

Introducción a la Teoría de Códigos, Teoría de la Información y Criptografía

José Galaviz Casas

Departamento de Matemáticas
Universidad Nacional Autónoma de México

Índice general

1 Preliminares	3
1.1 Aritmética modular	3
1.2 Cadenas de símbolos	10
1.2.1 Distancia de Hamming	12
1.3 Códigos	15
1.3.1 Códigos unívocamente decodificables	16
1.3.2 Códigos instantáneos	20
1.4 Anotaciones finales	25
2 Codificación eficiente	27
2.1 Árbol de decodificación	27
2.2 Codificación de Shannon-Fano	29
2.3 Codificación de Huffman	31

2.4	Longitud promedio de un código	36
2.5	La eficiencia del esquema de codificación de Huffman	37
3	Información y entropía	45
3.1	Cantidad de información	45
3.2	Entropía de información	49
3.3	Propiedades de la entropía	50
3.4	Extensiones de una fuente	53
3.5	El primer teorema de Shannon	56
	Intermezzo A Métodos de compresión	63
A.1	Métodos sin pérdida	64
A.1.1	Lempel-Ziv	64
A.1.2	Burrows-Wheeler	67
A.2	Métodos con pérdida	70
A.2.1	El estándar de JPEG	70
4	Codificación para detectar y corregir errores	79
4.1	Regla de decisión y canal de comunicación	80
4.2	Decodificación al vecino más cercano	87
4.3	Distancia mínima, capacidad de detección y corrección	88
4.4	Códigos maximales	96
4.5	Probabilidad de error al decodificar	97
4.6	Códigos de Hamming: idea e implementación	99
4.7	Esferas y códigos perfectos	102
4.8	Equivalencia de códigos	105
4.9	Tasa de transmisión y de corrección	107

4.10	El teorema de la codificación con ruido	109
4.11	Códigos de Golay: idea e implementación	111
5	Códigos lineales	115
5.1	Definición, características	115
5.2	La matriz generadora	117
5.3	Corrección de errores	119
5.4	Probabilidad de decodificar correctamente	122
5.5	Complemento ortogonal, código dual	124
5.6	Decodificación de síndrome	126
5.7	Códigos de Hamming y Golay revisados	131
	Intermezzo B Detección y corrección de errores	135
B.1	Códigos de Reed-Muller	135
B.2	Códigos de Reed-Solomon	138
6	Criptografía simétrica	141
6.1	Conceptos, sustitución monoalfabética	141
6.2	Sustitución polialfabética	144
6.3	Enigma	148
6.4	DES	152
7	Criptografía de llave pública	157
7.1	Funciones de un solo sentido	157
7.2	Factorización	158
7.3	El logaritmo discreto	160
7.4	Mensajes y números	162

7.5	Idea de la criptografía de llave pública	163
7.6	Intercambio de llaves (Diffie-Hellman)	165
7.7	Algunos preliminares	166
7.8	Transmisión de mensajes (Massey-Omura)	170
7.9	Cifrado y firma (ElGamal)	171
7.9.1	Cifrado de mensajes	172
7.9.2	Firma digital	173
7.10	RSA	173
Intermezzo C Software criptográfico		177
C.1	La contraseña en UNIX	177

1

Preliminares

1.1 Aritmética modular

En general en toda la teoría de códigos se suele trabajar en conjuntos cuyo alfabeto es bastante restringido. En las aplicaciones directas de la teoría de códigos, usadas en dispositivos de almacenamiento o transmisión de datos en equipo de cómputo por ejemplo, el alfabeto de símbolos posibles es el conjunto $\{0, 1\}$, los dígitos binarios (que llamaremos en adelante bits). En criptografía suele trabajarse sobre el conjunto de todos los enteros menores que un cierto número primo. En general se trata de subconjuntos de los enteros con una cierta estructura, lo que conocemos como las clases residuales módulo algún número entero. Recordemos algo de nuestro curso de álgebra superior en relación a las clases residuales.

Decimos que a es congruente con b módulo m , o en notación $a \equiv b \pmod{m}$, cuando:

- Se obtiene el mismo residuo al dividir a entre m y b entre m . Así que podemos escribir:

$$\begin{aligned} a &= c_1 m + r_1 \\ b &= c_2 m + r_2 \end{aligned} \tag{1.1.1}$$

de donde:

$$a - b = c_1 m + r_1 - c_2 m - r_2 = (c_1 - c_2)m + (r_1 - r_2)$$

como $r_1 = r_2$:

$$a - b = (c_1 - c_2)m = c_3 m \tag{1.1.3}$$

de donde concluimos el siguiente punto.

- $a - b$ es divisible por m , en notación $m \mid (a - b)$. Si reescribimos 1.1.3 como:

$$a = (c_1 - c_2)m + b$$

concluimos el último punto.

- Existe $k \in \mathbb{Z}$ tal que $a = km + b$.

Ejemplo 1.1

$$\begin{aligned} 7 &\equiv 3 \pmod{2} \\ -5 &\equiv 49 \pmod{6} \\ 96 &\equiv 4 \pmod{23} \end{aligned}$$

◁

Si m es un entero positivo \mathbb{Z}_m denota el conjunto de los enteros módulo m , es decir: $\mathbb{Z}_m = \{0, \dots, m - 1\}$.

Además las clases residuales módulo m constituyen una partición para todo \mathbb{Z} . A cada entero lo asociamos al único elemento de \mathbb{Z}_m que le corresponde por el residuo que se obtiene de dividirlo entre m . Es decir, dado un entero k cualquiera, hay exactamente un entero n en \mathbb{Z}_m tal que:

$$k \equiv n \pmod{m}$$

Otra cosa que debemos recordar es el algoritmo de la división. Como veremos más adelante es de fundamental importancia en los fundamentos de la criptografía de llave

pública. Por el momento lo usaremos sólo para asegurar la existencia de una única solución para una congruencia.

Dados $a, b \in \mathbb{Z}$ existen dos enteros únicos q y r tales que $a = bq + r$ con $0 \leq r < b$. A q normalmente le llamamos el cociente y a r el residuo. Este es la esencia del algoritmo de la división y asegura que existe una única solución para x en la congruencia $x \equiv a \pmod{m}$. Evidentemente x es el residuo que resulta de dividir a entre m .

Ejemplo 1.2

$$x \equiv 61 \pmod{7}$$

significa que $x = 5$ porque $61 = 7 \cdot 8 + 5$

◁

Podemos también darle una estructura más interesante a \mathbb{Z}_m definiendo operaciones en él. Por ejemplo la suma y el producto módulo m .

Definición 1.1 Sean $x, y \in \mathbb{Z}$. La suma de x y y en \mathbb{Z}_m es el residuo que resulta de dividir $x + y \in \mathbb{Z}$ entre m . Análogamente el producto de x y y en \mathbb{Z}_m es el residuo que resulta de dividir $x \cdot y \in \mathbb{Z}$ entre m .

Nótese que con esta definición \mathbb{Z}_m es un conjunto cerrado bajo la suma y el producto módulo m .

Ejemplo 1.3 En \mathbb{Z}_9 :

$$7 + 5 = 3$$

y

$$5 \cdot 4 = 2$$

◁

Definición 1.2 A la terna $(\mathbb{Z}_m, +, \cdot)$ es a lo que, en adelante, llamaremos *los enteros módulo m* . Generalmente, abusando de la notación, lo denotaremos simplemente como \mathbb{Z}_m .

Es bien conocido, por todos los dedicados a la computación el conjunto de los enteros módulo dos. La tabla de sumar en \mathbb{Z}_2 es:

+	0	1
0	0	1
1	1	0

y la de multiplicar:

·	0	1
0	0	0
1	0	1

Nótese que $4 \cdot 2 = 0$ en \mathbb{Z}_8 , pero 4 y 2 son ambos, distintos de cero. Sería deseable que cuando un producto sea cero al menos uno de los operandos sea cero. ¿Qué necesitamos para que esto ocurra?

Supongamos que $a \cdot b = 0 \in \mathbb{Z}_m$, es decir:

$$a \cdot b \equiv 0 \pmod{m}$$

para que esto ocurra necesitamos que $m \mid (a \cdot b)$. Nosotros quisiéramos que alguna de las dos siguientes condiciones fuera verdadera:

1. $a = 0$
lo que significa que $a \equiv 0 \pmod{m}$ y por tanto $m \mid a$.
2. $b = 0$
lo que análogamente significa que: $m \mid b$

Por otra parte, $m \mid (a \cdot b)$ ocurre si y sólo si m divide a, al menos uno, de los factores. Esto significa que m es un número primo.

Teorema 1.1 Sean $a, b \in \mathbb{Z}_p$. $a \cdot b = 0$ en \mathbb{Z}_p implica que $a = 0$ o $b = 0$ si y sólo si p es primo.

Dem.: Notemos que la estructura del teorema es $A \Rightarrow B \iff C$ donde A es $a \cdot b = 0$, B es $a = 0$ o $b = 0$ y C es la afirmación de que p es primo.

\Leftarrow Suponemos A y C , pretendemos concluir B .

Suponemos que $a \cdot b = 0$ en \mathbb{Z}_p . Esto significa que $(a \cdot b) \equiv 0 \pmod{p}$ y por tanto $p \mid (a \cdot b)$. Dado que p es primo entonces ocurre alguna de las dos cosas siguientes:

- $p \mid a$, lo que significa que $a \equiv 0 \pmod{p}$, es decir $a = 0$ en \mathbb{Z}_p
- $p \mid b$ lo que análogamente significa que $b = 0$ en \mathbb{Z}_p

\Rightarrow Demostrar $A \Rightarrow B \Rightarrow C$ es equivalente a demostrar $\neg C \Rightarrow \neg(A \Rightarrow B)$. Es decir, supondremos que p no es primo y que no es cierto que si el producto de a y b es cero, a o b son cero.

Si p no es primo entonces es posible dividirlo por algún número que no es ni el mismo ni la unidad, es decir lo podemos escribir como:

$$p = k \cdot r \quad (1.1.5)$$

donde $k, r \in \mathbb{Z}$ y tanto k como r son distintos de p y de 1, y deben ser menores que p entonces $2 \leq k, r \leq p - 1$. Así que son, de hecho elementos de \mathbb{Z}_p distintos de cero y por 1.1.5 tales que:

$$k \cdot r \equiv 0 \pmod{p}$$

Así que: $kr = 0$ en \mathbb{Z}_p .

□

En álgebra, a un conjunto cerrado bajo una operación producto definida en él, que sea asociativa se le denomina semigrupo. Si un semigrupo posee elemento identidad se denomina monoide y si además cada elemento tiene inverso multiplicativo, se denomina grupo. Si la operación es conmutativa el grupo es abeliano. Nuestros \mathbb{Z}_m son un grupo abeliano con la operación $+$ y con el producto (\cdot) son un semigrupo, esto en lenguaje algebraico es un anillo. Pero lo mejor viene cuando m es un número primo, en ese caso \mathbb{Z}_m es un campo.

Definición 1.3 Un campo es un conjunto no vacío F con dos operaciones binarias: adición $(+)$ y producto (\cdot) , que cumplen con las siguientes propiedades:

Sean a, b y c elementos cualesquiera de F

1. Asociatividad

$$a + (b + c) = (a + b) + c$$

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c$$

2. Conmutatividad

$$a + b = b + a$$

$$a \cdot b = b \cdot a$$

3. Distributividad

$$a \cdot (b + c) = a \cdot b + a \cdot c$$

4. Existe un elemento distinguido, denotado como 0 tal que:

$$0 + a = a + 0 = a$$

5. Existe un elemento distinguido, denotado como 1 tal que:

$$1 \cdot a = a \cdot 1 = a$$

6. Existe un inverso aditivo, denotado con $-a$ y un inverso multiplicativo, denotado con a^{-1} tales que:

$$a + (-a) = (-a) + a = 0$$

$$a \cdot (a^{-1}) = (a^{-1}) \cdot a = 1$$

Teorema 1.2 \mathbb{Z}_p es un campo si y sólo si p es primo.

No vamos a demostrar todas las propiedades. Pero sí la que tiene particular relevancia en criptografía y teoría de códigos en general.

Teorema 1.3 En \mathbb{Z}_p todos los elementos distintos de cero tienen un inverso multiplicativo si y sólo si p es primo.

Dem.: \Rightarrow La demostración en este sentido la haremos negando la tesis y demostrando la negación de la hipótesis. Es decir: si p no es primo entonces no todos los elementos de \mathbb{Z}_p tienen inverso.

Si p no es primo significa que es divisible entre algún número distinto de 1 y de p mismo. Es decir, podemos escribir $p = ab$ con a y b enteros distintos de cero y menores a p . Pero entonces $a, b \in \mathbb{Z}_p$ y como el producto de ellos es p entonces $a \cdot b = 0$ en \mathbb{Z}_p .

Ahora bien, si a tuviera inverso a^{-1} entonces tendríamos:

$$b = a^{-1} \cdot (a \cdot b) = a^{-1} \cdot 0 = 0$$

pero habíamos dicho que b era distinto de cero, así que esto es una contradicción. Por tanto a no tiene inverso, existe un elemento en \mathbb{Z}_p sin inverso.

\Leftarrow Supongamos ahora que p es primo, debemos deducir que todos los elementos de \mathbb{Z}_p tienen inverso.

Sea a un elemento cualquiera de \mathbb{Z}_p . Consideremos ahora todos los productos de la forma $a \cdot b$ donde $b \in \mathbb{Z}_p$. Si hubiera un par de elementos $b, b' \in \mathbb{Z}_p$ distintos entre sí pero que dieran el mismo producto al multiplicarse por a tendríamos que $b \neq b'$ y $a \cdot b = a \cdot b'$, es decir

$$(a \cdot b) \equiv c \pmod{p}$$

y

$$(a \cdot b') \equiv c \pmod{p}$$

Es decir al dividir $a \cdot b$ entre p nos da el mismo residuo que el que resulta de dividir $a \cdot b'$ entre p . Es decir:

$$ab = kp + c$$

$$ab' = k'p + c$$

Si el residuo es el mismo y el divisor también, el algoritmo de la división nos dice que los cocientes son los mismos. Esto es: $k = k'$.

Entonces b debe ser también igual a b' lo que contradice nuestra hipótesis. Así que no puede haber dos productos iguales, para toda $b \in \mathbb{Z}_p$ el producto $a \cdot b$ es un elemento distinto de \mathbb{Z}_p . Así que en total hay p productos distintos, todos ellos en \mathbb{Z}_p , entonces el conjunto de todos los posibles productos es exactamente todo \mathbb{Z}_p y debe haber algún producto que sea igual a 1. Es decir, existe alguna $b \in \mathbb{Z}_p$ tal que $a \cdot b = 1$, luego a , un elemento cualquiera de \mathbb{Z}_p tiene inverso.

□

De hecho en un conjunto \mathbb{Z}_m un elemento a tiene inverso multiplicativo si a es primo relativo con m . Esto es un poco más general que lo que acabamos de demostrar, porque si m es primo entonces cualquier elemento en \mathbb{Z}_m resultará primo relativo con m , así que todos los elementos de \mathbb{Z}_m tendrán inverso. Esta generalización será presentada más adelante en la parte de criptografía.

Ejemplo 1.4 En \mathbb{Z}_{11} $6 + 5 = 0$ así que el inverso aditivo de 6 es 5, en notación:

$$-6 = 5$$

Además en \mathbb{Z}_{11} $6 \cdot 2 = 1$, así que

$$6^{-1} = 2$$

Por supuesto, dado que 2 es primo \mathbb{Z}_2 es un campo, un campo curioso: cada elemento es su propio inverso aditivo: sumar es lo mismo que restar. ◁

1.2 Cadenas de símbolos

Codificar significa, esencialmente expresar cosas escritas con ciertos símbolos mediante el uso de otros símbolos. Así que la siguiente definición sentará las bases de todo lo que nos ocupará más adelante.

Definición 1.4 A un conjunto finito $S = \{s_1, s_2, \dots, s_n\}$ de n símbolos distintos le denominaremos un *alfabeto*. A una secuencia finita de elementos de S se le denomina una *cadena sobre S* . Al número de elementos en S le llamaremos el tamaño de S .

Hay que notar que la semántica nada tiene que ver con los símbolos o las cadenas. No nos interesa el significado de las cosas expresadas en un alfabeto, sólo nos interesan las palabras mismas.

Dado que una cadena es finita podemos hablar de su longitud. La longitud de una cadena $\mathbf{x} = x_1x_2 \dots x_k$ es el número de símbolos que la constituyen y la denotaremos con $\text{len}(\mathbf{x}) = k$.

La concatenación o yuxtaposición de dos cadenas \mathbf{x}, \mathbf{y} es la cadena \mathbf{z} obtenida de poner \mathbf{y} inmediatamente a continuación de \mathbf{x} . En ese caso diremos que \mathbf{x} y \mathbf{y} son, respectivamente, prefijo y sufijo de \mathbf{z} .

Por supuesto en caso de que $\mathbf{z} = \mathbf{xy}$ tenemos $\text{len}(\mathbf{z}) = \text{len}(\mathbf{x}) + \text{len}(\mathbf{y})$.

Con S^* denotaremos el conjunto de todas las posibles cadenas sobre un alfabeto S , incluyendo la cadena vacía θ .

Con S^n denotaremos el conjunto de todas las cadenas de longitud n sobre S y con S_n el conjunto de todas las cadenas de longitud n o menor sobre S .

Con $\mathbf{0}$ denotaremos la cadena constituida exclusivamente de ceros cuando nuestro alfabeto tenga al cero, por supuesto.

Con $|A|$ denotaremos la cardinalidad o el tamaño del conjunto A .

Teorema 1.4 Sea S un alfabeto de tamaño $k > 1$.

1. $|S^n| = k^n$
2. $|S_n| = \frac{k^{n+1} - 1}{k - 1}$

Dem.: Para demostrar el primer inciso basta notar que, dado que se trata de cadenas de longitud n entonces hay, en cada cadena, n maneras de escoger el símbolo a poner en la primera posición, cada vez que se elige esta hay n opciones para la siguiente, etc.

Para demostrar el segundo inciso usaremos el primero. El número de cadenas de longitud n y menor es:

$$|S_n| = |S^0| + |S^1| + \cdots + |S^n|$$

esto es:

$$|S_n| = 1 + k + k^2 + \cdots + k^n = \frac{k^{n+1} - 1}{k - 1}$$

El cálculo de la suma parcial es fácil de comprobar, sólo debemos restar a la suma original el producto de k veces ella misma.

□

Teorema 1.5 1. En \mathbb{Z}_2^n el número de cadenas que tienen exactamente k ceros es:

$$\binom{n}{k}$$

2. En \mathbb{Z}_r^n el número de cadenas que tienen exactamente k ceros es:

$$\binom{n}{k} (r-1)^{n-k}$$

Dem.: Esencialmente, de los n lugares disponibles en la cadena debemos elegir donde poner los k ceros, esto es elegir un subconjunto de k posiciones de las n posibles, de allí

$$\binom{n}{k}$$

Para el segundo inciso, ya que se eligieron los k lugares donde van los ceros, hay que elegir ahora que símbolos poner en los restantes $n - k$ lugares. Por supuesto ya no podemos poner ceros así que para todos los lugares podemos elegir sólo de entre $r - 1$ símbolos. □

Hay que notar que en el teorema anterior tenemos la posibilidad de elegir las posiciones de la cadena donde van los ceros, de allí el factor de las combinaciones de k en n . Si estos lugares estuvieran fijos, es decir, si quisiéramos saber cuántas cadenas hay que tengan k ceros en posiciones fijas el resultado es, por supuesto, diferente. En este caso ya no es

posible elegir donde poner los ceros y si sólo puede haber una posibilidad. Si queremos saber cuantas cadenas hay que tengan *al menos* k ceros en ciertas posiciones fijas entonces, debemos contar cuantas opciones hay para las restantes $n - k$ posiciones. En cada una de ellas podemos poner 0 o 1, así que el resultado es 2^{n-k} .

Ejemplo 1.5 En \mathbb{Z}_4^{12} , el número de cadenas que tienen exactamente tres ceros y dos unos es:

$$\binom{12}{3} \binom{9}{2} 2^7$$

primero elegimos tres de las 12 posiciones donde poner cero, luego 2 de las 9 restantes donde poner 1, el resto de las posiciones (7) las rellenamos con 2 o 3. \triangleleft

En general el número de cadenas en \mathbb{Z}_n^k (cadenas de longitud k en \mathbb{Z}_n) que contienen r ceros y t unos es:

$$\binom{k}{r} \binom{k-r}{t} (n-2)^{(k-r-t)}$$

En \mathbb{Z}_4^{12} , el número de cadenas con, al menos, 10 ceros es:

$$\binom{12}{10} 3^2 + \binom{12}{11} 3 + \binom{12}{12}$$

Generalizando, el número de cadenas en \mathbb{Z}_n^k con, al menos, r ceros es:

$$\binom{k}{r} (n-1)^{k-r} + \binom{k}{r+1} (n-1)^{k-r-1} + \dots + \binom{k}{k}$$

1.2.1 Distancia de Hamming

Ahora que tenemos un concepto de cadena de símbolos en un alfabeto, resulta conveniente definir una noción de distancia que nos permita expresar que tan diferentes son dos cadenas entre sí, que tan alejadas están. Para ello utilizaremos la *distancia de Hamming*.

Definición 1.5 La *distancia de Hamming* entre dos cadenas cualesquiera $\mathbf{x}, \mathbf{y} \in S^n$, denotada con $d(\mathbf{x}, \mathbf{y})$, es el número de posiciones en las que \mathbf{x} difiere de \mathbf{y} .

Ejemplo 1.6 La distancia entre 0343201 y 0322101 en \mathbb{Z}_5^7 es 3 dado que las cadenas coinciden en sus dos primeros y en sus dos últimos símbolos y difieren en los 3 centrales. \triangleleft

La distancia de Hamming resulta ser *realmente* una buena noción de distancia o métrica desde el punto de vista formal del término. Es decir, si $\mathbf{x}, \mathbf{y}, \mathbf{z}$ son elementos cualesquiera de S^n la distancia de Hamming cumple con las siguientes propiedades:

1. Positiva definida.

$$d(\mathbf{x}, \mathbf{y}) \geq 0$$

y $d(\mathbf{x}, \mathbf{y}) = 0$ si y sólo si $\mathbf{x} = \mathbf{y}$.

2. Simétrica.

$$d(\mathbf{x}, \mathbf{y}) = d(\mathbf{y}, \mathbf{x})$$

3. Desigualdad del triángulo.

$$d(\mathbf{x}, \mathbf{z}) \leq d(\mathbf{x}, \mathbf{y}) + d(\mathbf{y}, \mathbf{z})$$

No demostraremos las dos primeras propiedades, pero sí la tercera, la más interesante.

Teorema 1.6 *Si \mathbf{x}, \mathbf{y} y \mathbf{z} son tres cadenas cualesquiera en S^n y d denota la distancia de Hamming, entonces se satisface:*

$$d(\mathbf{x}, \mathbf{z}) \leq d(\mathbf{x}, \mathbf{y}) + d(\mathbf{y}, \mathbf{z})$$

Dem.: Las posiciones en las que \mathbf{x} difiere de \mathbf{y} se pueden dividir en dos conjuntos ajenos:

1. Las r posiciones en las que \mathbf{x} difiere de \mathbf{y} y \mathbf{y} no difiere de \mathbf{z} .
2. Las u posiciones en las que \mathbf{x} difiere de \mathbf{y} y \mathbf{y} difiere de \mathbf{z} .

Así que $d(\mathbf{x}, \mathbf{y}) = r + u$ con $u \geq 0$.

Por otra parte las posiciones en las que \mathbf{y} difiere de \mathbf{z} se pueden dividir también en dos conjuntos ajenos:

1. Las t posiciones en las que \mathbf{y} difiere de \mathbf{z} y \mathbf{x} no difiere de \mathbf{y} .
2. Las u posiciones mencionadas con anterioridad, en las que \mathbf{y} difiere de \mathbf{z} y \mathbf{x} difiere de \mathbf{y} .

Así que $d(\mathbf{y}, \mathbf{z}) = t + u$.

En el caso binario, cuando $S^n = \mathbb{Z}_2^n$, las posiciones en las que \mathbf{x} difiere de \mathbf{z} están constituidas por las mencionadas en los incisos (1). Es decir:

$$d(\mathbf{x}, \mathbf{z}) = r + t \leq r + t + 2u = d(\mathbf{x}, \mathbf{y}) + d(\mathbf{y}, \mathbf{z})$$

dado que $u \geq 0$.

En un caso más general en el que el alfabeto no es binario, el hecho de que \mathbf{x} difiera de \mathbf{y} y que \mathbf{y} difiera de \mathbf{z} no necesariamente significa que \mathbf{x} coincida con \mathbf{z} . Así que las posiciones en las que \mathbf{x} difiere de \mathbf{z} están constituidas por las de los incisos (1) y posiblemente por algunas de las restantes u posiciones. Digamos que difieren en w de esas u con $w \leq u$.

$$d(\mathbf{x}, \mathbf{z}) = r + t + w \leq r + t + u \leq r + t + 2u = d(\mathbf{x}, \mathbf{y}) + d(\mathbf{y}, \mathbf{z})$$

Dado que $0 \leq w \leq u$.

□

En caso de trabajar sobre un alfabeto binario, para fines prácticos de implementación, es conveniente pensar la distancia de Hamming entre dos cadenas como el número de unos contenidos en el resultado de aplicar la operación de disyunción exclusiva (a la que también podemos llamar *suma módulo dos*) entre las dos cadenas, la operación bit a bit conocida como OR *exclusivo* o XOR. La tabla de verdad para esta operación, denotada con \oplus , es la siguiente:

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

Notará el lector que el XOR vale 1 cuando ambos operandos son diferentes y 0 en otro caso. Así que el XOR nos “marca” las posiciones en las que difieren dos cadenas binarias. Sólo resta contar el número de 1’s en el resultado del XOR para conocer la distancia de Hamming.

Ejemplo 1.7 Sean $\mathbf{x} = 01001101$ y $\mathbf{y} = 10001110$ dos cadenas en \mathbb{Z}_2^8 . La distancia de Hamming es el número de unos en:

$$X \oplus Y = 11000011$$

por lo que $d(\mathbf{x}, \mathbf{y}) = 4$. ◁

A fin de cuentas podemos pensar a las cadenas binaria como la expresión binaria de un número entero; por supuesto podemos entonces pensar el resultado de un XOR como otro número escrito en binario. Así que contar el número de unos en cualquier cadena binaria consiste en saber cuantos unos tiene la expresión binaria del número que representa. En ciertas circunstancias¹ podemos hacer esto echando mano de la expresión obtenida en [12]:

$$w(n) = n - \sum_{i=1}^n \left\lfloor \frac{n}{2^i} \right\rfloor \quad (1.2.6)$$

donde n es el número del que se quiere saber el número de unos en su expresión binaria y $w(n)$ denota justamente el número de unos en la expresión binaria de n . A este número se le denomina frecuentemente el *peso* de n .

1.3 Códigos

Un código es un conjunto cualquiera de cadenas sobre un cierto alfabeto. Por ejemplo $C = \{0, 01, 101, 1101, 0010\}$ es un código sobre \mathbb{Z}_2 . Formalmente hablando:

Definición 1.6 Sea $A = \{a_1, a_2, \dots, a_r\}$ un conjunto finito de símbolos al que llamaremos *alfabeto del código*. Un código r -ario sobre A es un subconjunto C de A^* de todas las palabras sobre A . El número r es llamado la base del código.

Por supuesto los códigos sobre \mathbb{Z}_2 se llaman binarios.

Definición 1.7 Sea $S = \{s_1, s_2, \dots, s_q\}$ un conjunto finito de símbolos al que nos referiremos como *alfabeto fuente*. Sea C un código. Una *codificación* o *función de codificación* es una función biyectiva $f : S \mapsto C$. Si C es un código y f una codificación entonces (C, f) es llamado un esquema de codificación.

¹En general calcular la suma de la expresión 1.2.6 para un valor particular de n es más complicado que hacer $\log_2(n)$ desplazamientos para obtener cada bit de la expresión binaria de n . Pero si sabemos de antemano el valor máximo de n que se usará en un determinado problema, podemos guardar en un arreglo el peso de cada n que necesitemos usando la expresión.

Ejemplo 1.8 Sea $\{A, B, C, D, E, F\}$ el alfabeto fuente. Entonces f , definida como sigue es una codificación:

$$\begin{aligned} f(A) &= 1 \\ f(B) &= 2 \\ f(C) &= 3 \\ f(D) &= 11 \\ f(E) &= 22 \\ f(F) &= 23 \end{aligned} \tag{1.3.7}$$

◁

Pero hay que notar que la cadena AA se codifica igual que D , BB se codifica igual que E y que BC se codifica igual que F . No hay una única manera de leer la secuencia codificada 11 o 22 o 23. Por cierto, a las palabras escritas en símbolos de código asociadas con el alfabeto fuente se les llama *palabras del código* en nuestro ejemplo las palabras del código son todas las de la derecha del símbolo “=”.

También hay que notar que no todas las palabras del código miden lo mismo, las hay de uno y de dos símbolos. A este tipo de códigos se les denomina de *longitud variable*. Por supuesto también existen códigos en los que las palabras asociadas a los símbolos del alfabeto fuente son todas de la misma longitud. En ese caso al código se le llama de *longitud fija* o de *bloque*.

Representar datos mediante códigos de longitud fija tiene ventajas: Si se lee una secuencia de palabras de código se sabe exactamente donde termina una palabra y comienza la otra. Sin embargo, como veremos más adelante, no es la manera más eficiente de hacerlo.

1.3.1 Códigos unívocamente decodificables

Por supuesto lo que pretendemos con una codificación, es poder ir en ambos sentidos, de allí que sea requisito el que la función de codificación sea biyectiva, para asegurarnos de que sea invertible. Queremos poder codificar y decodificar; mapear los símbolos en el alfabeto fuente a palabras de código y luego poder recuperar los datos originales decodificando estas luego de haberlas transmitido a través de una línea de comunicación o luego de haberlas almacenado durante algún tiempo.

En nuestro ejemplo anterior ya nos percatamos de que no siempre hay una única lectura posible para una secuencia de palabras de código, pero desearíamos que así fuera para evitar

interpretaciones ambiguas. A esta propiedad se le llama decodificación única. Sólo hay una manera de interpretar una secuencia de palabras de código en términos de los símbolos del alfabeto fuente.

Definición 1.8 Un código C sobre un alfabeto A es unívocamente decodificable si, para toda cadena $\mathbf{x} = x_1x_2 \dots x_n$ sobre A existe a lo más una secuencia de palabras de código $\mathbf{c} = \mathbf{c}_1\mathbf{c}_2 \dots \mathbf{c}_m$ que coincide con \mathbf{x} . Es decir: $\mathbf{x} = \mathbf{c}$.

Parafraseando: Un código es unívocamente decodificable si no hay dos secuencias diferentes de palabras de código que representen la misma cadena sobre A , el alfabeto del código.

Ejemplo 1.9 $C_1 = \{\mathbf{c}_1 = 01, \mathbf{c}_2 = 11, \mathbf{c}_3 = 011101\}$ no es unívocamente decodificable dado que la cadena: 011101 puede ser interpretada como \mathbf{c}_3 o como la secuencia $\mathbf{c}_1 \mathbf{c}_2 \mathbf{c}_1$.

En cambio $C_2 = \{\mathbf{d}_1 = 01, \mathbf{d}_2 = 10, \mathbf{d}_3 = 110\}$ sí es unívocamente decodificable. \triangleleft

En un código de longitud variable hay palabras de código más largas que otras. Intuitivamente, si un código tiene muchas palabras cortas entonces es más fácil que ocurra que algunas de ellas sean subcadenas de las más largas, lo que haría que el código no fuera unívocamente decodificable.

Si por ejemplo un código binario tiene la palabra de longitud dos 01, entonces 0 y 1 no pueden ser palabras del código al mismo tiempo, porque 01 puede ver como las dos palabras de longitud uno concatenadas. Si las palabras 110, 100 están en el código entonces no es posible que estén 11, 10 y 0 ni 1, 10 y 00. Es decir no pueden estar todas las posibles palabras de longitud menor que una palabra cualquiera dada del código. Con base en esto formularemos el teorema de McMillan de 1956.

Teorema 1.7 Sea $C = \{\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_q\}$ un código r -ario y sea $\ell_i = \text{len}(c_i)$. Si C es unívocamente decodificable, entonces las longitudes de sus palabras de código deben satisfacer:

$$\sum_{k=1}^q \frac{1}{r^{\ell}} \leq 1$$

Dem.: Sea α_j el número de palabras del código cuya longitud es j . Sea $m = \max_i\{\ell_i\}$ la máxima longitud de las palabras del código.

Recordemos que el número total de cadenas de longitud k en un código r -ario es r^k .

La suma de la desigualdad la podemos escribir como sigue:

$$\sum_{k=1}^q \frac{1}{r^{\ell_k}} = \sum_{j=1}^m \frac{\alpha_j}{r^j} \quad (1.3.8)$$

la suma del lado izquierdo es sobre las q palabras del código, la suma del lado derecho agrupa las palabras de acuerdo a su longitud y por tanto recorre todas las posibles longitudes. Nótese que el cociente $\frac{\alpha_j}{r^j}$ es la proporción de palabras de longitud j que están en el código respecto a todas las posibles palabras de esa longitud.

Sea u un entero positivo cualquiera:

$$\begin{aligned} \left(\sum_{j=1}^m \frac{\alpha_j}{r^j} \right)^u &= \left(\frac{\alpha_1}{r} + \frac{\alpha_2}{r^2} + \cdots + \frac{\alpha_m}{r^m} \right)^u \\ &= \sum_{\{i_j\}} \left[\frac{\alpha_{i_1}}{r^{i_1}} \cdots \frac{\alpha_{i_u}}{r^{i_u}} \right] \\ &= \sum_{\{i_j\}} \frac{\alpha_{i_1} \alpha_{i_2} \cdots \alpha_{i_u}}{r^{i_1+i_2+\cdots+i_u}} \end{aligned} \quad (1.3.9)$$

i_j es una longitud de cadena, así que $1 \leq i_j \leq m$, dado que m es la máxima longitud. Así que $i_1 + i_2 + \cdots + i_u$ está entre u (si cada longitud fuera mínima) y um (si cada longitud fuera máxima):

$$u \leq i_1 + i_2 + \cdots + i_u \leq um$$

Si agrupamos ahora todos los términos de la suma que tienen el mismo denominador, es decir, el mismo valor de $i_1 + i_2 + \cdots + i_u$, podemos escribir 1.3.9 como:

$$\left(\sum_{j=1}^m \frac{\alpha_j}{r^j} \right)^u = \sum_{k=u}^{um} \frac{N_k}{r^k} \quad (1.3.10)$$

donde:

$$N_k = \sum_{i_1+i_2+\cdots+i_u=k} (\alpha_{i_1} \alpha_{i_2} \cdots \alpha_{i_u})$$

Sería bueno que analizáramos qué significa el producto de las α 's. La demostración ya es de suyo bastante poco didáctica. α_j es el número de palabras de código de longitud j , así que el producto es el número total de cadenas de longitud

$$k = i_1 + i_2 + \cdots + i_u$$

construidas con u palabras de las longitudes especificadas por las i_j 's.

Si ahora se suma sobre todas las posibles combinaciones de longitudes que sumen k entonces realmente estamos obteniendo la cantidad total de palabras de longitud k que pueden ser construidas con u palabras de código.

Es decir, dado un conjunto de u longitudes que suman k , el producto de las α 's es el número de palabras de longitud total k que pueden formarse con palabras del código de las longitudes del conjunto. Si luego ya no nos es dado el conjunto de u longitudes que sumen k y nos es encomendado encontrar todas las combinaciones de u longitudes de palabras que cumplan con ese requisito, estaremos buscando realmente todas las palabras de longitud k que pueden formarse pegando u palabras del código.

Dado que C es unívocamente decodificable, no hay dos secuencias de u palabras de código que generen la misma cadena de longitud k . Por supuesto hay a lo más r^k de tales secuencias, así que:

$$N_k \leq r^k \quad (1.3.11)$$

Usando 1.3.11, 1.3.8 y 1.3.10:

$$\begin{aligned} \left(\sum_{k=1}^q \frac{1}{r^{\ell_k}} \right)^u &= \left(\sum_{j=1}^m \frac{\alpha_j}{r^j} \right)^u \\ &= \sum_{k=u}^{um} \frac{N_k}{r^k} \\ &\leq \sum_{k=u}^{um} 1 \\ &\leq um \end{aligned} \quad (1.3.12)$$

Dados cualesquiera enteros m y $x > 1$ siempre es posible hallar un número entero u tal que $x^u > um$. Pero la expresión 1.3.12 indica que con

$$x = \sum_{k=1}^q \frac{1}{r^{\ell_k}}$$

no es posible hallar esa u , así que necesariamente:

$$\sum_{k=1}^q \frac{1}{r^{\ell_k}} \leq 1 \quad (1.3.13)$$

□

Hay que enfatizar que el teorema de McMillan NO nos dice cuando un código es unívocamente decodificable, sino cuándo NO lo es, negando el teorema.

A la desigualdad del teorema de McMillan se le denomina desigualdad de Kraft, así pues, el teorema de McMillan dice que todo código unívocamente decodificable satisface la desigualdad de Kraft pero NO que todo código que la satisface es unívocamente decodificable. Dado un código si la desigualdad de Kraft no se cumple en él entonces seguro no es unívocamente decodificable, si la satisface entonces no podemos decir nada.

1.3.2 Códigos instantáneos

Los códigos instantáneos son un caso particular de los códigos unívocamente decodificables.

Definición 1.9 Un *código instantáneo* es un código unívocamente decodificable en el que la decodificación puede hacerse leyendo secuencialmente de izquierda a derecha la cadena de palabras de código.

Ejemplo 1.10 El código $C = \{c_1 = 1, c_2 = 10\}$ es unívocamente decodificable, pero no instantáneo. Si se recibe la secuencia 1110 y se lee de izquierda a derecha, entonces ocurre lo siguiente:

1. Leemos el primer uno y no sabemos si es c_1 o el principio de c_2 .
2. Leemos el segundo uno y ahora ya sabemos que el anterior era c_1 , pero ahora no sabemos si este nuevo uno es otra vez c_1 o el principio de c_2 .
3. Se lee el tercer uno y ocurre otra vez lo mismo que en el inciso anterior.
4. Se lee el cuarto bit y se determina que la última palabra recibida es c_2 .

Varias veces hemos terminado de leer una palabra del código y no podemos decir aún si realmente la hemos terminado de leer o si es sólo el prefijo de alguna otra palabra. \triangleleft

Esta última afirmación nos da pretexto para formular una conjetura: En los códigos instantáneos ninguna palabra completa del código constituye un prefijo, una subcadena inicial, de otra palabra más larga; equivalentemente ninguna palabra del código comienza con otra palabra de código más corta.

Definición 1.10 Un código tiene la propiedad de prefijo si ninguna palabra de código es prefijo de alguna otra.

Es un poco paradójica la definición anterior: un código es prefijo o tiene la propiedad de prefijo si sus palabras de código no son prefijos de otras.

Ahora podemos formular nuestra conjetura más formalmente:

Teorema 1.8 *Un código C es instantáneo si y sólo si tiene la propiedad de prefijo.*

Dem.: \Rightarrow (Por reducción al absurdo).

Supongamos que C es instantáneo, en cuanto una palabra de código es leída puede ser interpretada. Supongamos que C no tiene la propiedad de prefijo, es decir, existe una palabra de código \mathbf{c} que es prefijo de la palabra de código más larga \mathbf{d} . Entonces al recibir la secuencia \mathbf{cc} , no es posible interpretar la primera \mathbf{c} hasta leer un símbolo más, dado que podría tratarse del principio de \mathbf{d} . Así C no es instantáneo lo que es una contradicción.

