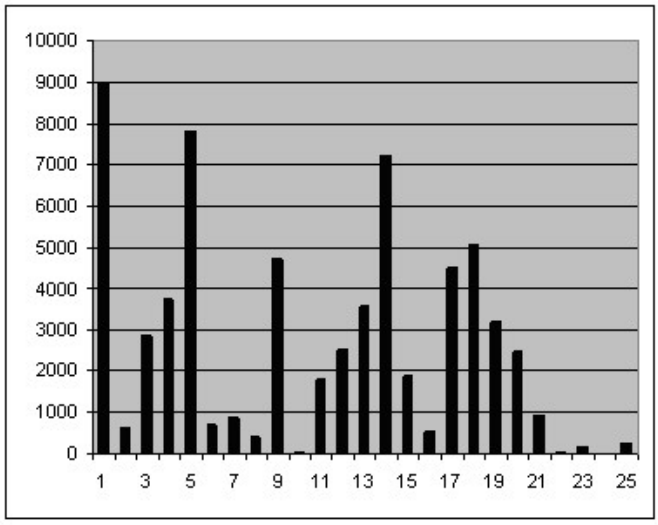


Figura 3.1: Distribuição dos símbolos em $\mathcal{M}$

O gráfico seguinte ilustra a distribuição dos símbolos do criptograma $\mathcal{C}$ obtido a partir da mensagem $\mathcal{M}$:

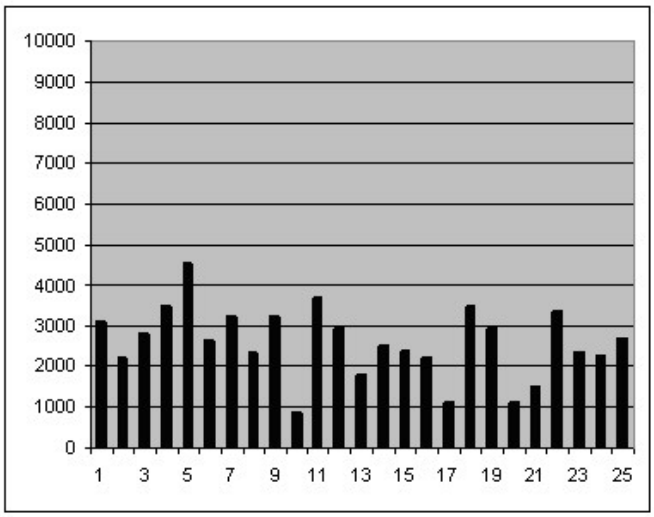


Figura 3.2: Distribuição dos símbolos em $\mathcal{C}$

Pode observar-se que ao efectuar a cifra da mensagem a distribuição dos símbolos do alfabeto no criptograma é de certa forma "suavizada" em relação à mensagem original.

Observe-se agora o caso da mensagem $\mathcal{M}_{\mathcal{A}}$ obtida aleatoriamente. O gráfico seguinte ilustra a distribuição dos símbolos na mensagem original:

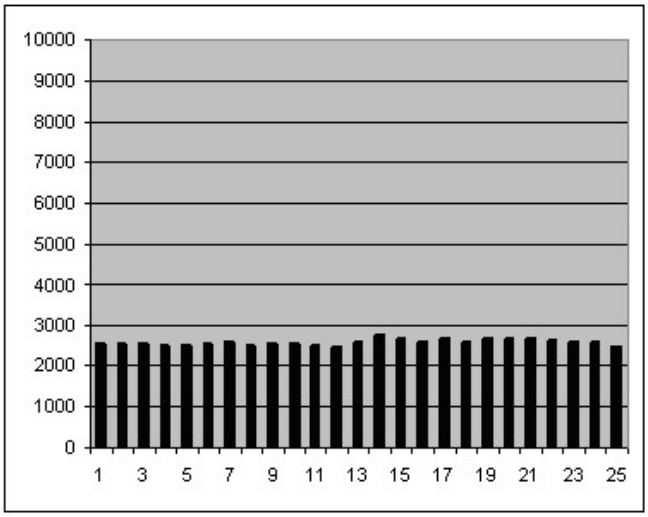


Figura 3.3: Distribuição dos símbolos em $\mathcal{M}_{\mathcal{A}}$

O gráfico a seguir representado ilustra a distribuição dos símbolos no criptograma obtido a partir da mensagem original $\mathcal{M}_{\mathcal{A}}$:

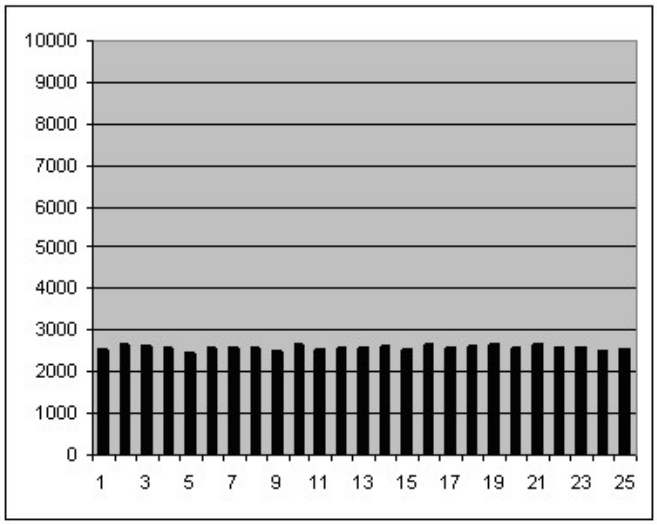


Figura 3.4: Distribuição dos símbolos em $\mathcal{C}_{\mathcal{A}}$

Neste caso a distribuição dos símbolos na mensagem original já não possuia grandes oscila-

ções, o que é natural dado que o texto foi construído aleatoriamente. E ao efectuar a cifra desta mensagem não foi intoduzida nenhuma oscilação significativa.

Pode concluir-se da observação dos dois casos anteriores que a cifra de uma mensagem impossibilita a obtenção de informação estatística sobre a mensagem original. Observe-se o caso dos quatro símbolos mais frequentes em $\mathcal{M}$: o primeiro, o quinto, o nono e o décimo-quarto, que correspondem respectivamnte a a, e, i e o. No criptograma a distribuição destes símbolos reduziu drasticamente não sendo possível identificá-los estudando as frequências no criptograma (excepto o caso do símbolo e que, de facto, tem a maior frequência no criptograma mas é apenas uma informação isolada). No caso dos símbolos cujo número de ocorrências é zero ou muito próximo de zero na mensagem original, acontece que a cifra da mensagem conduz a que, no criptograma, o número de ocorrências aumente significativamente de uma forma que torna impossível a identificação destes símbolos.

3.2 Análise das distâncias a que se encontram símbolos repetidos do alfabeto em criptogramas

Tendo em conta os ataques já referidos dirigidos à Bifida, apresentados em [4], outra das análises efectuadas consistiu em obter a distribuição das distâncias a que aparecem letras repetidas no criptograma. O gráfico da distribuição destas distâncias é, na maioria dos casos, uma sinusóide cujo período corresponde ao período da cifra. Nos casos em que este gráfico é "plano" o período pode ser obtido a partir da representação gráfica do desvio-padrão pois o período da cifra corresponde a metade do valor mais elevado.

Utilizaram-se novamente as mensagens $\mathcal{M}$ e $\mathcal{M}_{\mathcal{A}}$, de tamanho 65001, sendo $\mathcal{M}$ constituída por um texto em língua portuguesa, e $\mathcal{M}_{\mathcal{A}}$ constituída por um texto construído aleatoriamente. Os respectivos criptogramas são denotados por $\mathcal{C}$ e $\mathcal{C}_{\mathcal{A}}$.

Por um lado obteve-se a distribuição das distâncias a que se encontram letras repetidas no criptograma e, por outro, a distribuição dos comprimentos de bloco que se verificaram na cifra da mensagem em estudo.

Em primeiro lugar analise-se o caso da mensagem $\mathcal{M}$. O gráfico ilustra o número de letras repetidas que se encontram à distância d com $d < 100$.

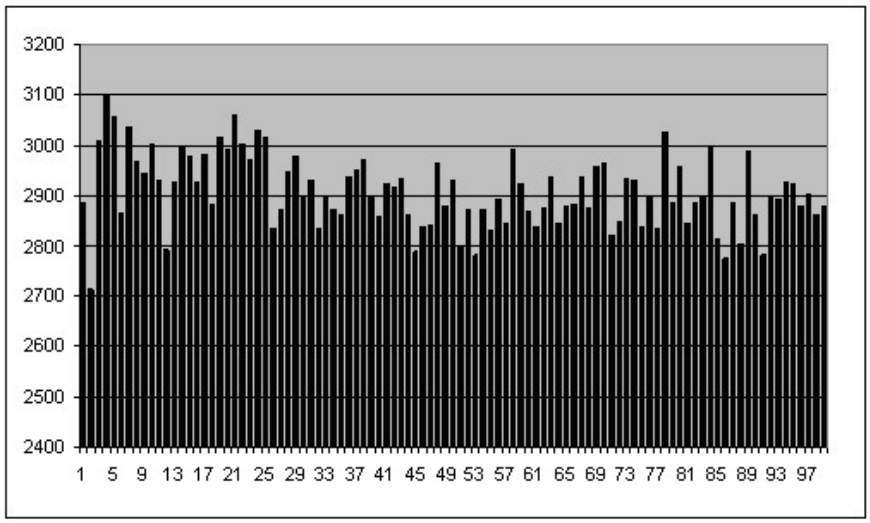


Figura 3.5: Distribuição das distâncias das letras repetidas em $\mathcal{C}$

A seguir apresenta-se o gráfico que ilustra a distribuição dos comprimentos de bloco, sendo que estes valores variam entre o valor mínimo $m = 5$ e o valor máximo $M = 50$.

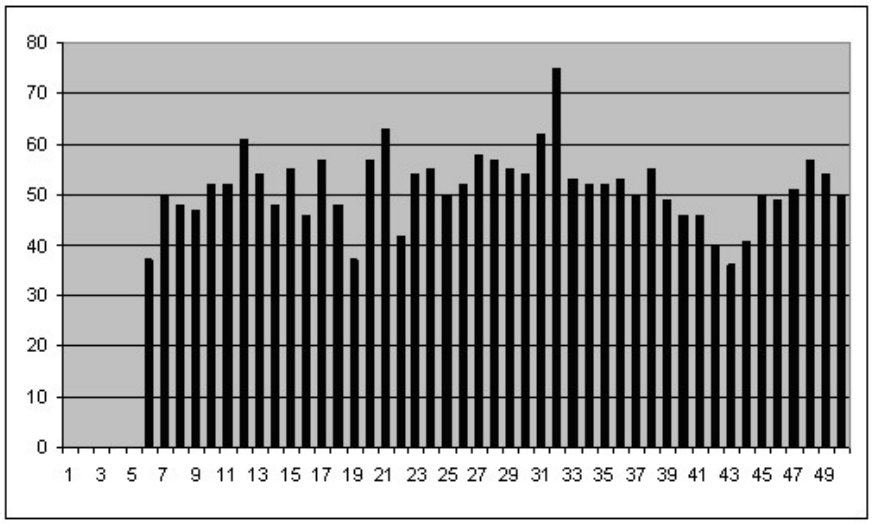


Figura 3.6: Distribuição dos comprimentos de bloco de $\mathcal{C}$

Analise-se agora o caso da mensagem $\mathcal{M}_{\mathcal{A}}$. O gráfico seguinte ilustra o caso do criptograma obtido a partir de uma mensagem em texto obtido aleatoriamente e mostra o número de letras repetidas que se encontram à distância d com $d < 100$.

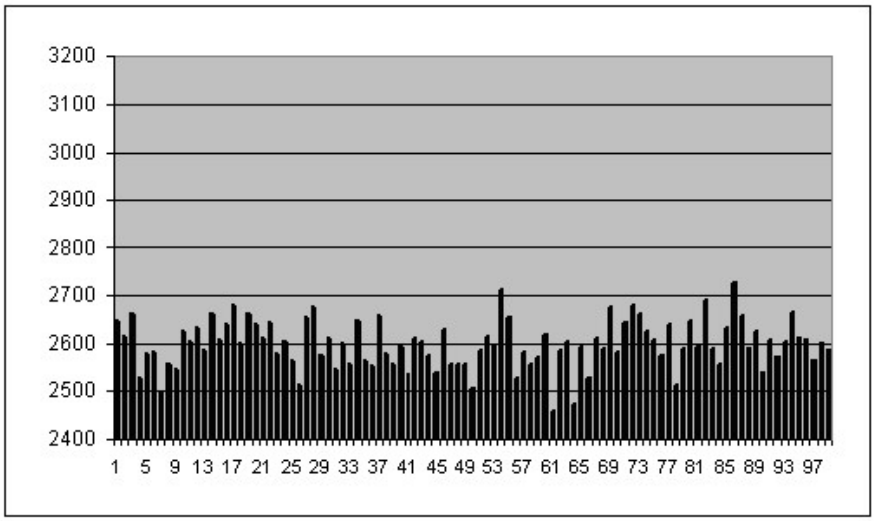


Figura 3.7: Distribuição das distâncias das letras repetidas em $\mathcal{C}_{\mathcal{A}}$

A seguir apresenta-se o gráfico que ilustra a distribuição dos comprimentos de bloco, sendo que estes valores variam entre o valor mínimo $m = 5$ e o valor máximo $M = 50$.

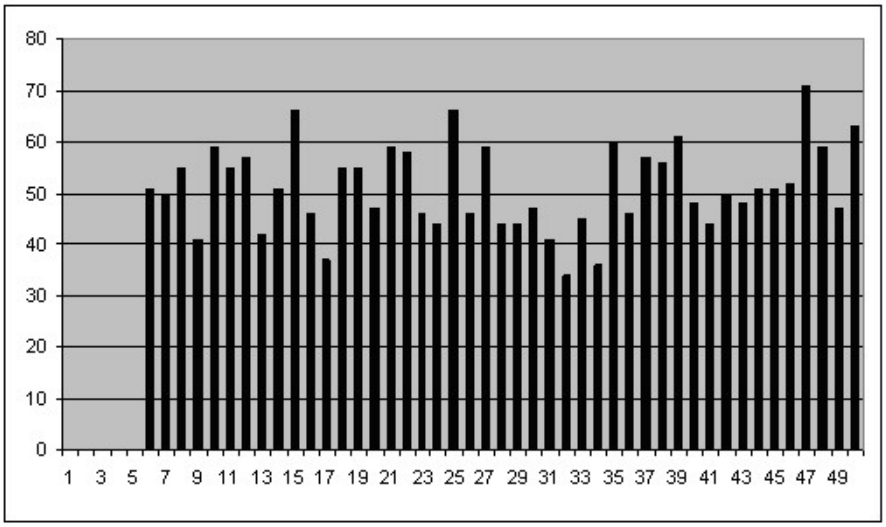


Figura 3.8: Distribuição dos comprimentos de bloco de $\mathcal{C}_{\mathcal{A}}$

É possível observar que a distribuição das distâncias a que se encontram as letras repetidas no criptograma não permite retirar informação sobre os períodos utilizados na cifra da mensagem.

Uma vez que não é possível determinar o comprimento de bloco da Medusa, por um método análogo ao utilizado para a Bífida em [4], resulta que os restantes ataques expostos na mesma

fonte não são aplicáveis pois pressupõem o conhecimento do comprimento de bloco.

3.3 Adaptações necessárias a uma implementação em contexto real

Nesta secção apresentar-se-á um esboço do que poderia ser feito para resolver duas questões relacionadas com a utilização da Medusa numa aplicação prática: o problema de o alfabeto ter de ser definido à partida entre as partes intervenientes na comunicação e o facto de, apesar do que foi observado nas secções anteriores deste capítulo, poderem persistir informações decorrentes do conhecimento estatístico da distribuição das letras na escrita numa determinada língua.

Para simplificação da utilização devia ser permitida a utilização de qualquer símbolo, ou seja, de qualquer alfabeto, na escrita das mensagens. Mas, foi visto que o tamanho do alfabeto da Medusa tem de ser um quadrado perfeito. Além disso, o código ASCII é esparso e se se tomásse um determinado intervalo entre 0 e 255 obter-se-ia números que não correspondem a nenhum símbolo válido.

Por outro lado, para evitar ataques baseados na distribuição das ocorrências dos símbolos torna-se necessário eliminar o mais possível a distribuição natural das letras na língua portuguesa. Apesar de, aparentemente, a cifra com a Medusa destruir essa distribuição a informação continua presente e não é possível excluir que ela seja obtida de outra forma diferente da apresentada nas secções anteriores. A aplicação de uma transformação ao texto antes da cifra que uniformizasse a distribuição dos símbolos seria aconselhável.

Assim, explorar-se-á a possibilidade de, por um lado, aplicar o algoritmo LZW [10] para obter uma distribuição mais uniforme dos símbolos da mensagem e, por outro lado, transformar a mensagem resultante numa sequência de bits e cifrá-la de uma forma em que passa a ser irrelevante o conjunto de símbolos que constitui o alfabeto da mensagem.

3.3.1 Algoritmo LZW

O algoritmo de compressão LZW(Lemple-Ziv-Welch) [10, 11] é um algoritmo de compressão de dados criado por Abraham Lempel, Jacob Ziv e Terry Welch e que constitui uma implementação melhorada do algoritmo LZ78, publicado em 1978 por A. Lempel e J. Ziv.

O algoritmo LZW constrói uma tabela de correspondência entre sequências de bits e sequências de caracteres a partir do texto que está a ser comprimido e, inversamente, no processo de descompressão recupera a tabela a partir do texto a ser descomprimido. A aplicação deste algoritmo, antes da cifra e depois da decifração, permitiria uniformizar a distribuição dos símbolos do alfabeto e assim melhorar a segurança da cifra.