\Leftarrow Supongamos que C tiene la propiedad de prefijo, sea n el número de palabras de código leídas e interpretadas.

1. $n = 1$ Se lee completamente una primera palabra \mathbf{c}_1 . Dado que no hay ninguna palabra que sea prefijo de otra entonces podemos estar seguros de haber leído \mathbf{c}_1
2. Suponemos que se puede leer e interpretar secuencialmente $n = k$ palabras. Si leemos una más, cualquiera, $n = k + 1$, dado que no dejamos nada pendiente y la $(k + 1)$ -ésima no es prefijo de ninguna otra entonces puede ser interpretada inmediatamente.

□

Ejemplo 1.11 Un ejemplo de código instantáneo es el *código coma*, llamado así porque al leer un mensaje escrito en código coma, las palabras del código poseen un separador, una especie de signo de puntuación que las delimita. El código coma de ocho palabras es el siguiente:

$$Coma_8 = \{0, 10, 110, 1110, 11110, 111110, 1111110, 1111111\}$$

El cero actúa como delimitador de las palabras del código, salvo en el caso de la última palabra, que es fácilmente distinguible dado que tiene una longitud predeterminada y es la única constituida de puro 1.

Este código es unívocamente decodificable y, más aún, es instantáneo, ninguna palabra es prefijo de otra.

En cambio el código siguiente, muy parecido al código coma, no es instantáneo:

$$C = \{0, 01, 011, 0111, 01111, 011111, 0111111, 01111111, 11111111\}$$

Hay palabras que son prefijo de otras, esto basta para decir que no es instantáneo. Sin embargo sí es unívocamente decodificable, basta leer cualquier mensaje de derecha a izquierda para decodificar de la única manera posible. \triangleleft

Aclaremos un poco las cosas. En un código unívocamente decodificable sólo hay una manera de dividir cualquier mensaje en palabras del código. En un código instantáneo sólo hay una posible división de el mensaje en palabras y esta división puede hacerse leyendo secuencialmente los símbolos del mensaje de izquierda a derecha, cada vez que es leído el último símbolo de una palabra esta puede ser interpretada sin más demora, nunca tenemos incertidumbre acerca de si realmente hemos acabado de leer una palabra o sólo hemos leído el principio de otra más larga.

Hay un teorema que nos permite decir si, dado un conjunto arbitrario de longitudes, existe un código instantáneo con palabras de esas longitudes justamente. Se lo debemos a Kraft (el de la desigualdad) quién lo formuló en 1949. Pero antes de proceder con el teorema haremos algunas operaciones que serán necesarias para demostrarlo.

Retomaremos la desigualdad de Kraft recordando la equivalencia que mostramos en la expresión 1.3.8:

$$\sum_{k=1}^q \frac{1}{r^{\ell_k}} = \sum_{j=1}^m \frac{\alpha_j}{r^j}$$

Donde α_j es el número de palabras de longitud j en el código. Si multiplicamos y dividimos por r^m tenemos:

$$\sum_{k=1}^q \frac{1}{r^{\ell_k}} = \frac{1}{r^m} \sum_{j=1}^m \alpha_j r^{m-j}$$

Así que la desigualdad de Kraft es equivalente a:

$$\frac{1}{r^m} \sum_{j=1}^m \alpha_j r^{m-j} \leq 1$$

es decir:

$$\sum_{j=1}^m \alpha_j r^{m-j} \leq r^m$$

En la suma el último término es: $\alpha_m r^{m-m} = \alpha_m$, así que podemos escribir:

$$\alpha_m + \sum_{j=1}^{m-1} \alpha_j r^{m-j} \leq r^m \quad (1.3.18)$$

como un equivalente a la desigualdad de Kraft.

Teorema 1.9 *Existe un código instantáneo r -ario $C = \{\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_q\}$ con palabras de longitudes $\ell_1, \ell_2, \dots, \ell_q$ si y sólo si estas longitudes satisfacen la desigualdad de Kraft:*

$$\sum_{k=1}^q \frac{1}{r^{\ell_k}} \leq 1$$

Dem.: \Rightarrow Sea $\mathbf{c} \in C$ una palabra en un código instantáneo C , $\text{len}(\mathbf{c}) = j \leq m - 1$, dado que \mathbf{c} no debe ser prefijo de ninguna otra palabra en C entonces ninguna palabra de la forma $\mathbf{c}\mathbf{x}$ con $\text{len}(\mathbf{x}) = m - j$ puede estar en C . Dada \mathbf{c} hay exactamente r^{m-j} palabras de la forma $\mathbf{c}\mathbf{x}$ de longitud m que deben ser excluidas de C . Esto para cada posible elección de \mathbf{c} de longitud j en el código C . Hay α_j posibles elecciones de \mathbf{c} y por cada una r^{m-j} palabras de longitud m que deben excluirse del código. Así que, en total, hay:

$$\sum_{j=1}^{m-1} \alpha_j r^{m-j}$$

palabras de longitud m que no deben estar en C .

Las que sí están (α_m) mas las que no deben estar (la suma anterior) deben ser, a lo más, tantas como todas las posibles (r^m). Así que:

$$\alpha_m + \sum_{j=1}^{m-1} \alpha_j r^{m-j} \leq r^m$$

que es justo la expresión 1.3.18, equivalente a la desigualdad de Kraft.

\Leftarrow Por inducción sobre q .

- $q \leq r$.

Podemos elegir q símbolos (hay r disponibles) con los cuales comenzar q palabras de longitudes $\ell_1, \ell_2, \dots, \ell_q$, por lo tanto ninguna palabra es prefijo de otra y el código es instantáneo.

- *hipótesis: dadas las longitudes $\ell_1, \ell_2, \dots, \ell_q$. Si*

$$\sum_{j=1}^q \frac{1}{r^{\ell_j}} \leq 1$$

entonces existe un código instantáneo con palabras de esas longitudes precisamente.

Sean $\ell_1, \ell_2, \dots, \ell_{q+1}$ longitudes tales que: $\ell_1 \leq \ell_2 \leq \dots \leq \ell_{q+1}$ y que cumplen con:

$$\sum_{j=1}^{q+1} \frac{1}{r^{\ell_j}} \leq 1$$

entonces, dado que $\ell_{q+1} > 0$, si lo excluimos de la suma tenemos:

$$\sum_{j=1}^q \frac{1}{r^{\ell_j}} < 1$$

Tenemos ahora un conjunto de longitudes $\ell_1, \ell_2, \dots, \ell_q$ que satisfacen la desigualdad de Kraft, de hecho la satisfacen estrictamente, por hipótesis de inducción existe entonces un código instantáneo con esas longitudes. En ese código que ahora sabemos que existe habrá entonces α_1 palabras de longitud 1, α_2 palabras de longitud 2 y en general α_j palabras de longitud j y sabemos que se satisface:

$$\alpha_m + \sum_{j=1}^{m-1} \alpha_j r^{m-j} < r^m \quad (1.3.22)$$

con la desigualdad estricta, dado que se cumplía estrictamente la de Kraft.

El primer sumando es el número de cadenas de longitud m que están en el código, la suma que le sigue es el número total de cadenas de longitud m que no están en el código y que no deben estar porque se construyen usando como prefijo a otras y el lado derecho de la desigualdad es el número total de posibles cadenas de longitud m .

Dado que la desigualdad anterior es estricta entonces concluimos que falta al menos una palabra de longitud m que no está en el código pero que podría estar, porque en la suma contamos a todas las que no pueden estar. Así que incluimos esa palabra, que ya sabemos que existe y que tiene longitud m y ya tenemos un código con $q + 1$ palabras de longitudes $\ell_1, \ell_2, \dots, \ell_{q+1}$ instantáneo.

□

Ejemplo 1.12 Si se nos plantea la pregunta ¿existirá algún código instantáneo binario con cinco palabras de longitudes 3, 3, 2, 2, 2? podemos utilizar el teorema de Kraft.

$$\sum_{k=1}^q \frac{1}{r^{\ell_k}} = 2 \left(\frac{1}{2^3} \right) + 3 \left(\frac{1}{2^2} \right) = 4/4 = 1$$

la desigualdad de Kraft se cumple, de hecho se cumple con la igualdad estricta en este caso, así que sí existe un código binario instantáneo con las longitudes dadas. ◁

El hecho de que se cumpla la igualdad estricta significa que no es posible añadirle al código instantáneo una palabra más, cualquier añadidura ocasionaría que el código resultante ya no fuera instantáneo. En casos como el del ejemplo, en los que se cumple la igualdad estricta se dice que el código en cuestión es maximal.

Definición 1.11 Un código r -ario instantáneo C es *maximal* si no está contenido en otro código r -ario instantáneo de mayor tamaño.

Por supuesto “mayor tamaño” aquí, significa “con mayor número de palabras”.

1.4 Anotaciones finales

Es conveniente recapitular lo visto acerca de los teoremas de McMillan y Kraft.

El teorema de McMillan dice que si un código es unívocamente decodificable entonces cumple con la desigualdad de Kraft, es decir la desigualdad de Kraft es una condición necesaria para que un código sea unívocamente decodificable, pero no una condición suficiente. De allí que digamos que el teorema nos provee de un mecanismo para probar cuando un código no es unívocamente decodificable: si no cumple con algo que es necesario para serlo, no puede serlo.

El teorema de Kraft por su parte, nos dice que si damos un conjunto de longitudes que cumplan con la desigualdad de Kraft entonces existe un código instantáneo con palabras de esas longitudes precisamente y que si nos es dado un código instantáneo entonces debe cumplir con la desigualdad de Kraft. Pero no dice que si nos es dado un código que cumpla con la desigualdad entonces ese código es instantáneo, puede no ser precisamente ese, sólo sabremos que existe alguno con las mismas longitudes que el que nos dieron y que

si es instantáneo pero no dice nada respecto al que tenemos enfrente, para saber si ese precisamente es instantáneo habría que verificar que cumpla con la propiedad de no tener prefijos.

Codificación eficiente

2.1 Árbol de decodificación

El hecho de que en los códigos instantáneos, revisados en el capítulo anterior, sea posible leer secuencialmente un mensaje codificado de izquierda a derecha, símbolo a símbolo e irlo decodificando “al vuelo” nos hace pensar en algunas cosas. Probablemente a aquellos versados en la teoría de autómatas les recuerde una máquina de estados, conforme leemos símbolos de entrada nuestro estado cambia hasta llegar a uno en el que se ha terminado de reconocer una palabra de código y en ese instante se produce como símbolo de salida el símbolo del alfabeto de la fuente que le corresponde bajo el esquema de codificación usado.

Probablemente a algunos otros les haga pensar en un árbol de decisión. Cada vez que se recibe un símbolo en el alfabeto r -ario del código, se desciende desde un nodo de un árbol por una de sus r posibles ramas hasta otro nodo. El proceso de descenso continua hasta que se ha terminado de leer completamente una palabra de código, en ese momento el nodo

actual del proceso sería una hoja asociada a la palabra de código leída y que nos permite determinar el símbolo del alfabeto fuente que es codificado por ella.

Ambas visiones son igualmente correctas, pero para fines de demostraciones formales posiblemente sea un poco más útil el enfoque de árboles.

Cada nodo del árbol de decodificación tiene, a lo más, r hijos, donde r es la base del código. Cada arista de salida de un nodo tiene asociada como etiqueta un símbolo del alfabeto r -ario del código, si el símbolo leído del mensaje codificado es c entonces recorreremos la arista de salida del nodo actual etiquetada c , llegamos así a un nuevo nodo, hijo del anterior desde el que podemos repetir el proceso leyendo el siguiente símbolo del mensaje. Evidentemente al iniciar la lectura del mensaje el nodo inicial es la raíz del árbol r -ario de decodificación. Cada vez que se lee una palabra completa se llega a una hoja del árbol y antes de leer el siguiente símbolo se regresa a la raíz. Asociada con cada hoja hay una palabra de código y, usando la biyección de la función de codificación, también un símbolo del alfabeto fuente.

Ejemplo 2.1 El código $C = \{111, 110, 10, 011, 010, 00\}$ con la función de codificación:

$$\{f(A) = 111, f(B) = 110, f(C) = 10, f(D) = 011, f(E) = 010, f(F) = 00\}$$

posee el árbol de decodificación mostrado en 2.1. ◁

Asociado pues a todo código r -ario instantáneo tenemos un árbol de decodificación r -ario. Con esto en mente podemos empezar a pensar en cosas interesantes.

Qué tal si todas las palabras de un código instantáneo fueran del mismo tamaño. Si el número de palabras es q significa que las q hojas del árbol de decodificación asociado al código están a la misma profundidad en el árbol, para llegar a cualquiera de ellas desde la raíz hay que recorrer tantas aristas como la altura del árbol.

Ahora bien, en un árbol r -ario hay, en el primer nivel abajo de la raíz, r nodos, en el siguiente r^2 , r^3 en el siguiente y r^n en el n -ésimo nivel. Así que si todas las hojas están en un mismo nivel $r^n = q$ lo que significa que $n = \log_r(q)$. Además si todas las hojas están a profundidad n entonces las longitudes de las palabras de código asociadas a ellas son siempre n . ¿Como se ve la desigualdad de Kraft en este caso:

$$\sum_{k=1}^q \frac{1}{r^n} = \sum_{k=1}^q \frac{1}{r^{\log_r(q)}} = \sum_{k=1}^q \frac{1}{q} = q \left(\frac{1}{q} \right) = 1$$

bueno, ahora ya sabemos que si eso pasa estamos trabajando con un código maximal, la igualdad se cumple en la desigualdad de Kraft.

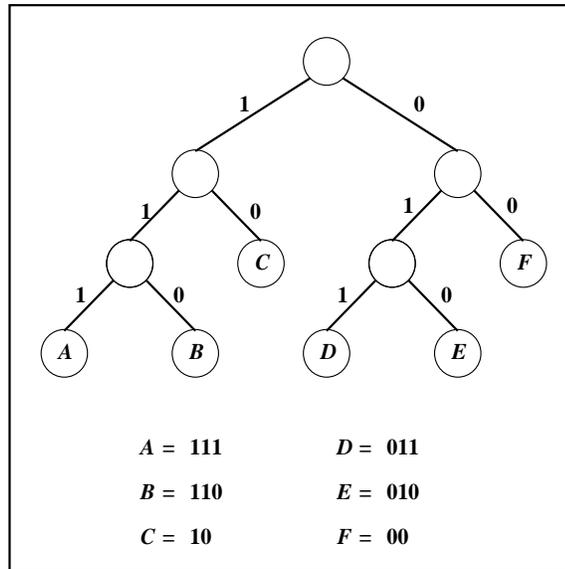


Figura 2.1: Árbol de decodificación para el código del ejemplo.

El ejercicio que acabamos de hacer no nos proporciona algo novedoso, pero sí novedoso punto de vista. Si nos topamos con un código r -ario instantáneo en el que todas las palabras sean de la misma longitud y tenga r^k palabras, para alguna k , entonces sabemos que debe ser maximal. Esto es, de hecho, un código de bloque o de longitud fija visto como un caso particular de código instantáneo.

En el capítulo anterior se presentó un ejemplo de código maximal de longitud variable instantáneo, sirva dicho ejemplo para dejar claro que no todos los códigos maximales tienen longitud fija y por tanto, sus árboles de decodificación tampoco tienen por qué tener todas sus hojas al mismo nivel.

2.2 Codificación de Shannon-Fano

El código de Shannon-Fano fue desarrollado independientemente por Claude Shannon, el padre de la teoría de la información, en Bell Labs y R. M. Fano en el MIT a finales de la década de 1940.

Es un código que busca la representación eficiente de los datos que codifica. Posee longitud variable y proporciona códigos más largos para los símbolos de la fuente menos probables y más cortos para los más probables. Por supuesto se requiere de conocer previamente la

distribución de probabilidad de los símbolos del alfabeto de la fuente.

Para obtener el código de Shannon-Fano de un alfabeto fuente dadas las probabilidades (o equivalentemente las frecuencias) de cada símbolo en él se procede como sigue:

1. Se ordena crecientemente¹ la lista de todos los símbolos del alfabeto fuente de acuerdo a su probabilidad.
2. Se selecciona un *punto de corte* que divida la lista en dos partes. El criterio para seleccionar el punto de corte es que la suma de las probabilidades de los símbolos en la sublista de un lado del corte sea lo más parecida posible a la suma de las probabilidades en la sublista del otro lado.
3. A cada uno de los símbolos en una de las sublistas se les asocia un cero, a los de la otra sublista un uno.
4. A cada sublista se le aplica recursivamente el mismo procedimiento descrito en los dos incisos anteriores.

Ejemplo 2.2 Supóngase que tenemos el alfabeto fuente $\{A, B, C, D, E, F\}$ con las siguientes frecuencias:

Sim.	Frec.
A	3
B	5
C	10
D	12
E	18
F	22

En la figura 2.2 se muestra el alfabeto ordenado (de hecho ya los estaba) y en las primeras tres columnas la diferencia entre la suma superior y la inferior, la suma inferior y la suma superior, respectivamente. Cuando la diferencia es menor (10 en el ejemplo) se elige ese como el punto de corte (señalado con una línea). Para las siguientes iteraciones ya no se muestran las sumas, sólo los puntos de corte. ◁

El algoritmo de Shannon-Fano se muestra a continuación escrito en pseudocódigo. En este algoritmo *idxini* e *idxfin* son, respectivamente el índice inicial y final de la partición

¹Adoptamos la convención de ordenar crecientemente, en realidad el orden puede ser creciente o decreciente.

	70	3	A	3	0 0 0 0
64	67	8	B	5	0 0 0 1
54	62	18	C	10	0 0 1
34	52	30	D	12	0 1
10	40	48	E	18	1 0
26	22	70	F	22	1 1

Figura 2.2: Obtención del código de Shannon-Fano para los símbolos del ejemplo.

a procesar de la lista original, evidentemente la primera vez que se llama al algoritmo son el índice inicial y el final de la lista completa. La función `suma` obtiene la suma de las frecuencias de los elementos en la lista entre los índices que recibe como argumentos. La función `arbolbin` construye un árbol binario de un solo nodo (la raíz) cuyo contenido es el argumento de la función. En la segunda llamada a `arbolbin` (línea 16) el argumento no importa (*cualquiersimb*). Las funciones `setSubIzq` y `setSubDer` establecen el subárbol izquierdo y derecho de la instancia de árbol binario que las llama, en este caso `arbol` es el identificador de dicha instancia. Hay que notar que en las líneas 17 y 18 se llama recursivamente al proceso de construcción, el `if` de la línea 1 establece la condición de terminación de la recursión. Al final el proceso regresa un árbol binario que constituye el árbol de decodificación de Shannon-Fano.

Es importante aclarar que el código de Shannon-Fano de un alfabeto dadas sus frecuencias, no es único. Basta con cambiar todos los ceros por unos, por ejemplo, para obtener otro esquema de codificación que, siendo de Shannon-Fano es diferente del que se obtuvo primero. Si hay dos símbolos con igual frecuencia basta con intercambiar sus códigos para obtener otra codificación de Shannon-Fano.

2.3 Codificación de Huffman

En 1952 Huffman publicó un método que, al igual que el de Shannon-Fano, busca representar los símbolos de un alfabeto fuente de acuerdo con sus frecuencias y también es

```
SHANNON-FANO(idxini, idxfin, lista)
1  if (idxini = idxfin) then
2    return (arbolbin(lista[idxini]))
3  else
4    idxpart = idxini
5    ssup = suma(lista, idxini, idxpart)
6    sinf = suma(lista, idxpart + 1, idxfin)
7    dif = | ssup - sinf |
8    difant = dif
9    while (difant ≥ dif) do
10   idxpart = idxpart + 1
11   ssup = suma(lista, idxini, idxpart)
12   sinf = suma(lista, idxpart + 1, idxfin)
13   difant = dif
14   dif = | ssup - sinf |
15  endwhile
16  arbol = arbolbin(cualquiersimb)
17  arbol.setSubIzq ( SHANNON-FANO (idxini, idxpart - 1, lista))
18  arbol.setSubDer ( SHANNON-FANO (idxpart, idxfin, lista))
19  return arbol
20 endif
21 end
```

Figura 2.3: Algoritmo de construcción del árbol de decodificación de Shannon-Fano.

instantáneo.

Al igual que en la codificación de Shannon-Fano se requiere una lista de todos los símbolos del alfabeto fuente con sus frecuencias. El procedimiento de codificación de Huffman es el siguiente:

1. Se ordena la lista crecientemente de acuerdo con la frecuencia.
2. Se crea un árbol con cada elemento de la lista de símbolos del alfabeto fuente. Evidentemente cada uno de estos árboles está constituido de un único nodo, la raíz.
3. Seleccionar a los dos símbolos de menor frecuencia (los dos primeros de la lista), eliminarlos de la lista y unir los árboles asociados a ellos a través de un nuevo nodo raíz.
4. Introducir un nuevo elemento a la lista, asociado con el árbol que se acaba de construir. El valor del símbolo no es relevante, la frecuencia debe ser la suma de las frecuencias de los símbolos que se unieron. Al introducir este nuevo elemento a la lista debe asegurarse que esta permanece ordenada.
5. Regresar al paso 3 hasta que en la lista haya un único elemento; el árbol asociado a este elemento es el árbol de decodificación de Huffman.

Ejemplo 2.3 Sea $A = \{a, b, c, d, e\}$ el alfabeto de una fuente con las frecuencias siguientes: $f(a) = 15$, $f(b) = 10$, $f(c) = 12$, $f(d) = 4$ y $f(e) = 6$. por conveniencia representaremos cada elemento en la lista como una pareja ordenada (*símbolo, frecuencia*). Así la lista ordenada es:

$$\{(d, 4), (e, 6), (b, 10), (c, 12), (a, 15)\}$$

En el primer paso seleccionaremos, como los dos símbolos menos frecuentes, a la d y a la e cuyas frecuencias sumadas son 10, así que los árboles (de hecho hojas) asociados a d y e se juntan en un solo árbol con d como hijo izquierdo y e como derecho, el peso de este nuevo árbol es de 10. La nueva lista ordenada es entonces:

$$\{(\delta_1, 10), (b, 10), (c, 12), (a, 15)\}$$

se ha introducido a δ_1 para denotar al nuevo símbolo, sin importancia, que se asocia al nuevo nodo.

En el siguiente paso se unen el árbol asociado a δ_1 y el árbol (de hecho otra vez una sola hoja) asociado a b . El resultado es un nuevo nodo etiquetado con δ_2 tal como se muestra

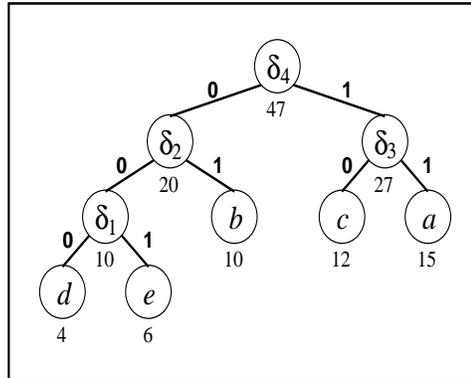


Figura 2.4: El árbol de decodificación de Huffman para el código del ejemplo. Bajo cada nodo se ha puesto la suma de las frecuencias del subárbol del que es raíz, cada nodo se ha etiquetado con un símbolo, en el caso de las hojas con un símbolo del alfabeto fuente, en el caso de los nodos interiores con un símbolo δ_i sin importancia. Las aristas de salida derechas se han asociado con “1” y las izquierdas con “0”.

en la figura 2.3. La lista ordenada es:

$$\{(c, 12), (a, 15), (\delta_2, 20)\}$$

En el siguiente paso se unirán c y a en un solo árbol cuya raíz, etiquetada δ_3 tiene un peso de 27. La lista resultante será:

$$\{(\delta_2, 20), (\delta_3, 27)\}$$

En el último paso se unen estos únicos dos símbolos para formar el árbol de decodificación que se muestra en la figura 2.3. El peso asociado a la raíz es de $20 + 27 = 47$. \triangleleft

El algoritmo de Huffman aparece en la figura 2.5 expresado en pseudocódigo. En él se está suponiendo que cada elemento de la lista es en realidad una estructura constituida por un árbol binario, al que denotaremos ab , y una frecuencia, a la que denotaremos $frec$. El atributo $frec$ contiene la suma de las frecuencias de aquellos símbolos que se encuentran en el atributo ab . La función (mejor dicho: método, en terminología orientada a objetos) $getMin$ se encarga de encontrar y regresar el elemento de la lista con la mínima frecuencia y $eliminaMin$ de eliminarlo; con vistas al análisis que haremos posteriormente, supondremos que el algoritmo recibe la lista, ordenada crecientemente respecto al atributo $frec$, en una estructura de datos óptima para llevar a cabo estas dos operaciones de manera óptima. Las

```
HUFFMAN(lista)
1  while (lista.longitud > 1) do
2      elem1 = lista.getMin
3      lista.eliminaMin
4      elem2 = lista.getMin
5      lista.eliminaMin
6      arbol = arbolbin(cualquiersim)
7      arbol.setSublq(elem1.getArbol)
8      arbol.setSubDer(elem2.getArbol)
9      elemnuevo.setArbol(arbol)
10     elemnuevo.setFrec(elem1.getFrec + elem2.getFrec)
11     lista.insertaOrden(elemnuevo)
12 endwhile
13 return (lista.getElem(0).getArbol)
14 end
```

Figura 2.5: Algoritmo de Huffman

funciones `setSublq` y `setSubDer` establecen al subárbol izquierdo y derecho, respectivamente, de la instancia de árbol que las llama, en el caso de la figura 2.5, el nombre de la instancia es *arbol*. `getArbol` regresa el atributo `ab` del elemento que llama a la función y `setArbol` establece dicho atributo, `getFrec` y `setFrec` hacen lo análogo para el atributo `frec`. `getElem` entrega el elemento de la lista indexado por el argumento de la función, supondremos que el índice inicial es el 0. `insertaOrden` inserta el elemento que recibe como argumento en la lista preservando el orden respecto al atributo `frec`.

Al igual que en el caso de la codificación de Shannon-Fano, el código de Huffman obtenido para un alfabeto dadas sus frecuencias, no es único, basta cambiar la convención *derecha* = 1, *izquierda* = 0 por la complementaria para tener otra codificación de Huffman.

Calculemos ahora la complejidad del algoritmo de Huffman mostrado en la figura 2.5 y que suponemos que recibe la lista de símbolos ya ordenada en una estructura de datos óptima para hacer búsquedas, un *heap* por ejemplo. Si denotamos con n el número de símbolos en el alfabeto fuente, en cada iteración solamente tenemos que buscar a los dos elementos de menor frecuencia, e insertar al símbolo resultante de “unirlos”, esto nos tomaría $O(3 \log(n)) = O(\log(n))$ operaciones de comparación. Esto lo hacemos en cada iteración y hay n iteraciones del ciclo, así que en síntesis la complejidad del algoritmo de Huffman es $O(n \log(n))$.

2.4 Longitud promedio de un código

Hemos presentado ya dos esquemas de codificación diferentes para representar datos eficientemente. Pero ¿cómo evaluamos qué tan bueno es un esquema? ¿habrá algún esquema mejor que otros? Para resolver estas preguntas debemos inventar una métrica que nos proporcione un criterio para decidir el grado de eficiencia de un código.

La información que poseemos es: la frecuencia de aparición de cada símbolo del alfabeto fuente y el código que le es asociado. Con base en las frecuencias podemos estimar la probabilidad de aparición de cada símbolo. Mientras mayor sea la cantidad de datos producidos por la fuente que han sido observados y considerados para contar las frecuencias, mayor será nuestra confianza de que esas frecuencias nos permiten estimar la distribución de probabilidades de los símbolos. Si en total hemos observado N símbolos producidos por una fuente y de ellos F_i han sido el símbolo i -ésimo entonces nuestro estimador para la probabilidad de aparición de i -ésimo es el cociente F_i/N al que denotaremos p_i . Debemos definir esto formalmente.

Definición 2.1 Una *fuerza de información* es una pareja ordenada $\mathcal{S} = (S, P)$ donde $S = \{s_1, s_2, \dots, s_q\}$ es el alfabeto de la fuente y P es una función de distribución de probabilidad $P : S \mapsto [0, 1]$ que asocia a cada símbolo s_i del alfabeto fuente una probabilidad $P(s_i) = p_i$.

Ahora podemos considerar la variable aleatoria *longitud del código asociado al símbolo producido por la fuente*. Cada símbolo posee un código y ese código una cierta longitud y ahora podemos calcular el valor esperado de la variable aleatoria mencionada como se define a continuación.

Definición 2.2 Sea $\mathcal{S} = (S, P)$ una fuerza de información y sea (C, f) un esquema de codificación para $S = \{s_1, s_2, \dots, s_q\}$. La *longitud promedio de palabra de código* de (C, f) es:

$$\text{AvgLen}(C, f) = \sum_{i=1}^q [P(s_i) \text{len}(f(s_i))] \quad (2.4.1)$$

Ejemplo 2.4 1. Sea $S = \{a, b, c, d\}$ con la siguiente distribución: $P(a) = 2/17$, $P(b) = 2/17$, $P(c) = 8/17$, $P(d) = 5/17$ (por supuesto la suma de las probabilidades es 1, requisito de toda distribución de probabilidades).

En el esquema (C_1, f_1) , donde C_1 es un código binario, definido como sigue: $f_1(a) = 11$, $f_1(b) = 0$, $f_1(c) = 100$, $f_1(d) = 1010$. La longitud promedio de palabra es:

$$\text{AvgLen}(C_1, f_1) = 2 \cdot \frac{2}{17} + 1 \cdot \frac{2}{17} + 3 \cdot \frac{8}{17} + 4 \cdot \frac{5}{17} = \frac{50}{17}$$

En el esquema (C_2, f_2) , donde C_2 es también un código binario, definido como sigue: $f_2(a) = 01010$, $f_2(b) = 00$, $f_2(c) = 10$, $f_2(d) = 11$. La longitud promedio de palabra es:

$$\text{AvgLen}(C_2, f_2) = 5 \cdot \frac{2}{17} + 2 \cdot \frac{2}{17} + 2 \cdot \frac{8}{17} + 2 \cdot \frac{5}{17} = \frac{40}{17}$$

Comparando el esquema (C_1, f_1) tiene una longitud promedio mayor que (C_2, f_2) , así que si codificáramos un mensaje cualquiera de la fuente, resultaría probablemente más corto si lo hiciéramos usando el segundo esquema de codificación que si lo hiciéramos con el primero. Hay que notar algo importante. Si codificáramos, por ejemplo, el mensaje $aa \dots a$ resultaría mejor usar el esquema 1, pero, dada la distribución de probabilidad de la fuente, este no es un mensaje representativo de la fuente. Nuestra afirmación de cuál esquema es mejor, más eficiente, se basa en la distribución de probabilidades, buscamos como codificar de la manera más eficiente posible aquellos mensajes que son más representativos de lo que suele producir la fuente.

2. Sea $S = \{A, B, C, D, E\}$ el alfabeto fuente y $\{15, 7, 6, 6, 5\}$ las frecuencias respectivas. Usaremos códigos binarios para codificar el alfabeto de esta fuente.

Usando el esquema de Shannon-Fano obtenemos: $SF(A) = 11$, $SF(B) = 10$, $SF(C) = 001$, $SF(D) = 01$, $SF(E) = 000$. Lo que nos da como resultado: $\text{AvgLen}(SF) = 2.28$

Usando el esquema de Huffman: $Huff(A) = 0$, $Huff(B) = 111$, $Huff(C) = 101$, $Huff(D) = 110$, $Huff(E) = 100$. Lo que nos da como resultado: $\text{AvgLen}(Huff) = 2.23$

Resulta que el código de Huffman es ligeramente mejor que el de Shannon-Fano. Esto siempre es verdadero como demostraremos más adelante. Sin embargo, en general, la diferencia entre las longitudes promedio de ambas codificaciones suele ser pequeña.

◁

2.5 La eficiencia del esquema de codificación de Huffman

Hay que notar una característica del esquema de Huffman dada por su construcción: *los símbolos del alfabeto S , están asociados biyectivamente a las hojas del árbol de Huffman. No hay símbolo sin hoja ni hoja sin símbolo.*

Demostraremos que no puede haber un esquema de codificación instantáneo más eficiente que los de Huffman. Para ello debemos primero demostrar un par de lemas que nos serán útiles.

Lema 2.1 Sea (C, f) un esquema de codificación instantáneo para una fuente $S = (S, P)$. Sean $\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_q$ las palabras de C asociadas a los símbolos con probabilidades: $p_1 \geq p_2 \geq \dots \geq p_q$ y longitudes: m_1, m_2, \dots, m_q respectivamente. Si el esquema (C, f) tiene longitud promedio de palabra mínima entonces $m_1 \leq m_2 \leq \dots \leq m_q$

Dem.: Por reducción al absurdo.

Supongamos que (C, f) es un esquema instantáneo de longitud promedio mínima y que $p_1 \geq p_2 \geq \dots \geq p_q$ pero no es cierto que $m_1 \leq m_2 \leq \dots \leq m_q$. Es decir, hay al menos una pareja (m_i, p_i) tal que $m_i > m_{i+1}$ y $p_i \geq p_{i+1}$.

La longitud promedio en este caso es:

$$\text{AvgLen}(C, f) = \sum_{j=1}^{i-1} p_j m_j + m_i p_i + m_{i+1} p_{i+1} + \sum_{j=i+2}^q p_j m_j$$

Definamos ahora un nuevo esquema de codificación (C, f') , idéntico a (C, f) salvo que intercambia \mathbf{c}_i con \mathbf{c}_{i+1} . En este caso:

$$\text{AvgLen}(C, f') = \sum_{j=1}^{i-1} p_j m_j + m_{i+1} p_i + m_i p_{i+1} + \sum_{j=i+2}^q p_j m_j$$

Si calculamos la diferencia:

$$\begin{aligned} \text{AvgLen}(C, f) - \text{AvgLen}(C, f') &= m_i p_i + m_{i+1} p_{i+1} - m_{i+1} p_i - m_i p_{i+1} \\ &= p_i (m_i - m_{i+1}) + p_{i+1} (m_{i+1} - m_i) \\ &= (p_i - p_{i+1})(m_i - m_{i+1}) \end{aligned}$$

Dado que $m_i > m_{i+1}$ y $p_i \geq p_{i+1}$ entonces:

$$\text{AvgLen}(C, f) - \text{AvgLen}(C, f') > 0$$

de donde $\text{AvgLen}(C, f) \geq \text{AvgLen}(C, f')$ por lo que (C, f) no es un esquema de longitud promedio mínima, lo que contradice nuestra suposición. \square

Lema 2.2 Sea (C, f) un esquema de codificación instantáneo para una fuente $S = (S, P)$. Sean $\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_q$ las palabras de C asociadas a los símbolos con probabilidades: p_1, p_2, \dots, p_q y longitudes: m_1, m_2, \dots, m_q respectivamente. Si el esquema (C, f) tiene longitud promedio de palabra mínima entonces $m_{q-1} = m_q$ y las palabras \mathbf{c}_{q-1} y \mathbf{c}_q difieren sólo en su último bit.

Dem.: Por reducción al absurdo.

Supóngase que el esquema (C, f) posee longitud promedio mínima de palabra con $C = \{\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_{q-1}, \mathbf{c}_q\}$ y $f(s_1) = \mathbf{c}_1, f(s_2) = \mathbf{c}_2, \dots, f(s_{q-1}) = \mathbf{c}_{q-1}, f(s_q) = \mathbf{c}_q$. Podemos suponer que $p_1 \geq p_2 \geq \dots \geq p_q$ y, por el lema anterior sabemos que entonces las longitudes están ordenadas: $m_1 \leq m_2 \leq \dots \leq m_{q-1} \leq m_q$.

Supóngase además que $m_{q-1} < m_q$. Consideremos la palabra \mathbf{c}_q en C y eliminemos su último bit, llamemos \mathbf{c}'_q a esta palabra cuya longitud es m_{q-1} .

Como C es instantáneo no existe ninguna palabra en C que sea igual a \mathbf{c}'_q porque de haberla sería prefijo de \mathbf{c}_q y el código no sería instantáneo. Además \mathbf{c}'_q es demasiado larga para ser prefijo de alguna otra palabra en C . Así que $C' = \{\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_{q-1}, \mathbf{c}'_q\}$ sigue siendo instantáneo.

Sea f' la función de codificación que asigna

$$f'(s_1) = \mathbf{c}_1, f'(s_2) = \mathbf{c}_2, \dots, f'(s_{q-1}) = \mathbf{c}_{q-1}, f'(s_q) = \mathbf{c}'_q$$

El esquema (C', f') es instantáneo y posee una longitud promedio de palabra menor que (C, f) dado que una de sus palabras, \mathbf{c}'_q , tiene una longitud menor. Así que (C, f) no tiene longitud promedio de palabra mínima, lo que contradice nuestra suposición inicial.

Ahora bien \mathbf{c}_{q-1} y \mathbf{c}_q son palabras de igual longitud y deben diferir en al menos uno de sus bits. Si ese bit no fuera el último de la palabra entonces podríamos eliminar el último bit de ambas palabras y se mantendrían diferentes. Obtendríamos entonces un código con dos palabras más cortas y por tanto una longitud promedio de palabra menor.

□

Ahora tenemos todo lo necesario para demostrar el teorema siguiente.

Teorema 2.1 *Sea S una fuente de información. Todos los esquemas de codificación de Huffman para S son instantáneos y tienen la longitud promedio de palabra mínima de entre todos los esquemas instantáneos.*

Dem.: Para demostrar que el esquema es instantáneo basta hacer notar que, por construcción del árbol de Huffman, no hay símbolo del alfabeto fuente que no tenga asociada una hoja en el árbol de Huffman y no hay hoja del árbol que no tenga asociado un símbolo de dicho alfabeto. Esto significa que siempre que nos encontramos un símbolo del alfabeto fuente en el árbol es porque hemos terminado en él un camino que desciende desde la raíz

hasta él y ya no podemos continuar más abajo en el árbol. No hay ningún camino que pase por un nodo del árbol asociado a un símbolo del alfabeto fuente y luego continúe hasta llegar a otro. Como los códigos están determinados por las aristas recorridas hasta llegar al nodo asociado al símbolo del alfabeto fuente, se concluye que no hay prefijos en el código y por lo tanto el código es instantáneo.

Para mostrar que la longitud promedio de palabra es mínima procederemos por inducción sobre q , el número de palabras en el código.

1. Nuestro caso base es $q = 2$.

Cuando se tienen sólo dos símbolos, como estos deben ser hojas de un árbol binario entonces el árbol está hecho de tres nodos, uno que constituye la raíz y dos hijos pendientes de él asociados a los dos símbolos del alfabeto de la fuente.

En este caso la longitud de palabra promedio es uno, que por supuesto, es menor o igual que cualquier otra.

2. *hipótesis: suponemos que para cualquier fuente con alfabeto de $q - 1$ símbolos un esquema de codificación de Huffman tiene longitud promedio mínima*

Sea (H, f) un esquema de codificación de Huffman para una fuente $\mathcal{S} = (S, P)$ con $S = \{s_1, s_2, \dots, s_{q-1}, s_q\}$ y $P = \{P(s_1) = p_1, P(s_2) = p_2, \dots, P(s_{q-1}) = p_{q-1}, P(s_q) = p_q\}$. Sean $\{\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_{q-1}, \mathbf{h}_q\}$ y $\{\ell_1, \ell_2, \dots, \ell_{q-1}, \ell_q\}$ las palabras de código y sus respectivas longitudes en el esquema de Huffman (H, f) . Establezcamos además que $f(s_i) = \mathbf{h}_i$. Sin pérdida de generalidad podemos suponer que: $p_1 \geq p_2 \geq \dots \geq p_{q-1} \geq p_q$.

Sea (C, g) un esquema de codificación instantáneo de longitud promedio mínima para la misma fuente $\mathcal{S} = (S, P)$. Sean $\{\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_{q-1}, \mathbf{c}_q\}$ y $\{m_1, m_2, \dots, m_{q-1}, m_q\}$ las palabras de código y sus respectivas longitudes en el esquema (C, g) . Análogamente $g(s_i) = \mathbf{c}_i$.

Por el lema 2.1 sabemos que: $m_1 \leq m_2 \leq \dots \leq m_{q-1} \leq m_q$ y por el lema 2.2 sabemos que, de hecho: $m_{q-1} = m_q$.

Sea \mathcal{S}' la fuente con el mismo alfabeto salvo que se han reemplazado los símbolos s_{q-1} y s_q por un único símbolo s con probabilidad $p_q + p_{q-1}$. Este nuevo alfabeto, \mathcal{S}' , tiene $q - 1$ símbolos.

En el primer paso del algoritmo de Huffman sobre \mathcal{S} se unían justamente s_q y s_{q-1} en un nuevo símbolo de probabilidad $p_q + p_{q-1}$. Luego de ese paso los árboles de Huffman para \mathcal{S} y para \mathcal{S}' son iguales, en el de \mathcal{S}' ese pequeño árbol de tres nodos es reemplazado por un solo nodo hoja asociado con el símbolo que hemos llamado s .

Si $\text{AvgLen}(H, f)$ denota la longitud de palabra promedio para la fuente original \mathcal{S} en el esquema de Huffman y $\text{AvgLen}(H', f')$ denota la longitud de palabra promedio para la fuente \mathcal{S}' en el esquema de Huffman que acabamos de construir, entonces:

$$\text{AvgLen}(H', f') = \text{AvgLen}(H, f) - (\ell_{q-1}p_{q-1} + \ell_q p_q) + (\ell_{q-1} - 1)(p_{q-1} + p_q)$$

Como los esquemas de Huffman son instantáneos, por el lema 2.2: $\ell_q = \ell_{q-1}$ así que:

$$\text{AvgLen}(H', f') = \text{AvgLen}(H, f) - (p_{q-1} + p_q) \quad (2.5.2)$$

Ahora consideremos el esquema instantáneo (C, g) de longitud promedio de palabra mínima, las últimas dos palabras de este son: $\mathbf{c}_q = x_1 x_2 \dots x_u 0$ y $\mathbf{c}_{q-1} = x_1 x_2 \dots x_u 1$, dado que es instantáneo

$\mathbf{c} = x_1 x_2 \dots x_u$ no es una palabra del código y podemos usarla entonces como palabra para el símbolo s en un esquema instantáneo (C', g') tal que $g'(s_i) = \mathbf{c}_i$ para toda $i \leq q-1$ y $g'(s) = \mathbf{c}$.

Entonces:

$$\begin{aligned} \text{AvgLen}(C', g') &= \text{AvgLen}(C, f) - (m_{q-1}p_{q-1} + m_q p_q) \\ &\quad + (m_{q-1} - 1)(p_{q-1} + p_q) \\ &= \text{AvgLen}(C, f) - (p_{q-1} + p_q) \end{aligned} \quad (2.5.3)$$

por hipótesis de inducción, dado que H' y C' tienen $q-1$ palabras:

$$\text{AvgLen}(H', f') \leq \text{AvgLen}(C', g')$$

Así que, sintetizando las expresiones 2.5.2 y 2.5.3:

$$\text{AvgLen}(H, f) \leq \text{AvgLen}(C, g)$$

□

Este teorema es importante, nos dice que, no importa cuál esquema de codificación instantáneo escojamos, siempre será, a lo más, tan bueno como un esquema de Huffman; no puede haber mejor.

Curiosamente en [8] podemos encontrar el siguiente ejemplo:

Ejemplo 2.5 En [8] se obtiene un código de Shannon-Fano y uno de Huffman para una fuente. Tanto los códigos obtenidos como la fuente misma se muestran aquí en la tabla 2.1.

Simb.	Prob.	Shannon-Fano	Huffman
x_1	0.10	001	101
x_2	0.05	0000	111
x_3	0.20	10	010
x_4	0.15	011	011
x_5	0.15	010	100
x_6	0.25	11	00
x_7	0.10	0001	110

Tabla 2.1: Códigos de Shannon-Fano y de Huffman mostrados en la tabla 2.19, pag. 110 de [8].

Con estos códigos se obtienen las siguientes longitudes promedio de palabra (pag. 109 [8]):

$$\text{AvgLen}(\textit{Shannon} - \textit{Fano}) = 2.7 \quad (2.5.4)$$

$$\text{AvgLen}(\textit{Huffman}) = 2.75 \quad (2.5.5)$$

¡La longitud promedio del código de Huffman es mayor que la del código de Shannon-Fano! ¿Qué ocurre? acabamos de demostrar que esto no puede ser.

En efecto algo está mal. Ambos promedios están bien calculados, no es un error numérico. El código de Shannon-Fano está bien construido, así que realmente la longitud del esquema de Shannon-Fano es 2.7. Pero el código de Huffman no está bien construido. El autor del libro cometió un error al pegar dos nodos en el paso 3 del proceso de construcción del árbol, uno de los símbolos que se “pegan” no tiene probabilidad mínima de entre los que quedaban en la lista de disponibles. Así que el código en la cuarta columna de la tabla 2.1, que aparece en la tabla 2.19 de [8], NO es un código de Huffman para la fuente usada como ejemplo. Un código de Huffman real para esa fuente aparece en la tabla 2.2, nótese que en este caso la longitud de palabra promedio es:

$$\text{AvgLen}(H) = 2.7$$

que coincide con la de Shannon-Fano y constituye el resultado esperado luego de demostrar el teorema 2.1.

No obstante el error operativo, el autor de [8] justifica el resultado cometiendo un error aún peor ([8], pag 110): “Aunque el código de Shannon-Fano es más eficiente dado que su longitud promedio es menor que la del código de Huffman, el lector debe notar que no

Simb.	Prob.	H
x_1	0.10	1111
x_2	0.05	1110
x_3	0.20	00
x_4	0.15	011
x_5	0.15	110
x_6	0.25	10
x_7	0.10	010

Tabla 2.2: Códigos de Huffman correcto para la fuente de la tabla 2.1. La longitud promedio de palabra es 2.7, igual a la de Shannon-Fano.

necesariamente es siempre más eficiente”².

◁

² Although the Shannon-Fano code is more efficient since its average code length is less than that of the Huffman code, the reader should note that it is not necessarily always more efficient.

Información y entropía

En el capítulo pasado vimos que es posible representar de manera breve una cierta cantidad de datos si sabemos que tan probable es que que aparezca cada uno de los símbolos posibles que aparecen en los datos. Para lograrlo representábamos con palabras más larga a los símbolos menos probables y con palabras más cortas a los más probables. En este capítulo nos avocaremos a determinar hasta donde podemos llegar, que tan breve puede ser la representación y para ello requeriremos de un par de conceptos fundamentales: cantidad de información y entropía de una fuente.

3.1 Cantidad de información

Seguramente todos tenemos una vaga idea intuitiva de lo que es cantidad de información. En nuestro contexto específico, si una palabra de código es muy larga intuimos que “tiene mucha información” y si es corta “tiene poca información”. Como hemos dicho que las

palabras largas están asociadas a símbolos poco probables entonces lo que estamos diciendo es que los símbolos menos probables son los que tienen más información y los más probables tienen menos. En este sentido la cantidad de información es algo así como una medida de la “sorpresa” que nos causa ver un símbolo cuando aparece producido por una fuente, a mayor sorpresa mayor información.

Una manera más sencilla de llegar al concepto de información y en la que también podemos apelar a nuestra idea intuitiva es esta: Supongamos que un amigo acaba de comprar una mascota y debemos adivinar que animal compro haciéndole preguntas cuya respuesta solo pueda ser un “sí” o un “no”. Hay muchos animales probables, pero unos son más comunes como mascotas que otros. Generalmente las personas preferimos a los mamíferos, son más cálidos e inteligentes, más próximos a nosotros. Así que podríamos comenzar preguntando ¿es un mamífero? si contesta que sí, podemos preguntar ahora si es de más de 20 centímetros de longitud, si la respuesta es si entonces podemos preguntar ya directamente si es un perro, si la respuesta a esta última pregunta es no, podemos preguntar si se trata de un gato; pero si la respuesta respecto al tamaño es no entonces estamos en problemas: muy probablemente escogió algún roedor y hay muchas posibilidades: hamster, ratón, cuyo, hurón, conejo, tendremos que hacer varias preguntas para determinar cual es ellos es. Si la respuesta a ¿es un mamífero? fue no, entonces también estamos en problemas puede ser un reptil, un anfibio, un pez, un arácnido, un ave y dentro de cada una de estas categorías hay muchas posibilidades, como podrá percatarse quien visite una tienda de mascotas y pueda apreciar la amplia gama de exóticos ejemplares que algunas pocas personas gustan de tener en casa. Esa es la clave del problema, en general sólo algunas pocas personas eligen como mascota una iguana o una tarántula y casi nadie tendrá un pitón albino de Madagascar. Si nuestro amigo tiene gustos extraños tendremos que hacerle más preguntas, tantas más cuanto más extraña sea su mascota, necesitamos que nos dé mucha información, que responda muchas preguntas, si eligió un dragón de Komodo y solo dos o tres si compró un pastor alemán. Para determinar las cosas más frecuentes se necesita menos información que para determinar las más raras.

Dijimos que en el juego de adivinanza nuestro amigo solo podía contestar “sí” o “no” a nuestras preguntas, el proceso adivinatorio se puede ver entonces como un árbol binario de decisión, cada vez que se responde una pregunta elegimos una de dos posibles rutas para formular la siguiente. Este árbol, por cierto, es análogo a nuestros árboles de decodificación del capítulo pasado para códigos binarios. En las hojas del árbol están las posibles respuestas, tanto más profundas cuanto más preguntas haya que hacer para llegar a ellas. Esto nos hace pensar en que es posible medir la cantidad de información necesaria para determinar la respuesta en bits: cuantos “sí” (1) y “no” (0) se requieren para llegar desde la raíz a una hoja. Podemos también pensar en definir una función que, dada una hoja o posible respuesta, nos diga cuantas preguntas hay que hacer para determinarla.

Ahora bien, si nuestro amigo compró una mascota para él y otra para su novia y queremos adivinar ambas, debemos hacer todas las preguntas para determinar una de ellas y luego volver a empezar para determinar la otra. El saber qué compró para él no nos ayuda, en general, a saber qué compró para su novia. Así que la cantidad de preguntas hechas en total es la suma de las hechas para determinar su mascota y las hechas para determinar la de su novia.

Es tiempo de ponernos formales. Queremos una función I que nos permita medir la cantidad de información de cada símbolo (s) de un alfabeto fuente dependiendo de su probabilidad (p). Esta función debe cumplir con los siguientes requisitos:

- $I(p) \geq 0$ para cualquier símbolo s del alfabeto fuente con probabilidad p .
- I es continua. Queremos que la cantidad de información de símbolos cuya probabilidad es muy similar sea también muy similar.
- $I(p_i p_j) = I(p_i) + I(p_j)$. La cantidad de información que se requiere para determinar que han ocurrido los símbolos de probabilidad p_i y p_j (que son producidos independientemente) es la necesaria para determinar que ha ocurrido p_i más la necesaria para determinar que ha ocurrido p_j , lo que decíamos para adivinar la mascota de nuestro amigo y la de su novia. Si la producción de un símbolo por parte de una fuente, es un evento independiente, entonces la probabilidad de que ocurra s_i (que es producido con probabilidad p_i) y luego ocurra s_j (que es producido con probabilidad p_j), es el producto de sus probabilidades $p_i p_j$.

Llama la atención que la función de información transforme un producto en una suma. Eso solo lo hace la función logaritmo, que además resulta ser continua. Parece un buen candidato, analicemos: si un símbolo es seguro, si siempre es producido por la fuente y ningún otro es producido, ese símbolo tiene probabilidad 1, no nos produce sorpresa alguna verlo aparecer, así que no se requiere nada de información para determinarlo. Si nuestro amigo nos informa de antemano que compró un perro no hay que preguntar nada. Así que la cantidad de información asociada a la probabilidad 1 es cero. En cambio si un símbolo es muy poco probable tiene mucha información, tanta más cuanto más cercana a cero sea su probabilidad. Queremos una función como la que se muestra en la figura 3.1.

Definición 3.1 La *cantidad de información* $I(p)$ obtenida de un símbolo fuente s con probabilidad $p > 0$, medida en bits de información, es:

$$I(p) = \log_2 \left(\frac{1}{p} \right)$$

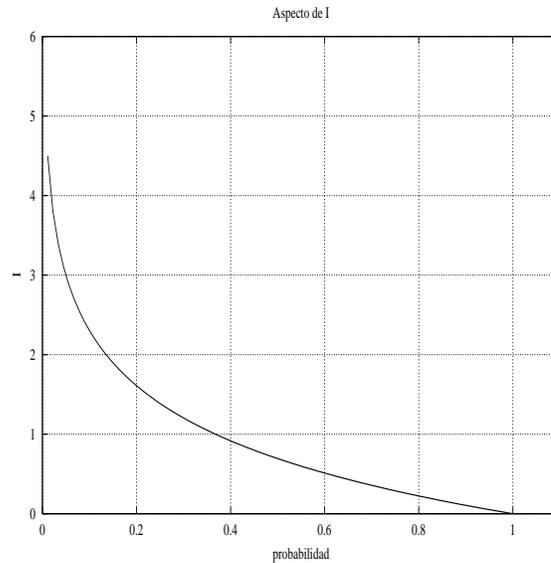


Figura 3.1: Aspecto que debe exhibir la función I (cantidad de información) en función de la probabilidad p (eje de las abscisas).

Las unidades de medida son bits de información, de allí la base (2) del logaritmo. Esto tiene que ver con las dos posibles respuestas a las preguntas que mencionamos arriba (sí o no, 1 o 0). La base es arbitraria y de hecho no tiene por que ser un valor entero, si usamos e por ejemplo estaremos midiéndola en *nepers* de información, dado que la base de los logaritmos neperianos o naturales es justamente e .

De hecho, si en algún momento requerimos expresar la cantidad de información en base b y solo poseemos el dato en base r podemos hacerlo traduciendo la expresión del logaritmo. Recordemos que un número w es $w = \log_b(x)$ si y sólo si $b^w = x$ de donde: $w \log_r(b) = \log_r(x)$ lo que significa que:

$$w = \frac{\log_r(x)}{\log_r(b)} = \log_b(x) \quad (3.1.1)$$

Ejemplo 3.1 En una imagen de $1024 \times 768 \times 256$ hay 256 posibilidades en cada uno de los 1024×768 píxeles. $256 = 2^8$ así que el número de posibles imágenes es: $N = 2^{(8)(1024)(768)} = 2^{6,291,456}$. Si cada imagen es igualmente probable: $p = 1/N$. Entonces la cantidad de información en una imagen es:

$$I(p) = 6,291,456 \text{ bits}$$

◁

3.2 Entropía de información

Si ahora pensamos en todos los posibles animales de la tienda de mascotas y en la cantidad de preguntas de respuesta binaria que hay que hacer para determinar cada animal nos encontraremos con que la variabilidad es enorme. Desde dos o tres preguntas para determinar a un perro hasta más de una decena para determinar un reptil extraño. Pero podemos hacer estadística y obtener una visión general del problema de determinar una mascota. Podemos obtener una medida que nos indique cuanta información se requiere en promedio para adivinar.

Definición 3.2 Sea $\mathcal{S} = (S, P)$ una fuente de información con distribución de probabilidad $P = \{p_1, p_2, \dots, p_q\}$. La *entropía de información* de \mathcal{S} es el valor esperado de la cantidad de información de los símbolos de \mathcal{S} :

$$H(\mathcal{S}) = \sum_{i=1}^q \left[p_i \log_2 \left(\frac{1}{p_i} \right) \right] = - \sum_{i=1}^q [p_i \log_2 (p_i)]$$

Con la convención de que si $p_i = 0$ entonces $p_i \log_2 (1/p_i) = 0$.

Ejemplo 3.2 1. Sea \mathcal{S} una fuente con alfabeto $S = \{s_1, s_2, \dots, s_q\}$ donde cada símbolo es equiprobable, es decir $p_i = 1/q$. En este caso la entropía es:

$$H(\mathcal{S}) = \frac{1}{q} \sum_{i=1}^q \log_2 (q) = \log_2 (q)$$

2. Sea \mathcal{S} una fuente con alfabeto $S = \{s_1, s_2, \dots, s_q\}$ donde $p_1 = 1$ y $p_i = 0$ para toda $i \in \{2, \dots, q\}$. En este caso la entropía es:

$$H(\mathcal{S}) = p_1 \log_2 \left(\frac{1}{p_1} \right) = 0$$

◁

En los ejemplos presentamos los dos casos extremos. En el primero tenemos una fuente totalmente incierta, no podemos apostar por ningún símbolo, todos pueden aparecer con igual probabilidad, La fuente es totalmente aleatoria. En este caso la entropía es máxima.

En el segundo ejemplo tenemos una fuente que solo produce un símbolo, salvo s_1 , el resto del alfabeto, para todo fin práctico, es como si no existiera. En todo momento sabemos con certeza que es lo que va a producir la fuente. En este caso la entropía es mínima.

Con estos dos ejemplos en mente podemos observar la analogía que existe entre el concepto de entropía en la teoría de la información y el concepto homónimo en física, ambos son una medida del desorden, de la desorganización de un sistema. En un sistema de partículas altamente desorganizado, con todas las partículas moviéndose aleatoriamente a distintas velocidades y en distintas direcciones la entropía es alta. En una fuente que produce símbolos aleatoriamente, sin ninguna estructura, la entropía de información es alta.

3.3 Propiedades de la entropía

Recordemos de nuestros cursos de cálculo que, para $x \geq 0$: $xe \leq e^x$ de donde: $x \leq e^x/e = e^{x-1}$. Finalmente:

$$\ln(x) \leq x - 1$$

De esto y la expresión 3.1.1 tenemos:

$$\log_2(x) \leq \frac{x-1}{\ln(2)} \quad (3.3.2)$$

la igualdad solo ocurre cuando $x = 1$.

Con esto en mente podemos demostrar el siguiente lema

Lema 3.1 *Sea $P = \{p_1, p_2, \dots, p_q\}$ una distribución de probabilidad. Si $R = \{r_1, r_2, \dots, r_q\}$ es un conjunto de números tales que:*

1. $0 \leq r_i \leq 1$ para toda i .

2.

$$\sum_{i=1}^q r_i \leq 1 \quad (3.3.3)$$

entonces

$$\sum_{i=1}^q p_i \log_2 \left(\frac{1}{p_i} \right) \leq \sum_{i=1}^q p_i \log_2 \left(\frac{1}{r_i} \right)$$

donde la igualdad solo se cumple si y sólo si $p_i = r_i$ para toda i .

Dem.: Por la expresión 3.3.2 tenemos que:

$$\begin{aligned}
 \sum_{i=1}^q p_i \log_2 \left(\frac{r_i}{p_i} \right) &\leq \sum_{i=1}^q p_i \frac{r_i - 1}{\ln(2)} \\
 &= \frac{1}{\ln(2)} \sum_{i=1}^q p_i \left(\frac{r_i}{p_i} - 1 \right) \\
 &= \frac{1}{\ln(2)} \sum_{i=1}^q (r_i - p_i) \\
 &= \frac{1}{\ln(2)} \left(\sum_{i=1}^q r_i - \sum_{i=1}^q p_i \right) \\
 &= \frac{1}{\ln(2)} \left(\sum_{i=1}^q r_i - 1 \right)
 \end{aligned}$$

Por la expresión 3.3.3, que es parte de nuestra hipótesis:

$$\frac{1}{\ln(2)} \left(\sum_{i=1}^q r_i - 1 \right) \leq 0$$

así que:

$$\sum_{i=1}^q p_i \log_2 \left(\frac{r_i}{p_i} \right) \leq 0 \quad (3.3.4)$$

Pero:

$$\begin{aligned}
 \log_2 \left(\frac{r_i}{p_i} \right) &= \log_2 (r_i) - \log_2 (p_i) \\
 &= -\log_2 \left(\frac{1}{r_i} \right) + \log_2 \left(\frac{1}{p_i} \right)
 \end{aligned}$$

así que la expresión 3.3.4 la podemos reescribir como:

$$\sum_{i=1}^q p_i \log_2 \left(\frac{1}{p_i} \right) - \sum_{i=1}^q p_i \log_2 \left(\frac{1}{r_i} \right) \leq 0$$

de donde finalmente:

$$\sum_{i=1}^q p_i \log_2 \left(\frac{1}{p_i} \right) \leq \sum_{i=1}^q p_i \log_2 \left(\frac{1}{r_i} \right)$$

la igualdad ocurre si $r_i/p_i = 1$ lo que significa que $r_i = p_i$.

□

Ahora estamos en condiciones de probar formalmente los extremos de la entropía que ya presentamos antes.

Teorema 3.1 Para una fuente $S = (S, P)$ en la que el alfabeto S posee q símbolos, la entropía $H(S)$ satisface:

$$0 \leq H(S) \leq \log_2(q)$$

Dem.: Sea $P = \{p_1, p_2, \dots, p_q\}$ la distribución de probabilidades de una fuente S y sea $R = \{1/q, 1/q, \dots, 1/q\}$ la distribución uniforme para q símbolos.

Aplicando el lema anterior a P y R obtenemos:

$$\begin{aligned} H(S) &= \sum_{i=1}^q p_i \log_2 \left(\frac{1}{p_i} \right) \\ &\leq \sum_{i=1}^q p_i \log_2 \left(\frac{1}{1/q} \right) \\ &= \sum_{i=1}^q p_i \log_2(q) \\ &= \log_2(q) \sum_{i=1}^q p_i \\ &= \log_2(q) \end{aligned}$$

Así que $H(S) \leq \log_2(q)$ y la igualdad ocurre cuando $p_i = 1/q$ para toda i .

□

El teorema confirma nuestra observación de que cuando la distribución de los símbolos de una fuente es uniforme la entropía es máxima. Por ejemplo, si una fuente es binaria, es decir solo puede producir dos símbolos, digamos 0 o 1, con probabilidad p y $1-p$ respectivamente, entonces el comportamiento de la entropía en función de p es el que se muestra en la figura 3.2. En este caso la entropía es:

$$H(p) = p \log_2 \left(\frac{1}{p} \right) + (1-p) \log_2 \left(\frac{1}{1-p} \right) \quad (3.3.6)$$

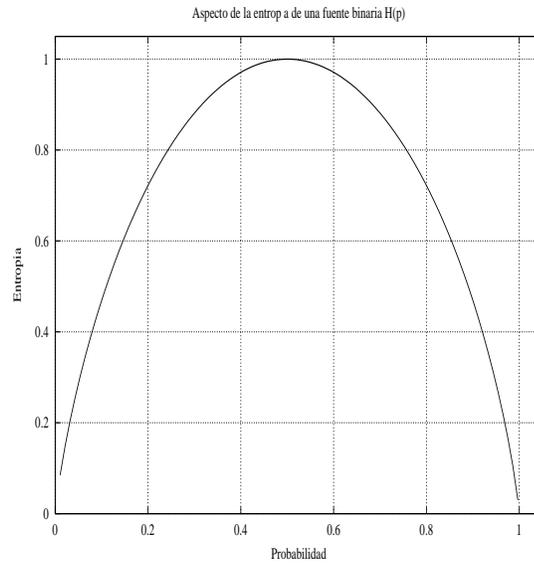


Figura 3.2: Aspecto de la entropía de una fuente binaria (con alfabeto $S = \{0, 1\}$) en la que $P(0) = p$ y $P(1) = 1 - p$. En el eje de las abscisas se ha puesto la variable p .

Recordemos que si u es una función de x entonces:

$$\frac{d \log(u)}{dx} = \frac{1}{u} \frac{du}{dx}$$

Así que en nuestro caso, derivando 3.3.6 obtenemos:

$$H'(p) = \log_2 \left(\frac{1}{p} \right) - \log_2 \left(\frac{1}{1-p} \right)$$

claramente la derivada es cero cuando $p = 1 - p = 1/2$, nótese además que tiende a infinito cuando p se acerca a cero o cuando se acerca a uno, lo que confirma el aspecto exhibido en la figura 3.2.

3.4 Extensiones de una fuente

Supongamos que tenemos una fuente $\mathcal{S} = (S, P)$ donde $S = a, b$ y $P(a) = 0.8$, $P(b) = 0.2$. El esquema de codificación de Huffman para esta fuente sería: $f(a) = 0$, $f(b) = 1$, dado que solo hay dos símbolos. La longitud promedio de este código es, evidentemente, 1.

Que tal si ahora nos fijamos no en los símbolos que produce la fuente, sino en todas las parejas de símbolos que puede producir: aa , ab , ba y bb . ¿Con qué probabilidad es producida cada pareja? dado que la producción de cada símbolo es un evento independiente, la probabilidad de producir cada pareja es el producto de las probabilidades de producir cada uno de sus símbolos. Es decir:

$$P'(aa) = 0.8 \cdot 0.8 = 0.64$$

$$P'(ab) = 0.8 \cdot 0.2 = 0.16$$

$$P'(ba) = 0.2 \cdot 0.8 = 0.16$$

$$P'(bb) = 0.2 \cdot 0.2 = 0.04$$

Sea $\mathcal{S}' = (S', P')$ una nueva fuente en la que $S' = \{aa, ab, ba, bb\}$ y P' es la distribución que aparece listada arriba. En este caso un esquema de codificación de Huffman nos entrega:

$$f'(aa) = 1$$

$$f'(ab) = 01$$

$$f'(ba) = 001$$

$$f'(bb) = 000$$

La longitud promedio de esta codificación es 1.56, pero en cada palabra de código se codifican dos símbolos de la fuente original así que la longitud promedio de palabra para representar los símbolos de la fuente original es $1.56/2 = 0.78$ lo que resulta menor que 1. Esto representa una mejora sobre el esquema de codificación de Huffman de la fuente original.

Analizaremos esto con cuidado, pero antes necesitamos algo de notación y definiciones.

Definición 3.3 Sea $\mathcal{S} = (S, P)$ una fuente de información. La n -ésima extensión de \mathcal{S} es la fuente $\mathcal{S}^n = (S^n, P^n)$ donde S^n es el conjunto de todas las cadenas de longitud n sobre S y P^n es la distribución de probabilidades definida como sigue: sea s una cadena cualquiera de S^n , $s = s_{i_1}s_{i_2} \dots s_{i_n}$, entonces $P^n(s) = p_{i_1}p_{i_2} \dots p_{i_n}$

Sean $S = \{s_1, s_2\}$ un alfabeto de símbolos y $P(s_1) = p_1$, $P(s_2) = p_2$ una distribución para una fuente $\mathcal{S} = (S, P)$. La segunda extensión de \mathcal{S} tiene el alfabeto $S^2 = \{s_1s_1, s_1s_2, s_2s_1, s_2s_2\}$ la distribución de la extensión es $P^2 = \{p_1p_1, p_1p_2, p_2p_1, p_2p_2\}$.

La entropía de esta fuente es:

$$\begin{aligned}
H(\mathcal{S}^2) &= p_1 p_1 \log_2 \left(\frac{1}{p_1 p_1} \right) + p_1 p_2 \log_2 \left(\frac{1}{p_1 p_2} \right) + \\
&\quad p_2 p_1 \log_2 \left(\frac{1}{p_2 p_1} \right) + p_2 p_2 \log_2 \left(\frac{1}{p_2 p_2} \right) \\
&= \left(p_1 p_1 \log_2 \left(\frac{1}{p_1} \right) + p_1 p_1 \log_2 \left(\frac{1}{p_1} \right) \right) + \\
&\quad \left(p_1 p_2 \log_2 \left(\frac{1}{p_1} \right) + p_1 p_2 \log_2 \left(\frac{1}{p_2} \right) \right) + \\
&\quad \left(p_2 p_1 \log_2 \left(\frac{1}{p_2} \right) + p_2 p_1 \log_2 \left(\frac{1}{p_1} \right) \right) + \\
&\quad \left(p_2 p_2 \log_2 \left(\frac{1}{p_2} \right) + p_2 p_2 \log_2 \left(\frac{1}{p_2} \right) \right) \\
&= \log_2 \left(\frac{1}{p_1} \right) (p_1 p_1 + p_1 p_1 + p_1 p_2 + p_2 p_1) + \\
&\quad \log_2 \left(\frac{1}{p_2} \right) (p_1 p_2 + p_2 p_1 + p_2 p_2 + p_2 p_2) \tag{3.4.8} \\
&= \log_2 \left(\frac{1}{p_1} \right) p_1 (p_1 + p_1 + p_2 + p_2) + \\
&\quad \log_2 \left(\frac{1}{p_2} \right) p_2 (p_1 + p_1 + p_2 + p_2) + \\
&= 2 p_1 \log_2 \left(\frac{1}{p_1} \right) + 2 p_2 \log_2 \left(\frac{1}{p_2} \right) \\
&= 2H(\mathcal{S})
\end{aligned}$$

De hecho esto ocurre en general para cualquier extensión.

Teorema 3.2 *Sea \mathcal{S} una fuente de información y sea \mathcal{S}^n su n -ésima extensión. Entonces:*

$$H(\mathcal{S}^n) = n H(\mathcal{S})$$

Dem.:

$$\begin{aligned}
 H(\mathcal{S}^n) &= \sum_{i_1, i_2, \dots, i_n} p_{i_1} p_{i_2} \dots p_{i_n} \log_2 \left(\frac{1}{p_{i_1} p_{i_2} \dots p_{i_n}} \right) \\
 &= \sum_{i_1, i_2, \dots, i_n} p_{i_1} p_{i_2} \dots p_{i_n} \log_2 \left(\frac{1}{p_{i_1}} \right) + \dots \\
 &+ \sum_{i_1, i_2, \dots, i_n} p_{i_1} p_{i_2} \dots p_{i_n} \log_2 \left(\frac{1}{p_{i_n}} \right)
 \end{aligned}$$

Hemos seguido el mismo procedimiento que llevamos a cabo para nuestro análisis de la segunda extensión de una fuente. Nuestra última expresión corresponde a la expresión 3.4.8.

Tal como hicimos en el análisis de la segunda extensión, el primer sumando se puede escribir como:

$$\sum_{i_1=1}^q p_{i_1} \log_2 \left(\frac{1}{p_{i_1}} \right) \times \sum_{i_2=1}^q p_{i_2} \times \dots \times \sum_{i_n=1}^q p_{i_n}$$

Como la suma de las p 's es 1, entonces este primer sumando es: $H(\mathcal{S})$.

Análogamente para cada sumando se tiene el mismo resultado. Así que:

$$H(\mathcal{S}^n) = n H(\mathcal{S})$$

□

3.5 El primer teorema de Shannon

¿Qué tenemos hasta ahora? Hemos definido algo llamado entropía, que es una propiedad de la fuente de información y que nos mide la *cantidad de información promedio de los símbolos de la fuente*. Si la fuente es totalmente aleatoria, si produce cualquiera de los q símbolos de su alfabeto con igual probabilidad, entonces su entropía es máxima y es $\log_2(q)$.

Por otra parte ya definimos una medida de la eficiencia de un código para representar los símbolos producidos por una fuente de información. La longitud promedio de un código binario es la *cantidad promedio de bits necesarios para decir una palabra de código, es decir, un símbolo del alfabeto fuente*.

Ahora debe parecernos que estos conceptos están relacionados. Ambos son *medidas de información promedio de los símbolos de la fuente*. La entropía mide la información

promedio que existe intrínsecamente en los símbolos de la fuente de información. La longitud promedio de código el promedio de información necesaria para decir cada uno de esos símbolos. Si imaginamos que tenemos el mejor código posible, el más eficiente, para codificar una cierta fuente, entonces es claro que nuestro código tendría que tener una cantidad de información promedio *para decir* los símbolos de la fuente igual o, en el peor de los casos, ligeramente superior, que la cantidad de información *intrínseca* en los símbolos de la fuente.

Hay que notar que aunque son conceptos relacionados son, por supuesto, diferentes. La entropía es una *propiedad de la fuente*, es inherente a ella. La longitud promedio mínima de palabra de un código es una *propiedad del esquema de codificación*. Si concebimos a nuestra fuente como una caja negra que produce símbolos por un extremo y a nuestra función de codificación como otra caja que se conecta por un extremo a la anterior, recibe cada símbolo que la fuente produce y por cada uno entrega una palabra de código. Entonces la entropía es algo que se le puede medir a la primera caja y la longitud promedio de palabra es algo que se le puede medir a la segunda. Si la segunda caja es muy eficiente en su codificación, entonces la medida de su longitud promedio de palabra de código deberá ser muy cercana a la medida de la entropía de la primera. Pero la segunda caja nunca puede tener una longitud promedio de palabra menor que la medida de entropía de la primera. No es posible decir con menos bits de información lo que la fuente produce.

Con esto en mente podemos plantear el siguiente teorema. Se lo debemos a Claude Shannon (1948) y fue llamado por él “el teorema fundamental para un canal sin ruido”. El nombre dado por Shannon significa que el teorema es verdadero cuando podemos ver exactamente lo que nuestra caja de codificación produce, sin alteración alguna ocasionada por el medio ambiente, suponemos que no ocurren errores de los que tengamos que preocuparnos. Más adelante estudiaremos el caso de canales de comunicación donde algo es introducido en un extremo y otra cosa es recibida en el otro extremo, canales en los que pueden ocurrir alteraciones aleatorias (errores) en la transmisión de datos.

En el teorema $H_r(\mathcal{S})$ denota la entropía de la fuente \mathcal{S} calculada en base r , es decir usando el logaritmo base r . Nosotros solemos escribir $H(\mathcal{S})$ para denotar lo que realmente es $H_2(\mathcal{S})$, la entropía binaria. Por supuesto, dada la expresión 3.1.1, podemos escribir:

$$H_r(\mathcal{S}) = \frac{H_2(\mathcal{S})}{\log_2(r)} \quad (3.5.9)$$

Teorema 3.3 (de la codificación sin ruido, versión 1). *Sea \mathcal{S} una fuente de información. Si $\text{MinAvgLen}_r(\mathcal{S})$ denota la mínima longitud promedio de palabra sobre todos los esquemas unívocamente decodificables r -arios para \mathcal{S} entonces:*

$$H_r(\mathcal{S}) \leq \text{MinAvgLen}_r(\mathcal{S})$$

Dem.: Sea $P = \{p_1, p_2, \dots, p_q\}$ la distribución de la fuente \mathcal{S} . Sea (C, f) un esquema de codificación r -ario unívocamente decodificable para \mathcal{S} con longitudes de palabra $\ell_1, \ell_2, \dots, \ell_q$ en correspondencia a las probabilidades de P .

Sea $r_i = 1/r^{\ell_i}$. Evidentemente $0 \leq r_i \leq 1$. Como C es unívocamente decodificable debe cumplir con el teorema de McMillan, así que:

$$\sum_{i=1}^q r_i = \sum_{i=1}^q \frac{1}{r^{\ell_i}} \leq 1$$

Se cumplen las hipótesis del lema 3.1 así que debe ocurrir que:

$$\begin{aligned} H(\mathcal{S}) &= \sum_{i=1}^q p_i \log_2 \left(\frac{1}{p_i} \right) \\ &\leq \sum_{i=1}^q p_i \log_2 \left(\frac{1}{r_i} \right) \\ &= \sum_{i=1}^q p_i \log_2 \left(r^{\ell_i} \right) \\ &= \sum_{i=1}^q p_i \ell_i \log_2 (r) \\ &= \log_2 (r) \sum_{i=1}^q p_i \ell_i \\ &= \log_2 (r) \text{AvgLen}(C, f) \end{aligned}$$

Dividiendo por $\log_2 (r)$:

$$H_r(\mathcal{S}) = \frac{H(\mathcal{S})}{\log_2 (r)} \leq \text{AvgLen}(C, f)$$

□

Ejemplo 3.3 Sea $\mathcal{S} = (S, P)$ una fuente con distribución uniforme con el alfabeto $S = \{0, 1, \dots, 9\}$. Entonces $H(\mathcal{S}) = \log_2 (10)$.

El teorema dice que la longitud promedio de una palabra de código unívocamente decodificable ternario es, al menos:

$$H_3(\mathcal{S}) = \frac{H(\mathcal{S})}{\log_2 (3)} = 2.0959$$

<

Este teorema nos puede auxiliar en la construcción de un código con palabras cortas, eficiente, para una fuente \mathcal{S} . Recordemos que si tenemos longitudes $\ell_1, \ell_2, \dots, \ell_q$ tales que satisfacen la desigualdad de Kraft:

$$\sum_{i=1}^q \frac{1}{r^{\ell_i}} \leq 1$$

entonces existe un código instantáneo C con palabras de esas longitudes para \mathcal{S} .

Si $P = \{p_1, p_2, \dots, p_q\}$ es la distribución de \mathcal{S} entonces podemos reemplazar el uno de la derecha de la desigualdad:

$$\sum_{i=1}^q \frac{1}{r^{\ell_i}} \leq 1 = \sum_{i=1}^q p_i \quad (3.5.11)$$

¡Ah! entonces si establecemos los valores de las longitudes de tal forma que para toda $i \in \{1, 2, \dots, q\}$ ocurra que:

$$\frac{1}{r^{\ell_i}} \leq p_i \quad (3.5.12)$$

se cumplirá automáticamente 3.5.11 y por tanto existirá un esquema de codificación instantáneo (C, f) para \mathcal{S} con esas longitudes justamente.

Pero 3.5.12 es equivalente a:

$$\log_r(r^{\ell_i}) \geq \log_r\left(\frac{1}{p_i}\right)$$

de donde:

$$\log_r\left(\frac{1}{p_i}\right) \leq \ell_i \quad (3.5.13)$$

Por supuesto las longitudes deben ser números enteros así que para satisfacer 3.5.13 y al mismo tiempo hacerlas lo más pequeñas posibles debemos tener que:

$$\ell_i = \left\lceil \log_r\left(\frac{1}{p_i}\right) \right\rceil$$

Pero esto significa que:

$$\log_r\left(\frac{1}{p_i}\right) \leq \ell_i < \log_r\left(\frac{1}{p_i}\right) + 1 \quad (3.5.15)$$

De donde, si usamos la cota superior:

$$\begin{aligned}
 \text{AvgLen}_r(C, f) &= \sum_{i=1}^q p_i \ell_i \\
 &< \sum_{i=1}^q p_i \left(\log_r \left(\frac{1}{p_i} \right) + 1 \right) \\
 &= \sum_{i=1}^q p_i \log_r \left(\frac{1}{p_i} \right) + \sum_{i=1}^q p_i \\
 &= H_r(\mathcal{S}) + 1
 \end{aligned}$$

Como (C, f) es un código instantáneo r -ario entonces la longitud promedio de sus palabras es, al menos la mínima de las longitudes promedio para los esquemas unívocamente decodificables. Así que finalmente:

$$\text{MinAvgLen}_r(\mathcal{S}) \leq \text{AvgLen}_r(C, f) < H_r(\mathcal{S}) + 1$$

Esto, junto con el la primera versión del teorema de la codificación sin ruido demuestra lo siguiente.