Seja D o dicionário cujas entradas são constituídas por sequências de caracteres (*strings*), denotadas por s. As palavras-código, denotadas por p, que serão obtidas pela aplicação do algoritmo são constituídas por sequências de bits e são referências às *strings* do dicionário. O dicionário é inicializado com uma entrada para cada *byte* possível, ou seja, com os símbolos cuja representação em ASCII corresponde aos valores entre 0 e 250. As restantes *strings* são adicionadas à medida que são construídas a partir da sequência de entrada (*input stream*). A palavra-código para uma nova *string* a ser adicionada ao dicionário é simplesmente o valor seguinte disponível.

A sequência-código (*encoded string*) é construída a partir da sequência de entrada, que é lida 1 *byte* de cada vez, e é utilizada para adicionar novas *strings* ao dicionário. Ao concatenar à *encoded string* a *string* correspondente ao novo *byte* lido obtém-se uma sequência que pode estar ou não no dicionário. Se estiver a *encoded string* actual passa a ser essa nova *string*. Se não estiver é criada uma nova entrada no dicionário para a nova *string*, a palavra-código correspondente à *encoded string* é adicionada à sequência de saída (*output stream*) e a *encoded string* actual passa a ser o novo *byte* lido.

Algoritmo 3.3.1 (Compressão com o Algoritmo LZW)
Entrada: Sequência de caracteres $\mathcal{S}$.
Saída: Sequência de palavras-código (bytes) $\mathcal{B}$.

1. *Inicializar o dicionário* D *com todas as entradas entre* 0 *e* 256. *Inicializar a encoded string com a string correspondente ao primeiro byte lido a partir da input stream* $\mathcal{S}$;
2. *Ler o byte seguinte a partir da input stream* $\mathcal{S}$;
3. *Se o byte lido for um indicador de fim de mensagem (EOF) ir para o passo 7;*
4. *Seja* s *a sequência obtida ao concatenar a encoded string com a string que corresponde ao novo byte;*
5. *Se* $s \in D$:
   - *Substituir a encoded string por* s;
   - *Ir para o passo 2.*
6. *Se* $s \notin D$:
   - *Adicionar* s *ao dicionário* D;
   - *Escrever a palavra-código da encoded string actual na output stream* $\mathcal{B}$;
   - *Substituir a encoded string pela string que corresponde ao novo byte;*
   - *Ir para o passo 2.*
7. *Escrever a palavra-código correspondente à encoded string na output stream* $\mathcal{B}$.

O algoritmo que permite descomprimir a mensagem consiste basicamente em efectuar o processo inverso ao método de compressão. A única informação necessária é a sequência resultante da compressão uma vez que o dicionário será obtido ao longo do processo.

Algoritmo 3.3.2 (Descompressão com o Algoritmo LZW)
Entrada: Sequência de palavras-código (bytes) $\mathcal{B}$.
Saída: Sequência de caracteres $\mathcal{S}$.

1. *Inicializar o dicionário* D *com todas as entradas entre* 0 *e* 256.
2. *Ler a primeira palavra-código a partir da input stream* $\mathcal{B}$ *e escrever a string correspondente na output stream* $\mathcal{S}$;
3. *Ler a palavra-código seguinte a partir da input stream* $\mathcal{B}$;
4. *Se o byte lido for um indicador de fim de mensagem (EOF) terminar;*
5. *Seja* s *a string que corresponde à palavra-código lida;*
6. - *Escrever* s *na output stream* $\mathcal{S}$;
   - *Adicionar ao dicionário a string obtida ao concatenar o primeiro caracter da string* s *com a string produzida pela palavra-código anterior;*
7. *Ir para o passo 3.*

Com a utilização deste algoritmo obtém-se as mensagens como sequência de bytes em que a distribuição dos valores possíveis é uniforme reforçando assim a segurança da Medusa ao protegê-la contra ataques baseados na frequência das letras.

3.3.2 Cifra e decifração da mensagem

Após a aplicação do algoritmo LZW tal como foi apresentado na secção anterior, a mensagem passa a ser constituída por uma sequência de palavras-código do dicionário utilizado, ou seja, uma sequência de *bytes*. Para efectuar a cifra desta mensagem ela será dividida em blocos de bits de um dado tamanho e o valor representado por cada um desses blocos de bits vai corresponder a um elemento da chave da cifra. De forma inversa, aquando da decifração, a sequência de bits recebida é fraccionada em blocos de bits que correspondem da mesma forma a elementos da chave da Medusa e a mensagem é assim decifrada. Após a decifração da mensagem, a sequência de bits obtida é encarada como uma sequência de bytes e é aplicada a descompressão com o algoritmo LZW recuperando desta forma a mensagem original.

Uma vez que a chave da Medusa é uma tabela quadrada onde são dispostos os símbolos do alfabeto é necessário que o tamanho do alfabeto seja um quadrado perfeito. Assim, é

necessário que as partes intervenientes na comunicação escolham *à priori* o valor de N. Feito isto o alfabeto utilizado pode ser qualquer um pois, como já foi referido, depois de aplicado o algoritmo LZW a mensagem a cifrar passa a ser uma sequência de bits.

Por exemplo, se o valor de N for fixado como $N = 256$ a construção da chave (que será feita da forma apresentada em 1.3) conduzirá a uma sequência dos números no intervalo $[0, \ldots, 255]$. Se a chave for vista como uma tabela então será um quadrado de lado 16 e conterá não símbolos de um alfabeto mas todos os valores no intervalo $[0, \ldots, 255]$.

Sendo Σ o alfabeto tal que $|\Sigma| = 256$ e $\sigma_i \in [0, \ldots, 255]$:

	0	1	$\cdots$	14	15
0	σ_0	σ_1	$\cdots$	σ_{14}	σ_{15}
1	σ_{16}	σ_{17}	$\cdots$	σ_{30}	σ_{31}
$\vdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$
15	σ_{240}	σ_{241}	$\cdots$	σ_{254}	σ_{255}

Para este valor de N a mensagem a cifrar será dividida em blocos de 8 bits. Por exemplo dada a mensagem $\mathcal{M} =$ "10101011101100000101010101010101110101010101010100" então:

$$\mathcal{M} = 10101011|10110000|01010101|01010101|11010101|01010100.$$

Depois de dividida a mensagem em blocos são atribuídas aos valores as coordenadas que lhe correspondem na chave e a cifra é efectuada de forma análoga à apresentada em 1.5. No exemplo dado:

$$\mathcal{M} = \ '171' \ '168' \ '85' \ '85' \ '213' \ '84'$$

$$\downarrow$$

'171'	'168'	'85'	'85'	'213'	'84'
x_0	x_1	x_2	x_3	x_4	x_5
y_0	y_1	y_2	y_3	y_4	y_5

Se os valores da tabela estiverem escritos em base 2 o criptograma obtido é uma sequência de bits, se os elementos da chave estiverem em base decimal o criptograma será obtido concatenando os bits resultantes de converter cada valor para a base 2. Note-se que o criptograma a ser enviado que resulta da aplicação do algoritmo LZW continuará a ser uma sequência de bits.

Para efectuar a decifração da mensagem que foi recebida divide-se a sequência de bits em blocos de 8 bits, efectua-se a decifração analogamente ao que foi apresentado em 1.6 e, por fim, aplica-se o algoritmo LZW.

Por exemplo, se o criptograma $\mathcal{C}$ recebido na troca de mensagens for a seguinte sequência de bits $\mathcal{C} =$ "0100100111101110000001010000110000101101010101 00" então divide-se $\mathcal{C}$ da seguinte forma:

$$\mathcal{C} = 01001001|11101110|00000101|00001100|00101101|01010100.$$

E depois faz-se corresponder aos valores as respectivas coordenadas da chave:

$$\mathcal{C} = \ '73' \ '238' \ '5' \ '12' \ '45' \ '84' \ \ldots$$

$$\downarrow$$

$'73'$	$'238'$	$'5'$	$'12'$	$'45'$	$'84'$	$\ldots$
x_0	x_1	x_2	x_3	x_4	x_5	$\ldots$
y_0	y_1	y_2	y_3	y_4	y_5	$\ldots$

Prosseguindo a decifração da forma descrita anteriormente. Por fim, à sequência obtida pela decifração é aplicado o algoritmo de descompressão LZW recuperando-se assim a mensagem original.

Conclusão e trabalho futuro

A implementação apresentada nos capítulos anteriores compreende o processo de construção de uma chave da Medusa, a partir de uma chave partilhada pelo Protocolo de Diffie-Hellman (e modificada de modo apropriado) e a cifra e decifração de mensagens. O processo de construção da chave tem como entrada um valor binário obtido através do Protocolo de Diffie-Hellman. A partir desse valor é construída a chave da cifra na forma de uma sequência. Uma vez construída a sequência, que constitui a chave da cifra, é possível cifrar mensagens dividindo-as em blocos de comprimento variável e intrinsecamente dependente do conteúdo da mensagem, à excepção do primeiro que é calculado a partir da chave e dos dois últimos, que dependem do tamanho restante da mensagem a cifrar. Por outro lado, ao receber uma mensagem cifrada e, tendo conhecimento do comprimento do primeiro bloco através da chave, decifra-se o primeiro bloco do criptograma cujo texto simples correspondente conduz ao conhecimento do comprimento de bloco seguinte.