Teorema 3.4 (de la codificación sin ruido, versión 2). Sea \mathcal{S} una fuente de información. Si $\text{MinAvgLen}_r(\mathcal{S})$ denota la mínima longitud promedio de palabra sobre todos los esquemas unívocamente decodificables r -arios para \mathcal{S} entonces:

$$H_r(\mathcal{S}) \leq \text{MinAvgLen}_r(\mathcal{S}) < H_r(\mathcal{S}) + 1$$

Si ahora tomamos en consideración las extensiones de la fuente, que son por sí mismas, fuentes y les aplicamos el teorema:

$$H_r(\mathcal{S}^n) \leq \text{MinAvgLen}_r(\mathcal{S}^n) < H_r(\mathcal{S}^n) + 1$$

Pero sabemos que $H_r(\mathcal{S}^n) = n H_r(\mathcal{S})$, así que:

$$n H_r(\mathcal{S}) \leq \text{MinAvgLen}_r(\mathcal{S}^n) < n H_r(\mathcal{S}) + 1$$

dividiendo por n obtenemos el teorema siguiente:

Teorema 3.5 (de la codificación sin ruido, versión 3). Sea \mathcal{S} una fuente de información y \mathcal{S}^n su n -ésima extensión. Si $\text{MinAvgLen}_r(\mathcal{S}^n)$ denota la mínima longitud promedio de palabra sobre todos los esquemas unívocamente decodificables r -arios para \mathcal{S}^n entonces:

$$H_r(\mathcal{S}) \leq \frac{\text{MinAvgLen}_r(\mathcal{S}^n)}{n} < H_r(\mathcal{S}) + \frac{1}{n}$$

Esto es lo que justifica lo que ocurrió en el ejemplo que hicimos, en el que el código de Huffman para la segunda extensión de la fuente codifica con un promedio menor los símbolos de la fuente original que el código de Huffman hecho para ella.

Nótese que si n crece entonces $1/n$ se hace pequeño, esto significa que podemos codificar tan eficientemente como queramos una fuente, solo hay que elegir un valor de n suficientemente grande. Esto significa que codificaremos bloques de símbolos de tamaño n , lo que pudiera no ser práctico, después de todo cada bloque de tamaño n es un “símbolo” de la extensión y debemos poder guardar su frecuencia o probabilidad, como ya sabemos, el número de posibles cadenas de longitud n es q^n , donde q es el número de símbolos en el alfabeto original, así que este número es considerablemente más grande que q , el número de frecuencias que había que guardar para la fuente original.

INTERMEZZO A

Métodos de compresión

Existen muchos métodos de compresión de datos. Solo algunos de ellos son aplicación directa de los teoremas y métodos de codificación que hemos visto. En este intermezzo nos dedicaremos a analizar algunos de estos métodos y los principios en que están basados.

Los métodos de compresión de datos se clasifican en dos: aquellos que comprimen los datos y pueden restaurarlos íntegramente y aquellos en los que, una vez comprimidos los datos, estos no pueden ser restaurados en su totalidad. A los primeros se les denomina *sin pérdida* (*lossless*) y a los segundos *con pérdida* (*lossy*).

Los métodos sin pérdida se utilizan en aplicaciones en las que es indispensable absolutamente toda la información contenida en los datos originales, sería impensable que, luego de comprimir temporalmente un programa ejecutable, este dejara de ser ejecutable o perdiera parte de su funcionalidad. Los métodos con pérdida se utilizan en aplicaciones en las que la información perdida no es relevante, al comprimir imágenes o música, por ejemplo, se pueden despreciar ciertos detalles imperceptibles para el ojo o el oído. La eliminación de

ciertos datos no repercute perceptiblemente en la calidad de la imagen o de la música.

A.1 Métodos sin pérdida

A.1.1 Lempel-Ziv

En 1977 se publicó un artículo ([23]) en el que dos investigadores del Technion (Israel), Abraham Lempel y Jacob Ziv propusieron un algoritmo para compresión de datos basado en diccionarios.

Si tenemos un texto en español y reemplazamos cada palabra por un par de números x y y de tal forma que x sea el número de la página de nuestro diccionario favorito de la lengua española donde aparece la palabra que reemplazamos y y sea el número de la palabra en esa página, entonces es posible reescribir todo nuestro texto en este código de parejas de números de forma más breve. Además podríamos transmitírselo a un amigo que posea el mismo diccionario y podrá leer el texto original luego de decodificarlo. Suena bien.

Por supuesto el requisito para que nuestro método de compresión basado en el diccionario funcione, es que tanto el codificador como el decodificador posean el mismo diccionario. Si hablamos en términos de programas compresores y descompresores de datos, entonces estamos diciendo que tanto el programa compresor como el descompresor deben poseer el mismo diccionario. Hay entonces dos posibilidades: o el diccionario es añadido al programa compresor/descompresor o es añadido a los datos. Podemos ser un poco más específicos aún:

1. El diccionario es añadido al programa compresor/descompresor.
 - (a) El diccionario es muy general, posee muchas “palabras” representativas de una considerable variedad de “lenguajes”.

Ventajas: Es posible comprimir una amplia variedad de tipos de datos.

Desventajas: La cantidad de información requerida para especificar cada referencia al diccionario es tanto mayor cuanto más grande sea el diccionario. Posiblemente haya una gran proporción de palabras en el diccionario que se usan escasamente.

- (b) El diccionario es muy específico, solo maneja el “lenguaje” de un contexto particular

Ventajas: Las referencias a las palabras del diccionario son más cortas dado que el catálogo de posibles palabras es pequeño.

Desventajas: Se pierde generalidad, el tratar de comprimir datos que no pertenecen al contexto puede generar una cantidad de información mayor que la original (tasa de compresión negativa).

2. El diccionario es añadido por el compresor a los datos comprimidos, el descompresor recupera el diccionario antes de decodificar.

Ventajas: Se pueden hacer diferentes versiones de los programas compresores y descompresores cambiando el diccionario y los programas comprimidos con una versión pueden ser descomprimidos con otra.

Desventajas: El tamaño de los datos comprimidos se incrementa notablemente por tener el diccionario “pegado”.

Como puede verse un esquema de compresión con diccionario puede ser útil pero hay que tener cuidado al diseñarlo, principalmente en el rubro de su generalidad. Si queremos un compresor muy general entonces estaremos perdiendo mucho en su tasa de compresión efectiva.

Lo ideal sería tener un diccionario *ad-hoc* para cada conjunto de datos que se nos presente, así siempre tendríamos referencias de longitud mínima dado que el tamaño del diccionario sería también el mínimo indispensable. Siguiendo esta idea hasta sus últimas consecuencias llegamos hasta el concepto de diccionario dinámico.

Se dice que un esquema de compresión de diccionario es estático o que posee un diccionario estático si el diccionario utilizado por el esquema está definido de antemano y permanece fijo sin importar los datos que se estén comprimiendo o descomprimiendo. Si, en cambio, el diccionario se va construyendo durante el proceso de compresión/descompresión, se dice que el esquema es dinámico o que utiliza un diccionario dinámico.

El esquema propuesto por Ziv y Lempel en 1977 es un esquema dinámico en el que el diccionario cambia constantemente. De hecho el diccionario está constituido por la cadena de los últimos k caracteres leídos. A este tipo de esquema se le denomina también de *ventana deslizante*.

Durante la lectura y compresión de un archivo, el algoritmo de Ziv y Lempel de 1977, comúnmente llamado LZ77, mantiene una ventana de unos cientos de caracteres, los más recientemente procesados y una ventana mucho más pequeña, de unas decenas de caracteres, delante de la primera a la que se le llama en inglés *look-ahead*. El objetivo del algoritmo es tratar de encontrar coincidencias entre las subcadenas contenidas en la segunda ventana y las subcadenas contenidas en la primera. En caso de hallar una coincidencia, se hace referencia a la posición y longitud de la misma dentro de la ventana grande.

Pongamos un ejemplo. Supongamos que tenemos el texto que aparece a continuación:

```

cuando tuve
yo te tuve
te mantuve
y te dí

```

supongamos además que estamos usando un algoritmo LZ77 con ventana deslizante de longitud 11 y 6 caracteres de *look-ahead*. Cuando comenzamos a leer el texto con intención de comprimirlo no podemos hacer compresión alguna hasta que hemos leído los primeros 18 caracteres. Así que nuestro texto comprimido en ese momento es: “cuando tuve←yo te ” (nótese que se toma en cuenta el cambio de línea y el espacio final). En ese preciso momento la ventana comienza a la izquierda de la 't' del primer tuve y llega hasta el espacio luego de “te” (11 caracteres). También en ese preciso momento el *look-ahead* es la cadena “tuve←t” al final del segundo verso y la 't' al inicio del tercero. Entonces el algoritmo se percatará de que la subcadena del *look-ahead* “tuve←” ya la ha visto antes y de hecho la vió hace 11 caracteres, se da cuenta de esto porque la subcadena del *look-ahead* mencionada empata con el segmento inicial de la cadena de la ventana en ese momento. Así que en vez de poner la cadena “tuve←” en el texto comprimido simplemente debe decir: en que posición de la ventana (contando desde el extremo derecho de la misma) se encuentra el inicio de la subcadena que sigue, cuántos caracteres a partir de esa posición constituyen la cadena que sigue y, por último, que caracter hay que poner luego de la cadena que sigue. En nuestro ejemplo el texto comprimido sería ahora:

cuando tuve←yo te (11,5,'t')

la terna (11,5,'t') significa, los 5 siguientes caracteres se obtienen copiándolos a partir de la posición 11, contada a partir del extremo derecho de la ventana (el espacio después de “te” es el caracter 1), y luego se pone el caracter 't'.

Cuando el extremo derecho de la ventana sea la 'n' de la palabra “mantuve”, nuevamente será posible encontrar la cadena “tuve←” en la ventana, coincidentemente a partir de la posición 11, así que nuestro texto comprimido se vería así:

cuando tuve←yo te (11,5,'t')e man(11,5,'y') te dí←

Nótese que para saber si empata alguna subcadena del *look-ahead* con alguna subcadena de la ventana actual hay que buscar el empate más largo posible recorriendo toda la ventana. James Storer y Thomas Symanski inventaron una variante de LZ77, que se denomina LZSS, en la que las frases que se buscan empatar con la ventana se almacenan en un árbol binario de búsqueda, lo que reduce significativamente la complejidad para encontrar el empate más largo posible.

El conocido programa **gzip** está basado en LZ77 e incorpora algunas mejoras desarrolladas por Jean-Loup Gailly y Mark Adler con base en estadísticas.

Al año siguiente de la aparición del primer artículo, Ziv y Lempel (en ese orden, por alguna extraña razón en el nombre de los algoritmos las iniciales aparecen transpuestas) publicaron otro [24]. El algoritmo se llama, por supuesto, LZ78.

Una deficiencia importante en LZ77 es que si se encuentran un par de repeticiones de una cadena, suficientemente alejadas como para que la primera aparición haya salido de la ventana al tiempo en que es hallada la segunda, entonces el algoritmo no se da cuenta de que puede comprimir haciendo referencia al pasado. Simplemente la palabra ya no está en el diccionario cuando se la necesita [16]. Así que en LZ78 el diccionario deja de ser una ventana deslizante y se convierte en una entidad que se modifica carácter con carácter.

Cada vez que un nuevo carácter es leído, se busca en el diccionario la cadena que se ha ido construyendo con las lecturas hechas desde el último empate con el diccionario. Si la cadena actual se encuentra se hace una referencia al diccionario, en caso contrario la cadena se incorpora al diccionario y se continúa leyendo.

En este caso el inconveniente es que el diccionario crece aceleradamente y entonces hacer la búsqueda se convierte en un problema. Para evitar esto LZ78 almacena el diccionario en un árbol de búsqueda.

En 1984 Terry Welch del *Sperry Research Center* hizo una implementación bastante eficiente de LZ78, por lo que a su variante se le llama LZW. LZW es utilizado por el programa `compress` que viene normalmente en Unix y fue adoptado por Compuserve como el formato estándar para intercambio de gráficos GIF y fue adoptado también como el estándar de compresión V.42 bis por el CCITT. Sperry se fusionó con Burroughs para formar Unisys, así que Unisys es el dueño de la patente de LZW y por supuesto exigió el pago de derechos por el uso del algoritmo. De hecho en parte por esa razón fue que se desarrolló `gzip`, como la contraparte libre de el `compress` usual.

A.1.2 Burrows-Wheeler

En 1983 al profesor David Wheeler de la Universidad de Cambridge en el Reino Unido, se le ocurrió un algoritmo para transformar datos de tal forma que fueran susceptibles de una alta compresión. en ese momento Wheeler trabajaba en los *Bell labs* de AT&T y once años más tarde, en colaboración con M. Burrows de el centro de investigación en sistemas de la desaparecida DEC (hoy Compaq), puso todo en blanco y negro en un artículo ([5]). Actualmente el algoritmo de compresión Burrows-Wheeler es el utilizado por el programa `bzip2` que forma parte de las distribuciones de *Linux*.

El algoritmo de Burrows-Wheeler opera en tres fases [5, 6]:

1. BWT o transformación Burrows-Wheeler. Esta es la fase fundamental para lograr alta tasa de compresión. Básicamente, como veremos, consiste en reacomodar los datos de entrada para poder comprimirlos.
2. Algoritmo *Move-to-front*. En esta fase, a partir de la salida producida por la anterior, se producen datos a los que se les puede comprimir eficazmente con algoritmos conocidos que se aplican en la etapa siguiente.
3. Codificador basado en la entropía. en esta última fase se aplica un algoritmo de compresión que puede ser el de Huffman o el de compresión aritmética (que no vimos en el texto, el lector interesado puede consultar [16]).

El desempeño del algoritmo de compresión Burrows-Wheeler es bastante bueno en general aunque en la página del manual de `bzip2` se advierte que en archivos de datos completamente aleatorios puede ocurrir que el tamaño del archivo comprimido sea mayor que el del archivo original. En las comparaciones presentadas en [5] se aprecia que, con los *benchmarks* de compresión *Calgary Compression Corpus*, BW supera en un 10% a *gzip* (LZ77) y en un 33% al *compress* convencional de Unix (LZW, LZ78) en su tasa de compresión promedio sobre los 14 archivos del *benchmark*.

A grandes rasgos el algoritmo opera como sigue:

1. BWT. Se toma un bloque de datos (en teoría mientras más grande sea este bloque mejor, en la práctica si es muy grande puede volverse un problema por la cantidad de memoria necesaria) de N bytes. A partir de ese bloque se genera una matriz de $N \times N$ en la que el renglón inicial (de índice cero) contiene el bloque (cadena) de caracteres leídos tal cual y en su renglón i -ésimo aparece el mismo bloque rotado i lugares a la derecha. Para hacer una rotación simplemente desplazamos el contenido del bloque un lugar a la derecha y el caracter que sale por el extremo derecho “regresa” por la izquierda del bloque.

Considerando cada renglón como una cadena de caracteres, se ordenan los renglones crecientemente, en el renglón de índice cero de la matriz resultante aparece el renglón de valor mínimo lexicográficamente hablando. Por supuesto al menos un renglón de la matriz es el bloque original, llamemos r al índice de ese renglón.

La salida de esta etapa consiste de:

- r , el índice del renglón donde quedó la cadena original.
- L , la última columna de la matriz ordenada (la del extremo derecho).

La transformación hecha en esta etapa es reversible como se muestra en [5, 15]. Es decir se puede regenerar la cadena original a partir de la salida producida por esta etapa.

¿Por que es útil esta transformación? aparentemente estamos como al principio solo que con los datos desacomodados.

Consideremos que ocurre con una palabra que se repita mucho. Supongamos que una de esas palabras es “pelota” y que también aparecen por allí “rota” y “bota” por ejemplo. Como “pelota” es muy frecuente, entonces muchos de los renglones de la matriz ordenada que comienzan con “o” terminarán con “l” y estarán contiguos, así que uno esperaría que en la última columna de la matriz ordenada aparecieran corridas de “l” más o menos largas, a lo mejor interrumpidas por apariciones de algunas otras letras como la “r” o la “b” dado que tenemos “rota” y “bota” en nuestro bloque original.

Ya con lo que tenemos hasta ahora podríamos comprimir, pero ciertamente no ganaríamos nada más que si lo hubiéramos hecho al principio, la última columna simplemente tendrá, como hemos dicho los mismos símbolos originales solo que desacomodados. Lo realmente importante es el hecho de que hay corridas de letras, como veremos en la siguiente fase.

2. *Move-to-front*. Esto significa que, cuando vemos una letra decimos hace cuanto que no la vemos, si es que la hemos visto antes, si no la hemos visto previamente, simplemente decimos la letra misma. Es decir, vamos metiendo en una pila las letras y si volvemos a encontrar a alguien que ya está en la pila decimos su profundidad actual y la ponemos otra vez en el tope de la pila, como si la acabáramos de ver (lo que de hecho ocurrió). Cabe aclarar que no duplicamos letras en la pila, cuando reencontramos alguien solo lo extraemos de donde está (violando las reglas de acceso a la pila) y poniéndolo hasta arriba.

Imaginemos que tenemos $L = (abaagefabbaa)$. Cuando encontramos la primera “a” decimos “a” (formalmente hablando su código ASCII o Unicode o algo así), luego vemos la “b” y la decimos, hasta hora la pila tiene dos elementos la “b” en el tope y la “a” a profundidad 1. Luego volvemos a ver “a” así que decimos 1 (la profundidad de la “a” en la pila) y ponemos “a” en el tope, lo que nos deja la pila con “ab” donde “b” es la letra más profunda. Luego volvemos a ver “a” que está en el tope, así que decimos 0 y no hacemos nada con la pila. Si usamos los códigos ASCII convencionales entonces la secuencia en L quedaría traducida por el algoritmo *Move-to-front* en:

$$M(L) = (97, 98, 1, 0, 113, 101, 102, 3, 4, 0, 1, 0)$$

Lo que va a ocurrir es que la secuencia de números resultante deberá tener corridas largas de ceros, corridas largas de unos, aunque esperaríamos que más cortas que las

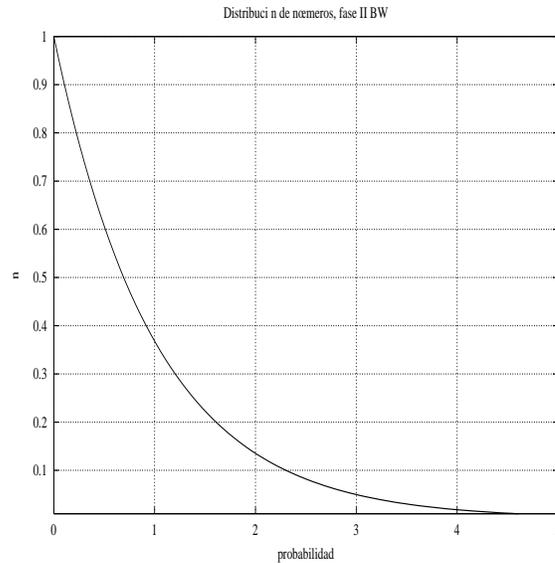


Figura A.1: Aspecto esperado de la distribución de los números (n) resultantes de la fase dos del algoritmo Burrows-Wheeler.

de ceros, corridas largas, aunque menos que las de unos, de 2, y así sucesivamente. Es decir la frecuencia de aparición de los números se vería como la figura A.1.

- Ahora se puede aplicar a $M(L)$ la codificación de Huffman (que es la utilizada en la implementación original de Burrows y Wheeler y en la versión 0.9.5 de `bzip2` que venía con Linux RedHat 6.2).

El aspecto de mostrado en la gráfica A.1 nos hace pensar en que un método como el de Huffman funcionará bien. Hay pocos números de $M(L)$ muy probables y muchos números poco probables.

A.2 Métodos con pérdida

A.2.1 El estándar de JPEG

Seguramente el lector está familiarizado con los espacios vectoriales y ha trabajado extensamente con \mathbb{R}^2 utilizando sus cualidades de espacio vectorial. Seguramente también sabe que todo espacio vectorial tiene conjuntos de vectores distinguidos llamados bases.

Las bases en los espacios vectoriales son importantes porque cualquier vector del espacio se puede escribir como una suma de los vectores la base multiplicados por escalares, es decir, cualquier vector del espacio vectorial puede ser escrito como una combinación lineal de los vectores de la base.

Pues bien, las funciones continuas también forman un espacio vectorial. Así que tendremos conjuntos de funciones que constituyen una base para el espacio vectorial de funciones continuas. Cualquier función se puede escribir como una combinación lineal de las funciones en la base.

Hay muchas bases diferentes para el espacio vectorial de funciones: los polinomios de Chebyshev o las funciones de Fourier por solo poner un par de ejemplos. Las de Fourier en particular expresan una función como combinación lineal de senos y cosenos, cada función de Fourier es ortogonal a todas las demás (la integral del producto es cero) y su integral (definida en cierto intervalo) vale 1, así que son ortonormales.

Hay un conjunto de funciones emparentadas con las de Fourier y que se utilizan para el mismo propósito. Son un conjunto de funciones coseno que permiten expresar funciones como combinación lineal de ellas. De hecho funciones discretas, por lo que a la transformación de la función en términos de la base se le denomina *transformación de coseno discreto* o DCT por sus siglas en inglés. En la figura A.2 se muestra una función $f(x)$, en este caso, solo como ejemplo, la función no es discreta sino continua. La función de la figura puede ser escrita como una combinación lineal de funciones de la forma:

$$f_i(x) = \cos\left(\frac{(2x+1)i\pi}{2N}\right)$$

en el intervalo $[0, N-1]$, donde i es un entero no negativo. En las figuras A.3 a la A.6 se muestran las funciones de la base. La función $f(x)$ puede escribirse entonces como

$$f(x) = \sum_{i=0}^{N-1} c_i f_i(x)$$

donde $i \in \{1, 2, \dots, N-1\}$ y $x \in [0, N-1]$, los coeficientes c_i (los escalares en la combinación lineal) deben ser determinados. En el caso de nuestra función ejemplo los coeficientes son: $c_0 = 12.3$, $c_1 = 10.5$, $c_2 = 8.4$, $c_3 = 6.74$, $c_4 = 4.3$, $c_5 = 1.67$, $c_6 = 0.2$ y $c_7 = 0.03$.

Hay que notar que las frecuencias altas, en nuestro caso las que corresponden a los índices 6 y 7 de las funciones de la base, contribuyen muy poco al aspecto general de la función; pesan más las componentes de baja frecuencia. Así que podríamos pensar en eliminarlos. En ese caso ya no tendremos nuestra función original, pero si una buena aproximación a ella; perderemos detalles, pero si realmente el peso de las componentes de alta frecuencia,

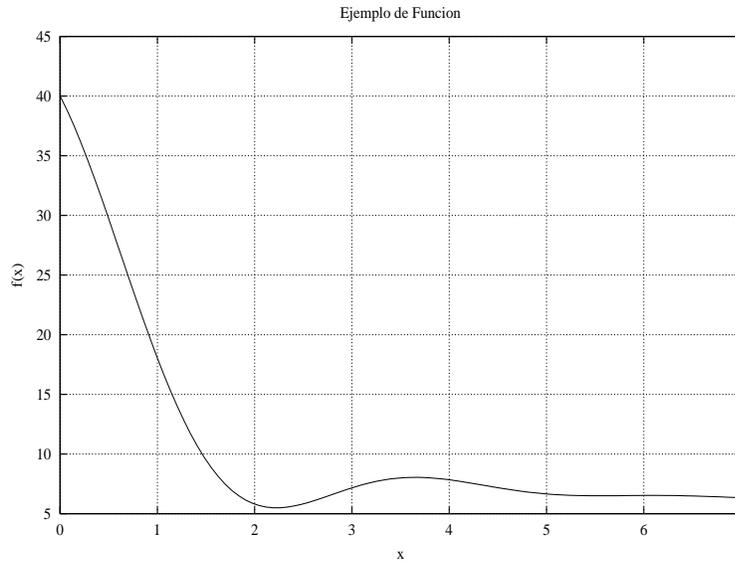


Figura A.2: Función de ejemplo

es decir, los escalares que multiplican a las funciones coseno de índice grande, son pequeños respecto a los demás, entonces lo que perdemos no es muy relevante. Compárense las gráficas de la función real (A.2) y de la aproximación que resulta de eliminar la contribución de las funciones de índice 6 y 7 (A.7), como se puede ver el aspecto general de la función está dado por las componentes de baja frecuencia. Hemos obtenido una aproximación bastante buena a nuestra función con unos cuantos coeficientes. Es por esto que nos interesan este tipo de transformaciones en general y la de coseno discreto en particular; solo con unos cuantos coeficientes podemos caracterizar toda una función con infinidad de puntos y además podemos aproximarla tan bien como queramos, solo debemos decidir cuantos coeficientes son suficientes.

Por supuesto dado que lo que se obtiene es una aproximación, es posible que algunos valores de la función no coincidan perfectamente con ella, es posible que algo de detalle se pierda, el objetivo es hacer que lo que se pierde no sea perceptible.

Estamos ahora en posibilidad de aplicar esto a la compresión de datos, pero debemos enfatizar que no podemos esperar que si solo aproximamos los datos originales mediante una base de funciones, algo puede perderse. ¿Para qué nos sirve entonces? ¿en que contexto puedo darme el lujo de perder algo de la información original?.

El oído humano solo capta sonidos en frecuencias de 200 a 2000 Hz. aproximadamente,

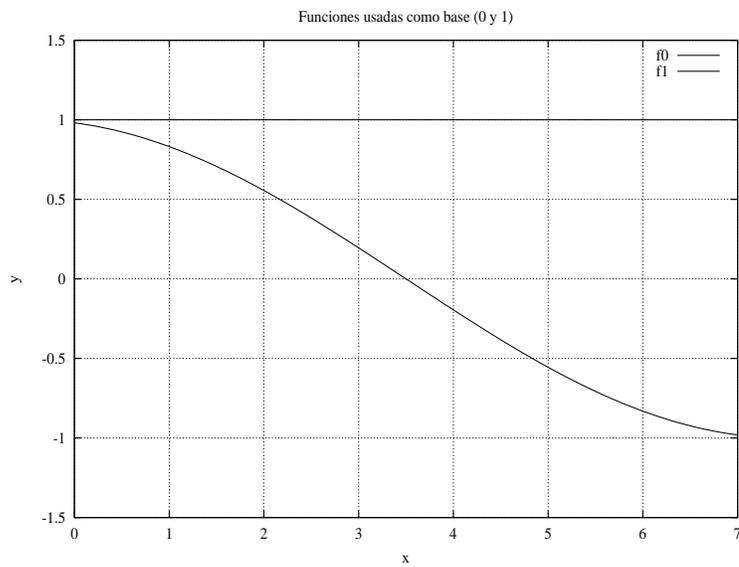


Figura A.3: Las funciones de índices 0 y 1 de la base usada en la transformación de coseno

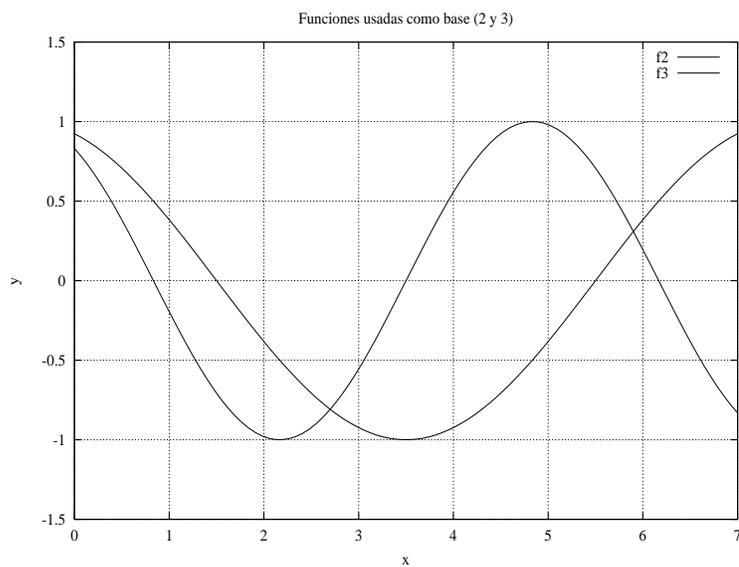


Figura A.4: Las funciones de índices 2 y 3 de la base usada en la transformación de coseno

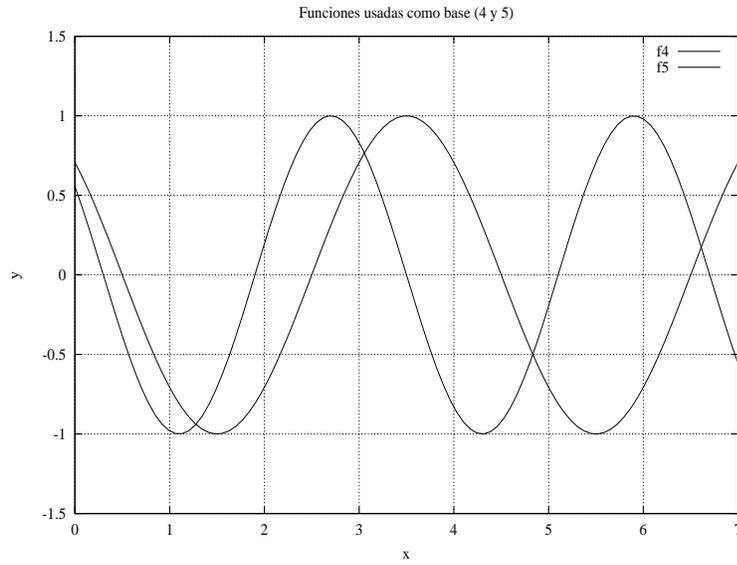


Figura A.5: Las funciones de índices 4 y 5 de la base usada en la transformación de coseno

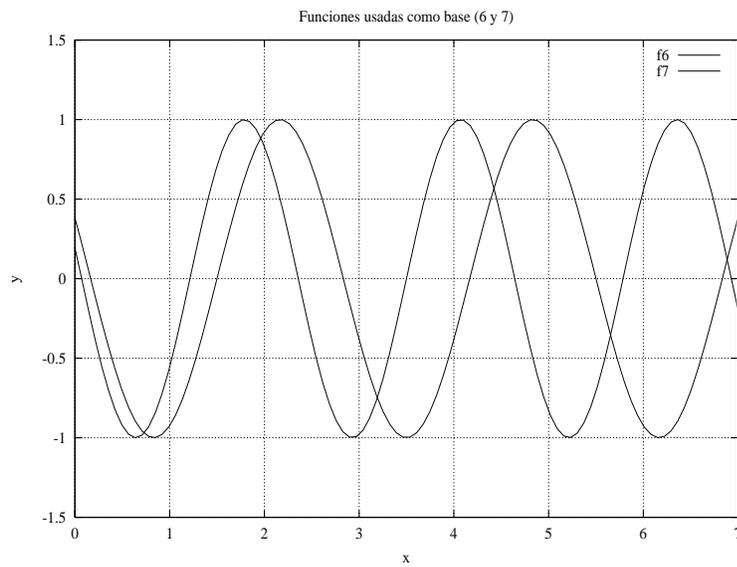


Figura A.6: Las funciones de índices 6 y 7 de la base usada en la transformación de coseno

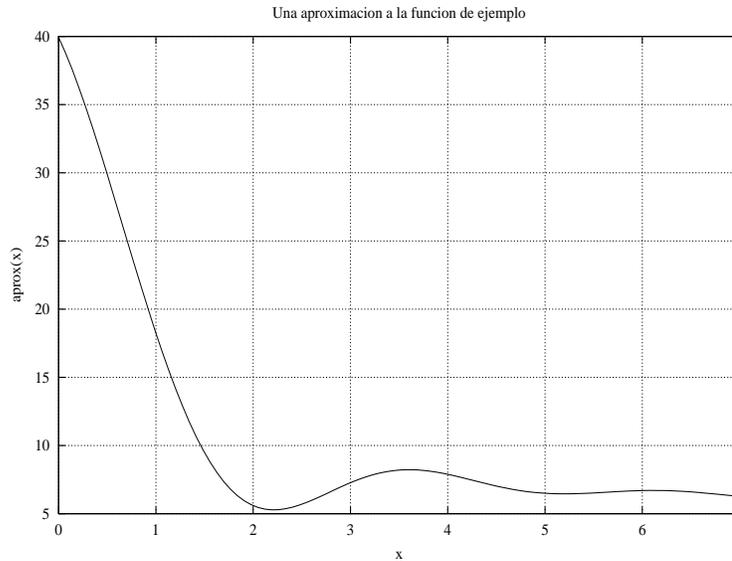


Figura A.7: Aproximación a la función de ejemplo. Se han eliminado los términos correspondientes a las funciones de índice 6 y 7 de la base. Compárese con la gráfica de la figura A.2.

sin embargo nuestros dispositivos de grabación de sonido pueden grabar frecuencias más altas o más bajas que las de ese rango. Así que si tenemos música grabada, como a fin de cuentas esta hecha para ser escuchada por oídos humanos, podemos perder frecuencias inaudibles y comprimir la información digitalizada de los sonidos; algo que hacemos cuando usamos MP3. Imaginemos ahora que tenemos una fotografía muy detallada de un bonito paisaje donde se ve una montaña cubierta de nieve, algunos pinos, el cielo azul. Se alcanzan a distinguir algunas rocas de la ladera de la montaña, incluso algunas que apenas se perciben por la distancia, todo alrededor de ellas es blanco y gris muy claro, son solo unos pequeños puntitos más oscuros, podríamos prescindir de ellos son que la calidad de nuestra imagen se vea sensiblemente afectada. Si tenemos un conjunto de funciones que sumadas nos dicen cuál es el color de cada pixel en nuestra imagen digitalizada, probablemente podríamos perder detalles insignificantes que están dados por algunos de los coeficientes que le dan relevancia a ciertos términos de la suma; esto es lo que hacemos al codificar imágenes en formato JPEG (*Joint Photographic Experts Group*). De hecho JPEG utiliza la transformación del coseno discreto para comprimir imágenes.

Por supuesto nuestras imágenes son bidimensionales y nuestra transformación de coseno discreto es unidimensional. Debemos definirla en dos dimensiones para poder aplicarla a imágenes.

En una imagen cada pixel tiene un valor, esencialmente un código que indica el color del pixel, pero si lo vemos como un simple valor, entonces una imagen es una función discreta que al punto de coordenadas (x, y) o columna x , renglón y , le asocia un valor $f(x, y)$ (el color del pixel), una altura en un imaginario eje z en la tercera dimensión. Esta función está dada explícitamente, para cada punto nos dice exactamente cuanto vale, lo que deseamos es aproximar esta función mediante una regla de correspondencia a la que uno le da (x, y) y regresa el valor aproximado de $f(x, y)$. Si usamos la transformación del coseno discreto ahora nuestras funciones son:

$$D_{i,j}(x, y) = \cos\left(\frac{(2x+1)i\pi}{2N}\right) \cos\left(\frac{(2y+1)j\pi}{2N}\right)$$

y hay que determinar los coeficientes $c_{i,j}$ tales que:

$$f(x, y) = \sum_{i=0}^{N-1} c_{i,j} D_{i,j}(x, y)$$

En el estándar de JPEG una imagen es dividida en cuadros de 8×8 pixeles y cada cuadrado de estos es considerado como una función. Esto hace que todas las funciones que se desean aproximar tienen un dominio que es $\{0, \dots, 7\} \times \{0, \dots, 7\}$ (los índices posibles de columna y renglón en el cuadrado). Así que cualquier función en esa malla de 64 puntos puede ser escrita *exactamente* con solo 64 diferentes funciones de la base coseno discreto. El problema consiste entonces en encontrar los 64 coeficientes de cada una de las funciones. Nótese que la frecuencia horizontal de las funciones crece conforme crece i y la vertical lo hace cuando crece j , así que si acomodamos estos 64 coeficientes en una matriz de 8×8 usando a i como índice de renglón y a j como índice de columna entonces los coeficientes cercanos a la esquina superior izquierda de la matriz corresponden a frecuencias bajas y las entradas cercanas a la esquina inferior derecha corresponden a frecuencias altas, a detalles de la imagen.

El primer paso en JPEG consiste pues en hallar los 64 coeficientes y acomodarlos en la matriz. Hay que notar que esto aun no comprime nada, de hecho estamos un poco peor que antes porque ahora tenemos un valor en punto flotante para cada coeficiente, curiosamente tenemos tantos coeficientes como pixeles, como en general el valor de un pixel es más corto que un número en punto flotante (a veces no, pero bueno), tenemos más datos que antes.

El segundo paso de JPEG se denomina cuantización, es en esta fase en la que realmente se comprimen los datos. El término cuantización se usa en el sentido en el que se utilizaría en física, hacer discreta la información que se tiene. Cada coeficiente de la matriz es dividido por un número predeterminado por el estándar de JPEG, elegido de acuerdo a que tan importantes consideran los expertos del grupo ciertas frecuencias, luego de hacer la

división el resultado se redondea, antes era un número en punto flotante y ahora es un entero. Ahora si poseemos menos datos que al principio y, de hecho, menos información. El paso previo no pierde nada, los coeficientes en la matriz expresaban exactamente, con todo detalle, la imagen original. Al final de la fase de cuantización, en cambio, ya no tenemos la misma información hemos disminuido la importancia de algunos coeficientes y los hemos redondeado, ahora tenemos una aproximación a la imagen original.

En la tercera fase de JPEG se recorre la matriz de coeficientes en zig-zag como se muestra en la figura A.8. Nótese que el recorrido va de frecuencias bajas a frecuencias altas. El primer coeficiente se escribe tal como está en la matriz, para escribir el segundo en realidad se escribe su diferencia respecto al primero, en general para escribir el i -ésimo coeficiente encontrado en el recorrido se escribe la diferencia entre él mismo y el $(i - 1)$ -ésimo. El objetivo de escribir los coeficientes de esta manera es escribir con mayor frecuencia números más pequeños. Como en general los primeros coeficientes serán grandes, de magnitud similar, su diferencia será pequeña y como los últimos serán muy pequeños o cero su diferencia será también pequeña. La frecuencia del número cero será alta y mientras mayor sea un número su frecuencia será mucho menor. Esto nos permitirá lograr aun mayor compresión en la última fase.

Por último JPEG utiliza un codificador de entropía como el de Huffman, para comprimir la secuencia de números que se obtuvo de la fase anterior.

Recuperar la imagen original, o mejor dicho, una buena aproximación a ella, consiste en decodificar cada una de las secuencias de números pequeños obtenidas en la última fase de compresión, Para hacer esto solo hay que decodificar usando el algoritmo de Huffman o el que se haya tenido a bien utilizar. Esto por cada uno de los cuadrados de 64 píxeles. Luego hay que reconstruir la matriz de coeficientes para cada cuadrado recorriendo la secuencia y poniendo la suma de el último coeficiente obtenido y la diferencia que viene en la secuencia. Ahora con los coeficientes solo hay que evaluar la combinación lineal de cosenos discretos en las coordenadas (x, y) para obtener el valor del píxel en esas coordenadas.

Actualmente se está trabajando en el estándar JPEG 2000 que ya no divide la imagen en cuadrados de 8×8 sino que la trata entera y que ya no utiliza la transformación de coseno discreto, sino otra llamada *Discrete Wavelet Transform* o DWT [21], el objetivo es el mismo, expresar la imagen como una combinación lineal de funciones periódicas. La DWT tiene la cualidad de descomponer la expresión de la función en dos partes: una de baja frecuencia y otra de alta. Como la DWT se aplica tanto a los renglones como a las columnas de la imagen, la matriz de sus coeficientes se divide en cuatro: (L, L) baja frecuencia para columnas, baja para renglones; (L, H) , baja para columnas, alta para renglones; (H, L) , alta para columnas, baja para renglones y (H, H) alta para renglones y columnas; pudiendo a su vez ser dividida recursivamente, lo que facilita el tener diferentes niveles de calidad de imagen, siempre en

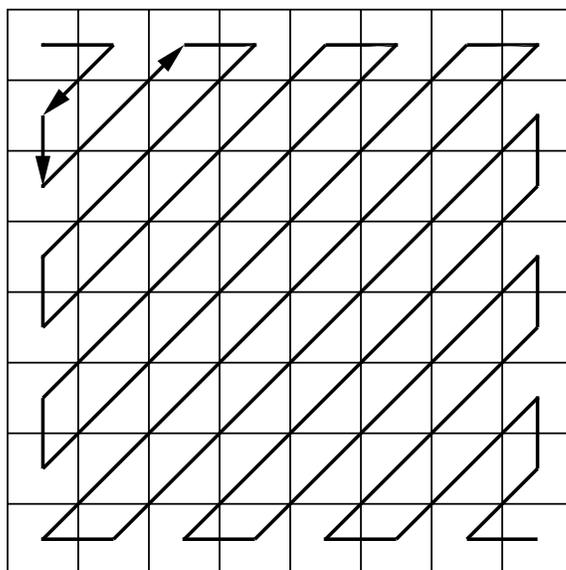


Figura A.8: Recorrido de la matriz de coeficientes.

correspondencia inversa al grado de compresión. Además DWT proporciona mejor calidad de imagen para tasas de compresión similares a las de la transformación de coseno discreto.

4

Codificación para detectar y corregir errores

Hasta ahora nos hemos dedicado a codificar datos buscando representarlos de la manera más breve posible. Hemos estado suponiendo que lo que se escribe por un lado del canal de transmisión (ya sea este espacial o temporal) puede ser leído correctamente del otro lado; lo que se lee es exactamente lo que se escribió. Pero el panorama cambia drásticamente si el canal de transmisión no es confiable, si no estamos seguros de que lo que se escribe de un lado puede ser leído con toda exactitud del otro lado entonces nuestras prioridades cambian, ahora nuestra preocupación fundamental no es abreviar la representación de los datos, sino asegurar que estos puedan ser recuperados.

Ahora nuestra intención es hacer posible que, luego de transmitir los datos de interés, podamos percatarnos de que ocurrieron errores durante la transmisión y en el mejor de los casos, revertir las alteraciones que estos errores ocasionaron a los datos.

Los códigos que usaremos ahora son de longitud fija, también llamados de bloque, todas las palabras del código miden lo mismo.

Definición 4.1 Sea $A = \{a_1, \dots, a_r\}$ un conjunto de tamaño (cardinalidad) r al que llamaremos el *alfabeto del código* y cuyos elementos se llaman *símbolos del código*. Un *código de bloque r -ario* C sobre A es un subconjunto no vacío de A^n (el conjunto de todas las cadenas de longitud n sobre A). Los elementos de C se denominan *palabras de código*, n es la *longitud del código* (formalmente de cada una de sus palabras). Al número M de palabras en el código C se le llama el *tamaño del código*. Un código con M palabras de longitud n sobre un alfabeto de r símbolos lo denominaremos un (n, M) -código r -ario.

Por el canal de comunicación también viajan símbolos, podemos suponer que uno a la vez (serial). Así que cada símbolo que viaja por el canal puede ser alterado y convertido en algún otro símbolo. Si esto ocurre y solo un símbolo ha viajado por el canal entonces no podemos darnos cuenta de que ha ocurrido un error, lo que recibimos es tan válido como lo que nos fue enviado. Si en cambio ocurre un error en un símbolo dentro de una palabra de código posiblemente podamos percatarnos de ello. Si al ocurrir el error un símbolo fue cambiado por otro y la cadena resultante está en A^n pero no en el catálogo de palabras del código C , entonces *podemos estar seguros* de que han ocurrido errores. Si luego de que ocurren el o los errores el resultado es una cadena A^n que si está en C entonces no podremos percatarnos de que han ocurrido alteraciones indeseables en los datos porque, igual que como ocurría en el caso de un solo símbolo, el resultado de las alteraciones es algo tan válido como lo que nos fue enviado en realidad y bien hubiera podido ser eso lo que se nos quería transmitir.

4.1 Regla de decisión y canal de comunicación

Ahora bien ¿qué hacemos con lo que recibimos? si la cadena recibida es una de las del código entonces simplemente la aceptaremos, puede que hayan ocurrido errores o no, pero nunca lo sabremos a menos que haya algún mecanismo externo a nuestro sistema (el conjunto de palabras de código y el canal) para determinarlo (por ejemplo, retransmisiones de los mismos datos o algunos datos extra para verificar todos los anteriores, etc.). Pero si lo que recibimos no es una de las palabras del código entonces hay dos opciones: señalamos que hubo errores durante la transmisión de los datos o tratamos de corregir los símbolos en la cadena que presumimos fueron alterados. Formalmente hablando lo que estamos haciendo en cualquiera de los tres casos es aplicar una regla de decisión.

Definición 4.2 Una regla de decisión es un procedimiento (hablando como computólogos)

que decide, dada la cadena recibida, que cadena del código le corresponde o si han ocurrido errores.

Si formuláramos la definición en los términos de un matemático diríamos que una regla de decisión es una función f que mapea una cadena cualquiera $s \in A^n$ en $f(s) \in C \cup \{?\}$, donde $?$ es un símbolo utilizado para denotar que han ocurrido errores.

Ejemplo 4.1 Sea $C = \{00000, 11111\}$ un $(5, 2)$ -código. Supongamos que la cadena recibida es: 11011. Una posible regla de decisión es: $f_1(11011) = 11111$. Otra posible regla es: $f_2(11011) = 00000$ y por supuesto otra es: $f_3(11011) = ?$. ¿Cuál es la mejor si queremos recuperar la información que originalmente nos fue transmitida y que desconocemos? responder esto no es trivial, para hacerlo necesitamos saber algo del canal por el que viajó la información. Si por ejemplo sabemos que el canal casi nunca altera los datos que transitan por él, entonces podríamos decir que *con alta probabilidad* lo que en realidad nos fue enviado es el resultado de f_1 , si en cambio sabemos que el canal es muy malo y casi siempre altera lo que viaja por él entonces *muy probablemente* la mejor regla de decisión sea f_2 . Por supuesto si no queremos arriesgarnos podemos optar por f_3 , más seguro, pero menos interesante. \triangleleft

En general la elección de la mejor regla de decisión depende de las características del canal y hay que enfatizar que *nunca* podemos estar completamente seguros de que la regla de decisión nos entrega la palabra de código que realmente nos fue enviada. Lo único cierto es la incertidumbre, el objetivo es minimizarla.

La característica más desagradable de nuestros canales de comunicación y nuestros dispositivos de almacenamiento (canales de comunicación temporales), es que se ven influidos por el azar. Las cosas serían más sencillas si supiéramos de antemano cuáles símbolos serán alterados en un cable de red o en un disco duro, pero no es así. De pronto se enciende el motor de aire acondicionado que está cerca de donde pasa el cable de red y su campo magnético altera brevemente algunos de los datos que pasan por allí o hay una pequeña variación de densidad en el material de un disco o una impureza que altera lo que allí se pretendía almacenar o un rayo da al traste con una transmisión de microondas, en fin; la naturaleza sí da saltos y dios prefiere jugar a los dados antes que tocar el violín.

Así que en nuestros canales los errores ocurren aleatoriamente y para modelarlos necesitaremos de la probabilidad, para variar. De un lado del canal se nos envía un símbolo a_i , del conjunto de símbolos que pueden viajar por el canal y que en general podemos pensar que son los del alfabeto del código, del otro lado del canal se recibe un símbolo a_j que puede o no coincidir con a_i . Así que para saber como se comporta el canal necesitamos tener las probabilidades:

$$P(a_j \text{ recibido} \mid a_i \text{ enviado})$$

para todos los símbolos posibles a_i y a_j que pueden viajar por el canal.

Definición 4.3 Un canal de comunicación r -ario consiste de un conjunto de símbolos llamado alfabeto de canal $A = \{a_1, \dots, a_r\}$ y un conjunto de probabilidades de transición $P(a_j \text{ recibido} \mid a_i \text{ enviado})$ que satisfacen:

$$\sum_{j=1}^r P(a_j \text{ recibido} \mid a_k \text{ enviado}) = 1$$

(nótese que k permanece fija durante toda la suma). Este conjunto de probabilidades no cambia con el tiempo.

Otra característica que añadiremos a nuestro modelo de canal es la falta de memoria. El resultado en la transmisión de cualquier símbolo en un momento dado no se ve influido por alguna transmisión previa, es decir la transmisión de un símbolo es un evento independiente. Así que si $\mathbf{c} = c_1 c_2 \dots c_n$ es una palabra de C enviada por un extremo del canal y $\mathbf{d} = d_1 d_2 \dots d_n$ es una cadena de A^n recibida en el otro extremo entonces:

$$P(\mathbf{d} \text{ recibido} \mid \mathbf{c} \text{ enviado}) = \prod_{i=1}^n P(d_i \text{ recibido} \mid c_i \text{ enviado})$$

Como nuestro modelo de canal está compuesto de los símbolos del alfabeto y las probabilidades de transición entonces un canal puede ser completamente descrito mediante una gráfica bipartita en la que los dos conjuntos de vértices a cada lado son los símbolos del alfabeto $A = \{a_1, \dots, a_r\}$ y el peso de cada arista (a_i, a_j) en la gráfica es la probabilidad de transición $P(a_j \text{ recibido} \mid a_i \text{ enviado})$.

Hay algunos canales de uso frecuente en teoría de la información, algunos de ellos están representados gráficamente en la figura 4.1.

Estos modelos de canales son útiles pero por supuesto no son verosímiles en toda circunstancia. Muchos canales de comunicación reales no podríamos ajustarlos a alguno de estos modelos, en general suelen ser más complicados. En la vida real el comportamiento de un canal está determinado por la fuente de distorsión de datos. Por ejemplo el ruido causado por la atmósfera o por el movimiento de los electrones en el cable que transmite (ruido térmico) se denomina *ruido blanco* o ruido blanco gaussiano. La luz blanca se hace mezclando todos los colores (todas las frecuencias del espectro visible) en la misma medida, el ruido es “blanco” porque, análogamente, contiene de todas las frecuencias en la misma medida. Podemos “ver” y oír el ruido blanco cuando se acaba la programación de una estación de televisión y queda la pantalla llena de puntitos. Otro tipo de ruido ocurre

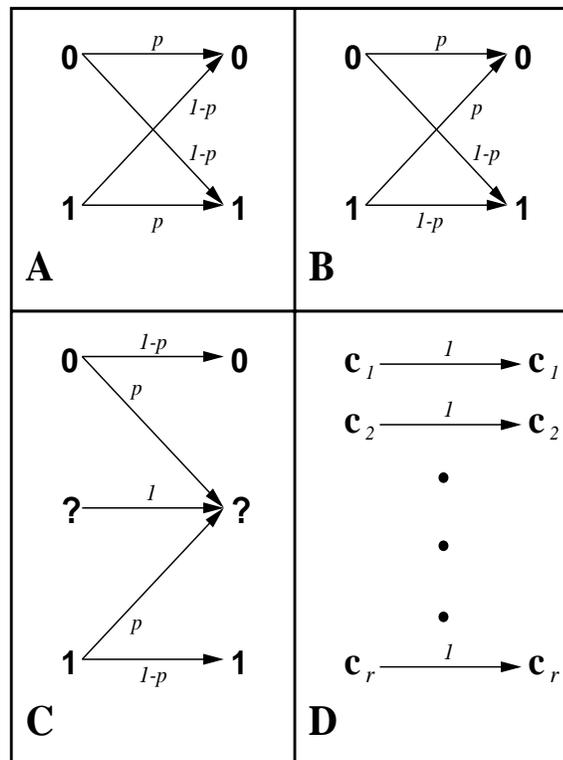


Figura 4.1: Algunos canales usuales. En **A** se muestra un canal simétrico binario, en **B** un canal inútil: si p es grande casi siempre que se envía cero se recibe cero, pero casi siempre que se envía uno se recibe cero, así que si se recibe cero ¿que habrá sido enviado?. En **C** se muestra un *canal de borrado*: o el símbolo transmitido llega bien o se borra, no hay confusión, por cierto se ha incluido solo por completez el símbolo “?” en el lado izquierdo, en realidad nunca se envía “?”. En **D** se muestra un canal sin ruido. El teorema de Shannon visto en el capítulo 3 supone que la transmisión de datos se lleva a cabo a través de un canal sin ruido.

cuando nuestros cables de red, por ejemplo, pasan cerca de un motor que no funciona permanentemente: cuando el motor se enciende ocasiona perturbaciones electromagnéticas a su alrededor, luego el motor se apaga y todo vuelve a la normalidad, el ruido se da en ráfagas que afectan, casi siempre sin remedio, grupos compactos de bits; a este tipo de ruido se le llama *ruido de impulso*. El ruido es pues una alteración electromagnética y como tal se puede ver como la superposición de muchas ondas diferentes (el ruido blanco suma sobre todas las frecuencias en la misma medida) y estas a su vez se suman a las que constituyen nuestras transmisiones.

Evidentemente no podemos evitar el ruido, así que debemos vivir con el y tratar de contrarrestar sus efectos. Lo que deseamos es poder encontrar una regla de decisión que maximice la probabilidad de decodificar correctamente, es decir necesitamos una regla de decisión f que maximice la probabilidad de que $f(x)$ haya sido enviada cuando recibimos x . Que se equivoque lo menos posible. supongamos que se nos envía una cierta palabra de código $\mathbf{c} \in C$ y que algo afecta la transmisión de la palabra, lo que ocasiona que el receptor reciba una cadena $x \in A^n$, desearíamos tener una regla de decisión f que nos dijera $f(x) = \mathbf{c}$. Hay que tener en cuenta que nuestra regla de decisión es estática, es decir si recibimos en dos instantes de tiempo diferentes la misma cadena y y la primera vez nuestra decisión fue $f(y) = \mathbf{c}_i$ entonces la segunda vez la decisión es la misma. Bajo estas condiciones lo que queremos es una regla de decisión que nos dijera $f(x) = \mathbf{c}$ en el ejemplo anterior, siempre y cuando \mathbf{c} sea la palabra de entre todas las del código, con la mayor probabilidad de haber sido transformada en x por el canal. Es decir:

$$P(\text{decodificar correctamente}) = \sum_{x \in A^n} P(f(x) \text{ enviado} \mid x \text{ recibido}) P(x \text{ recibido}) \quad (4.1.1)$$

Así que para maximizar el lado izquierdo de esta expresión debemos maximizar cada uno de los sumandos del lado derecho, dado que $P(x \text{ recibido})$ está fija, que debemos maximizar: $P(f(x) \text{ enviado} \mid x \text{ recibido})$. Esto significa que dada la cadena recibida $x \in A^n$ debemos tener una lista de las probabilidades:

$$\begin{aligned} &P(\mathbf{c}_1 \text{ enviado} \mid x \text{ recibido}) \\ &P(\mathbf{c}_2 \text{ enviado} \mid x \text{ recibido}) \\ &\quad \vdots \\ &P(\mathbf{c}_n \text{ enviado} \mid x \text{ recibido}) \end{aligned}$$

en la que podamos buscar para que palabra de código \mathbf{c}_k el valor de la probabilidad en la lista es mayor. De no ser porque esto es imposible sería bastante complicado: necesitaríamos tener almacenadas $|A|^n$ probabilidades para cada palabra de código. Sin embargo

podríamos garantizar que en la mayoría de los casos, dada la cadena recibida, adivinamos correctamente cuál fue la palabra de código que nos fue enviada.

Pero realmente esto es imposible. Nótese que las probabilidades de la lista solo puede calcularlas alguien que observa lo que se envía y lo que se recibe: podríamos decirle “Dios” u *observador ideal*.

Definición 4.4 Cualquier esquema de decisión f con la propiedad de que:

$$P(f(x) \text{ enviado} \mid x \text{ recibido}) = \max_{\mathbf{c} \in \mathcal{C}} \{P(\mathbf{c} \text{ enviado} \mid x \text{ recibido})\}$$

para todas las posibles cadenas recibidas x , es llamado un *observador ideal*.

Ejemplo 4.2 En un sistema de comunicación recibimos la cadena 1010 y en nuestra última charla con Dios el nos dijo que el 90% de las veces que recibíamos 1010 en realidad nos habían enviado 1011. Así que en la mejor regla de decisión que podríamos tener, la cadena 1010 se mapea en 1011. \triangleleft

Bien ahora, como todo filosofo medieval que se respete, nos preguntamos ¿como podemos acercarnos a la divinidad? ¿como hacemos nosotros, simples mortales contingentes e imperfectos, para procurar tener una regla de decisión lo más cercana posible al observador ideal?

Podríamos tratar de poner cada una de las probabilidades condicionales en la lista en términos cosas que si podemos conocer usando el teorema de Bayes:

$$P(\mathbf{c} \text{ enviado} \mid x \text{ recibido}) = \frac{P(x \text{ recibido} \mid \mathbf{c} \text{ enviado}) P(\mathbf{c} \text{ enviado})}{\sum_{i=1}^M [P(x \text{ recibido} \mid \mathbf{c}_i \text{ enviado}) P(\mathbf{c}_i \text{ enviado})]} \quad (4.1.2)$$

bien, ahora tenemos involucradas las probabilidades del canal, las *forward probabilities*. Sin embargo también tenemos involucradas las probabilidades:

$$P(\mathbf{c}_1 \text{ enviado}), P(\mathbf{c}_2 \text{ enviado}), \dots, P(\mathbf{c}_M \text{ enviado})$$

que es la distribución de entrada al canal y que, por supuesto, no conoce el receptor. Ahora sí ya no hay nada que hacer, como siempre los pobres mortales no podemos siquiera aproximarnos a la suprema perfección.

Para no vernos tñ patéticos podemos hacer suposiciones acerca de la distribución de entrada y la suposición menos descabellada es considerarla uniforme. Es decir, suponer que

$$P(\mathbf{c}_i \text{ enviado}) = \frac{1}{M}$$

para toda $\mathbf{c}_i \in C$, donde M es el número de palabras en el código. En este caso la expresión 4.1.2 se transforma en:

$$\begin{aligned} P(\mathbf{c} \text{ enviado} \mid x \text{ recibido}) &= \frac{P(x \text{ recibido} \mid \mathbf{c} \text{ enviado}) (1/M)}{(1/M) \sum_{i=1}^M [P(x \text{ recibido} \mid \mathbf{c}_i \text{ enviado})]} \\ &= \frac{P(x \text{ recibido} \mid \mathbf{c} \text{ enviado})}{\sum_{i=1}^M [P(x \text{ recibido} \mid \mathbf{c}_i \text{ enviado})]} \end{aligned} \quad (4.1.4)$$

y maximizar esta última expresión consiste en maximizar las probabilidades del canal, esto tiene un nombre.

Definición 4.5 Una regla de decisión f que maximiza las probabilidades del canal (*forward probabilities*), es decir:

$$P(x \text{ recibido} \mid f(x) \text{ enviado}) = \max_{\mathbf{c} \in C} P(x \text{ recibido} \mid \mathbf{c} \text{ enviado})$$

para toda $x \in A^n$, es llamada una *regla de decisión de máxima verosimilitud*.

Así que, si la distribución de entrada al canal fuera uniforme (todas las palabras de C tiene probabilidad $1/|C|$ de ser enviadas), entonces el observador ideal sería justamente el de máxima verosimilitud.

Ejemplo 4.3 Supongamos que en un canal hay una probabilidad $p = 0.05$ (por lo que $1 - p = 0.95$) de que se estropee (invierta) un bit. Sea $C = \{0000, 1111\}$ un código y supongamos que, luego de que se nos envía una de las palabras de C recibimos 0100.

Tenemos:

$$\begin{aligned} P(0100 \text{ recibido} \mid 0000 \text{ enviado}) &= P(1 \text{ recibido} \mid 0 \text{ enviado}) \times \\ &\quad P(0 \text{ recibido} \mid 0 \text{ enviado})^3 \\ &= 0.05 \times (0.95)^3 \approx 0.0428 \\ P(0100 \text{ recibido} \mid 1111 \text{ enviado}) &= P(1 \text{ recibido} \mid 1 \text{ enviado}) \times \\ &\quad P(0 \text{ recibido} \mid 1 \text{ enviado})^3 \\ &= 0.95 \times (0.05)^3 = 0.000118 \end{aligned}$$

Así que la regla de decisión de máxima verosimilitud (f) dice: $f(0100) = 0000$. Esta sería la regla de observador ideal si suponemos que $P(0000 \text{ enviado}) = P(1111 \text{ enviado}) = 1/2$.

Pero sí: $P(0000 \text{ enviado}) = 0.0001$ y $P(1111 \text{ enviado}) = 0.9999$, entonces (usando el teorema de Bayes):

$$P(0000 \text{ enviado} \mid 0100 \text{ recibido}) = \frac{0.0428 \times 0.0001}{0.000122} = 0.035$$

y

$$P(1111 \text{ enviado} \mid 0100 \text{ recibido}) = \frac{0.000118 \times 0.9999}{0.000122} = 0.965$$

¡que cambio! ¿no?

◁

4.2 Decodificación al vecino más cercano

Ya habíamos mencionado la posibilidad de que al enviar una palabra de código, esta sea alterada por errores en el canal y sea recibida, por pura casualidad, otra palabra válida del código. Si esto pasa, no es posible para el receptor, determinar si ocurrieron o no errores de transmisión. Por otra parte nuestros canales de comunicación son bastante confiables, en condiciones normales las tasas de error de los canales que solemos utilizar no son muy altas. No es descabellado suponer entonces que los datos recibidos difieren poco de los enviados. Teniendo estas dos cosas en mente pensemos que condiciones son deseables en el código utilizado para enviar los datos a fin de que se facilite la labor de detectar o corregir errores.

Sabemos que la diferencia entre la palabra enviada y la recibida es proporcional a la tasa de error del canal y sabemos que si lo recibido es una palabra de código no podemos percatarnos de nada; conclusión: cuanto mayores sean las diferencias entre las palabras que se pueden enviar (las del código) tanto más fácil será percatarse de los errores. Si las palabras del código son muy diferentes entre ellas, reducimos la probabilidad de que, luego de algunos errores, se obtenga otra palabra del código, lo que impediría que nos diéramos cuenta de los errores. Mientras mayores sean las diferencias mayor será la tasa de errores que se pueden percibir. Ahora bien ¿que herramienta poseemos para medir las diferencias entre dos cadenas de símbolos? pues la distancia de Hamming.

Durante esta sección estaremos trabajando siempre sobre un canal simétrico binario y examinaremos la decodificación de máxima verosimilitud para este tipo de canal suponiendo que la tasa de error es menor que la tasa de acierto. Recordemos que la probabilidad de que un bit sea alterado por un canal simétrico binario (también llamada probabilidad de *crossover*) la denotamos por p , así que estaremos suponiendo que $p < 1 - p$, es decir $p < 0.5$. En estas condiciones la probabilidad de que una palabra de n bits no sea alterada es:

$$P(\text{no error}) = (1 - p)^n$$

y la probabilidad de que ocurran exactamente $k \leq n$ errores en posiciones predeterminadas de una palabra (nótese que son posiciones predeterminadas, por lo que en el siguiente cálculo no aparece la expresión para elegir las k posiciones de entre las n disponibles) es:

$$P_k = p^k(1 - p)^{n-k}$$

Así que si una palabra x de longitud n sobre un alfabeto binario difiere en k posiciones de una palabra de código binario \mathbf{c} de longitud n entonces:

$$P(x \text{ recibido} \mid \mathbf{c} \text{ enviado}) = p^k (1 - p)^{n-k} \quad (4.2.5)$$

Teorema 4.1 *Para un canal simétrico binario con probabilidad de error $p < 1/2$, la regla de decisión de máxima verosimilitud es la que elige la palabra de código cuya distancia de Hamming a la palabra recibida x es mínima.*

Dem.: Sea x una palabra binaria de longitud n recibida. En la expresión 4.2.5, es claro que $P(x \text{ recibido} \mid \mathbf{c} \text{ enviado})$ es mayor mientras más grande sea el exponente $n - k$, dado que $p < 1 - p$.

Pero $n - k$ es tanto más grande cuanto menor sea k , que es la distancia de Hamming entre x y la palabra de código \mathbf{c} . Así que la regla de decodificación de máxima verosimilitud (la que maximiza $P(x \text{ recibido} \mid \mathbf{c} \text{ enviado})$) es la que elige decodificar x como la palabra de código \mathbf{c} con la distancia de Hamming más pequeña a x . \square

A las palabras de código con mínima distancia de una palabra x se les denomina *las vecinas más cercanas de x* y a la regla de decisión que acabamos de formular se le denomina por tanto *de vecino más cercano*.

Nótese que dado que $p < 1 - p$ al decodificar mediante la regla de decisión de vecino más cercano estamos suponiendo implícitamente que el número de errores es justamente la distancia de Hamming entre lo que recibimos y lo que es entregado por la regla de decisión.

Ejemplo 4.4 Sea $C = \{00000, 00101, 11000, 00011\}$ un código binario usado para transmitir datos a través de un canal. Sea $x = 01101$ una palabra recibida. Entonces la decodificación de vecino más cercano f decodifica x como $f(x) = 00101$ ya que la segunda palabra del código dista de x en uno y el resto de las palabras distan 3 unidades de ella. \triangleleft

4.3 Distancia mínima, capacidad de detección y corrección

Comencemos esta sección con un ejemplo ilustrativo.

Ejemplo 4.5 Supongamos que tenemos el código binario de longitud 10 siguiente $C = \{\mathbf{c}_1 = 0000000000, \mathbf{c}_2 = 1111100000, \mathbf{c}_3 = 1111111111\}$, que se nos envía la palabra \mathbf{c}_1 y que recibimos la cadena $x_1 = 1100000000$, nuestra regla de decodificación de vecino más cercano f nos entrega entonces $f(x_1) = 0000000000 = \mathbf{c}_1$ lo cual es correcto. Pero si en cambio recibimos $x_2 = 1110000000$ entonces $f(x_2) = 1111100000 = \mathbf{c}_2$ lo que ya no es correcto. Nuestro código puede corregir dos errores pero no tres en este caso. Sin embargo si la palabra recibida es $x_3 = 0000011111$ entonces $f(x_3) = 0000000000 = \mathbf{c}_1$ que es correcto. Tenemos un problema, un código cualquiera puede corregir algunas veces más errores que otras. El código que utilizamos como ejemplo puede corregir algunas veces hasta cuatro errores, pero no siempre, sería bueno determinar cuál es el mínimo número de errores que puede corregir, es decir cuantos errores podemos estar seguros de que puede corregir *siempre*. \triangleleft

Sabemos que el número de errores que puede detectar o corregir un código está en función de que tan diferentes, o para usar un lenguaje más propio, que tan distantes (en términos de Hamming) están sus palabras. Así que para determinar cual es el mínimo número de errores que puede detectar y/o corregir un código deberemos hablar de la mínima de las distancias entre sus palabras.

Definición 4.6 Sea C un código con, al menos, dos palabras. La *distancia mínima del código*, que denotaremos con $d(C)$ es la distancia más pequeña entre las distintas palabras del código:

$$d(C) = \min\{d(\mathbf{c}_i, \mathbf{c}_j) \mid \mathbf{c}_i, \mathbf{c}_j \in C, \mathbf{c}_i \neq \mathbf{c}_j\}$$

Evidentemente, dado que $\mathbf{c}_i \neq \mathbf{c}_j$ $d(C) \geq 1$.

Por cierto esta definición nos permite ser más precisos al describir un código.

Definición 4.7 Un código r -ario con M palabras diferentes, de longitud n y distancia mínima d es un (n, M, d) -código r -ario.

Ahora procederemos a definir formalmente que significa que un código detecte errores.

Definición 4.8 Sea u un entero positivo. Un código C es *detector de u errores* si cuando una palabra cualquiera $\mathbf{c} \in C$ es enviada a través de un canal y le ocurren a lo más u errores de símbolo, la palabra recibida x no pertenece a C . Un código es *detector de exactamente u errores* si es detector de u errores pero no de $u + 1$ errores.

Hay que hacer notar que una vez que una palabra es recibida, de alguna manera se compara con las que constituyen el código, si no empata con alguna podemos estar seguros de que ocurrieron errores, pero, como ya habíamos dicho, si empata con alguna no podemos saber si ocurrieron o no errores. Esto es equivalente a decir que una vez que ha sido recibida una palabra, el receptor nunca sabe cuantos errores ocurrieron durante su transmisión.

También tenemos una definición para código corrector de errores.

Definición 4.9 Sea v un entero positivo. Un código C es corrector de v errores si cuando una palabra cualquiera $\mathbf{c} \in C$ es enviada a través de un canal y le ocurren a lo más v errores de símbolo, la regla de decisión de vecino más cercano f mapea la palabra recibida en \mathbf{c} . Un código es *corrector de exactamente v errores* si es corrector de v errores pero no de $v + 1$ errores.

Recordemos nuestro ejemplo al inicio de esta sección. Si solo pretendemos detectar errores con ese código ¿cuantos podemos detectar? Supongamos que es enviada la palabra $\mathbf{c}_1 = 000000000$ y que es recibida la palabra $x_4 = 111100000$, en este caso podemos percatarnos de que han ocurrido errores. El receptor no sabe cuantos, de hecho si decodifica usando el vecino más próximo se equivocará suponiendo que le fue enviada \mathbf{c}_2 , pero si no decodifica y solo quiere reportar errores, en este caso podrá hacerlo bien. Si hubiera cinco unos al principio de la palabra recibida ya no podría hacerlo, es decir si hubieran ocurrido cinco errores. En ese caso la palabra recibida es una de las del código (\mathbf{c}_2) y no habría por qué dudar de que esa fuera la palabra enviada. Así que podemos detectar cuatro errores ¿será ese el mínimo número de errores detectables con el código del ejemplo? ciertamente lo es. En general si la distancia mínima de un código es d significa que las palabras más cercanas difieren en d posiciones y que si se cometen d errores justo en esas posiciones entonces no serán detectables, el número máximo de errores detectables de $d - 1$.

Ahora bien, el mínimo número de errores que puede corregir es 2. ¿Por qué? bueno, porque como vimos si la palabra recibida es $x_1 = 110000000$ todo sale bien, pero no cuando es $x_2 = 111000000$ y eso ocurre porque el tercer uno cambia la palabra de código que es vecino más cercano de la palabra. La distancia mínima del código es 5 y si observamos x_1 nos damos cuenta de que es una palabra que esta entre \mathbf{c}_1 , la palabra enviada, y \mathbf{c}_2 pero más cercana a la primera. Es decir si nos fijáramos en algo así como “el punto medio” entre \mathbf{c}_1 y \mathbf{c}_2 resulta que x_1 está en la mitad más próxima a \mathbf{c}_1 , su distancia a \mathbf{c}_1 es menor que la mitad de la distancia entre \mathbf{c}_1 y \mathbf{c}_2 , que por cierto es la distancia mínima del código. Cuando la palabra recibida es x_2 nos pasamos a la mitad dominada por \mathbf{c}_2 , cambia el vecino más cercano y por tanto la decodificación es errónea.

Con esto en mente podemos formalizar.

Lema 4.1 *Un código C es detector de u errores si y sólo si $d(C) \geq u + 1$.*

Dem.: \Rightarrow Supongamos que C es un código detector de u errores, eso significa que si se envía una palabra cualquiera $\mathbf{c} \in C$ y ocurren $k \leq u$ errores entonces la palabra recibida x no está en C . Dado que x se obtiene de \mathbf{c} alterando k posiciones cualesquiera, entonces la distancia de Hamming entre x y \mathbf{c} es tal que $d(x, \mathbf{c}) = k \leq u$. Dado que x no está en C eso significa que en el código no hay ninguna palabra que diste k unidades de \mathbf{c} (x puede ser cualquiera de las palabras que distan k de \mathbf{c}).

Lo anterior ocurre además para cualquier palabra enviada $\mathbf{c} \in C$, así que entre las palabras del código no hay ningún par que diste $k \leq u$ unidades, por lo que la distancia mínima del código debe ser $d(C) > u$, lo que podemos reescribir como $d(C) \geq u + 1$

\Leftarrow Supongamos ahora que tenemos un código C tal que $d(C) \geq u + 1$. Es decir, para toda pareja de palabras $\mathbf{c}, \mathbf{c}' \in C$ ocurre que $d(\mathbf{c}, \mathbf{c}') \geq u + 1$.

Sea $\mathbf{c} \in C$ una palabra cualquiera enviada y supóngase que ocurren $k \leq u$ errores por lo que es recibida una palabra x tal que $d(\mathbf{c}, x) = k \leq u$. Pero acabamos de concluir que en C no existe ninguna palabra \mathbf{c}' tal que: $d(\mathbf{c}, \mathbf{c}') \geq u$, así que $x \notin C$.

□

Lema 4.2 *Si \mathbf{c} es una palabra de un (n, M, d) -código C , tal que $d(C) > 2v$ con $2v \leq n$ y x es una cadena de longitud n tal que $d(\mathbf{c}, x) \leq v$ entonces \mathbf{c} es la palabra del código C más cercana a x .*

Dem.: Por reducción al absurdo.

Supongamos que tenemos un código C tal que $d(C) > 2v$. Sean $\mathbf{c} \in C$ y x tal que $d(\mathbf{c}, x) \leq v$. Supongamos también que existe otra palabra $\mathbf{c}' \in C$ tal que $d(\mathbf{c}', x) \leq v$ y $\mathbf{c} \neq \mathbf{c}'$.

Por la desigualdad del triángulo:

$$d(\mathbf{c}, \mathbf{c}') \leq d(\mathbf{c}, x) + d(x, \mathbf{c}') \leq v + v = 2v$$

Pero esto significa que hay dos palabras de C , \mathbf{c} y \mathbf{c}' que distan a lo más $2v$, lo que contradice que la distancia mínima del código es mayor que $2v$. □

Lema 4.3 *Un código C es corrector de v errores si y sólo si $d(C) \geq 2v + 1$*

Dem.: \Rightarrow Por reducción al absurdo.

Supongamos que C es un código de palabras de longitud n , corrector de v errores, es decir: siempre que al enviar una palabra se cometen v errores o menos, la decodificación de vecino más cercano entrega justo la palabra enviada. Supongamos también que: $d(C) < 2v + 1$, es decir: $d(C) \leq 2v$. Esto significa que existen dos palabras $\mathbf{c}_r, \mathbf{c}_t \in C$, tales que $d(\mathbf{c}_r, \mathbf{c}_t) \leq 2v$.

Como C es corrector de v errores entonces además debe ocurrir que $d(\mathbf{c}_r, \mathbf{c}_t) > v$, porque de otra forma \mathbf{c}_r podría transformarse en \mathbf{c}_t al ser enviada y sufrir a lo más v errores, lo que significa que los errores no podrían ser detectados y mucho menos corregidos.

Así que tenemos que hay dos palabras $\mathbf{c}_r, \mathbf{c}_t \in C$, tales que:

$$v < d(\mathbf{c}_r, \mathbf{c}_t) \leq 2v$$

Sea $m = d(\mathbf{c}_r, \mathbf{c}_t)$. Podemos reescribir la desigualdad como:

$$v + 1 \leq m \leq 2v \tag{4.3.6}$$

Podemos clasificar las n posiciones de las cadenas en dos subconjuntos:

- El subconjunto $D = \{i_1, i_2, \dots, i_m\}$ de $m = d(\mathbf{c}_r, \mathbf{c}_t)$ posiciones en las que \mathbf{c}_r difiere de \mathbf{c}_t .
- El subconjunto $E = \{1, 2, \dots, n\} \setminus D$ de las $n - m$ posiciones en las que \mathbf{c}_r coincide con \mathbf{c}_t .

Sea x una cadena con las siguientes características:

1. Coincide con \mathbf{c}_r en las posiciones del conjunto E . Es decir en todas las posiciones en que \mathbf{c}_r coincide con \mathbf{c}_t .
2. Difiere de \mathbf{c}_r en un subconjunto D_1 de v posiciones, contenidas en D (el conjunto en que \mathbf{c}_r difiere de \mathbf{c}_t).
3. Difiere de \mathbf{c}_t en un subconjunto D_2 de $m - v$ posiciones, contenidas en D .

Supongamos que en algún momento se envía la palabra \mathbf{c}_r a través de un canal y que, luego de v errores se recibe justamente la cadena x , que hemos definido arriba y que satisface $d(\mathbf{c}_r, x) = v$.

Por otra parte x difiere de \mathbf{c}_t en $m - v$ posiciones. Por la expresión 4.3.6 tenemos que:

$$1 \leq d(\mathbf{c}_t, x) = m - v \leq v$$

Tenemos entonces una palabra x que difiere de \mathbf{c}_r , la palabra enviada en v posiciones y que difiere de \mathbf{c}_t en, a lo más v posiciones. Así que la decodificación de vecino más cercano puede dar un resultado erróneo eligiendo \mathbf{c}_t en vez de la palabra enviada, lo que contradice que C sea un código corrector de v errores.

\Leftarrow Supongamos que C es un código tal que $d(C) \geq 2v+1$. Sea \mathbf{c} una palabra de C enviada y alterada por, a lo más v errores resultando x la cadena recibida. Entonces $d(\mathbf{c}, x) \leq v$. Por el lema 4.2 \mathbf{c} es la palabra de código más cercana a x y por tanto la decodificación de vecino más cercano será correcta.

□

Los lemas 4.1 y 4.3 los podemos sintetizar en el siguiente teorema.

Teorema 4.2 *Si C es un (n, M, d) -código entonces es detector de exactamente $d-1$ errores y corrector de exactamente $\lfloor \frac{d-1}{2} \rfloor$ errores.*

Dem.: Por el lema 4.1 tenemos que C es detector de u errores, donde $d \geq u+1$. Es decir $u \leq d-1$. Si es detector de exactamente u errores significa que no es detector de $u+1$ así que u adquiere su valor máximo, es decir $u = d-1$.

Por el lema 4.3 tenemos que C es corrector de v errores donde $d \geq 2v+1$, de donde:

$$v \leq \frac{d-1}{2} \tag{4.3.7}$$

Para que sea corrector de exactamente v errores no debe ser corrector de $v+1$ así que v debe tener el valor más grande posible que satisfaga 4.3.7, es decir:

$$v = \left\lfloor \frac{d-1}{2} \right\rfloor$$

□

Ejemplo 4.6 Un método históricamente importante para detectar errores en la transmisión o almacenamiento de información es el de añadir bits de paridad a los datos.

Supongamos que nuestro código de n bits consiste en todos los números expresables en binario en n bits. La distancia mínima del código es entonces 1 (la distancia entre el 0 y el 1 es 1, por ejemplo). Así que nuestro código puede detectar $1-1=0$ errores.

000	000 0
001	001 1
010	010 1
011	011 0
100	100 1
101	101 0
110	110 0
111	111 1

Tabla 4.1: Esquema de paridad par. A las palabras en el código original (columna izquierda) se les añade un bit adicional para completar siempre un número par de unos en cada palabra del nuevo código (columna derecha). Con negrita se denota el bit de paridad agregado.

Los esquemas de añadir bit de paridad a cada palabra son dos posibles: *paridad par* y *paridad impar*. En ambos esquemas se cuentan el número de unos en la palabra a enviar, este número de unos puede ser par o impar y dependiendo de ello se añade un bit adicional en 1 o en 0. En el esquema de paridad par se añade un 1 a todas las palabras que en el código original tienen un número impar de unos y un cero a las restantes, el objetivo es completar un número par de unos en toda palabra del código con el bit añadido. En el esquema de paridad impar el objetivo es el opuesto, se completa un número impar de bits en todas las palabras del nuevo código.

Supongamos que las palabras que deseamos enviar son las de la primera columna de la tabla 4.1. Si bien la distancia en el código original era uno, en el código con bit de paridad la distancia mínima es dos, lo que significa que ya es posible detectar $2 - 1 = 1$ bit erróneo por cada palabra de 4 bits.