A implementação desenvolvida pressupõe que o alfabeto utilizado na troca de mensagens seja fixo, e determinado à partida, de onde resulta que seja definida uma variável global que constitui o alfabeto que deve ser utilizado. Esta opção deve-se, por um lado, à necessidade de que o tamanho do alfabeto utilizado na cifra seja um quadrado perfeito e, por outro, à impossibilidade de se corresponder um símbolo diferente a todos os números entre 0 e 256, ou outro intervalo com limite superior menor, sem que se verificassem descontinuidades no intervalo, ou seja, números que não correpondem a um símbolo "válido".

Numa implementação que se pretendesse ser mais "realista" devia ser permitida a utilização de qualquer símbolo, ou seja, de qualquer alfabeto. E, por outro lado, para evitar ataques baseados na distribuição das ocorrências dos símbolos poderia ser utilizado o algoritmo de compressão LZW(Lemple-Ziv-Welch) [10] que é um algoritmo de compressão de dados criado por Abraham Lempel, Jacob Ziv e Terry Welch e que permitiria uniformizar a distribuição dos símbolos do alfabeto e assim melhorar a segurança da cifra. A implementação da Medusa de forma a que seja possível uma aplicação prática deste método seguro de troca de mensagens sms implica possivelmente outras adaptações que ultrapassam o âmbito desta dissertação e constituí assim um possível trabalho futuro.