El esquema de verificación de paridad era muy usual en las comunicaciones digitales durante las décadas pasadas y aún hoy se utiliza para conexiones por módem en una línea telefónica. También las antiguas unidades de cinta magnética utilizaban este esquema aunque con una variante que permite ya no solo detectar sino corregir algunos errores

Supongamos que utilizamos el código mostrado en la tabla ya referida (4.1). Cada palabra posee su bit de verificación. Ahora añadimos una palabra completa luego de cada tres palabras del código, de tal forma que cada bit de la palabra añadida verifique los bits verticalmente tal como lo hacen los bits de paridad de cada palabra, como se muestra en la figura 4.6, en la que hay tres palabras a enviar a las que se añade una cuarta cuyos bits verifican verticalmente los de cada palabra en una posición determinada. Supongamos que en el bloque mostrado en la figura, el tercer bit más significativo de la primera palabra fuera alterado quedando: 0111. Al recibir esta palabra nos percataríamos de que es errónea y

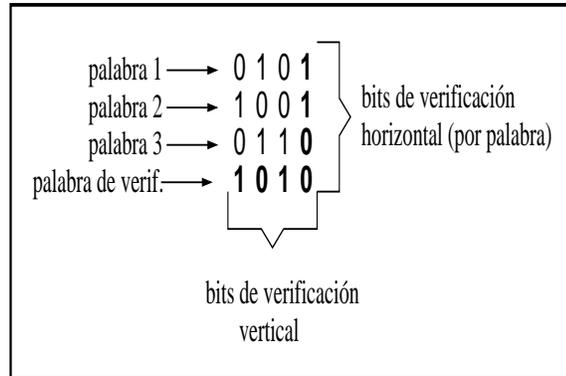


Figura 4.2: Verificación de paridad horizontal y vertical

más tarde cuando recibiéramos la palabra de verificación: 1010 nos daríamos cuenta de que hay un error en el tercer bit más significativo, tenemos las coordenadas: primera palabra, tercer bit y procedemos a negar el bit que se encuentre en esa posición, lo que corregiría el error.

Por supuesto si se cometen dos errores en el renglón y dos en una misma columna entonces el error en la intersección pasa desapercibido.

◁

Ejemplo 4.7 Otro ejemplo de código detector de errores es el llamado *International Standard Book Number* o ISBN. El ISBN es un $(10, 10^9, 2)$ -código por lo que puede detectar un error en alguno de sus dígitos, que por cierto son decimales. Para comprender este código pongamos un ejemplo: El libro de Baylis que aparece en las referencias [3], tiene el ISBN 0 – 412 – 78690 – 7. El cero inicial significa que el libro fué producido en los Estados Unidos de Norteamérica, el 412 que le sigue es el código de la editorial (Chapman & Hall) los siguientes cinco dígitos son el código del libro en la editorial y el último dígito es de verificación. Para calcularlo se ejecuta el algoritmo siguiente:

1. Para cada dígito i del código ISBN del libro (salvo el último $i = 10$ que es el que queremos calcular) sumar el valor de los anteriores $i - 1$ dígitos. Llamemos a este número s_i .
2. Calcular $S = \sum_{i=1}^9 s_i$.
3. Encontrar el valor de x más pequeño tal que: $S + s_9 + x \equiv 0 \pmod{11}$. es decir $S + s_9 + x$ debe ser el múltiplo más pequeño posible de 11.

Dígito	Suma	Suma de suma
0	0	0
4	4	4
1	5	9
2	7	16
7	14	30
8	22	52
6	28	80
9	37	117
0	37	154

Tabla 4.2: Cálculo del último dígito de un código ISBN. En la columna izquierda se muestran los primeros 9 dígitos del código, en la columna de en medio la suma acumulada parcial de los dígitos y en la columna derecha la suma acumulada parcial de las sumas de la columna previa.

El proceso completo para el libro de Baylis se muestra en la tabla 4.2. En síntesis se requiere el valor más pequeño de x tal que $191 + x$ sea múltiplo de 11, el múltiplo de 11 más cercano a 191 es 198, por lo que $x = 7$ es la solución y el valor del último dígito del código ISBN. Nótese que la operación es módulo 11, así que un posible residuo (valor para el último dígito) es 10, para escribir el 10 en un sólo dígito se representa en romano con una “X”.

◁

4.4 Códigos maximales

Recordemos el código analizado en el ejemplo 4.5. Ese código es capaz de corregir, ocasionalmente, más errores de los que garantiza que puede corregir siempre (como cuando la palabra recibida es x_3 y la enviada es \mathbf{c}_1 , en este caso se pueden corregir tres errores, lo que es superior a los dos garantizados). Esto no ocurriría si en el código estuviera incluida la palabra $\mathbf{c}_4 = 0000011111$, en ese caso x_3 sería decodificada como \mathbf{c}_4 y no como \mathbf{c}_1 . Vamos a especificar bajo que condiciones un código puede corregir solamente el número de errores garantizados y nunca más que esos.

Definición 4.10 Un (n, M, d) -código C es *maximal* si no está contenido en código más grande con la misma distancia mínima. Es decir si no existe un $(n, M + 1, d)$ -código, C' tal

que $C \subset C'$.

Teorema 4.3 *Un (n, M, d) -código C , sobre un alfabeto A es maximal si y sólo si para toda $x \in A^n \setminus C$ existe $\mathbf{c} \in C$ tal que $d(x, \mathbf{c}) < d$.*

Dem.: \Rightarrow Por reducción al absurdo.

Sea C un (n, M, d) -código maximal y supongamos que existe $x \in A^n \setminus C$ tal que para toda $\mathbf{c} \in C$ ocurre que $d(x, \mathbf{c}) \geq d$.

Si esto es cierto entonces el código $C' = C \cup \{x\}$ es un (n, M_1, d) -código, ya que la distancia mínima no se vería afectada por la inclusión de x .

Entonces $C \subset C'$ por lo que C no es maximal, lo que contradice nuestra hipótesis.

\Leftarrow Sea C un (n, M, d) -código. Supongamos que para toda $x \in A^n \setminus C$ existe una palabra del código $\mathbf{c} \in C$ tal que $d(x, \mathbf{c}) < d$. Esto significa que la inclusión de cualquier palabra x que no esté en el código reduce la distancia mínima de C , porque en el código ya sabemos que existe alguna \mathbf{c} tal que su distancia a x es menor que el mínimo actual d .

□

Tener códigos maximales tiene su ventaja y su desventaja. La ventaja es que en un código maximal tenemos el mayor número posible de cadenas con una distancia mínima dada. Entonces tenemos el acervo más grande posible de palabras y por tanto el poder máximo de representatividad del código. La desventaja es que con un código no maximal ocasionalmente podemos detectar/corregir un número mayor de errores de los que indica la distancia mínima.

4.5 Probabilidad de error al decodificar

Por lo visto en la sección antepasada ahora ya sabemos exactamente cuantos errores es capaz de detectar o corregir un código en función de su distancia mínima. Ahora bien ¿qué tan útil es realmente un código capaz de corregir v errores? por supuesto la respuesta depende de la tasa de error del canal que se este utilizando. Nuestros modelos de canal son probabilísticos, así que necesariamente la medida de la utilidad de un código en particular estará dada también en términos de probabilidad, concretamente, en términos de la probabilidad de que al decodificar no acertemos a la palabra que fue enviada. Nos avocaremos a calcular dicha probabilidad.

La probabilidad de que ocurran $k \leq n$ errores en una palabra de longitud n transmitida por un canal simétrico binario con probabilidad de error p es:

$$P_k = \binom{n}{k} p^k (1-p)^{n-k}$$

(el número de posibles selecciones de k posiciones de un total de n que tiene la palabra por la probabilidad de que esas k posiciones sean erróneas y las restantes $n - k$ no).

La ocurrencia de error en una posición de la palabra es un evento independiente, por lo que la probabilidad de que ocurran d o más errores en una palabra queda expresado en el lado derecho de la siguiente expresión:

$$P(\text{error decodificación}) \geq \sum_{k=d}^n \binom{n}{k} p^k (1-p)^{n-k} \quad (4.5.10)$$

lo que queremos expresar con el lado izquierdo de la expresión es la probabilidad de que al decodificar no se obtenga la palabra enviada, es decir la probabilidad de error al decodificar, lo que en un código de distancia mínima d ocurre cuando el número de errores excede la distancia mínima, de allí los límites de la suma.

Por otra parte un (n, M, d) -código puede corregir hasta $\lfloor \frac{d-1}{2} \rfloor$ así que no hay error al decodificar una palabra que tenga $\lfloor \frac{d-1}{2} \rfloor$ o menos errores.

$$P(\text{decodificar correctamente}) \geq \sum_{k=0}^{\lfloor \frac{d-1}{2} \rfloor} \binom{n}{k} p^k (1-p)^{n-k}$$

de donde:

$$P(\text{error decodificación}) \leq 1 - \sum_{k=0}^{\lfloor \frac{d-1}{2} \rfloor} \binom{n}{k} p^k (1-p)^{n-k} \quad (4.5.12)$$

Es posible sintetizar 4.5.10 y 4.5.12 en un teorema, que de hecho acabamos de demostrar.

Teorema 4.4 *Para un canal simétrico binario y usando decodificación de vecino más cercano, la probabilidad de error al decodificar una palabra, esto es, la probabilidad de que la palabra decodificada no sea la enviada, satisface:*

$$\sum_{k=d}^n \binom{n}{k} p^k (1-p)^{n-k} \leq P(\text{error decodificación}) \leq 1 - \sum_{k=0}^{\lfloor \frac{d-1}{2} \rfloor} \binom{n}{k} p^k (1-p)^{n-k} \quad (4.5.13)$$

4.6 Códigos de Hamming: idea e implementación

Los códigos de Hamming fueron descubiertos en 1949 por Golay y en 1950 por Richard Hamming. La distancia mínima de estos códigos siempre es 3. De donde determinamos fácilmente, por lo visto en este capítulo, que son detectores de hasta dos errores y correctores de un error (un bit erróneo) por palabra. Estos códigos los retomaremos un poco más adelante y con mayor generalidad más adelante, cuando sepamos algo de códigos lineales, pero es conveniente entrar en contacto con ellos.

La idea central de los códigos de Hamming es, como explica el mismo Hamming en [7], añadir a cada palabra de longitud n tantos bits como sea necesario para poder expresar cualquier número entre 0 y $n + 1$, de tal forma que sea posible expresar en esos bits el índice del bit equivocado, suponiendo que solo sea uno. Para hacer esto se generaliza el concepto de bit de paridad que ya revisamos. Por cada palabra se añaden varios bits de paridad, cada uno de ellos verifica la paridad de un subconjunto de bits de la palabra. Estos subconjuntos tienen intersecciones no vacías pero ninguno de ellos puede ser obtenido sumando algunos otros.

Si un bit de paridad, por ejemplo, verifica las posiciones 2, 3 y 5 de una palabra, otro verifica las posiciones 1, 3 y 7, y otro más las posiciones 3, 4 y 6, y los tres bits de paridad dicen que alguno de los bits que verifican está mal, evidentemente el bit erróneo es el 3. Este es el objetivo de que los conjuntos verificados por cada bit de paridad no sean ajenos. Pero si por ejemplo, tenemos un bit que verifica las posiciones 1, 3, 4 y 5, otro que verifica las posiciones 2, 3, 4, y 6 y un tercero que verifica las posiciones 1, 2, 5 y 6; entonces el tercer bit es completamente inútil, no proporciona información adicional, si sumamos las posiciones verificadas por los otros dos obtenemos las del tercero.

Nos avocaremos a describir un ejemplo sencillo de código de Hamming, el de palabra de longitud siete. ¿Cuántos bits son necesarios para decir cualquier número entre 0 y 7 inclusive? tres. Así que en el código de Hamming de longitud 7, tres de los bits son de verificación de paridad y por tanto los cuatro restantes son utilizados para almacenar el dato verdadero que se quería enviar o almacenar. El proceso en general es el siguiente:

1. El emisor recibe entonces una cuarteta de bits a enviar o almacenar, con base en estos bits calcula tres bits más, los bits de paridad.
2. El emisor forma una palabra de siete bits intercalando los bits de paridad en la cuarteta de bits de datos como veremos más adelante. Esta es la palabra de código.
3. Se envía la palabra de código de siete bits a un receptor.
4. El receptor recibe los siete bits y procede a hacer el cálculo de un número de tres bits

(estos tres bits no son los mismos que los de paridad ya mencionados, solo dependen de ellos). Este número de tres bits es llamado *síndrome*.

5. El síndrome, en el caso particular de los códigos de Hamming, resulta ser: cero si no se detectó ningún error (la palabra recibida es del código) o bien, si está entre 1 y 7, el índice del bit erróneo en caso de que haya ocurrido sólo un error.
6. Con base en el valor del síndrome se procede a negar el bit de la palabra recibida indicado por él o bien nada.
7. Se recuperan los cuatro bits de datos de la palabra recibida y corregida.

En la tabla 4.3 aparecen todos los posibles síndromes de tres bits. Salvo el 000 los demás son índices de la palabra de siete bits, que señalan el bit erróneo. Prestemos atención al bit menos significativo de los números que aparecen en la tabla (este es el procedimiento adoptado por Hamming en [7]), este bit está “prendido” en los índices 1, 3, 5 y 7, es decir, cuando el bit erróneo es el 1, 3, 5 o 7 de la palabra debe valer 1 el bit menos significativo del síndrome. Si ahora nos fijamos en el bit de en medio notaremos que está prendido en los índices 2, 3, 6 y 7, así que cuando el bit erróneo sea alguno de esos debe prenderse el segundo bit del síndrome. Siguiendo el mismo procedimiento vemos que el bit más significativo del síndrome debe prenderse cuando el bit erróneo sea el 4, 5, 6 o 7.

Perfecto, ahora sabemos que posiciones deben ser verificadas por cada bit de paridad. Pondremos un bit de paridad en la posición 1 de la palabra (es decir el bit más significativo de ella), ese bit de paridad completa un número par de unos considerando las posiciones 1, 3, 5 y 7. El segundo bit de paridad aparece en la posición 2 y hace que el número de unos de las posiciones 2, 3, 6 y 7. El último bit de paridad se pone en la posición 4 y completa un número par de unos en las posiciones 4, 5, 6 y 7.

Al momento de recibir una palabra de siete bits debemos obtener su síndrome: hay tres maneras de hacerlo, por supuesto las tres son equivalentes solo que dos de ellas son atajos para hacerlo fácil. Formalmente hablando hay que multiplicar la palabra recibida (escrita como vector columna, el bit más significativo arriba) por la matriz siguiente, conocida como $H_2(3)$ (matriz de Hamming binaria de tres renglones):

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \quad (4.6.14)$$

Esto es equivalente a calcular el bit de paridad que debería estar en la posición 1 (que corresponde a las posiciones 1, 3, 5 y 7), si es igual al que viene en la palabra recibida

Índice	Síndrome
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Tabla 4.3: Los posibles síndromes en un código de Hamming de siete bits.

entonces poner un cero como bit menos significativo del síndrome, si no poner 1. Calcular el de la posición 2 (posiciones 2, 3, 6 y 7), si coincide con el que viene en la palabra recibida poner cero en el segundo bit del síndrome, si no poner 1. Calcular el último (posiciones 4, 5, 6 y 7), y si coincide con el bit 4 de la palabra recibida poner 0 de bit más significativo del síndrome, de otro modo poner 1. Esto es equivalente a calcular el XOR de la terna de paridad calculada sobre la palabra recibida y la terna de paridad recibida. Lo que podemos abreviar usando el tercer método para calcular el síndrome.

En [1] los códigos correctores y detectores de errores se presentan desde un punto de vista práctico, siempre orientado a su implementación en circuitos digitales, así que para calcular el síndrome se presenta un circuito que calcula lo siguiente: Llamemos r a la palabra recibida considerada como vector binario, con $r[i]$ denotamos el i -ésimo bit de la palabra, el índice 1 corresponde al bit más significativo, el siete al menos significativo. Sea s el síndrome de tres bits considerado como vector.

$$\begin{aligned}
 s[1] &= r[4] \oplus r[5] \oplus r[6] \oplus r[7] \\
 s[2] &= r[2] \oplus r[3] \oplus r[6] \oplus r[7] \\
 s[3] &= r[1] \oplus r[3] \oplus r[5] \oplus r[7]
 \end{aligned}$$

Por cierto en [1] también se muestra un circuito para calcular la palabra enviada (a la que denotaremos con e y cuya convención de índices es la misma que ya usamos). Llamemos

c a la cuarteta de bits de datos que se quieren enviar.

$$\begin{aligned} e[1] &= c[1] \oplus c[2] \oplus c[4] \\ e[2] &= c[1] \oplus c[3] \oplus c[4] \\ e[3] &= c[1] \\ e[4] &= c[2] \oplus c[3] \oplus c[4] \\ e[5] &= c[2] \\ e[6] &= c[3] \\ e[7] &= c[4] \end{aligned}$$

Ejemplo 4.8 Supongamos que deseamos enviar 1101. Entonces la palabra enviada es: 1010101. Supongamos que se estropea el bit 6 de la palabra es decir se recibe la cadena 1010111. Por lo que el síndrome es:

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} = (110)$$

lo que por cierto es un seis escrito en binario. Para corregir el bit erróneo podemos hacer una máscara llena de cero, salvo el bit de índice especificado por el síndrome, es decir el sexto bit: 0000010 y hacer un XOR de esto con la palabra recibida, el XOR tiene la cualidad de que si se usa una máscara con unos en ciertas posiciones como operando, entonces el resultado del XOR es el otro operando con los bits que tiene prendidos la máscara negados. Así en este caso: $1010111 \oplus 0000010 = 1010101$ que es justo lo que nos enviaron. \triangleleft

4.7 Esferas y códigos perfectos

Hemos estado tratando con cadenas de código y luego, ayudados por la noción de distancia, adoptamos reglas de decisión que mapean cualquier cadena recibida de longitud n sobre nuestro alfabeto, a una cadena válida del código, la más cercana a la recibida. Es conveniente formular más formalmente estas nociones de “vecindad” de una palabra, un concepto que abarque todas las cadenas que distan de la palabra central en menos que una cierta distancia.

Definición 4.11 Sea \mathbf{x} una cadena en \mathbb{Z}_r^n y sea $\varrho \geq 0$. La *esfera* $S_r^n(\mathbf{x}, \varrho)$ con centro en \mathbf{x} y radio ϱ es el conjunto de todas las cadenas en \mathbb{Z}_r^n cuya distancia a \mathbf{x} es, a lo más, ϱ .

$$S_r^n(\mathbf{x}, \varrho) = \{\mathbf{y} \in \mathbb{Z}_r^n \mid d(\mathbf{x}, \mathbf{y}) \leq \varrho\}$$

Ejemplo 4.9 En \mathbb{Z}_2^3 la esfera $S_2^3(101, 2) = \{101, 001, 111, 100, 011, 000, 110\}$ ◁

Por supuesto podemos hablar del volumen de una esfera.

Definición 4.12 El volumen de una esfera $S_r^n(\mathbf{x}, \varrho)$ es el número de cadenas en ella.

Por supuesto el volumen de una esfera de un cierto radio no depende de la cadena en la que la esfera está centrada, sólo del radio, la longitud de las cadenas y el alfabeto. El número de cadenas r -arias que distan k unidades de otra es:

$$\binom{n}{k} (r-1)^k$$

Así que el volumen de una esfera de radio ϱ es:

$$V_r^n(\varrho) = \sum_{k=0}^{\varrho} \binom{n}{k} (r-1)^k$$

en \mathbb{Z}_2^n

$$V_2^n(\varrho) = \sum_{k=0}^{\varrho} \binom{n}{k}$$

Ahora tenemos un concepto útil: sabemos que un código tiene cierta capacidad de detectar y corregir errores. Que corrija errores significa que cuando una palabra es recibida luego de haber sido alterada por el canal, el resultado es una palabra diferente de aquella que fue enviada, si ocurrieron k errores durante la transmisión la palabra recibida estará a una distancia k de la palabra de código enviada, si k es menor que nuestro famoso número $v = \lfloor \frac{d-1}{2} \rfloor$ significa que ocurrió un número de errores que es posible corregir, la palabra recibida está en una esfera de radio menor a v , que como veremos tiene nombre especial.

Definición 4.13 Sea C un (n, M, d) -código r -ario. El radio de empaque de C , que denotaremos con $\text{Pr}(C)$ es el radio más grande posible para un conjunto de esferas disjuntas centradas, dada una, en una palabra de código. El conjunto de esferas $S_r^n(\mathbf{c}_i, \text{Pr}(C))$ ($\mathbf{c}_i \in C$), es llamado *las esferas de empaque para C* .

Por supuesto el valor del radio de empaque depende de la distancia mínima del código. Necesitamos encontrar el radio más grande posible tal que NINGUNA esfera de ese radio, $\text{Pr}(C)$, se interseque con otra. Habrá en el código dos palabras cuya distancia es la mínima de las distancias entre las palabras del código, cada una de ellas tendrá a su alrededor una esfera de radio $\text{Pr}(C)$ y deseamos que las esferas no se intersecten, así que el radio de empaque tiene que ser menor a la mitad de la distancia entre esas dos palabras. Si fuera exactamente la mitad o más, las esferas alrededor de la pareja de palabras de código más cercanas, se intersectarían.

Teorema 4.5 *El radio de empaque de un (n, M, d) -código es $\text{Pr}(C) = \lfloor \frac{d-1}{2} \rfloor$.*

Dem.: Sea d la distancia mínima de un código C , sean \mathbf{x} y \mathbf{y} las dos palabras de código más cercanas en C :

$$d = d(\mathbf{x}, \mathbf{y})$$

Si d es par el radio $d/2$ hace que las esferas se intersecten en una cadena en el punto medio. El radio de empaque de C debe ser $\lfloor \frac{d-1}{2} \rfloor$ con lo que las esferas alrededor de \mathbf{x} y \mathbf{y} no se intersectan (de hecho hay una palabra de código que no pertenece a ninguna de las dos esferas).

Si d es impar el radio de empaque debe ser $\frac{d-1}{2}$ con lo que ambas esferas permanecen disjuntas y no queda ninguna cadena fuera de esfera entre ambas.

Ambos casos quedan contemplados en $\text{Pr}(C) = \lfloor \frac{d-1}{2} \rfloor$. □

Tenemos ahora un nuevo lenguaje con el que se puede reformular un teorema ya conocido.

Corolario 4.1 *Un código es corrector de exactamente v errores si y sólo si $\text{Pr}(C) = v$.*

En la demostración del teorema ya vimos que es posible que algunas cadenas de \mathbb{Z}_r^n no estén contenidas en algunas de las esferas de empaque, podrían salirse muchas de ellas, si sale una entre las palabras más cercanas entonces sale al menos una por cada pareja. Cuando esto no ocurre, cuando el radio de empaque logra que las esferas contengan a todo \mathbb{Z}_r^n , se dice que el código es *perfecto*: el radio de empaque es impar y la distancia entre parejas de palabras de código es la misma.

Definición 4.14 Un (n, M, d) -código r -ario $C = \{\mathbf{c}_1, \dots, \mathbf{c}_M\}$ con alfabeto \mathbb{Z}_r es perfecto si las esferas de empaque son una partición de \mathbb{Z}_r^n .

Si las esferas de empaque son una partición entonces el volumen sumado de ellas debe ser la cardinalidad total de \mathbb{Z}_r^n .

$$M V_r^n \left(\left\lfloor \frac{d-1}{2} \right\rfloor \right) = r^n$$

o lo que es lo mismo:

$$M \sum_{k=0}^{\lfloor \frac{d-1}{2} \rfloor} \binom{n}{k} (r-1)^k$$

Ejemplo 4.10 Sea $C = \{00000, 11111\}$ a un código como este se le denomina *código de repetición* en este caso de longitud 5. En un alfabeto r -ario un código de repetición de longitud n , denotado $\text{Rep}_r(n)$, consta de r palabras, cada una con n repeticiones de cada uno de los símbolos del alfabeto. En nuestro caso $C = \text{Rep}_2(5)$. En nuestro caso $d(C) = 5$, (en general en un código de repetición la distancia mínima es la longitud de las palabras). La suma de las cadenas dentro de las esferas de empaque de radio $\lfloor \frac{5-1}{2} \rfloor = 2$ es:

$$2 \sum_{k=0}^2 \binom{5}{k} = 32$$

que es justamente el número total de cadenas binarias de longitud 5, por lo tanto el código de repetición $\text{Rep}_2(5)$ es perfecto. En general el código de repetición es perfecto cuando $r = 2$ y $n = 2m + 1$. \triangleleft

4.8 Equivalencia de códigos

Ejemplo 4.11 Sea $C = \{\mathbf{c}_1 = 00100, \mathbf{c}_2 = 00011, \mathbf{c}_3 = 11111, \mathbf{c}_4 = 11000\}$, C_1 es un $(5, 4, 3)$ -código binario.

También $D = \{\mathbf{d}_1 = 00000, \mathbf{d}_2 = 01101, \mathbf{d}_3 = 10110, \mathbf{d}_4 = 11011\}$ es un $(5, 4, 3)$ -código binario.

Así que, con lo que sabemos hasta ahora, en cualquier aplicación en la que es útil C es igualmente útil D . Tienen la misma capacidad de detección y corrección de errores, son equivalentes.

Formalmente hablando la noción de equivalencia de códigos es un poco más estricta, no es cierto que todos los (n, M, d) -códigos sean equivalentes. En particular nuestros códigos C y D realmente lo son. \triangleleft

Definición 4.15 Dos códigos son equivalentes si uno puede ser obtenido del otro por una combinación de operaciones de los dos tipos siguientes:

1. Permutar las posiciones de todas las palabras del código.
2. Dada una posición específica, aplicar una permutación a los símbolos en esa posición.

Probablemente al lector le ocurra lo que le ocurrió al autor al leer esta definición. Es difícil entenderla por primera vez, de hecho probablemente no se entienda hasta ver un ejemplo, pero podemos reformularla en otros términos [9].

Supongamos que tenemos un (n, M, d) -código r -ario C y que acomodamos cada una de las palabras del código en un renglón de una matriz de n columnas y M renglones a la que llamaremos \mathbf{M} . Las reglas del juego de equivalencia son las siguientes:

1. Cambiar el orden de las columnas de la matriz.
2. Dada una columna y una permutación de los símbolos en el alfabeto A del código (es decir una función biyectiva de A en A): aplicar la permutación a los elementos de la columna.
3. Cambiar un renglón por otro. Esta última regla no aparece arriba, pero es evidente, solo altera el orden en el que son listadas las palabras del código.

Ejemplo 4.12 Retomemos nuestro ejemplo anterior. Si acomodamos las palabras de C en una matriz obtenemos:

$$\mathbf{C} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Si cambiamos el renglón 3 por el 4 (la numeración de los renglones y de las columnas se comienza en 1), obtenemos:

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Ahora cambiamos la columna 2 por la 4:

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Ahora, aplicando la operación que puede parecer más extraña, invertimos todos los bits de la columna 3.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \end{pmatrix}$$

Por cierto los renglones de esta última matriz corresponden a las palabras en el código D del ejemplo anterior, lo que confirma que C y D son equivalentes.

Hay que notar que en el código C no tiene la palabra 0 y el código D sí.

◁

Teorema 4.6 *Si el alfabeto A de un código contiene el símbolo 0 entonces cualquier código sobre A es equivalente a uno que contiene la palabra $\mathbf{0}$.*

4.9 Tasa de transmisión y de corrección

¿Qué características son deseables en un código? es bueno tener una gran expresividad, poder decir muchas cosas, mientras más palabras de código tengamos mejor. Así que dada la longitud n de las palabras de un código r -ario nos gustaría tener un número de palabras cercano a r^n .

Por otra parte nos gustaría tener un código muy versátil, que pueda ser usado en una gran variedad de canales, con muchas tasas de error diferentes, esto es, nos gustaría tener un código que pueda corregir y/o detectar muchos errores, es decir, con una gran distancia mínima.

Por una parte entonces queremos tener muchas palabras de código, tantas como sea posible, mientras más nos acerquemos a A^n mejor. Pero por otra parte deseamos tener una gran distancia mínima, conforme más nos acerquemos a A^n la distancia mínima se reduce.

Así que hay un compromiso entre las dos cualidades deseables en un código: eficiencia y robustez.

Para medir ambas cosas de manera relativa a la longitud de las palabras necesitamos un par de definiciones:

Definición 4.16 La *tasa de transmisión* de un (n, M) -código C , r -ario es:

$$R(C) = \frac{\log_r M}{n} \leq 1$$

Nótese que $\log_r M$ nos dice que tantos símbolos, que longitud de palabra, es realmente necesaria para expresar M cosas en base r y n es la longitud de palabra que realmente tenemos.

Definición 4.17 La *tasa de corrección de error* de un (n, M, d) -código C , es:

$$\delta(C) = \frac{\lfloor \frac{d-1}{2} \rfloor}{n}$$

Número de errores que pueden corregirse en cada palabra entre el tamaño total de la palabra.

Ejemplo 4.13 Recordemos nuestros códigos de repetición, aquellos que solo tienen dos palabras una hecha puramente de 0 y otra de 1. El código binario de repetición de longitud n , $Rep_2(n)$ tiene las siguientes características:

$$R(Rep_2(n)) = \frac{1}{n}$$

para n impar, de la forma $n = 2u + 1$

$$\delta(Rep_2(n)) = \frac{u}{n} = \frac{1}{2 + \frac{1}{u}}$$

cuando n crece, u también lo hace y $\delta(Rep_2(n))$ tiende a $1/2$. Es decir, en el límite es capaz de corregir tantos errores como la mitad de la longitud de palabra. \triangleleft

Ahora podríamos preguntarnos: dados ciertos valores para la longitud de palabra, n , y para la distancia mínima, d , ¿cuál es el código más grande (con más palabras) que podemos

tener? n y d son los valores que determinan la eficiencia y la robustez de un código así que esos son los valores que le interesarían a cualquiera que diseñara un código.

Lo que deseamos encontrar es:

$$A_r(n, d) = \max\{M \mid \text{existe un } (n, M, d) \text{ - código } r \text{ - ario}\}$$

Este es uno de los problemas más importantes en teoría de códigos y por supuesto es tema de investigación permanente. Hasta ahora se conocen sólo unos pocos resultados, por ejemplo[18]:

$$A_2(4, 3) = 2$$

o

$$A_2(5, 3) = 4$$

Algunos otros resultados evidentes:

$$\begin{aligned} A_r(n, d) &\leq r^n \\ A_r(n, 1) &= r^n \\ A_r(n, n) &\leq r \end{aligned}$$

4.10 El teorema de la codificación con ruido

Al igual que el teorema de la codificación sin ruido, el de la codificación con ruido se lo debemos Claude Shannon. Apareció en el mismo trabajo de 1948 que constituye la tesis doctoral de Shannon. No nos daremos a la tarea de demostrarlo, solo lo enunciaremos y procuraremos motivarlo.

Supongamos que tenemos un $(24, 4096, 8)$ -código binario y que lo utilizamos sobre un canal simétrico binario con probabilidad de error de 0.1. El valor esperado de bits erróneos por cada palabra de código es 2.4 así que nuestro código es bastante útil, en promedio tendremos menos bits erróneos de los que nuestro código puede corregir. Si el canal tuviera una probabilidad de error de 0.13 entonces estaríamos en problemas, el valor esperado de bits alterados por palabra es de $0.13 \times 24 = 3.12$ lo que excede la capacidad del código para corregir errores (3). ¡Ah! pero en ese caso podríamos usar un código de Hamming. ¿Pero que dices insensato? dirá probablemente el lector: el código de Hamming tiene distancia mínima 3, puede corregir hasta 1 error. Ciertamente, *un error en una palabra de siete bits* es decir $1/7 = 0.1428$. Nuestro canal tiene una probabilidad de error de 0.13, es decir, en una palabra de siete bits se espera que haya $0.13 \times 7 = 0.91 < 1$ errores, así que resulta bastante útil.

De lo que estamos hablando aquí es, por una parte, de la ya mencionada tasa de transmisión de un código. Si esta es suficientemente baja entonces nuestro código es útil. Por otra parte estamos hablando de que tan ruidoso es el canal. Para continuar debemos definir la información mutua entre dos símbolos.

Definición 4.18 La información mutua de los símbolos a_i y b_j , $I(a_i | b_j)$ es:

$$I(a_i, b_j) = \log_2 \left(\frac{1}{p(a_i)} \right) - \log_2 \left(\frac{1}{p(a_i | b_j)} \right)$$

Hay que notar que $I(a_i, b_j)$ es tanto más grande cuanto más pequeño sea $\log_2 \left(\frac{1}{p(a_i | b_j)} \right)$, es decir cuanto más grande sea $p(a_i | b_j)$, es decir, mientras más frecuente sea la ocurrencia de a_i luego de que ha ocurrido b_j , o sea, mientras más ligados estén a_i y b_j . Si casi siempre que ocurre b_j ocurre a_i , entonces $p(a_i | b_j)$ es grande y entonces cuando vemos que ha ocurrido b_j sabemos que es muy probable que haya ocurrido a_i . Otra manera de verlo es: la información mutua de a_i y b_j es el número de preguntas que hay que hacer, a partir de nada, para deducir a_i menos el número de preguntas que hay que hacer para deducir a_i si ya sabemos que ha ocurrido b_j . Si nos fijáramos en el árbol de decisión para determinar a_i , hay un cierto número de aristas que recorrer si partimos de la raíz, pero si ya sabemos que ha ocurrido b_j , si ya tenemos algo de información previa, probablemente ya llevemos algo del recorrido hecho y estemos parados en algún nodo cercano al asociado a a_i , el camino que resta, desde donde estemos hasta ese nodo es la información adicional que necesitamos para determinar a_i a partir de b_j , mientras más pequeño sea ese camino adicional significa que a_i y b_j están más ligadas, b_j nos da mucha información acerca de a_i .

Si nos fijáramos ahora en la información de cada posible cadena en Σ^n (donde Σ es el alfabeto), con base en la de sus símbolos constitutivos y luego nos fijáramos en el promedio de la información conjunta de dos cadenas cualesquiera, una de ellas en un código y la otra en todo A^n podemos pensar en:

$$I(A, B) = H(A) - H(A | B)$$

donde A está en el código y B puede o no estarlo.

Imaginemos ahora que B en la expresión anterior es una palabra recibida, está en Σ^n , y A es una palabra del código. Si el canal es casi completamente confiable, sin ruido, entonces las probabilidades condicionales son casi cero y por tanto $H(A | B)$ es casi cero y entonces $I(A, B)$ es casi $H(A)$ la entropía de la fuente original que produce los datos que entran al canal. Si en cambio el canal no es confiable $H(A | B)$ crece y $I(A, B)$ disminuye. $I(A, B)$ es entonces una manera de cuantificar que tan confiable es un canal, que tanto trasmite

íntegramente la información que entra en él. A esto se le llama la *capacidad del canal* cuando se toma el máximo.

Definición 4.19 La *capacidad de un canal* \mathcal{C} es:

$$C = \max_{\text{distrib. de entrada}} I(A | B)$$

donde A es una cadena de entrada y B es una cadena de salida.

Lo que hemos estado diciendo en esta sección es que, dado un canal con cierta capacidad, es posible corregir los errores que el canal introduce si el código usado para transmitir la información tiene suficiente redundancia, es decir, si la tasa de transmisión del código es suficientemente baja.

Por cierto, la capacidad de un canal arbitrario no es trivial de calcular. Pero para un canal simétrico binario con probabilidad de error p es:

$$\text{Cap}(p) = 1 + p \log_2(p) + (1 - p) \log_2(1 - p)$$

Teorema 4.7 (De la codificación ruidosa para un canal simétrico binario) Para un canal simétrico binario con probabilidad de error p y capacidad $\text{Cap}(p)$. Si $R < \text{Cap}(p)$, entonces para toda $\varepsilon > 0$ existen n suficientemente grande y un (n, M) -código C tal que la tasa de transmisión de C es, al menos, R y $P(\text{error de decodificación}) < \varepsilon$.

4.11 Códigos de Golay: idea e implementación

Más adelante retomaremos los códigos de Golay, pero por ahora veremos como implementar uno de ellos. El código de Golay que revisaremos es un $(24, 4096, 8)$ -código y se le utilizó para transmitir las fotografías a color enviadas por las sondas Voyager en la década de los 1980's.

El código está relacionado con la matriz \mathbf{A} siguiente, en la que hemos puesto . para

denotar un 0 por claridad:

$$\mathbf{A} = \begin{pmatrix} . & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & . & 1 & 1 & 1 & . & . & . & 1 & . \\ 1 & 1 & . & 1 & 1 & 1 & . & . & . & 1 & . & 1 \\ 1 & . & 1 & 1 & 1 & . & . & . & 1 & . & 1 & 1 \\ 1 & 1 & 1 & 1 & . & . & . & 1 & . & 1 & 1 & . \\ 1 & 1 & 1 & . & . & . & 1 & . & 1 & 1 & . & 1 \\ 1 & 1 & . & . & . & 1 & . & 1 & 1 & . & 1 & 1 \\ 1 & . & . & . & 1 & . & 1 & 1 & . & 1 & 1 & 1 \\ 1 & . & . & 1 & . & 1 & 1 & . & 1 & 1 & 1 & . \\ 1 & . & 1 & . & 1 & 1 & . & 1 & 1 & 1 & . & . \\ 1 & 1 & . & 1 & 1 & . & 1 & 1 & 1 & . & . & . \\ 1 & . & 1 & 1 & . & 1 & 1 & 1 & . & . & . & 1 \end{pmatrix} \quad (4.11.16)$$

Si con \mathbf{I}_{12} denotamos la matriz identidad de orden 12 entonces a la matriz que resulta de colocar unidas la identidad de orden 12 y la matriz 4.11.16 le llamaremos $\mathbf{G}_{24} = (\mathbf{I}_{12} \mid \mathbf{A})$, esta es la matriz de Golay de 24 columnas y 12 renglones.

Hay que notar que la matriz \mathbf{A} es simétrica (es igual a su transpuesta) y también la identidad lo es. Así que si en vez de poner la identidad a la izquierda y \mathbf{A} a la derecha pudiéramos la identidad arriba y la matriz \mathbf{A} abajo obtendríamos \mathbf{G}_{24}^t , la transpuesta de \mathbf{G}_{24} .

Si multiplicáramos un vector \mathbf{x} de 12 entradas binario, dispuesto como un renglón, por \mathbf{G}_{24}^t obtendríamos un vector columna \mathbf{c} de 24 entradas también binario. Esto es lo que hay que hacer para obtener el código de Golay \mathcal{G}_{24} de \mathbf{x} . Es decir, el código de Golay \mathcal{G}_{24} de una palabra \mathbf{x} es el producto $\mathbf{c} = \mathbf{G}_{24}^t \mathbf{x}$. Así que tomamos 12 bits de datos, los codificamos en 24 bits y los enviamos.

Codificar es fácil, pero decodificar ya no lo es tanto. El código de Golay que estamos revisando tiene distancia mínima 8, por lo que puede corregir hasta 3 errores. El procedimiento en general es el siguiente:

1. Determinar, a partir de la palabra recibida \mathbf{d} , de 24 bits, la palabra que se debió haber recibido \mathbf{c} , (o mejor dicho la que consideremos que es más probable que se nos haya enviado), también de 24 bits.
2. Buscar en una lista la palabra \mathbf{c} para obtener los 12 bits de datos reales.

Por supuesto lo difícil es hacer lo primero y para hacerlo necesitamos calcular el patrón probable de error. Es decir una palabra \mathbf{e} de 24 bits de longitud, con unos en las posiciones

de error probables y cero en las correctas, de tal forma que $\mathbf{e} \oplus \mathbf{d} = \mathbf{c}$. Para determinar el patrón de error se hace lo siguiente:

1. Se calcula el síndrome $\mathbf{s} = \mathbf{G}_{24}\mathbf{d}$
2. Si $w(\mathbf{s}) \leq 3$ (el número de unos en el síndrome es, a lo más, tres), entonces el patrón de error es \mathbf{s} desplazado 12 bits a la izquierda $\mathbf{e} = \mathbf{s} \ll 12$.
3. Si $w(\mathbf{s}) > 3$ entonces buscamos un índice $i \in \{0, \dots, 11\}$ tal que la columna (o renglón, al cabo es simétrica) i -ésima de la matriz \mathbf{A} difiera de \mathbf{s} en, a lo más, 2. Si existe dicha columna el patrón de error es $\mathbf{e} = ((\mathbf{s} \oplus \mathbf{A}[i]) \ll 12) + \mathbf{I}_{12}[i]$, donde $+$ denota el OR bit a bit.
4. Si no hubo tal columna obtenemos $\mathbf{s}_1 = \mathbf{A}\mathbf{s}$
5. Si $w(\mathbf{s}_1) \leq 3$ entonces $\mathbf{e} = \mathbf{s}_1$
6. Si $w(\mathbf{s}_1) > 3$ entonces buscamos un índice $i \in \{0, \dots, 11\}$ tal que la columna (o renglón) i -ésima de la matriz \mathbf{A} difiera de \mathbf{s}_1 en, a lo más, 2. Si existe dicha columna el patrón de error es $\mathbf{e} = (\mathbf{s}_1 \oplus \mathbf{A}[i]) + (\mathbf{I}_{12}[i] \ll 12)$, donde $+$ denota el OR bit a bit.
7. Si esto no ocurre estamos perdidos, ocurrieron muchos errores.