Apêndice A

- Programa em C++

```
#pragma warning(disable:4786)
#include <iostream>
#include <string>
#include <stdlib.h>
#include <stdio.h>
#include <cmath>
#include <time.h>
#include <fstream>
#include <vector>
#include <map>
#include <iostream>
#define NBITS 16
#define NNBITS 65535
#define NSIMB 81
#define max(X,Y) (X>=Y?X:Y)

using namespace std;

string ALFABETO="!&%()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXY
                                              Z_abcdefghijklmnopqrstuvwxyz";
int K = sqrt(NSIMB);

typedef struct intpiece{
  unsigned long val;
  unsigned short ord;
  struct intpiece *prev;
  struct intpiece *next;
} IntPiece;
```

```
//ARITMETICA DOS INTPIECES
IntPiece *SumIntP(IntPiece *li,IntPiece *lj);
IntPiece *SubIntP(IntPiece *li,IntPiece *lj);
IntPiece *KaratR(IntPiece *li, IntPiece *lj);
IntPiece *FactorialIntP( IntPiece *pt );
//calcula: li^lj mod lk
IntPiece *ModExpIntP( IntPiece *li, IntPiece *lj , IntPiece *lk );
//DIVIDE DOIS INTPIECES DE TAMANHOS QQ
void EuclDivIntP( IntPiece *li, IntPiece *lj, IntPiece **quo, IntPiece **res );
//DIVIDE UM INTPIECE DE TAMANHO QQ POR UM INTPIECE DE TAMANNHO 1
void EuclDivIntPbysmallint( IntPiece *li, IntPiece *lj,
IntPiece **quo, IntPiece **res);
IntPiece *QUOEuclDivIntP( IntPiece *li, IntPiece *lj);//SO DEVOLVE O QUOCIENTE
IntPiece *RESEuclDivIntP( IntPiece *li, IntPiece *lj);//SO DEVOLVE O RESTO

//AUXILIARES (ARITMETICA IP)
unsigned long int ConvertBinaryStringToInteger( string S );
string IntP2DecimalStr(IntPiece *li);
IntPiece *DecimalStr2IntP(string S);
string Num(int nbits);//cria strings aleatorias comecadas por 1
IntPiece *Str2IntP(string S);
IntPiece *smallint2IntP(int i);
IntPiece *ZeroIntP(int n);

void PrintIntP(IntPiece *li);
void PrintInFileIntP(IntPiece *li);
int LenIntP(IntPiece *li);
int LessEqualBiggerIntP( IntPiece *li, IntPiece *lj );

void DeleteIntP(IntPiece *li);
IntPiece *SplitIntP(IntPiece *li, int lastord);
IntPiece *MulIntP(IntPiece *li,IntPiece *lj);
IntPiece *DuplicateIntP(IntPiece *i);
IntPiece *ShiftIntP(int i, IntPiece *li);
IntPiece *GoToLastOrd(IntPiece *li);
IntPiece *GoToGivenOrd(IntPiece *li, int i);
void CutLastOrdPasteToFirstOrd( IntPiece **li, IntPiece **lj );
IntPiece *AddValInLastOrdIntP( IntPiece *li , int i);

//#####################################################################
IntPiece *SumIntP(IntPiece *li, IntPiece *lj){
```

```
  int o=0;
  unsigned long carry=0, sum=0;
  IntPiece *pt1=new IntPiece, *pt = pt1;

  while(li || lj || carry){
    if(o){
      pt->next = new IntPiece;
      (pt->next)->prev = pt;
      pt = pt->next;
    }
else pt->prev = NULL;
    sum = (li?li->val:0) + (lj?lj->val:0) + carry;
    pt->val = sum & NNBITS;
    carry = sum>>NBITS;
    pt->ord = o++;
    li = li?li->next:NULL;
    lj = lj?lj->next:NULL;
  }
  pt->next = NULL;
  return pt1;
}

IntPiece *SubIntP(IntPiece *li,IntPiece *lj){
  int o=0;
  unsigned long carry=0, sub=0, a, b;
  IntPiece *pt1=new IntPiece, *pt = pt1, *pt2;

  while(li || lj || carry){
    if(o){
      pt->next = new IntPiece;
      (pt->next)->prev = pt;
      pt = pt->next;
    } else
      pt->prev = NULL;
    a = (li?li->val:0);
    b = (lj?lj->val:0);
    if (a<(b+carry)){
      sub = NNBITS+1+a-b-carry;
      carry = 1;
    } else {
      sub = a-b-carry;
      carry = 0;
```

```
    }
    pt->val = sub;
    pt->ord = o++;
    li = li?li->next:NULL;
    lj = lj?lj->next:NULL;
  }
  pt->next = NULL;

  while((!pt->val) && pt->prev){
    (pt->prev)->next = NULL;
    pt2 = pt;
    pt = pt->prev;
    delete pt2;
  }
  return pt1;
}


IntPiece *KaratR(IntPiece *li, IntPiece *lj){
  int ti, tj, n;
  static int xxx=0;
  ti = LenIntP(li);
  tj = LenIntP(lj);
  n = max(ti,tj);
  if(n > 1){
    IntPiece *li1, *lj1, *li0, *lj0, *lie, *lje, *kl0, *kl1, *r,
      *t1, *t2, *t3, *t4, *t5, *t6;
    n = (n+1)/2-1;
    li0 = DuplicateIntP(li);
    lj0 = DuplicateIntP(lj);
    li1 = SplitIntP(li0,n);
    lj1 = SplitIntP(lj0,n);
    lie = SumIntP(li0,li1);
    lje = SumIntP(lj0,lj1);
    kl0 = KaratR(li0,lj0);
    DeleteIntP(li0);
    DeleteIntP(lj0);
    kl1 = KaratR(li1,lj1);
    DeleteIntP(li1);
    DeleteIntP(lj1);
    t4  = ShiftIntP(2*n+2,kl1);
    t3 = SumIntP(kl0,kl1);
```

```
    t5 = KaratR(lie,lje);
    DeleteIntP(lie);
    DeleteIntP(lje);
    t2 = SubIntP(t5,t3);
    DeleteIntP(t3);
    DeleteIntP(t5);
    t1 = ShiftIntP(n+1,t2);
    t6 = SumIntP(t1,t4);
    DeleteIntP(t4);
    DeleteIntP(t1);
    r = SumIntP(kl0,t6);
    DeleteIntP(kl0);
    DeleteIntP(t6);
    return r;
  } else
    return MulIntP(li,lj);
}

IntPiece *FactorialIntP( IntPiece *pt ){
IntPiece *l, *pt1 = pt;
    if ((!pt1->ord) && (!pt1->val))
{
   l=smallint2IntP(1);
}
    if ( ( pt1->val > 0 )){
l = KaratR( pt1, FactorialIntP(SubIntP(pt1, smallint2IntP(1)) ));
}
return l;
}

IntPiece *ModExpIntP( IntPiece *li, IntPiece *lj , IntPiece *lk ){
  IntPiece *pt1 = li, *pt2 = lj, *pt3 = lk ,
    *pt4 = new IntPiece , *pt5, *pt6;
  int cont = 0;
  pt6 = smallint2IntP(1);
  IntPiece *ptemp = smallint2IntP(0);

  while ( LessEqualBiggerIntP( pt2 , smallint2IntP(1) ) > 0 ){
    pt4 = QUOEuclDivIntP( pt2 , smallint2IntP(2) );
pt5 = SubIntP( pt2 , KaratR( pt4 , smallint2IntP(2) ) ) ;
    pt2 = DuplicateIntP( pt4 );
    if ( LessEqualBiggerIntP( pt5 , smallint2IntP(1) ) == 1  ){
```

```
      pt6 = KaratR( pt6 , pt1 );
  pt6 = RESEuclDivIntP( pt6 , pt3 );
}
    pt1 = RESEuclDivIntP( KaratR( pt1 , pt1 ) , pt3 );
    cont++;
  }
  return pt6;
}

void EuclDivIntPbysmallint( IntPiece *li, IntPiece *lj, IntPiece **quo,
IntPiece **res){
  IntPiece *pt1 = li, *pt2 = lj, *pt3 = new IntPiece;
  int j=0;
  unsigned long int w, r=0;
  int n=LenIntP(pt1);
  int nn = LessEqualBiggerIntP(pt1,pt2);
  pt1 = GoToLastOrd(pt1);
  if (! nn){
    *quo=smallint2IntP(0);
*res=DuplicateIntP(pt1);
  }
  if (nn == 1){
    *quo=smallint2IntP(1);
*res=smallint2IntP(0);
  }
  if (nn == 2){
    pt3 = ZeroIntP(0);
while(pt1 && j<n){
  w = r * (NNBITS+1) + (pt1->val);
  w = w / (pt2->val);
  pt3 -> val = w;
      r = r * (NNBITS+1) + pt1->val;
  r = r % (pt2->val);
  j++;
  pt1 = pt1->prev;
  if (pt1) pt3=ShiftIntP(1,pt3);
}
*quo=DuplicateIntP(pt3);
*res=smallint2IntP(r);
  }
}
```

```
void EuclDivIntP( IntPiece *li, IntPiece *lj , IntPiece **quo, IntPiece **res){
  IntPiece *pt1 , *pt2 , *pt3 = new IntPiece ,
   *ptd = new IntPiece, *pt= new IntPiece ;
  pt1 = DuplicateIntP(li);//vai ser alterado mas assim nao altera o input
  pt2 = DuplicateIntP(lj);//vai buscar o valor + sig e dp aponta p o inicio de lj
  pt3 = smallint2IntP(0);//vai ser duplicado para dar o quo
  int t = LessEqualBiggerIntP(pt1,pt2);
  if (! t){
    *quo=smallint2IntP(0);
*res=DuplicateIntP(pt1);
  }
  if ( t == 1){
    *quo=smallint2IntP(1);
*res=smallint2IntP(0);
  }
  if ( t == 2){
   int n = LenIntP(pt2);
   if(n<=1) EuclDivIntPbysmallint( pt1, pt2, quo, res);
 else{
    unsigned long int temp, w, d, uj0, uj1, uj2, v0, v1;
  IntPiece *ptP = new IntPiece, *ptlast = new IntPiece;
  //buscar o v0, o valor + sig do dividendo
  pt2 = GoToLastOrd(pt2);
  v0 = pt2->val;
  pt2 = GoToGivenOrd(pt2,0);
    //calculo de d e atribuicao a um intpiece
  d = (NNBITS+1) / ( (v0) + 1);
  ptd = smallint2IntP(d);
  //normalizacao dos valores do Divisor e do dividendo
  pt1 = KaratR( pt1 , ptd );
    pt2 = KaratR( pt2 , ptd );
  //buscar os "novos" 2 valores + sig do dividendo
  pt2 = GoToLastOrd(pt2);
  v0 = pt2->val;
  v1 = (pt2->prev)->val;
  pt2 = GoToGivenOrd(pt2,0);

  int N = LenIntP(pt1), m = N - n , j=0;
  int lastord = N - n - 1;//ultima ordem a ficar em pt1

  //o pt vai ser o Divisor temporario em cada iteracao
  //pt fica so c as ordens a partir de lastord exclusive (pode nao ter +)
```

```
 pt = SplitIntP(pt1, lastord);

 //comparar o Divisor com o dividendo para saber
 //se tomo + posicao ou acrescento zero
 t = LessEqualBiggerIntP(pt,pt2);

   //se pt<pt2 acrescenta-se outro cartao e lastord e m teem de ser alterados
 if (t == 0 ){
  CutLastOrdPasteToFirstOrd( &pt1 , &pt );
lastord = lastord - 1;
  m = m - 1;
 }
 //se pt>=pt2 acrescenta-se um zero na posicao + sig
 else AddValInLastOrdIntP(pt,0);

 while( j <= m){
  if( j ){
     pt3 = ShiftIntP( 1 , pt3 );
 CutLastOrdPasteToFirstOrd( &pt1 , &pt );
 lastord = lastord - 1;
}
int b = n - LenIntP( pt );
int k = 0;
//para obter os valores + sig de pt este tem de ter +1 posicao do
//que pt2 vou acrescentar zeros no fim
while ( k < b+1){
    pt = AddValInLastOrdIntP(pt,0);
  k++;
}
ptlast = GoToLastOrd(pt);
uj0 = ptlast->val;
uj1 = (ptlast->prev)->val;
uj2 = ((ptlast->prev)->prev)->val;
if ( ( uj0 ) == (v0) ){
 w = NNBITS;
 temp = ( ( uj0 ) * (NNBITS + 1)  + ( uj1 ) ) ;
}
else{
 temp = ( ( uj0 ) * (NNBITS + 1)  + ( uj1 ) ) ;
w = temp / v0;
}
    if ( ( ( temp - v0*w ) <= NNBITS  ) ){
```

```
      if ( ( v1*w ) > ( ( temp - v0*w ) * (NNBITS + 1) + uj2 ) )   w = w - 1;
 }
     if ( ( ( temp - v0*w ) <= NNBITS  ) ){
  if ( ( v1*w ) > ( ( temp - v0*w ) * (NNBITS + 1) + uj2 ) )   w = w - 1;
 }
 ptP = KaratR( smallint2IntP(w) , pt2  );
 t = LessEqualBiggerIntP( pt , ptP  );
 if ( t  ==  0 ){
  w = w - 1;
ptP = SubIntP( ptP , pt2 );
 }
 t = LessEqualBiggerIntP( pt , ptP  );
 pt = SubIntP (  pt , ptP );
 pt3 ->val = w;
 j++;
  }//fecha o ciclo while do j

*quo = DuplicateIntP(pt3);
*res = smallint2IntP(0);
IntPiece *ptresto = new IntPiece;
ptresto = smallint2IntP(0);
EuclDivIntPbysmallint( pt , ptd , res, &ptresto );

 }//fecha o else
  }//fecha t == 2
}

//SO DEVOLVE O QUOCIENTE
IntPiece *QUOEuclDivIntP( IntPiece *li, IntPiece *lj){
  IntPiece *pt1 , *pt2 , *pt3 = new IntPiece ,
   *ptd = new IntPiece, *pt= new IntPiece ;
  //vai ser alterado mas assim nao altera o input
  pt1 = DuplicateIntP(li);
  //vai buscar o valor + sig e dp vai manter-se a apontar para o inicio de lj
  pt2 = DuplicateIntP(lj);
  //vai ser duplicado para dar o quo
  pt3 = smallint2IntP(0);
  int t = LessEqualBiggerIntP(pt1,pt2);
  if (! t) pt3 = smallint2IntP(0);
  if ( t == 1) pt3 = smallint2IntP(1);