5

Códigos lineales

5.1 Definición, características

Hasta ahora nuestro análisis general acerca de códigos para detectar y corregir errores no ha supuesto gran cosa acerca del conjunto de palabras del código. No hemos exigido ningún requisito a este conjunto, salvo el de que todas las palabras sean del mismo tamaño y que constituyan un espacio métrico (con la distancia de Hamming). Pero en este capítulo le daremos al conjunto de las palabras de código una estructura más interesante, con la que podemos hacer detección y corrección de errores de manera más eficiente.

Definición 5.1 Un código $C \subseteq \mathbb{Z}_p^n$ (con p primo) es un *código lineal* si C es un subespacio vectorial de \mathbb{Z}_p^n . Si C tiene dimensión k y distancia mínima d se dice que es un $[n, k, d]$ -código.

Por supuesto esta definición implica que \mathbb{Z}_p^n es un espacio vectorial, los escalares que estaremos usando son elementos de \mathbb{Z}_p y las operaciones (suma y producto) se realizan también sobre \mathbb{Z}_p . Un espacio vectorial tiene definidas operaciones de suma y producto por escalares, existe el neutro aditivo (vector cero, que denotamos $\mathbf{0}$) y el inverso aditivo de cualquier elemento, el producto por escalar es asociativo y distribuye a la suma. El hecho de que un código sea un subespacio vectorial significa que es, por sí mismo, un espacio vectorial, lo que significa que debe contener al vector cero.

Hay que notar que en la notación para códigos lineales no se pone la cardinalidad del código, solo su dimensión, es decir el número mínimo de elementos en una base del subespacio vectorial que constituye el código. Pero evidentemente estas dos cantidades están relacionadas.

Si se tienen k vectores en la base de un código, $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k\}$, cualquier otra palabra del código \mathbf{x} debe ser expresable como combinación lineal de los vectores de la base, es decir deben existir $\alpha_1, \alpha_2, \dots, \alpha_k \in \mathbb{Z}_p$ tales que:

$$\mathbf{x} = \alpha_1 \mathbf{x}_1 + \alpha_2 \mathbf{x}_2 + \dots + \alpha_k \mathbf{x}_k$$

como cada α_i está en \mathbb{Z}_p , hay p posibles valores para elegirla, así que en total hay $M = p^k$ palabras de código.

Hemos demostrado el siguiente teorema:

Teorema 5.1 *Un $[n, k, d]$ -código sobre \mathbb{Z}_p , es un (n, p^k, d) -código*

Ejemplo 5.1 $C_1 = \{0000, 1011, 0110, 1101\}$ en \mathbb{Z}_2^4 es un código lineal y una base para él es $B = \{1011, 0110\}$ así que es un $[4, 2]$ -código y un $(4, 4)$ -código. \triangleleft

Para determinar la distancia mínima de un código arbitrario hay que calcular la distancia entre cualesquiera dos palabras de él y seleccionar la mínima de estas distancias, es decir hay que calcular:

$$\binom{M}{2}$$

distancias. En un código lineal esta labor es más sencilla, basta con contar el número de unos (el peso), de la palabra distinta de la palabra (vector) cero, que menos unos tenga y ese número, llamado el peso del código, es la distancia mínima del mismo. Solo hay que calcular M pesos.

Definición 5.2 El peso de un código C , denotado $w(C)$, es el mínimo peso de las palabras distintas de cero en C .

Teorema 5.2 Si C es un código lineal $d(C) = w(C)$.

Dem.: Sea C un código lineal y sean $\mathbf{c}, \mathbf{d} \in C$ tales que $d(C) = d(\mathbf{c}, \mathbf{d})$. Como \mathbf{c} y \mathbf{d} están en un código lineal y este es un subespacio vectorial, entonces $\mathbf{c} - \mathbf{d} \in C$. Pero $\mathbf{c} - \mathbf{d}$ es una palabra que tiene cero en las posiciones en las que \mathbf{c} y \mathbf{d} coinciden y algo distinto de cero en las que difieren. Así que $\mathbf{c} - \mathbf{d}$ es una palabra del código y su peso es $w(\mathbf{c} - \mathbf{d}) = d(\mathbf{c}, \mathbf{d})$, por lo tanto:

$$d(C) = d(\mathbf{c}, \mathbf{d}) = w(\mathbf{c} - \mathbf{d}) \geq w(C)$$

Por otra parte como $w(C)$ es el mínimo peso de una palabra en C , debe existir una palabra \mathbf{e} en C tal que $w(\mathbf{e}) = w(C)$, entonces:

$$w(C) = w(\mathbf{e}) = w(\mathbf{e} - \mathbf{0}) = d(\mathbf{e}, \mathbf{0}) \geq d(C)$$

dado que la palabra $\mathbf{0}$ también está en el código y por tanto $\mathbf{e} - \mathbf{0}$ también. \square

5.2 La matriz generadora

Dado que un código lineal es un subespacio vectorial de dimensión k , entonces queda completamente descrito si se dan k vectores que constituyan una base del código. Es decir para especificar completamente un código lineal basta con dar k palabras de él y no las p^k que lo constituyen en su totalidad. El resto de las palabras son todas las combinaciones lineales de la base.

Definición 5.3 Sea C un código lineal con base $B = \{\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_k\}$ si $\mathbf{b}_i = b_{i1}b_{i2} \dots b_{in}$ denota la i -ésima palabra de la base, entonces la matriz de $k \times n$:

$$\begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \dots & \vdots \\ b_{k1} & b_{k2} & \dots & b_{kn} \end{pmatrix}$$

es la *matriz generadora* de C .

De nuestro curso introductorio de álgebra lineal sabemos que la cualidad de los elementos de una base de un espacio vectorial es que son vectores linealmente independientes, ninguno de ellos puede escribirse como combinación de los demás. Así que el siguiente teorema es evidente.

Teorema 5.3 Sea \mathbf{G} una matriz con elementos en \mathbb{Z}_p . \mathbf{G} es una matriz generadora de un código lineal si y sólo si las cadenas en los renglones de \mathbf{G} vistas como vectores son linealmente independientes.

Ejemplo 5.2 Los renglones de la matriz:

$$\begin{pmatrix} 0 & 1 & 2 & 1 \\ 2 & 2 & 1 & 0 \end{pmatrix}$$

son linealmente independientes sobre \mathbb{Z}_3 , así que forman una base para un $[4, 2]$ -código lineal ternario C . De hecho:

$$C = \{\alpha(0121) + \beta(2210) \mid \alpha, \beta \in \mathbb{Z}_3\}$$

◁

Dado un conjunto de vectores $V = \{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_s\}$ denotaremos como $\langle \mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_s \rangle$ el conjunto de todas las palabras que pueden expresarse como combinación lineal de los elementos de V , este es el *subespacio generado por V* .

Ejemplo 5.3 Sea $C = \langle 1101, 01101, 10100 \rangle$. Los vectores dentro de los paréntesis triangulares no son linealmente independientes, por ejemplo:

$$10100 = 1 \ 11001 + 1 \ 01101$$

así que no son una base. En cambio el conjunto $B = \{11001, 01101\}$ sí lo es, así que:

$$C = \langle 1101, 01101, 10100 \rangle = \langle 11001, 01101 \rangle$$

De ver la cardinalidad de B podemos decir que C es un código bidimensional, es decir es un $[5, 2]$ -código.

Su matriz generadora esta formada, como ya lo hemos dicho, por las palabras en una base de C (como las de B), por ejemplo:

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

De hecho: $C = \{00000, 11001, 01101, 10100\}$

◁

Algo importante que hay que notar porque lo usaremos con frecuencia, es el hecho de que un código lineal C está constituido por TODAS las combinaciones lineales de elementos en la base de C (que por cierto, como lo hemos dado a entender, no es única). Si sumamos dos elementos cualesquiera de C el resultado estará en C porque la suma también puede escribirse como combinación lineal de los elementos de la base. Si en cambio, sumamos dos elementos que no están en el código, o uno que sí está y otro que no, la suma tiene dos opciones:

1. No está en el código.
2. Sí está en el código y es la palabra $\mathbf{0}$.

Es decir, la única posibilidad de que la suma de dos elementos esté en el código, dado que al menos uno de los sumandos no está en él, es que la suma sea cero.

5.3 Corrección de errores

Ya sabemos que el método de decodificación de máxima verosimilitud es el de vecino más cercano. Así que dada una palabra recibida \mathbf{x} , ¿cómo encontramos la palabra del código más cercana a \mathbf{x} ? En un código general tendríamos que calcular la distancia de \mathbf{x} a todas y cada una de las palabras del código (en general sí, aunque podríamos detenernos antes si encontramos una palabra con distancia uno), eso significa calcular M distancias, donde M es el número de palabras en el código. En un código lineal la solución es más simple.

Comenzaremos con un ejemplo.

Ejemplo 5.4 Sea $C = \{0000, 1011, 0110, 1101\}$, entonces el complemento de C es el conjunto:

$$\overline{C} = \mathbb{Z}_2^4 \setminus C = \{1000, 0100, 0010, 0001, 0011, 0101, 1001, 1010, 1100, 1110, 0111, 1111\}$$

Nótese que hemos puesto los elementos ordenados por peso.

Construyamos ahora una tabla. En el primer renglón de la tabla ponemos a todos los elementos de C y luego, procedemos a poner los elementos de \overline{C} colocando el elemento de menor peso que podamos encontrar y que no haya sido incluido ya, al inicio de cada renglón. El resto del renglón lo determinamos sumando el elemento que encabeza la columna con el

que encabeza el renglón.

0000	1011	0110	1101
1000	0011	1110	0101
0100	1111	0010	1001
0001	1010	0111	1100

En la tabla hay 16 elementos, todos distintos, así que tenemos a todo \mathbb{Z}_2^4 . Además si elegimos a cualquier elemento de la tabla, el vecino más cercano a él que está en el código es el que encabeza la columna donde aparece. \triangleleft

A la tabla mostrada en el ejemplo anterior se le denomina el *arreglo estándar* del código. A cada renglón se le denomina *coset*¹ y al elemento inicial de cada renglón se le llama *líder del coset*.

Interesante, si tenemos un código lineal simplemente construimos su arreglo estándar como sigue:

1. El primer renglón del arreglo estándar son todas las palabras del código empezando por la palabra $\mathbf{0}$.
2. El primer elemento del renglón i -ésimo es aquel elemento de peso mínimo de entre aquellos que están en $\mathbb{Z}_p^n \setminus C$ y que aún no han sido incluidos en el arreglo.
3. El elemento en la columna j del i -ésimo renglón se calcula sumando el primer elemento del renglón (líder del coset) con el que encabeza la columna j (un elemento del código, por cierto).

y luego, cada vez que se recibe una palabra \mathbf{x} la buscamos en el arreglo y suponemos que en realidad nos fue enviada la palabra que encabeza la columna donde encontramos a \mathbf{x} .

Es interesante analizar el tercer renglón del arreglo estándar en nuestro ejemplo. Notará el lector que el tercer elemento del renglón es 0010 una palabra que, bien podría haber aparecido al inicio del renglón, o al inicio de algún otro renglón, fue solo por azares del destino que no fue elegida para encabezar un renglón dado que tiene el mismo peso que el resto de los líderes de coset. Esto engendra una ambigüedad, si recibimos 0010 decodificamos como 0110, pero bien podría ser 0000, al fin y al cabo la palabra recibida dista lo mismo de cualquiera de las dos posibilidades. Esta ambigüedad no puede ser evitada a menos que ningún elemento en un renglón i tenga el mismo peso que su líder. Más adelante veremos que se necesita para que esto ocurra.

¹Acrónimo de *Complete Ordered Set* un conjunto completamente ordenado: dados dos elementos cualesquiera podemos decir si uno es mayor o igual que el otro.

El arreglo estándar se genera sumando elementos del código con elementos que no están en él. Además los elementos que sumamos a los del código son siempre lo más “ligeros” posible. Decodificamos un elemento como el encabezado de su columna. Lo que estamos haciendo es suponer que en realidad nos fue enviado el elemento que encabeza la columna y que los errores que ocurrieron en el camino hicieron que esta palabra se modificara hasta quedar la cosa que recibimos. Para que esto ocurra debieron de ocurrir los errores justo en el patrón especificado por el líder del coset. Si lo analizamos en binario es más fácil de ver: supongamos que nos fue enviada la palabra 0110 con el código de nuestro ejemplo anterior y que recibimos 0111 eso significa que el patrón de error es 0001, la palabra que sumada a la enviada nos da la recibida.

Hay algunas características del arreglo estándar que conviene hacer notar:

1. Están todas las cadenas de \mathbb{Z}_p^n por construcción.
2. Como hay p^k columnas y en total deben ser p^n entradas, debe haber p^{n-k} renglones.
3. Como están todos los elementos de \mathbb{Z}_p^n y solo aparecen una vez entonces el conjunto de renglones constituye una partición de \mathbb{Z}_p^n , por supuesto el conjunto de columnas también. Cada elemento en una columna está en la misma clase de equivalencia del encabezado de la columna, equivalentemente cada elemento en un renglón está en la misma clase de equivalencia del líder del coset.

Teorema 5.4 *Dos palabras $\mathbf{x}, \mathbf{y} \in \mathbb{Z}_p^n$ están en el mismo coset si y sólo si $\mathbf{x} - \mathbf{y}$ es una palabra de código.*

Dem.: \Rightarrow Sean \mathbf{x} y \mathbf{y} dos palabras de $\mathbb{Z}_p^n \setminus C$ en el mismo coset. Es decir $\mathbf{x} = \mathbf{f}_i + \mathbf{c}_j$ y $\mathbf{y} = \mathbf{f}_i + \mathbf{c}_r$, nótese que \mathbf{f}_i , el líder de coset, es común a ambas expresiones.

Entonces:

$$\mathbf{x} - \mathbf{y} = \mathbf{c}_j - \mathbf{c}_r \in C$$

\Leftarrow Sean $\mathbf{x}, \mathbf{y} \in \mathbb{Z}_p^n \setminus C$ tales que $\mathbf{x} - \mathbf{y} \in C$. \mathbf{x} y \mathbf{y} deben estar en el arreglo estándar, así que las podemos escribir como $\mathbf{x} = \mathbf{f}_i + \mathbf{c}_j$ y $\mathbf{y} = \mathbf{f}_r + \mathbf{c}_s$. Entonces sabemos que:

$$(\mathbf{f}_i + \mathbf{c}_j) - (\mathbf{f}_r + \mathbf{c}_s) = \mathbf{c} \in C$$

de donde:

$$\mathbf{f}_i + \mathbf{c}_j - \mathbf{f}_r - \mathbf{c}_s = \mathbf{c} \in C$$

$$\mathbf{f}_i - \mathbf{f}_r = \mathbf{c} - \mathbf{c}_j + \mathbf{c}_s$$

el lado derecho de la igualdad está en C porque es resultado de operar con elementos de C . Así que $\mathbf{f}_i - \mathbf{f}_r$ está en C . Pero ningún líder de coset está en C por definición, así que estas \mathbf{f} 's no están en C . La única manera en que puede estar en C una suma (resta) de elementos que no están en C es que la suma sea cero, así que:

$$\mathbf{f}_i - \mathbf{f}_r = 0$$

de donde

$$\mathbf{f}_i = \mathbf{f}_r$$

por lo que \mathbf{x} y \mathbf{y} tienen entonces el mismo líder de coset y por tanto están en el mismo coset.

□

Ahora es oportuno recordar nuestro conocido $v = \lfloor \frac{d-1}{2} \rfloor$, el número de errores que un código con distancia mínima d puede corregir. Transportado a nuestro contexto actual significa que entre los líderes de coset deben estar aquellas palabras de peso menor o igual a v y si el código es perfecto entonces TODOS los líderes de coset tienen peso menor o igual a v . Por cierto si el código es perfecto no ocurre la ambigüedad mencionada a propósito del ejemplo, las esferas de empaque, que se ven como columnas del arreglo estándar, tienen intersecciones vacías.

5.4 Probabilidad de decodificar correctamente

Para códigos en general tenemos expresiones que acotan por arriba y por abajo la probabilidad de decodificar correctamente. En el caso de códigos lineales es posible obtener el valor exacto de esta probabilidad. Otra ventaja más de tener estructura.

Sea C un $[n,k]$ -código lineal con arreglo estándar:

$$\begin{array}{cccc} 0 & \mathbf{c}_2 & \dots & \mathbf{c}_M \\ \mathbf{f}_2 & \mathbf{f}_2 + \mathbf{c}_2 & \dots & \mathbf{f}_2 + \mathbf{c}_M \\ \vdots & \vdots & & \vdots \\ \mathbf{f}_q & \mathbf{f}_q + \mathbf{c}_2 & \dots & \mathbf{f}_q + \mathbf{c}_M \end{array} \quad (5.4.1)$$

Supongamos que tenemos un canal simétrico binario con probabilidad de error de símbolo p . Si un líder de coset tiene peso $w(\mathbf{f}_i)$ entonces la probabilidad de que una palabra de código tenga errores precisamente en las posiciones donde \mathbf{f}_i es distinto de cero es:

$$P(\mathbf{f}_i = error) = p^{w(\mathbf{f}_i)}(1-p)^{n-w(\mathbf{f}_i)} \quad (5.4.2)$$

Ahora podemos sintetizar esto en el siguiente teorema.

Teorema 5.5 Sea C un $[n, k]$ -código y sean $\mathbf{f}_1, \mathbf{f}_2, \dots, \mathbf{f}_q$ los líderes de coset del arreglo estándar de C . Si se transmite a través de un canal simétrico binario con probabilidad de error de símbolo p entonces la probabilidad de que una palabra recibida sea decodificada correctamente es:

$$P(\text{decodificar correctamente}) = \sum_{i=0}^q p^{w(\mathbf{f}_i)} (1-p)^{n-w(\mathbf{f}_i)} \quad (5.4.3)$$

Si contamos el número de líderes de coset de peso i y lo llamamos w_i entonces podemos reescribir 5.4.3 como:

$$P(\text{decodificar correctamente}) = \sum_{i=0}^n w_i p^i (1-p)^{n-i}$$

Ejemplo 5.5 Sea $C = \{0000, 1011, 0110, 1101\}$, el mismo código de nuestro ejemplo anterior. El arreglo estándar de este código es:

0000	1011	0110	1101
1000	0011	1110	0101
0100	1111	0010	1001
0001	1010	0111	1100

hay una sola palabra con peso cero, tres con peso uno y cero con peso cuatro entre los líderes de coset. Así que la suma 5.4.3 es:

$$P(\text{decodificar correctamente}) = 1(1-p)^4 + 3p(1-p)^3 = (1-p)^3(1+2p)$$

si $p = 0.01$

$$P(\text{decodificar correctamente}) = 0.9897$$

lo que resulta mayor que la probabilidad de decodificar correctamente si solo usamos los dos bits indispensables para decir cuatro cosas diferentes. En ese caso la probabilidad de decodificar correctamente es la de que no se cometa ningún error, es decir: $(1-p)^2 = 0.9801$. \triangleleft

En el caso de un código perfecto es aún más fácil. Sabemos que los líderes de coset son cadenas de peso $v = \lfloor \frac{d-1}{2} \rfloor$ o menor y hay exactamente $\binom{n}{i}$ cadenas de peso i en \mathbb{Z}_2^n así que el número de líderes de coset de peso i es:

$$w(i) = \binom{n}{i}$$

El número de líderes de coset de peso mayor a v es cero, así que en síntesis, para un código perfecto y un canal simétrico binario con probabilidad de *crossover* p :

$$P(\text{decodificar correctamente}) = \sum_{i=0}^{\lfloor \frac{d-1}{2} \rfloor} \binom{n}{i} p^i (1-p)^{n-i}$$

5.5 Complemento ortogonal, código dual

Así que el conjunto de todas las cadenas ortogonales a una cadena arbitraria cualquiera a en un espacio vectorial \mathbb{Z}_p^n es un código lineal. Entonces evidentemente el conjunto de las cadenas ortogonales a todas las cadenas de un código lineal es también un código lineal, aquel definido por la intersección de los subespacios ortogonales a cada una de las cadenas.

Teorema 5.6 *El complemento ortogonal A^\perp de un código A es un código lineal llamado el código dual de A .*

Sea $A = \{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_s\}$ un código lineal, entonces $A^\perp = \{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_s\}^\perp = \{\mathbf{a}_1\}^\perp \cap \{\mathbf{a}_2\}^\perp \cap \dots \cap \{\mathbf{a}_s\}^\perp$ es el código dual de A .

Hay que notar que el código dual de A está hecho de todas aquellas cadenas que son ortogonales simultáneamente a todas las cadenas en A . Es decir, un vector $\mathbf{x} = x_1x_2 \dots x_n$ está en el código dual de A si satisface:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n &= 0 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n &= 0 \\ &\vdots \\ a_{s1}x_1 + a_{s2}x_2 + \dots + a_{sn}x_n &= 0 \end{aligned} \tag{5.5.4}$$

donde $\mathbf{a}_1 = a_{11}a_{12} \dots a_{1n}$, $\mathbf{a}_2 = a_{21}a_{22} \dots a_{2n}$, ..., $\mathbf{a}_s = a_{s1}a_{s2} \dots a_{sn}$ son los vectores de A .

A las ecuaciones en 5.5.4 se les denomina *las ecuaciones de verificación de paridad de A^\perp* . Si pensamos en base 2 el nombre es evidente, si hay un número par de unos en las posiciones en las que un vector \mathbf{a}_i coincide con \mathbf{x} el resultado de la suma es 0.

Podemos reescribir el sistema de ecuaciones 5.5.4 en notación matricial:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{s1} & a_{s2} & \dots & a_{sn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \quad (5.5.5)$$

La matriz de $s \times n$ de la izquierda es llamada la matriz de verificación de paridad para A^\perp .

Teorema 5.7 Sea A un código y $\mathbf{x} = x_1x_2\dots x_n$ una cadena en \mathbb{Z}_p^n . \mathbf{x} está en A^\perp si y sólo si:

$$\mathbf{Ax} = \mathbf{0}$$

donde \mathbf{A} es la matriz de verificación de paridad para A^\perp .

Ejemplo 5.6 Sean $\mathbf{a}_1 = 1110$ y $\mathbf{a}_2 = 1001$ las dos cadenas de un código. Las ecuaciones de verificación de paridad para el código dual son:

$$\begin{array}{rcccc} x_1 + & x_2 + & x_3 + & & = 0 \\ x_1 + & & & x_4 & = 0 \end{array} \quad (5.5.6)$$

Las soluciones de la segunda ecuación son:

1. $x_1 = x_4 = 0$ en este caso hay dos soluciones para la primera:
 - (a) $x_2 = x_3 = 0$
 - (b) $x_2 = x_3 = 1$
2. $x_1 = x_4 = 1$ en este caso también hay dos soluciones para la primera:
 - (a) $x_2 = 1, x_3 = 0$
 - (b) $x_2 = 0, x_3 = 1$

en síntesis: $\{0000, 0110, 1101, 1011\}$

◁

Hay que notar que en la matriz de verificación de paridad de un código no necesariamente sus renglones son linealmente independientes, pero puede ocurrir.

Algunas cosas interesantes:

- Si C y D son dos códigos y $C \subseteq D$ entonces $D^\perp \subseteq C^\perp$
- Si \mathbf{G} es la matriz generadora de un $[n, k]$ -código C , entonces también es la matriz de verificación de paridad de C^\perp .
- Si C es un $[n, k]$ -código C^\perp es un $[n, n - k]$ -código (la dimensión del dual es $n - k$).
- Si C es un código lineal $C^{\perp\perp} = C$.

Ejemplo 5.7 $C = \{0000, 1011, 0110, 1101\}$ subespacio de \mathbb{Z}_2^4 , base $\beta = \{1011, 0110\}$, matriz generadora:

$$\mathbf{G} = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

Entonces C^\perp está determinado por las soluciones al sistema:

$$\begin{array}{rcccc} x_1 + & & x_3 + & x_4 & = & 0 \\ & x_2 + & x_3 & & = & 0 \end{array}$$

Las soluciones son $\{1110, 0111, 1001, 0000\} = C^\perp = \langle 1110, 1001 \rangle$

◁

Con lo que sabemos podemos afirmar:

- Cualquier matriz es la de verificación de paridad de algún código lineal.
- Cualquier matriz con renglones linealmente independientes es la matriz generadora de algún código y la de verificación de paridad de algún otro.
- Una matriz generadora para un código C es la de verificación de paridad para C^\perp .
- Cualquier código lineal C tiene una matriz de verificación de paridad. En particular, una matriz generadora para el código dual C^\perp es una matriz de verificación de paridad para C .

5.6 Decodificación de síndrome

En el capítulo anterior presentamos un caso particular de código de Hamming. Por cierto los códigos de Hamming son lineales. Recordará el lector que para decodificar determinamos

el índice del bit erróneo, en caso de haberlo, multiplicando una matriz por el vector (palabra) recibido. A este producto se le denomina síndrome como ya hemos señalado. En los códigos lineales en general es posible y útil calcular el síndrome.

Definición 5.4 Sea \mathbf{P} una matriz de verificación de paridad de un código $C \subseteq \mathbb{Z}_p^n$. El *síndrome* $S(\mathbf{x})$ de una cadena $\mathbf{x} \in \mathbb{Z}_p^n$, es el producto $\mathbf{P}\mathbf{x}$.

Esta definición nos hace pensar en que la matriz que usamos en el capítulo anterior al presentar el código de Hamming, la matriz $H_2(3)$, es la de verificación de paridad de el código de Hamming de siete bits, algo que debe parecernos claro, dado que nos permite calcular la paridad de tres diferentes subconjuntos de los siete bits de la palabra recibida.

Ejemplo 5.8 Recurriremos a nuestro ya muy usado código:

$$C = \{0000, 1011, 0110, 1101\}$$

Una matriz generadora para C es:

$$\mathbf{G} = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \quad (5.6.7)$$

y la matriz generadora de C^\perp y por tanto verificadora de paridad de C :

$$\mathbf{P} = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \quad (5.6.8)$$

Sea $\mathbf{x} = 0111$. El síndrome de esta palabra es:

$$S(\mathbf{x}) = \mathbf{P}\mathbf{x} = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

El síndrome es la cadena 01, nótese que la palabra 0111 no está en el código.

Si en cambio calculamos el síndrome de la palabra 1011 que sí está en el código:

$$S(\mathbf{x}) = \mathbf{P}\mathbf{x} = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

el resultado es la palabra 00.

◁

La curiosidad mostrada en el ejemplo es algo que siempre ocurre. El síndrome de una palabra del código siempre es cero y el de una palabra que no está en el código es distinto de cero. En el caso particular de los códigos de Hamming, además de no ser cero, el síndrome de una palabra que no está en el código nos dice cuál bit hay que modificar para transformarla en la más cercana a ella que sí está en el código, pero esto solo ocurre para los códigos de Hamming.

En el caso de los códigos de Golay, que también son lineales, la matriz de verificación de paridad es la \mathbf{G}_{24} y el síndrome nos indica directa o indirectamente el patrón de error, el líder del coset donde se encuentra la palabra recibida.

¿Por qué siempre ocurre que el síndrome de una palabra del código es cero? bueno, la matriz de verificación de paridad de un código C es equivalente² a la matriz generadora del código dual C^\perp , es decir el subespacio generado por la matriz generadora del dual y el generado por la matriz de verificación de paridad son el mismo y son, de hecho, el dual de C . Por definición el dual de C es el conjunto de todos los vectores ortogonales a todos los elementos de C , lo que significa que si efectuamos el producto interior de un vector en el dual de C con uno de C el resultado es $\mathbf{0}$. En particular los renglones de la matriz de verificación de paridad de C están en C^\perp , así que el producto interior de cualquier renglón de la matriz con un elemento de C es cero porque son ortogonales.

De nuestros conocimientos de álgebra lineal sabemos que $S(\mathbf{x})$ es, de hecho, una transformación lineal. Es decir:

1. $S(\mathbf{x} + \mathbf{y}) = S(\mathbf{x}) + S(\mathbf{y})$.
2. $S(\alpha\mathbf{x}) = \alpha S(\mathbf{x})$.

Estas características hacen del síndrome algo aún más útil. No solo sirve para saber si una palabra está o no en el código, nos puede dar más información.

Teorema 5.8 *Sea C un código lineal. Dos cadenas \mathbf{x} y \mathbf{y} están en el mismo coset si y sólo si tienen el mismo síndrome.*

Dem.: Sea C un código con arreglo estándar:

$$\begin{array}{cccc}
 0 & \mathbf{c}_2 & \dots & \mathbf{c}_M \\
 \mathbf{f}_2 & \mathbf{f}_2 + \mathbf{c}_2 & \dots & \mathbf{f}_2 + \mathbf{c}_M \\
 \vdots & \vdots & & \vdots \\
 \mathbf{f}_q & \mathbf{f}_q + \mathbf{c}_2 & \dots & \mathbf{f}_q + \mathbf{c}_M
 \end{array} \tag{5.6.9}$$

²Se puede llegar de la matriz de verificación de paridad a una generadora para C^\perp mediante reducción de Gauss-Jordan.

Sean $\mathbf{x} = \mathbf{f}_i + \mathbf{c}_r$ y $\mathbf{y} = \mathbf{f}_j + \mathbf{c}_s$. supongamos que:

$$S(\mathbf{x}) = S(\mathbf{y}) \quad (5.6.10)$$

Por una parte tenemos que:

$$S(\mathbf{x}) = S(\mathbf{f}_i + \mathbf{c}_r) = S(\mathbf{f}_i) + S(\mathbf{c}_r) = S(\mathbf{f}_i)$$

la última igualdad de la cadena se debe a que el síndrome de \mathbf{c}_r es cero dado que esta palabra está en el código.

Análogamente:

$$S(\mathbf{y}) = S(\mathbf{f}_j + \mathbf{c}_s) = S(\mathbf{f}_j) + S(\mathbf{c}_s) = S(\mathbf{f}_j)$$

Así que 5.6.10 es cierta si y sólo si:

$$S(\mathbf{f}_i) = S(\mathbf{f}_j)$$

de donde:

$$0 = S(\mathbf{f}_i) - S(\mathbf{f}_j) = S(\mathbf{f}_i - \mathbf{f}_j)$$

¡Ah! algo cuyo síndrome es cero, eso significa que el algo está en el código, es decir: $\mathbf{f}_i - \mathbf{f}_j \in C$.

Ahora tenemos una suma de dos elementos que no están en el código (recordemos que las \mathbf{f} 's son líderes de coset y no están en el código) y cuya suma sí está, como habíamos dicho la única posibilidad de que esto ocurra es que $\mathbf{f}_i - \mathbf{f}_j = \mathbf{0}$, de donde finalmente:

$$\mathbf{f}_i = \mathbf{f}_j$$

lo que significa que \mathbf{x} y \mathbf{y} están en el mismo coset. □

Así que dos elementos con el mismo síndrome están en el mismo coset del arreglo estándar, esto significa dos cosas útiles:

- Tienen el mismo líder de coset y por tanto el mismo patrón de error.
- Cada coset está caracterizado por el síndrome de sus elementos ya que es el mismo para todos. Así que no es necesario guardar todo el arreglo estándar, basta guardar la lista de los síndromes y los líderes de coset.

¡Excelente! Ahora para decodificar tendríamos que hacer lo siguiente:

1. En vez de almacenar todo el arreglo estándar del código C que se usará, simplemente se guarda una tabla con dos columnas: los líderes de coset y los valores de síndrome de cada coset.
2. Dada una cadena \mathbf{x} recibida, calculamos $S(\mathbf{x})$.
3. Buscamos $S(\mathbf{x})$ en la tabla hasta encontrar un líder de coset \mathbf{f}_i tal que $S(\mathbf{f}_i) = S(\mathbf{x})$.
4. \mathbf{f}_i es el patrón de error, así que decodificamos a vecino más cercano entregando $\mathbf{c} = \mathbf{x} - \mathbf{f}_i$.

Ejemplo 5.9 En nuestro ejemplo recurrente $C = \{0000, 1011, 0110, 1101\}$, el arreglo estándar es:

0000	1011	0110	1101
1000	0011	1110	0101
0100	1111	0010	1001
0001	1010	0111	1100

de donde obtendríamos la tabla:

Líder	Síndrome
0000	00
1000	11
0100	10
0001	01

Si recibimos $\mathbf{x} = 1110$ que no está en C , su síndrome es: $\mathbf{P}\mathbf{x} = 11$, en nuestra tabla el líder de coset con síndrome 11 es 1000 así que decodificamos: $1110 - 1000 = 0110$ que es la palabra que suponemos que nos fue enviada. \triangleleft

Hay una característica curiosa más en la matriz de verificación de paridad que nos será útil. Resulta que la distancia mínima del código es justamente el número mínimo de columnas de la matriz de paridad linealmente dependientes. Es decir: cualquier subconjunto de columnas con cardinalidad estrictamente menor a la distancia mínima del código es un conjunto linealmente independiente.

Teorema 5.9 Sea \mathbf{P} la matriz de verificación de paridad de un $[n.k.d]$ -código lineal, C . La distancia mínima, d , es el entero más pequeño r para el que hay r columnas linealmente dependientes en \mathbf{P} .

Dem.: El producto de matriz de verificación de paridad por un vector $\mathbf{c} = c_1 c_2 \dots c_n$ es:

$$\mathbf{P}\mathbf{c} = \begin{pmatrix} p_{11} & p_{12} & \dots & p_{1n} \\ p_{21} & p_{22} & \dots & p_{2n} \\ \vdots & \vdots & & \vdots \\ p_{q1} & p_{q2} & \dots & p_{qn} \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} = \begin{pmatrix} p_{11}c_1 + p_{12}c_2 + \dots + p_{1n}c_n \\ p_{21}c_1 + p_{22}c_2 + \dots + p_{2n}c_n \\ \vdots \\ p_{q1}c_1 + p_{q2}c_2 + \dots + p_{qn}c_n \end{pmatrix}$$

esto lo podemos reescribir como:

$$\mathbf{P}\mathbf{c} = c_1\mathbf{P}_1 + c_2\mathbf{P}_2 + \dots + c_n\mathbf{P}_n \quad (5.6.12)$$

donde \mathbf{P}_i es la i -ésima columna de \mathbf{P} .

Sea r el mínimo número de columnas linealmente dependientes. Es decir existe $\mathbf{c} = c_1, c_2 \dots c_n \in \mathbb{Z}_p$ tal que r de sus componentes son distintas de cero y:

$$\mathbf{P} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} = \mathbf{0} \quad (5.6.13)$$

Así que $w(\mathbf{c}) = r \geq w(C) = d$.

Por otra parte si $\mathbf{c} = c_1 c_2 \dots c_n \in C$ es de peso mínimo entonces $d = w(\mathbf{c}) = w(C)$. Dado que \mathbf{c} está en el código y \mathbf{P} es la matriz de verificación de paridad, ocurre 5.6.13; de donde se deduce que las d columnas que están multiplicando a los elementos de \mathbf{c} distintos de cero son linealmente dependientes (el producto es cero sin que todos ellos sean cero) así que $r \leq d$. \square

5.7 Códigos de Hamming y Golay revisados

Con el último teorema formulado podemos hacer un análisis de los códigos de Hamming en general.

Ya mencionamos que todo código de Hamming tiene una distancia mínima de 3. En términos de nuestro último teorema esto significa que existe un conjunto de tres columnas de la matriz de verificación de paridad de cualquier código de Hamming que resulta ser linealmente dependiente y ese es el mínimo número de columnas que es posible encontrar con esa propiedad. Lo que significa que cualquier conjunto de dos columnas de la matriz de verificación de paridad resulta ser linealmente independiente, es decir, cualquier pareja de columnas es linealmente independiente: ninguna columna es múltiplo de otra.

En un código de Hamming binario como el tratado en el capítulo anterior esta propiedad es evidente: la i -ésima columna es la representación binaria del número i , que no puede ser obtenida multiplicando por un bit cualquiera, alguna otra columna. En códigos de Hamming p -arios ya no es tan trivial.

Para $p = 3$ por ejemplo, podemos incluir la columna 001 si pretendemos construir una matriz de verificación de paridad de tres renglones, pero entonces no podemos incluir la columna 002 que se obtiene multiplicando por 2 la anterior. Si se incluye la columna 012 ya no puede ser incluida la 021 ya que se obtiene de 012 multiplicándola por 2. Si analizamos esto con cuidado nos percatamos de que es posible construir una matriz de verificación de paridad para un código de Hamming p -ario de h renglones si incluimos en ella sólo las columnas cuya posición más significativa sea 1. Por ejemplo $H_3(3)$ la matriz de Hamming ternaria de tres renglones:

$$H_3(3) = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 \\ 1 & 0 & 1 & 2 & 0 & 1 & 2 & 0 & 1 & 2 & 0 & 1 & 2 \end{pmatrix}$$

Dados p , la base del código y h el número de renglones, podemos determinar n , el número de columnas, es decir, la longitud de las palabras del código.

Supongamos que la posición más significativa de una columna es $m \in \{1, \dots, h\}$ (de arriba hacia abajo). Así que la posición m de la columna es 1, antes de eso, las primeras $m - 1$ posiciones son cero, y hacia abajo, las restantes $h - m$ posiciones pueden ser cualesquiera dígitos de \mathbb{Z}_p . Hay p posibilidades en cada una de esas posiciones, así que en total hay p^{h-m} columnas que pueden y deben estar en la matriz de verificación de paridad. Ahora sumemos sobre todos los posibles valores de m :

$$n = p^{h-1} + \dots + p^1 + p^0 = \frac{p^h - 1}{p - 1}$$

Ahora bien ¿cuántas palabras tiene el código? la longitud de cada palabra es n , la que acabamos de calcular, el número de bits de verificación que se incluyen en una palabra es justamente el número de renglones en la matriz de verificación de paridad, h . Así que hay en total p^{n-h} palabras de código, esto es un subespacio de dimensión $n - h$. Hemos demostrado el siguiente teorema.

Teorema 5.10 *La matriz de Hamming en base p y con h renglones es la matriz de paridad de un $[n, k, d]$ -código lineal con parámetros: $k = n - h$, $d = 3$ y*

$$n = \frac{p^h - 1}{p - 1}$$

Por supuesto cualquier código de Hamming puede corregir hasta un error exactamente, como habíamos dicho. En el caso binario los parámetros se convierten en: $n = 2^h - 1$, $k = n - h$ y $d = 3$.

Por cierto en el caso general el síndrome de una palabra errónea en un código de Hamming en base $p \neq 2$ no es el índice del símbolo erróneo, eso sólo funciona en el caso binario, pero sí proporciona un múltiplo de la posición, solo hay que buscar una columna de la matriz de verificación tal que el síndrome sea un múltiplo de ella y el índice de esa columna sí es la posición del error.

Los códigos de Hamming son perfectos y únicos, en el sentido de que cualquier código con parámetros de la forma especificada en el teorema anterior resulta ser equivalente a un código de Hamming.