  if ( t == 2){
    int n = LenIntP(pt2);
```

```
    if(n<=1){
  IntPiece *ptres = new IntPiece;
  EuclDivIntPbysmallint( pt1, pt2, &pt3, &ptres);
}
else{
      unsigned long int temp, w, d, uj0, uj1, uj2, v0, v1;
  IntPiece *ptP = new IntPiece, *ptlast = new IntPiece;
  pt2 = GoToLastOrd(pt2);
  v0 = pt2->val;
  pt2 = GoToGivenOrd(pt2,0);
  d = (NNBITS+1) / ( (v0) + 1);
  ptd = smallint2IntP(d);
  pt1 = KaratR( pt1 , ptd );
  pt2 = KaratR( pt2 , ptd );
  pt2 = GoToLastOrd(pt2);
  v0 = pt2->val;
  v1 = (pt2->prev)->val;
  pt2 = GoToGivenOrd(pt2,0);
  int N = LenIntP(pt1), m = N - n , j=0;
  int lastord = N - n - 1;//ultima ordem a ficar em pt1
      //pt fica so c as ordens a partir de lastord exclusive (pode nao ter +)
  pt = SplitIntP(pt1, lastord);
  t = LessEqualBiggerIntP(pt,pt2);
  if (t == 0 ){
    CutLastOrdPasteToFirstOrd( &pt1 , &pt );
lastord = lastord - 1;
m = m - 1;
  }
      else AddValInLastOrdIntP(pt,0);
  while( j <= m){
    if( j ){
  pt3 = ShiftIntP( 1 , pt3 );
  CutLastOrdPasteToFirstOrd( &pt1 , &pt );
  lastord = lastord - 1;
}
int b = n - LenIntP( pt );
int k = 0;
//para obter os valores + sig de pt este tem de ter +1 posicao
//do que pt2 vou acrescentar zeros no fim
while ( k < b+1){
  pt = AddValInLastOrdIntP(pt,0);
  k++;
```

```
}
ptlast = GoToLastOrd(pt);
uj0 = ptlast->val;
uj1 = (ptlast->prev)->val;
uj2 = ((ptlast->prev)->prev)->val;
if ( ( uj0 ) == (v0) ){
  w = NNBITS;
  temp = ( ( uj0 ) * (NNBITS + 1)  + ( uj1 ) ) ;
}
else{
  temp = ( ( uj0 ) * (NNBITS + 1)  + ( uj1 ) ) ;
  w = temp / v0;
}
if ( ( ( temp - v0*w ) <= NNBITS  ) ){
          if ( ( v1*w ) > ( ( temp - v0*w ) * (NNBITS + 1) + uj2 ) ){
    w = w - 1;
  }
 }
  if ( ( ( temp - v0*w ) <= NNBITS  ) ){
            if ( ( v1*w ) > ( ( temp - v0*w ) * (NNBITS + 1) + uj2 ) ){
  w = w - 1;
}
  }
ptP = KaratR( smallint2IntP(w) , pt2  );
t = LessEqualBiggerIntP( pt , ptP  );
if ( t  ==  0 ){
w = w - 1;
ptP = SubIntP( ptP , pt2 );
}
t = LessEqualBiggerIntP( pt , ptP  );
pt = SubIntP (  pt , ptP );
pt3 ->val = w;
j++;

}//fecha o ciclo while do j
}//fecha o else
}//fecha t == 2

return pt3;
}

//SO DEVOLVE O RESTO
```

```
IntPiece *RESEuclDivIntP( IntPiece *li, IntPiece *lj){
  IntPiece *pt1 , *pt2 , *pt3 = new IntPiece ,
   *ptd = new IntPiece, *pt= new IntPiece, *ptr= new IntPiece ;
  //vai ser alterado mas assim nao altera o input
  pt1 = DuplicateIntP(li);
  //vai buscar o valor + sig e dp vai manter-se a apontar para o inicio de lj
  pt2 = DuplicateIntP(lj);
  pt3 = smallint2IntP(0);
  //vai ser o valor devolvido - o resto
  ptr = smallint2IntP(0);

  int t = LessEqualBiggerIntP(pt1,pt2);
  if (! t) ptr = DuplicateIntP(pt1);
  if ( t == 1) ptr = smallint2IntP(0);
  if ( t == 2){
    int n = LenIntP(pt2);
if(n<=1){
      IntPiece *ptquo = new IntPiece;
  EuclDivIntPbysmallint( pt1, pt2, &ptquo, &ptr);
}
else{
      unsigned long int temp, w, d, uj0, uj1, uj2, v0, v1;
  IntPiece *ptP = new IntPiece, *ptlast = new IntPiece;
  pt2 = GoToLastOrd(pt2);
  v0 = pt2->val;
  pt2 = GoToGivenOrd(pt2,0);
  d = (NNBITS+1) / ( (v0) + 1);
  ptd = smallint2IntP(d);
  pt1 = KaratR( pt1 , ptd );
  pt2 = KaratR( pt2 , ptd );
  pt2 = GoToLastOrd(pt2);
  v0 = pt2->val;
  v1 = (pt2->prev)->val;
  pt2 = GoToGivenOrd(pt2,0);
  int N = LenIntP(pt1), m = N - n , j=0;
  int lastord = N - n - 1;//ultima ordem a ficar em pt1
      //pt fica so c as ordens a partir de lastord exclusive (pode nao ter +)
  pt = SplitIntP(pt1, lastord);
  t = LessEqualBiggerIntP(pt,pt2);
  if (t == 0 ){
    CutLastOrdPasteToFirstOrd( &pt1 , &pt );
lastord = lastord - 1;
```

```
m = m - 1;
  }
  else AddValInLastOrdIntP(pt,0);
  while( j <= m){
    if( j ){
  pt3 = ShiftIntP( 1 , pt3 );
  CutLastOrdPasteToFirstOrd( &pt1 , &pt );
  lastord = lastord - 1;
}
int b = n - LenIntP( pt );
int k = 0;
while ( k < b+1){
      pt = AddValInLastOrdIntP(pt,0);
  k++;
}
ptlast = GoToLastOrd(pt);
uj0 = ptlast->val;
uj1 = (ptlast->prev)->val;
uj2 = ((ptlast->prev)->prev)->val;
if ( ( uj0 ) == (v0) ){
  w = NNBITS;
  temp = ( ( uj0 ) * (NNBITS + 1)  + ( uj1 ) ) ;
}
else{
  temp = ( ( uj0 ) * (NNBITS + 1)  + ( uj1 ) ) ;
  w = temp / v0;
}
  if ( ( ( temp - v0*w ) <= NNBITS  ) ){
          if ( ( v1*w ) > ( ( temp - v0*w ) * (NNBITS + 1) + uj2 ) ){
    w = w - 1;// isto nao stava aqui!!!!!
  }
 }
  if ( ( ( temp - v0*w ) <= NNBITS  ) ){
            if ( ( v1*w ) > ( ( temp - v0*w ) * (NNBITS + 1) + uj2 ) ){
  w = w - 1;// isto nao stava aqui!!!!!
}
  }
          ptP = KaratR( smallint2IntP(w) , pt2  );
  t = LessEqualBiggerIntP( pt , ptP  );
  if ( t  ==  0 ){
    w = w - 1;
ptP = SubIntP( ptP , pt2 );
```

```
  }
  t = LessEqualBiggerIntP( pt , ptP  );
  pt = SubIntP (  pt , ptP );
  pt3 ->val = w;
  j++;
  }//fecha o ciclo while do j

      ptr = QUOEuclDivIntP( pt , ptd );
}//fecha o else
  }//fecha t == 2
return ptr;
}

//#################AUXILIARES#################################################
unsigned long int ConvertBinaryStringToInteger(string S){
  unsigned int valor = 0;
  int i=0;
  while( S[i] != '\0' ){
    int d = (S[i] == '0') ? 0 : 1;
    valor <<= 1;
    valor += d;
    i++;
   }
  return valor;
}


string IntP2DecimalStr(IntPiece *li){
  IntPiece *pt1 = li, *pt2 = new IntPiece, *pt3 = new IntPiece,
   *pt = new IntPiece, *ptd = new IntPiece;
  string DC = "";
  pt2 = smallint2IntP(10);
  pt = smallint2IntP(0);
  ptd = DuplicateIntP(pt);
  //fica a apontar para o inicio do resultado na forma de IntPiece
  pt3 = ptd;
  while( LessEqualBiggerIntP( pt1 , pt2 ) > 0 ){
    EuclDivIntPbysmallint(  pt1, pt2 , &pt1, &pt);

ptd->val = pt->val;
ptd->next = new IntPiece;
(ptd->next)->prev = ptd;
```

```
(ptd->next)->next = NULL;
    (ptd->next)->ord = ptd->ord + 1;

ptd = ptd->next;
  }

   ptd->val = pt1->val;
   ptd->next = NULL;

   while(ptd){
     int temp = ptd->val;
 DC.append(1, temp + 48);
 ptd = ptd -> prev;

   }
  cout<<"\n O IntPiece: \n";
  PrintIntP(pt3);
  cout<<"\n A string: "<<DC<<endl;
  return DC;
}

//cria strings aleatorias comecadas por 1 com nbits bits
string Num(int nbits){
    string Num="1";
int c_Num=Num.length();
srand( (unsigned)time( NULL ) );
while(c_Num<nbits)
{
int r=rand();//obtem um n aleatorio q depende do time
r=r%2;
Num.append(1,r+48);//cola os bits em Num
c_Num=Num.length();
}
return Num;
}

IntPiece *Str2IntP(string S){
  IntPiece *l=new IntPiece, *li = l;
  int c_S=S.length();
  int i=0;
  int base=NBITS;
```

```
  li->prev=NULL;
  while(c_S>base){
    string sub_S = S.substr(c_S-base,base);
    li->val = ConvertBinaryStringToInteger(sub_S);
    li->ord = i;
    li->next = new IntPiece;
    (li->next)->prev=li;
    li = li->next;
    S = S.erase(c_S-base,base);
    c_S = S.length();
    i++;
  }
  li->ord = i;
  li->val = ConvertBinaryStringToInteger(S);
  li->next = NULL;
  return l;
}

IntPiece *smallint2IntP(int i){
  IntPiece *pt = new IntPiece;
  pt->val = i;
  pt->ord = 0;
  pt->next = NULL;
  pt->prev = NULL;
  return pt;
}

IntPiece *ZeroIntP(int n){
  IntPiece *pt=new IntPiece, *pt1=pt;
  int i=0;
  pt->val = 0;
  pt->ord =0;
  pt->prev = NULL;
  while(i++<n-1){
    pt1->next = new IntPiece;
    (pt1->next)->prev = pt1;
    pt1 = pt1->next;
    pt1->ord = i;
    pt1->val = 0;
  }
  pt1->next = NULL;
  return pt;
```

```
}

void PrintIntP(IntPiece *l){
  int base = NBITS;
  IntPiece *li = l;
  cout<<"\nO valor: "<<li->val<<" cuja ordem e: "<<li->ord<<"  \n";
  while(li->next){
    li = li->next;
    cout<<"O valor: "<<li->val<<" cuja ordem e: "<<li->ord<<"  "<<endl;
  }
}

void PrintInFileIntP(IntPiece *l){
char fileName[10]="IntPiece";
    ofstream fout(fileName);//abre o ficheiro
int base = NBITS;
IntPiece *li = l;
fout<< li->val<<",";
while(li->next){
    li = li->next;
    fout<<li->val<<",";
  }
  fout.close();//fecha o ficheiro
}

int LenIntP(IntPiece *pt){
  while(pt->next){
    pt = pt->next;
  }
  return pt->ord+1;
}

//devolve 0,1 ou 2 conforme Less, Equal,Bigger
int LessEqualBiggerIntP( IntPiece *li, IntPiece *lj){
  IntPiece *pt1 = li, *pt2 = lj;
  int dif_len = ( LenIntP(pt1) - LenIntP(pt2));
  int flag=0;
  if ( dif_len > 0 )  flag=2;
  if ( ! dif_len ){
    pt1 = GoToLastOrd(pt1);
pt2 = GoToLastOrd(pt2);
while( (pt1->val >= pt2->val) && !flag ){
```

```
  if ( (pt1->val > pt2->val) ) flag=2;
  else{
    if( pt1->prev ){
  pt1 = pt1->prev;
  pt2 = pt2->prev;
}
else flag=1;
  }
}
  }
return flag;
}


void DeleteIntP(IntPiece *pt){
  while(pt->next){
    pt=pt->next;
    delete pt->prev;
  }
  delete pt;
}

IntPiece *SplitIntP(IntPiece *li, int lastord){
  IntPiece *pt=li, *pt1;
  if( lastord < 0 ){
  pt1 = DuplicateIntP(pt);
  pt = ZeroIntP(1);
  return pt1;
  }
  else{
    if(lastord+1 >= LenIntP(li)) return ZeroIntP(1);
while(pt->ord <= lastord) pt = pt->next;

(pt->prev)->next = NULL;
pt->prev = NULL;

pt1 = pt;
int i = 0;
do{
  pt->ord = i++;
  pt = pt->next;
}while(pt);
```

```
return pt1;
  }
}

IntPiece *MulIntP(IntPiece *li, IntPiece *lj){
  unsigned long p = (li->val)*(lj->val);
  unsigned long M2=NNBITS<<NBITS;
  IntPiece *pt, *pt1;

  if(!(p & M2))
    return smallint2IntP(p);
  else{
    pt = smallint2IntP(p & NNBITS);
    pt1 = smallint2IntP(p>>NBITS);
    pt1->prev = pt;
    pt->next = pt1;
    pt1->ord = 1;
    return pt;
  }
}

IntPiece *DuplicateIntP(IntPiece *pt1){
  IntPiece *pt=new IntPiece, *pt2=pt;

  pt2->val = pt1->val;
  pt2->prev = NULL;
  pt2->ord = 0;

  while(pt1->next){
    pt2->next = new IntPiece;
    (pt2->next)->prev = pt2;
    pt2 = pt2->next;
    pt1 = pt1->next;
    pt2->val = pt1->val;
    pt2->ord = pt1->ord;
  }
  pt2->next = NULL;
  return pt;
}

IntPiece *ShiftIntP(int i, IntPiece *li){
  IntPiece *pt, *pt1;
```

```
  if(LenIntP(li)==1  && !li->val) return li;
  pt1 = ZeroIntP(i);
  pt = pt1;
  while(pt->next) pt = pt->next;
  pt->next = li;
  li->prev = pt;
  pt = li;
  do{
    pt->ord += i;
    pt = pt->next;
  }while(pt);
  return pt1;
}

IntPiece *GoToLastOrd(IntPiece *li){
  IntPiece *pt1 = li;
  while(pt1->next)  pt1 = pt1->next;
  return pt1;
}

IntPiece *GoToGivenOrd(IntPiece *li, int i){
  IntPiece *pt1 = li;
  if(pt1->ord <i){
    while(pt1->ord < i) pt1 = pt1->next;
  }
  else{
    while(pt1->ord > i)  pt1 = pt1->prev;
  }
  return pt1;
}

void CutLastOrdPasteToFirstOrd( IntPiece **li, IntPiece **lj ){
  IntPiece *pt1 = *li, *pt2 = *lj;
  int n = LenIntP( pt1 );
  pt1 = GoToLastOrd( pt1 );
  pt2 = ShiftIntP( 1, pt2 );
  pt2->val = pt1->val;
  pt1 = GoToGivenOrd( pt1 , 0 );
  if ( n-2 >= 0 ) SplitIntP( pt1, n-2);
  else pt1 = ZeroIntP(1);
  *li = DuplicateIntP(pt1);
  *lj = DuplicateIntP(pt2);
```

```
}

IntPiece *AddValInLastOrdIntP( IntPiece *li , int i){
  IntPiece *pt = li, *pt1 = pt;
  pt1 = GoToLastOrd( pt1 );
  pt1->next = new IntPiece;
  (pt1->next)->val = i;
  (pt1->next)->ord = (pt1->ord)+1;
  (pt1->next)->next = NULL;
  (pt1->next)->prev = pt1;
return pt;
}

//#############CRIACAO DA CHAVE#########################
vector <int> KeyConstr(IntPiece *li );
vector <int> Val2Coef( IntPiece *li);
vector <int> Coef2Key( vector <int> coef );
vector <int> Key2Coef (vector <int> key );
IntPiece *Key2Val (vector <int> key );

//AUXILIARES (CONSTR CHAVE)
int Count_if_MI(vector <int> v , int begin , int end , int value);
int Count_if_Min(vector <int> v , int begin , int end , int value);

vector <int> KeyConstr(IntPiece *li ){
  IntPiece *pt1 = li, *pt2 = new IntPiece , *pt3 = new IntPiece;
  vector <int> key;
  key = Coef2Key( Val2Coef( li ) );
  return key;
}

vector <int> Val2Coef( IntPiece *li){
  IntPiece *pt1 = li, *pt2 = new IntPiece , *pt3 = new IntPiece;
  int c;
  vector <int> coef;
  pt2 = FactorialIntP( smallint2IntP( NSIMB - 1 ) );

  for(int i=0;i<NSIMB;i++){
    if ( i == NSIMB-1 ) pt3 = QUOEuclDivIntP( pt1 , pt2 );
else{
  EuclDivIntP( pt1 , pt2 , &pt3 , &pt1 );
  pt2 = QUOEuclDivIntP( pt2 , smallint2IntP( NSIMB - 1 - i) );
```

```
}
c = pt3->val;
    coef.push_back(c);
  }
  return coef;
}

vector <int> Coef2Key( vector <int> coef ){
  vector <int> key;
  //para assinalar com 1 os que ja foram escolhidos
  map<int, int> Contr;
  for (int i=0;i<NSIMB;i++) Contr[i] = 0;
  key.push_back(coef[0]);
  for( i=1;i<NSIMB;i++){
    int c , temp = coef[i] , n = temp;
    int FLAG = 0;
while (!FLAG){
      //conta quantos sao menores ou iguais a temp entre as posicoes 0 e i
  c = Count_if_MI( key , 0 , i-1 , n );
  if( ( (n - temp - c) == 0 ) && ( Contr[n] == 0  )   ) FLAG = 1;
  else n +=1;
}
  key.push_back(n);
  Contr[n]=1;
  }
  return key;
}

vector <int> Key2Coef (vector <int> key ){
  vector <int> coef;
  coef.push_back(key[0]);
  for( int i=1;i<NSIMB;i++){
    int c , temp = key[i] ;
    //conta quantos sao menores a temp entre as posicoes 0 e i
c = Count_if_Min( key , 0 , i-1 , temp );
temp = temp - c;
coef.push_back(temp);
  }
  return coef;
}

IntPiece *Key2Val (vector <int> key ){
```

```
  IntPiece *pt1 = new IntPiece , *pt2 = new IntPiece, *pt3 = new IntPiece;
  int c;
  pt3 = ZeroIntP(1);
  vector <int> coef;
  coef = Key2Coef (key );
  pt2 = FactorialIntP( smallint2IntP( NSIMB - 1 ) );
  int i = 0;
  while(i<NSIMB-1){
    c = coef[i];
pt3 = SumIntP( pt3 , KaratR( pt2 , smallint2IntP(c) ));
pt2 = QUOEuclDivIntP( pt2 , smallint2IntP( NSIMB - 1 - i) );
i++;
  }
  c = coef[i];
  pt3 = SumIntP( pt3 , KaratR( pt2 , smallint2IntP(c) ));
  return pt3;
}

int Count_if_MI(vector <int> vec , int begin , int end , int value){
  int contador = 0 , i = begin;
  while(i < end+1){
    if( vec[i] <= value ) contador++;
i++;
  }
return contador;
}

int Count_if_Min(vector <int> vec , int begin , int end , int value){
  int contador = 0;
  for (int i= begin; i < end+1; i++ ){
    if( vec[i] < value ) contador++;
  }
  return contador;
}

//###################CIFRAR E DECIFRAR##########################
string CifraM(string MO , vector <int> key);
string DeCifraM(string MC , vector <int> key);

//as seguintes teem em conta o tamamnho maximo de uma mensagem(sms)
string CifraTextoSimples(string TS , vector <int> key, int S );
string DeCifraCriptograma(string CR , vector <int> key , int S);
```

```
//AUXILIARES (CIFRAR/DECIFRAR)
int LenIniBl(vector <int> key );
void LenMAXmin( int *M , int *m );

string CifraM(string MO , vector <int> key){
  //minimo e Maximo para os comprimentos de bloco
  int m = 0, M = 0 ;
  LenMAXmin( &M , &m );
  int b=LenIniBl(key);//comp. bloco inicial
  vector <int> coord_l(NSIMB), coord_c(NSIMB);
  for(int j=0;j<NSIMB;j++){
    coord_l[key[j]]=j/K;
    coord_c[key[j]]=j%K;
  }
  string MC;
  basic_string <char>::size_type pos;
  int a = 0; //posicao inicial
  int len_MO = MO.length();
  if( b > len_MO ) b = len_MO;
  int res = len_MO; //res = len_MO-1-b+1
  if ( res < (2*M) ) b = res/2;
  while (res > 0){
   res = res  - b; //res = len_MO-1-b+1
   vector <int> coord(2*b);
 int sum = 0;
   for(j=0; j<b;j++){
  pos = ALFABETO.find(MO[a+j],0);
  coord[j] = coord_l[pos];
  coord[j+b] = coord_c[pos];
  sum+=pos;
 }
 if ( res < (2*M) ) sum = res/2;
 if ( res < M ) sum = res;
   else sum = sum % ( M-m+1 ) + m;
 for(j=0; j<b;j++){
  MC.append(1, ALFABETO[key[K*coord[2*j]+coord[2*j+1]]]);
 }
 a+=b;
 b = sum;
}
  return MC;
```

```
}

string DeCifraM(string MC , vector <int> key){
  //minimo e Maximo para os comprimentos de bloco
  int m = 0 , M =0 ;
  LenMAXmin( &M , &m );
  int b=LenIniBl(key);//comp. bloco inicial
  vector <int> coord_l(NSIMB), coord_c(NSIMB);
  for(int j=0;j<NSIMB;j++){
    coord_l[key[j]]=j/K;
    coord_c[key[j]]=j%K;
  }
  string MO;
  basic_string <char>::size_type pos;
  int a = 0;//posicao inicial
  int len_MC = MC.length();
  if( b > len_MC ) b = len_MC;
  int res = len_MC;
  if ( res < (2*M) ) b = res/2;
  while (res > 0){
    res = res- b; //res = len_MC-1-b+1
    vector <int> coord(2*b);
int sum = 0;
for(j=0; j<b;j++){
  pos = ALFABETO.find(MC[a+j],0);
  coord[2*j] = coord_l[pos];
      coord[2*j+1] = coord_c[pos];
}
for(j=0; j<b;j++){
char t = ALFABETO[key[K*coord[j]+coord[j+b]]];
MO.append(1, t);
pos = ALFABETO.find(t,0);
sum+=pos;
}
if ( res < (2*M) ) sum = res/2;
if ( res < M ) sum = res;
else sum = sum % ( M-m+1 ) + m;
    a = a + b;
b = sum;
  }
  return MO;
}
```

```
int LenIniBl(vector <int> key ){
  int b, min, Max;
  LenMAXmin( &Max , &min );
  b = (key[0]*(Max-min))/(NSIMB-1)+min;
  return b;
}

void LenMAXmin( int *M , int *m ){
*M = 50 , *m = 5 ;
}

//cifra em varios blocos cada um com o tamanho maximo de uma mensagem
string CifraTextoSimples(string TS , vector <int> key , int S){
  string CR;
  int a = 0; //posicao inicial
  int len_TS = TS.length();
  int len_M = S;// S = tamanho fixo de uma mensagem
  if( len_M > len_TS ) len_M = len_TS;
  int res = len_TS;
  int flag = 0;//para entrar no ciclo

  while ( !flag ){
    string blocoC;
basic_string <char> blocoO = TS.substr ( a , len_M );
    blocoC = CifraM( blocoO , key );
    a+=len_M;
    if( len_M > res ) len_M = res;
res = res - len_M;
if ( res == 0 ) flag = 1;
CR = CR + blocoC;
  }
return CR;
}

//decifra os blocos com o tamanho maximo de uma mensagem
string DeCifraCriptograma(string CR , vector <int> key , int S){
  string TS;
  int a = 0; //posicao inicial
  int len_CR = CR.length();
  int len_M = S;// S = tamanho fixo de uma mensagem
  if( len_M > len_CR ) len_M = len_CR;
```

```
  int res = len_CR;
  int flag = 0;//para entrar no ciclo
  while ( !flag ){
    string blocoO;
basic_string <char> blocoC = CR.substr ( a , len_M  );
    blocoO = DeCifraM( blocoC , key );
    a+=len_M ;
if( len_M  > res ) len_M  = res;
res = res - len_M ;
if ( res == 0 ) flag = 1;
TS = TS + blocoO;
  }
return TS;
}

//*********************************************************
int main()
{
//s_A tem o valor em binário do valor K que originará a chave
  string s_A="10001111101011011110101111110001111100110011011010111001000
00100001101010110000010010110010010110001001101011100100110100001010100110
01100000010111010000000101101110110100001100000010111111100100111111011001
11101101100110111111010011001110111111100101001110101000000011011111111001
11100101100101110010000011111011010110100111111111111111111111111111111111
11111111111111111111111111111111111111111111111111";

  IntPiece *ValorK;

  //o valor de s_A convertido para IntPiece, ie, base 2^16
   ValorK = Str2IntP(s_A);
   cout<<"\nO valor K:\n";
   PrintIntP(ValorK);

   vector <int> Chave(0);
   Chave = KeyConstr(ValorK);
   cout<<"\n A CHAVE OBTIDA: \n"<<endl;
   for ( int k=0;k<NSIMB;k++) cout<<Chave[k]<<endl;

  //*************************************************************
  vector <int> teste;
  teste = Key2Coef( Chave );
  IntPiece *TesteChave;
```

```
  TesteChave = Key2Val(Chave);
  if(LessEqualBiggerIntP( ValorK , TesteChave ) == 1 ){
    cout<<"\n Teste chave  CORRECTO!" << endl;
  }
  else cout<<"\n Teste chave  ***ERRADO!***" << endl;
 //*********************************************************

  string s_TS, s_C, s_REC;
s_TS="ANAANAANAfilipa_SEQ!19-01-1977+!1234567890ABCDEFGHIJKLMNOPRSTUVWXYZA
NAfilipa_SEQ!19-01-1977+!1234567890ABCDEFGHIJKLMNOPRSTUVWXYZANAfilipa_SEQ!1
9-01-1977+!1234567890ABCDEFGHIJKLMNOPRSTUVWXYZ";

 //*******ESCREVER A CHAVE EM QUADRADO*****************************
cout <<endl<< "A CHAVE e': \n" <<endl;
cout<<"\t";
for (int i=0;i<9;i++) cout<<i<<"\t";
cout<<endl<<"  ";
for (i=0;i<9;i++) cout<<"--------";
cout<<endl;
for (int h=0; h<9;h++){
cout << h << "|";
for(int hh=0;hh<9;hh++) printf("\t%c",ALFABETO[Chave[h*9+hh]]);
cout<<endl<<endl;
}
 //*************************************************************

 cout<<"\n O texto simples: \n"<<s_TS<<endl;

 string s_CR = CifraTextoSimples(s_TS , Chave , 150);
 cout<<"\n A mensagem cifrada: \n"<<s_CR<<endl;
 string s_TS_MREC;
 s_TS_MREC = DeCifraCriptograma(s_CR , Chave , 150);
 cout<<"\n A mensagem recuperada: \n"<<s_TS_MREC<<endl;
 //***************************COMPARAÇAO**************************
 if ( s_TS_MREC == s_TS ) cout<<"\n CORRECTO!!\n ";
 else cout<<"\n ERRADO!!\n ";
 //*************************************************************
 cout<<"\n  *********** THE END ***********"<<endl;
 return 0;
}
```

Bibliografia

[1] W. Diffie and M. Hellman. *New Directions in Cryptography*, IEEE Transactions on Information Theory 22 (1976) 644-654.

[2] Joachim von Zur Gathen and Jurgen Gerhard, *Modern Computer Algebra*, Cambrigde University Press (1999).

[3] Donald E. Knuth, *The art of computer programming*, Addison-Wesley Publishing Company (1981).

[4] António Machiavelo and Rogério Reis, *Automated Ciphertext-Only Cryptanalysis of the Bifid Cipher*, Cryptologia 31 (2007) 112-124.

[5] Maurice Mignotte, *Mathematics for Computer Algebra*, Springer-Verlag (1992).

[6] Steve Oualline, *Praticall C++Programming*, O'Reilly & Associates (1995).

[7] Jean-François Raymong and Anton Stiglic, *Security Issues in the Diffie-Hellman Key Agreement Protocol* IEEE Transactions on Information Theory (1998) 1-17.

[8] http://www.cplusplus.com (consultado em 20/03/2007).

[9] http://msdn2.microsoft.com/en-us/library/default.aspx (consultado em 20/03/2007).

[10] http://marknelson.us/1989/10/01/lzw-data-compression (consultado em 20/03/2007).

[11] http://michael.dipperstein.com/lzw/index.html (consultado em 22/06/2007).