En lo que respecta al código de Golay presentado en el capítulo anterior, \mathcal{G}_{24} hay que añadir que es un código auto-dual, es decir $\mathcal{G}_{24} = \mathcal{G}_{24}^\perp$, verificar esto implica mucho trabajo porque hay que calcular el producto interior de cualesquiera dos renglones distintos de \mathbf{G}_{24} . Con lo que se concluye que $\mathcal{G}_{24} \subseteq \mathcal{G}_{24}^\perp$. Luego, dado que la dimensión de ambos subespacios debe ser 12, entonces deben ser el mismo subespacio. Además como la matriz \mathbf{G}_{24} es la de verificación de paridad de \mathcal{G}_{24} y este es igual a su dual entonces la matriz es también su generadora. \mathcal{G}_{24} no es un código perfecto, pero su hermano cercano con palabras de 23 bits \mathcal{G}_{23} sí lo es.

Cualquier renglón de la matriz \mathbf{G}_{24} tiene un peso divisible por 4, de hecho cualquier palabra de \mathcal{G}_{24} tiene la misma propiedad. \mathcal{G}_{24} es un $[24, 12, 8]$ -código lineal binario. Eso significa que puede corregir hasta 3 errores.

INTERMEZZO B

Detección y corrección de errores

B.1 Códigos de Reed-Muller

En las misiones Mariner a Marte entre 1969 y 1977 se tomaron fotografías en blanco y negro, formalmente hablando en 32 diferentes tonos de gris, luego de digitalizar estas imágenes había que transmitir las a la tierra, lo que significaría que la transmisión sería afectada probablemente por el viento solar. Para asegurar que, por lo menos, algunos de estos errores fueran corregidos se utilizó un código de Reed-Muller binario de primer orden para codificar cada pixel de las imágenes.

En general los códigos de Reed-Muller binarios de primer orden son $(2^m, 2^{m+1}, 2^{m-1})$ -códigos. El que se utilizó en las misiones Mariner fue un $[32, 6, 16]$ -código binario lineal. Como el código queda completamente descrito dada m , los códigos de Reed-Muller suelen denotarse con un solo número, en esa notación el usado por la NASA es el código $\mathcal{R}(5)$.

La construcción de los códigos de Reed-Muller es muy sencilla, está definida inductiva-

mente.

Definición B.1 El código de Reed-Muller binario $\mathcal{R}(m)$ está definido como sigue:

1. $\mathcal{R}(1) = \{00, 01, 10, 11\}$ que son todas las cadenas binarias de longitud dos.
- 2.

$$\mathcal{R}(m) = \{\mathbf{c}\mathbf{c} \mid \mathbf{c} \in \mathcal{R}(m-1)\} \cup \{\mathbf{c}\bar{\mathbf{c}} \mid \mathbf{c} \in \mathcal{R}(m-1)\}$$

cada cadena de $\mathcal{R}(m)$ es una de $\mathcal{R}(m-1)$ concatenada consigo misma o con su negación.

Ejemplo B.1 Según nuestro procedimiento de construcción:

$$\mathcal{R}(2) = \{0000, 0011, 0101, 0110, 1010, 1001, 1111, 1100\}$$

este código tiene dimensión $m+1 = 2+1 = 3$, de hecho una matriz generadora para el es:

$$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Por cierto, una matriz generadora de $\mathcal{R}(1)$ es:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

Esta matriz aparece dos veces, en la parte inferior de la matriz anterior. Algo curioso que ahora aclararemos. \triangleleft

Dado el proceso de construcción de los códigos binarios de Reed-Muller, es fácil también construir sus matrices generadoras.

1. La matriz generadora de $\mathcal{R}(1)$ es

$$\mathbf{R}_1 = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

2. Si \mathbf{R}_{m-1} denota la matriz generadora de $\mathcal{R}(m-1)$ entonces la matriz \mathbf{R}_m para $\mathcal{R}(m)$ es:

$$\mathbf{R}_m = \begin{pmatrix} 0 \dots 0 & 1 \dots 1 \\ \mathbf{R}_{m-1} & \mathbf{R}_{m-1} \end{pmatrix}$$

¡Perfecto! un proceso recursivo de construcción.

Un código de Reed-Muller¹ es un $(2^m, 2^{m+1}, 2^{m-1})$ -código binario, es decir un $[2^m, m+1, 2^{m-1}]$ -código lineal binario, lo que significa que puede corregir hasta:

$$\left\lfloor \frac{2^{m-1} - 1}{2} \right\rfloor = \lfloor 2^{m-2} - 1/2 \rfloor = 2^{m-2} - 1$$

errores. Sin embargo su arreglo estándar contendrá $2^{2^m - m - 1}$ cosets, algo impensable de almacenar aún para m pequeña. Afortunadamente hay una manera más sencilla de decodificar. Como cualquier palabra del código \mathbf{c} , debe ser expresable como combinación lineal de los elementos de la base $\{\mathbf{b}_1, \dots, \mathbf{b}_k\}$ tenemos que:

$$\mathbf{c} = \alpha_1 \mathbf{b}_1 + \dots + \alpha_k \mathbf{b}_k$$

si tenemos los elementos en la base podemos encontrar el valor de α_i a partir de las componentes de \mathbf{c} de 2^{m-1} maneras diferentes, todas ellas deben dar el mismo valor. En el caso de nuestro ejemplo, para cada elemento de $\mathbf{c} = c_0 c_1 c_2 c_3 \in \mathcal{R}(2)$ necesitamos tres escalares α_1, α_2 y α_3 (dado que la cardinalidad de la base es tres) y sus valores están determinados por:

$$\begin{aligned} \alpha_1 &= c_0 + c_2 \\ \alpha_2 &= c_0 + c_1 \\ \alpha_3 &= c_0 \end{aligned}$$

o bien por:

$$\begin{aligned} \alpha_1 &= c_1 + c_3 \\ \alpha_2 &= c_2 + c_3 \\ \alpha_3 &= c_1 + c_2 + c_3 \end{aligned}$$

Si las calculamos por cualquiera de los dos métodos debemos obtener el mismo resultado. Si esto no ocurre sabemos que la palabra recibida está mal.

Por desgracia $\mathcal{R}(2)$ no puede corregir errores. Pero para $\mathcal{R}(3)$ la situación es diferente, puede corregir un error.

Ejemplo B.2 La matriz generadora para $\mathcal{R}(3)$ es:

$$\mathbf{R}_3 = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

¹Formalmente hablando nos estamos refiriendo a códigos de Reed-Muller de primer orden.

En este caso el primer coeficiente de la combinación lineal, calculado en función de los bits (de izquierda a derecha: $c_0 c_1 \dots c_7$) es:

$$\alpha_1 = c_0 + c_4$$

$$\alpha_1 = c_1 + c_5$$

$$\alpha_1 = c_2 + c_6$$

$$\alpha_1 = c_3 + c_7$$

el segundo sería:

$$\alpha_2 = c_0 + c_2$$

$$\alpha_2 = c_1 + c_3$$

$$\alpha_2 = c_4 + c_6$$

$$\alpha_2 = c_5 + c_7$$

el tercero:

$$\alpha_3 = c_0 + c_1$$

$$\alpha_3 = c_2 + c_3$$

$$\alpha_3 = c_4 + c_5$$

$$\alpha_3 = c_6 + c_7$$

el último;

$$\alpha_4 = c_0$$

$$\alpha_4 = c_1 + c_2 + c_3$$

$$\alpha_4 = c_1 + c_4 + c_5$$

$$\alpha_4 = c_1 + c_6 + c_7$$

Hay que notar que en este caso tenemos cuatro ecuaciones diferentes para calcular cada coeficiente, así que una vez recibida una palabra procedemos a calcular cada coeficiente por cada uno de los cuatro mecanismos disponibles. En caso de haber errores algunas ecuaciones darán un resultado incorrecto, pero tenemos suficiente redundancia como para decidir los valores correctos por *mayoría de votos*. Es por eso que a este mecanismo de decodificación se le llama *decodificación mayoritaria*. \triangleleft

B.2 Códigos de Reed-Solomon

Los códigos de Reed-Muller y de Golay se vieron desplazados en las misiones espaciales más recientes por otros códigos, del mismo tipo, por cierto, que los usados hoy en día por nuestros lectores de discos CD-ROM y de audio. En los CD-ROM se utilizan de hecho varios códigos correctores de errores, los más robustos constituyen un CIRC (*Cyclic Interleaved Reed-Solomon Code*). De hecho son dos códigos de Reed-Solomon entrelazados.

Los códigos de Reed-Solomon son, de hecho, un caso particular de otros códigos más generales descubiertos paralelamente de modo independiente, por Hocquenghem en 1959 y por Bose y Chaudhuri conjuntamente en 1960. La cualidad fundamental de los códigos BCH, llamados así por las iniciales de sus descubridores, es el hecho de que es posible, dada una distancia mínima requerida, diseñar un código con esa distancia mínima y por tanto con capacidad de detectar y corregir errores predeterminada. Por supuesto lo mismo ocurre para los códigos de Reed-Solomon, descubiertos también en 1960.

Para comprender aunque sea superficialmente los códigos de Reed-Solomon, a los que llamaremos RS por brevedad, hay que cambiar el modo en el que concebimos nuestras cadenas binarias (y r -arias en general, aunque nos avocaremos al caso binario). En vez de verlas como simples cadenas de bits podemos concebirlas como coeficientes polinomiales. Es decir, cada cadena de longitud n sobre \mathbb{Z}_2 será vista como lista de coeficientes binarios de un polinomio de grado menor o igual a $n - 1$. Así, por ejemplo, la cadena 1001011 en \mathbb{Z}_2^7 puede verse como el polinomio de grado 6: $1x^0 + 0x^1 + 0x^2 + 1x^3 + 0x^4 + 1x^5 + 1x^6$ si conservamos nuestras operaciones módulo 2 entre cadenas lo que estaremos haciendo, viéndolas como polinomios, es nunca salirnos del conjunto de polinomios de grado menor o igual a $n - 1$ y coeficientes en \mathbb{Z}_2 (en \mathbb{Z}_p con p primo en general). Este conjunto es finito, por supuesto, solo tiene 2^n (p^n en el caso general) elementos.

Esto tiene más relevancia de la que parece. Ya habíamos visto que \mathbb{Z}_p es un campo siempre que p es primo, pero estos no son los únicos campos finitos que existen, de hecho existe un único campo, salvo isomorfismo, con p^r elementos, donde p es primo y r es cualquier número natural. De hecho nuestro conjunto de polinomios sobre \mathbb{Z}_2 es un campo finito. Por cierto que sus operaciones se ven extrañas, estamos acostumbrados a trabajar con polinomios cuyos coeficientes están en \mathbb{R} , pero en \mathbb{Z}_2 las cosas son diferentes, por ejemplo si $a(x) = x^2 + x$ y $b(x) = x$ la suma es:

$$a(x) + b(x) = x^2 + x + x = x^2$$

no $x^2 + 2x$ como ocurre en \mathbb{R} .

En estos campos finitos, por cierto, existen polinomios irreducibles (que no es posible factorizar) tales que cualquier otro polinomio del campo puede escribirse como múltiplo (con operación módulo por supuesto) de uno de estos polinomios irreducibles especiales. A estos se les llama *generadores*.

La idea de los códigos de Reed-Solomon se basa en que, dados k puntos $(x_0, y_0), \dots, (x_{k-1}, y_{k-1})$, existe uno y solo un polinomio de grado a lo más $k - 1$ tal que pasa por esos puntos precisamente. Esto ocurre tanto si estamos trabajando con coeficientes en \mathbb{R} o en \mathbb{Z}_p .

Dados los puntos, calcular el polinomio que pasa por ellos es una labor que puede ser hecha por diversos métodos, resolviendo un sistema de ecuaciones lineales, usando el método

de Lagrange, etcétera. Lo que hay que notar es que, dado que por cada k puntos existe un único polinomio, es entonces completamente equivalente tener los coeficientes del polinomio de grado $k - 1$ o un conjunto de k evaluaciones del mismo. Si de antemano fijamos los posibles valores de la variable independiente: x_0, \dots, x_{k-1} , entonces tenemos completamente determinado el polinomio $p(x) = a_0 + a_1x + \dots + a_{k-1}x^{k-1}$ si tenemos los coeficientes a_0, \dots, a_{k-1} o si tenemos los valores y_0, \dots, y_{k-1} tales que $y_i = p(x_i)$. Tenemos la misma información, podemos reconstruir el polinomio si nos dan las evaluaciones.

Entonces podríamos pensar en la siguiente situación: consideramos a nuestras cadenas binarias de longitud n como cadenas de coeficientes polinomiales de polinomios de grado menor o igual a $n - 1$, si debemos enviar una de nuestra cadenas (polinomio) por una canal de comunicación, enviamos, en vez de los n coeficientes del polinomio (bits de la cadena), un conjunto de n evaluaciones del polinomio, de hecho un conjunto de n valores de la variable dependiente, si nos hemos puesto de acuerdo con el receptor en el conjunto de valores de la variable independiente en los que vamos a evaluar.

Con esta situación por supuesto, no ganamos nada, de todos modos enviamos n bits.

Pero si pensamos que nuestro canal puede cometer errores entonces la situación cambia. Supongamos que nuestro canal puede cometer hasta s errores por cada palabra de n bits. Si solo enviamos n coeficientes o n evaluaciones y una se daña el polinomio original no puede ser recuperado. Pero si mandamos $n + 2s$ evaluaciones del polinomio entonces queda suficiente información como para recuperar el polinomio por algo como la decodificación de mayoría usada en la sección anterior.

Esa es la idea original de Reed y Solomon [17], aunque es difícil de implementar, la labor matemática es considerable y casi impensable en hardware. Afortunadamente en 1967 Berlekamp descubrió un algoritmo eficiente para decodificar códigos de Reed-Solomon y de hecho códigos BCH en general y más tarde se han inventado otros algoritmos para decodificar eficientemente basados en métodos matemáticos no tan sencillos. En general, siguiendo la regla de oro de *hacer más rápido lo más común* no se envían las $n + 2s$ evaluaciones sino que, realmente se envían los coeficientes del polinomio, en el supuesto de que el canal comete pocos errores esto significa que la decodificación, en el mejor de los casos, es sencilla. Se añaden las evaluaciones como verificaciones de paridad y solo se usan en caso de que los coeficientes se hayan dañado.

Criptografía simétrica

6.1 Conceptos, sustitución monoalfabética

Hemos visto ya la utilidad de los códigos para lograr representaciones eficientes de la información y para lograr representaciones que permitan verificar y preservar la integridad de la información. Ahora nos avocaremos a buscar representaciones que contribuyan a preservar la seguridad de la información, mecanismos que eviten que personas no autorizadas tengan acceso a información que se pretende mantener en secreto.

Hay que señalar que la criptografía ya no pertenece a la teoría de la información ni a la teoría de códigos, al menos no en general. Es toda una rama por sí misma, con sus propios problemas y métodos, la intersección entre ambas cosas es no vacía, claro, pero la criptografía no es subconjunto de aquellas disciplinas. De hecho la criptografía es el subconjunto de la criptología, que se dedica al diseño, fundamentos e implementación de algoritmos y protocolos de cifrado de datos. El otro subconjunto de la criptología se dedica

al estudio de las debilidades y los medios que permitan acceder de manera no autorizada a información cifrada, el criptoanálisis. También hay un pariente cercano a la criptografía llamado esteganografía.

Para aclarar el concepto de criptografía veamos primero un ejemplo de esteganografía, de hecho el primer ejemplo del que tenemos conocimiento.

Alrededor del año 400 a.d.c los persas pretendían invadir Europa, en particular Grecia. Esparta era la ciudad-estado griega de mayor poderío militar. Un exiliado griego en medio oriente se enteró de los planes persas y decidió enviar un cargamento de tablillas de madera, donde labró el mensaje de advertencia, a Esparta. Para que nadie se enterara de lo que había hecho cubrió las tablillas con cera, al llegar a Esparta la joven hija del rey despostilló una de las tablillas y notó que tenían un mensaje escrito, descubrieron todas ellas y quedaron sobre aviso de la próxima invasión. Esparta ganó, la batalla de las Termopilas probablemente hubiera tenido un resultado muy diferente de no ser por el aviso oculto en las tablillas.

La intención en el caso que acabamos de describir es ocultar la existencia del mensaje, cubrirlo (cubrir en griego se dice *steganos*), es lo mismo que se pretendía con los microfilms, que aparentaban ser defectos del papel, usados en la segunda guerra mundial.

La intención de la criptografía no es ocultar la existencia del mensaje, sino su contenido. El objetivo es que si alguien captura el mensaje no pueda entenderlo.

Uno de los primeros mecanismos criptográficos utilizados proviene, por cierto, de Esparta. Toda legión tiene un estandarte, el estandarte se pone en un bordón, un asta, que generalmente tiene el mismo diámetro para todas las legiones. Con esto en mente los espartanos idearon la *scytala*, un mensaje es escrito verticalmente en una tira de cuero enrollada alrededor de un bastón de cierto diámetro. Cuando se desenrolla, el mensaje es ininteligible. Solo alguien que tenga un bastón del mismo diámetro que el emisor puede leer el mensaje enrollando la tira alrededor de el.

Otro de los métodos antiguos de cifrado es el ideado por Julio César y que utilizó en la guerra de las Galias. Para empezar Cesar utilizó el alfabeto griego y no el romano. Colocando el alfabeto en el orden usual y luego poniendo bajo este el mismo alfabeto pero recorrido (rotado) tres lugares a la derecha obtenemos una función que asocia a cada letra del alfabeto puesto arriba una letra del de abajo. Si tenemos un mensaje basta buscar cada letra de él en el alfabeto superior y reemplazarla por la que aparezca bajo ella en el inferior. El mensaje cifrado no puede ser inmediatamente leído por nadie que no sepa cuantos lugares se ha rotado el alfabeto.

En estos dos casos distinguimos varios elementos comunes. Para empezar, la existencia de la criptografía supone la existencia de dos personas o grupos de ellas que pretenden

comunicarse, un emisor y un receptor. Supone también la existencia de una tercera persona o grupo, el enemigo, aquel al que el emisor y el receptor pretenden ocultarle el significado de los mensajes que intercambian. Supone también la existencia de un medio de comunicación que no es seguro, que puede ser intervenido en cualquier momento por el enemigo. Otros elementos fundamentales que podemos distinguir son: el mensaje que se pretende enviar, un algoritmo de cifrado que mapea ese mensaje en un mensaje cifrado con la ayuda de una clave y finalmente un algoritmo de descifrado que mapea el mensaje cifrado al mensaje original con ayuda de la clave. El objetivo es que quien no conozca la clave no pueda recuperar el mensaje original suponiendo que posee el mensaje cifrado.

En el párrafo anterior mencionamos la clave como un elemento indispensable para cifrar y para descifrar el mensaje, hablamos de una sola clave que se usa para ambas operaciones. Esto no necesariamente es así siempre, en el siguiente capítulo hablaremos de algoritmos criptográficos en los que la clave para descifrar es diferente de la usada para cifrar. Cuando ambas claves son la misma se dice que el sistema criptográfico es *simétrico*. De esos métodos hablaremos en este capítulo.

En el caso del cifrado de César la clave es el número de posiciones que se desplaza el alfabeto, al parecer César siempre usó tres. En “2001: Una odisea espacial” Arthur C. Clarke usa sólo una letra de desplazamiento (hacia atrás) cuando codifica “IBM” como “Hal”, la compañía fabricante de la computadora de la novela. Una generalización del esquema aparece en el *Kama Sutra*. En este antiguo documento se dice que las mujeres deben cultivar 64 diferentes artes y la cuadragésimo quinta de ellas es la criptografía, con lo que se conseguirá mantener en secreto los amoríos. El esquema descrito es esencialmente el de César pero el número de posiciones que se desplaza el alfabeto es arbitrario. Por supuesto podemos generalizar aún más y colocar una permutación arbitraria del alfabeto original bajo el y esto dificultará aún más la lectura no autorizada del mensaje. Podríamos de hecho usar cualquier otro conjunto de símbolos como el alfabeto de cifrado.

Los esquemas como el de César y sus generalizaciones, en los que, dada una letra del alfabeto del mensaje original, esta se reemplaza por otra, y siempre la misma, letra (símbolo para ser más general) del alfabeto de cifrado, se denominan *monoalfabéticos* se usa una única asociación de símbolos para todo el mensaje.

Alguien que tenía mucha experiencia en descifrar esquemas monoalfabéticos fue Edgar Allan Poe. En *El escarabajo dorado* nos deja ver el método típico para descifrar mensajes cifrados con este tipo de algoritmos. Dado que cada aparición de una letra del texto cifrado se reemplaza siempre por otra letra del alfabeto de cifrado, las frecuencias de aparición de los símbolos del alfabeto original se mapean a las frecuencias de los símbolos que les corresponden en el alfabeto cifrado. Nuestros idiomas no poseen una distribución de letras uniforme. Hay letras más frecuentes que otras, en español, por ejemplo, la letra más usual

Letra	Español	Inglés	Letra	Español	Inglés
A	0.1234	0.0820	N	0.0705	0.0670
B	0.0178	0.0150	O	0.0937	0.0750
C	0.0422	0.0280	P	0.0239	0.0190
D	0.0493	0.0430	Q	0.0109	0.0010
E	0.1304	0.1270	R	0.0666	0.0600
F	0.0060	0.0220	S	0.0750	0.0630
G	0.0109	0.0200	T	0.0443	0.0910
H	0.0115	0.0610	U	0.0413	0.0280
I	0.0574	0.0700	V	0.0110	0.0100
J	0.0056	0.0020	W	0.0002	0.0240
K	0.0004	0.0080	X	0.0010	0.0020
L	0.0609	0.0400	Y	0.0113	0.0200
M	0.0300	0.0240	Z	0.0047	0.0010

Tabla 6.1: Probabilidad de las letras en español e inglés.

es la “e”, también en inglés, pero en general las frecuencias de aparición son muy diferentes entre ambos idiomas. Las diez letras más frecuentes en español son, en orden decreciente: “E”, “A”, “O”, “S”, “N”, “R”, “L”, “I”, “D” y “T”; en cambio en inglés son: “E”, “T”, “A”, “O”, “I”, “N”, “S”, “H”, “R” y “D” (véase la tabla 6.1). Si tenemos un texto cifrado grande, para el que se usó una sustitución monoalfabética entonces es muy probable que el símbolo más frecuente del texto cifrado corresponda a la letra más frecuente del idioma en el que estaba escrito el texto original. En 1987 nos enteramos de que un árabe *Al-Kindi*, que vivió en Estambul en el siglo noveno de nuestra era, ya conocía este hecho, que se constituye en el método de criptoanálisis por excelencia para romper la seguridad de los métodos criptográficos de sustitución monoalfabética.

6.2 Substitución polialfabética

Una posibilidad para evitar el criptoanálisis de frecuencias consiste en cambiar el alfabeto que asocia al alfabeto original con cierta frecuencia. El primero en inventar un mecanismo de cifrado con esta característica fue Leon Baptista Alberti en el siglo XV. Alberti fabricó una par de círculos concéntricos de meta, uno de ellos menor que el otro. En el disco más grande grabó el alfabeto original en el orden lexicográfico usual y en el menor el alfabeto de cifrado en un orden arbitrario (ambos alfabetos eran, esencialmente el mismo, solo que el del círculo menor estaba en minúsculas mientras que el del exterior estaba en mayúsculas).

Cuando se colocan las letras de los discos en correspondencia nos dan una función de substitución, si la usamos para cifrar todo un mensaje estamos haciendo una substitución monoalfabética convencional, pero si luego de haber cifrado algunas letras giramos el disco interior a otra posición, obtenemos otra substitución monoalfabética diferente de la primera, si este procedimiento es repetido periódicamente obtenemos una substitución polialfabética: no siempre una letra dada del alfabeto original se reemplaza por la misma letra del alfabeto cifrado. Si nuestro alfabeto tiene 26 letras, como el usual en inglés, entonces tenemos 26 posibles substituciones monoalfabéticas que podemos mezclar en un solo mensaje. Esto, en principio, hace que el análisis de frecuencias sea inútil para romper el sistema criptográfico.

La clave en el sistema del disco de Alberti consiste en especificar cuál letra del disco interior es puesta bajo una letra predeterminada del disco exterior y durante cuantas letras será usada la substitución definida por esta correspondencia. Algo que se dá por sentado es que tanto el emisor como el receptor del mensaje poseen discos de Alberti idénticos, de otro modo la construcción del disco interior es también parte de la clave.

Una generalización del sistema del disco de Alberti fue inventada por Blaise de Vigenère en el siglo XVI. En este esquema el cambio de alfabeto se dá automáticamente con cada letra del texto original, el alfabeto cambia constantemente. Los alfabetos que se utilizan son, como en el caso de Cesar, el alfabeto original rotado un cierto número de posiciones a la derecha. Para determinar cuál de este catálogo de alfabetos se debe utilizar para una letra en una posición dada del texto original, se asocia el texto original con una palabra clave escrita en el mismo alfabeto.

Es conveniente ilustrar este mecanismo de cifrado con un ejemplo.

Ejemplo 6.1 Supongamos que tenemos el mensaje que deseamos transmitir de manera segura es *heladote* y que nos hemos puesto de acuerdo con el receptor en que la palabra clave que usaremos es *orale*. Lo primero que hacemos es aparear las letras del texto a cifrar con las de la palabra clave, como se muestra en la figura 6.1. Este apareamiento nos proporciona las parejas (r, c) siguientes: (o, h) , (r, e) , (a, l) , (l, a) , (e, d) , (o, o) , (r, t) , (a, e) . Estas las vamos a considerar como parejas ordenadas (renglón, columna) de una matriz que construiremos a continuación.

Para construir la matriz hacemos lo siguiente:

1. El primer renglón de la matriz es el alfabeto que estamos utilizando en un orden arbitrario, en nuestro caso hemos usado el orden lexicográfico convencional.
2. El i -ésimo renglón de la matriz es el mismo alfabeto rotado (al estilo de Cesar) i lugares a la derecha.

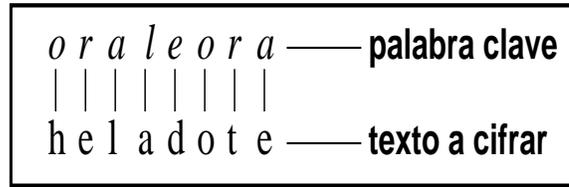


Figura 6.1: Apareamiento de la palabra clave y el texto a cifrar en el esquema de Vigenère.

	a	d	e	h	l	o	r	t
a	a	d	e	h	l	o	r	t
d	d	e	h	l	o	r	t	a
e	e	h	l	o	r	t	a	d
h	h	l	o	r	t	a	d	e
l	l	o	r	t	a	d	e	h
o	o	r	t	a	d	e	h	l
r	r	t	a	d	e	h	l	o
t	t	a	d	e	h	l	o	r

Tabla 6.2: Matriz de Vigenère para el alfabeto del ejemplo.

El resultado de este proceso se muestra en la tabla 6.2. Nótese que además tanto en el renglón superior como en la columna de la extrema izquierda de la tabla aparece el alfabeto en el orden lexicográfico creciente usual, esto nos servirá para indexar las posiciones la tabla.

Para cifrar cada letra del mensaje original nos fijamos en la entrada de la matriz indicada por la pareja ordenada (r, c) , determinada por la letra que deseamos cifrar (c): el renglón de la matriz indicado por (cuyo índice es) r y la columna indicada por c . En nuestro caso las entradas de la matriz determinadas por nuestras parejas ordenadas son:

- $(o, h) = a$
- $(r, e) = a$
- $(a, l) = l$
- $(l, a) = l$
- $(e, d) = h$
- $(o, o) = e$

- $(r, t) = o$
- $(a, e) = e$

así que nuestro mensaje cifrado es “**aallheoe**”.

El receptor recibe el mensaje cifrado y para descifrarlo lo pone otra vez en correspondencia con la palabra clave previamente acordada. Ahora quedan parejas de la forma (r, e) (renglón, entrada), el receptor procede entonces a buscar en el renglón indicado por cada pareja, aquella entrada que coincida con la letra especificada por el segundo elemento de la pareja, e ; cuando lo encuentra lo decodifica como aquella letra que aparezca encabezando la columna donde lo halló. En nuestro caso las parejas (r, e) y los encabezados (índices) de las columnas son:

- $(o, a) = h$
- $(r, a) = e$
- $(a, l) = l$
- $(l, l) = a$
- $(e, h) = d$
- $(o, e) = o$
- $(r, o) = t$
- $(a, e) = e$

con lo que se recupera el mensaje original.

◁

En nuestro ejemplo es claro que una sola letra del texto original es reemplazada por más de una del alfabeto de cifrado, por ejemplo la primera “e” del mensaje es reemplazada por una “a” mientras que la segunda “e” es reemplazada por una “e”, esto echa a perder el análisis de frecuencias tradicional. Sin embargo hay que notar algo: cada vez que se termina la palabra clave y se vuelve a poner la primera letra de ella se utiliza el mismo alfabeto que se utilizó en la repetición anterior, en nuestro caso cada vez que la “o” de la palabra clave se empata con alguna letra del texto original, se utiliza el mismo renglón, es decir, el mismo alfabeto para cifrar la letra en cuestión. Lo mismo para cada vez que se reutiliza cualquier letra de la clave. La reutilización de una substitución monoalfabética es periódica, el periodo es, a lo más, la longitud de la palabra clave (si no tiene letras repetidas). Esto

nos lleva a pensar dos cosas: la primera es que si logramos dividir el mensaje cifrado en submensajes que utilicen la misma substitución monoalfabética podemos aplicar nuestro ya conocido análisis de frecuencias para descifrar cada submensaje, la segunda es que, para dificultar esto debemos usar una palabra clave larga.

Este procedimiento de criptoanálisis que consiste en encontrar el periodo de reuso de la clave y por tanto su longitud fue descubierto por Charles Babbage, que se dedicaba al criptoanálisis como pasatiempo y que, de hecho, planeaba escribir un libro acerca de ello, algo que nunca ocurrió como suele ser en el caso de Babbage, nos enteramos de su descubrimiento hasta el siglo XX cuando se estudió su correspondencia. Sin embargo el procedimiento fue redescubierto por un oficial del ejercito prusiano Friedrich Wilhelm Kasiski en el siglo XIX. Otro mecanismo de criptoanálisis aplicable al esquema de Vigenère y a los cifrados polialfabéticos en general fue descubierto por el estadounidense William Friedman en el siglo XX, además Friedman estableció una prueba estadística para determinar si un cifrado es monoalfabético o polialfabético.

La prueba de Friedman se basa en el hecho de que, como ya dijimos, cada idioma tiene su muy particular distribución de probabilidades en el uso de las letras del alfabeto. Así que si se mide la entropía típica de una fuente que produce texto en español, por ejemplo, y se compara con la entropía de un texto cifrado monoalfabéticamente y que originalmente estaba en español, ambas deben ser muy similares. Si las entropías son muy diferentes significa que el proceso criptográfico es más que un reetiquetamiento de las letras del alfabeto, es un proceso que redistribuyó las probabilidades de cada símbolo, como ocurre en un cifrado polialfabético.

6.3 Enigma

En 1918 el alemán Arthur Scherbius patentó una máquina de cifrado que también generalizaba la idea del disco de Alberti. La máquina se llamó *enigma* y constituiría uno de los mecanismos de cifrado más célebres de la historia.

La máquina de cifrado de Alberti es, en esencia, un permutador de símbolos. En el esquema original planteado por Alberti, una vez que se definía una permutación, esta era utilizada para cifrar algunos símbolos y luego la permutación se cambiaba manualmente. La idea de Scherbius era conectar varios dispositivos de permutación, encadenarlos, además de hacer que el cambio de permutación fuera automático. El funcionamiento era similar al de un odómetro de automóvil, el dispositivo que mide el kilometraje recorrido, cada metro se mueve el contador de metros, cada 10 metros el contador de decenas de metros ..., cada mil el contador de kilómetros. En *enigma* cada vez que se cifra una letra se mueve un primer

permutador, cada vez que el primer permutador regresa a su posición original se mueve el segundo permutador y así sucesivamente. A los permutadores les llamaremos en adelante *rotores*, dado que sus movimientos consistían justamente en rotar una posición cada vez.

Cada rotor de enigma era un dispositivo electromecánico, tenía tantas entradas como salidas de alambres, cada alambre de entrada del primer rotor tenía asociada una letra de un teclado, cada salida del primer rotor era entrada del segundo y así sucesivamente hasta completar tres o cuatro rotores. Esto se muestra esquemáticamente en la figura 6.2.

En la máquina original de Alberti, si cada vez que se cifra un símbolo se rota una posición el disco interior, luego de haber cifrado un número de símbolos igual al tamaño del alfabeto (24 símbolos en el caso de Alberti), el disco interior regresa a su primera posición. Es decir hay un total de 24 posibles cifrados, esta es una medida de la complejidad del sistema criptográfico y de la dificultad para romperlo.

Si se conectan dos discos con alfabeto de 26 símbolos, como es usual en inglés, entonces por cada una de las 26 posiciones del primer disco (rotor), hay 26 posibilidades en el segundo, esto nos da un total de $26^2 = 676$ posibilidades, si se conectan tres rotores consecutivos, como en las máquinas enigma más sencillas, el número de posibilidades es $26^3 = 17576$, algo mucho más difícil de romper.

Cada rotor tenía una configuración fija, el alambrado dentro de él no podía ser cambiado. Sin embargo, para añadir un elemento de complejidad extra, enigma tenía rotores intercambiables, es decir, cada uno de los tres rotores podía ser colocado en cualquiera de las tres posiciones disponibles para ellos. Eso nos da $3! = 6$ posibles posiciones, ahora tenemos $6 \times 17576 = 105456$ posibilidades.

Para complicarlo aún más algunas máquinas enigma poseían un tablero de conexiones frontal, un conjunto de orificios asociados a las letras del alfabeto, era posible, mediante un cable, conectar cualesquiera dos de esos orificios. Los operarios de enigma eran equipados con seis cables, por lo que era posible conectar seis pares de letras. El objetivo de este panel, conectado entre el teclado y el primer rotor, era intercambiar letras antes de llegar al primer rotor, es decir, conectar la “a” con la “h” significa que cuando el operario presionara la tecla etiquetada con “a” en realidad esta se transformaba en “h” y bajo ese disfraz entraba al primer rotor, esto hace que el número de posibilidades crezca astronómicamente. Por si fuera poco, algunas máquinas enigma, a pesar de tener solo tres o cuatro ranuras para insertar los rotores tenían cinco diferentes rotores para insertar en ellas.

Una cualidad adicional que hacía que enigma fuera fácil de utilizar era que al teclear un mensaje previamente cifrado con enigma se obtenía el mensaje descifrado. Es decir, automáticamente se invertía el proceso de cifrado, si es que se iniciaba con la misma clave con la que se cifró. Por supuesto la clave estaba constituida por:

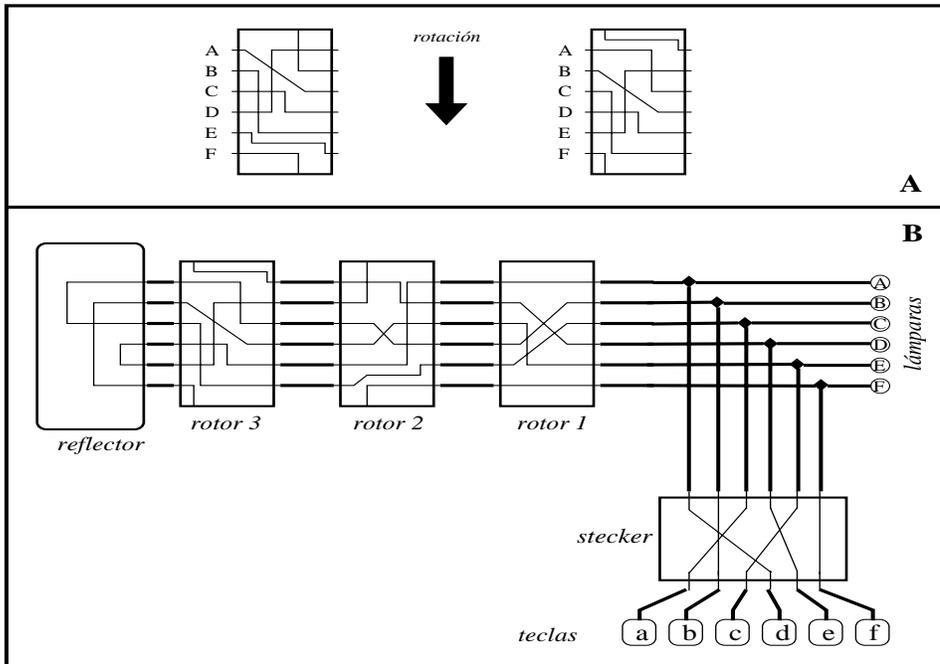


Figura 6.2: Representación esquemática de la máquina *enigma*. En **A** se ilustra un rotor antes y después de hacer una rotación en la dirección mostrada, cuando una línea se pierde por arriba del rotor “regresa” por abajo y viceversa. En **B** se muestra la máquina completa (sin el panel frontal mencionado en el texto). Nótese que si se pulsa una tecla dada el recorrido por la máquina llega hasta una lámpara y que si se pulsa la tecla que corresponde a la lámpara prendida se recorre el circuito anterior a la inversa llegando a la lámpara asociada a la primera tecla, esto hace que el teclear un texto cifrado resulte en el texto descifrado y viceversa.

1. Las seis conexiones hechas en el panel frontal.
2. Las posiciones de los rotores. Cada rotor era numerado, así la especificación 312 significa: el rotor 3 en la primera ranura (a la izquierda), el rotor 1 en la ranura de en medio y el rotor 2 en la ranura de la derecha.
3. Las orientaciones iniciales de los rotores. Cada posición, de las 26 posibles de cada rotor estaba identificada por una letra de las 26 grabadas en el rotor y que debía quedar hacia arriba. Por supuesto se debían especificar entonces tres letras.

La reversibilidad automática del proceso de cifrado proviene de una pieza de la máquina llamada *reflector* que recibía las líneas de salida del último rotor. Cada línea de entrada era sacada por el mismo lado del reflector, pero en otra posición de donde regresaba al conjunto de rotores atravesándolos para llegar finalmente a alguna de las lámparas etiquetadas con las letras, la lámpara que se prendía era la que cifraba a la que se había tecleado. Ambas letras constituían un circuito completo teclear una de ellas prendía la lámpara de la otra y viceversa, probablemente es más claro esto al observar la figura 6.2.

Enigma es famosa por ser la máquina de cifrado que utilizaron, en diferentes versiones, el ejército y la armada alemanas durante la segunda guerra mundial. Para utilizarla los alemanes tenían un catálogo de las claves que se debían utilizar durante un mes. Así existía la *clave del día* con la que se cifraban, en principio todos los mensajes transmitidos durante ese día en particular. Al iniciar el día todos los operarios de máquinas enigma debían establecer la clave de sus máquinas con la que indicaba el libro.

Aparentemente alguien pensó que esto era muy inseguro: proporcionar una gran cantidad de información cifrada con la misma clave podría debilitar el sistema facilitando el criptoanálisis de los aliados, en particular de los ingleses. Así que se decidió cambiar el esquema. Cada mensaje debía estar cifrado con una clave diferente. Pero esto implica que el emisor debe informar de la clave al receptor, para que este pueda descifrar el mensaje, así que cada mensaje era iniciado con la transmisión de la clave usada para cifrar ese mensaje en particular, por supuesto la clave era, a su vez cifrada con la clave del día. Ya hemos visto que nuestros medios de comunicación son falibles y pueden introducir errores en las transmisiones, así que para evitar errores en la transmisión de la clave esta era duplicada en el encabezado de cada mensaje. Por cierto que la clave de mensaje solo especificaba las orientaciones de los rotores, las conexiones del panel y las posiciones de los discos en las ranuras eran las mismas que las especificadas en la clave del día. Esto realmente debilitó el esquema criptográfico alemán: cada mensaje empezaba con seis letras, que constituían la misma clave de tres letras duplicada. Información que les fué útil a los aliados, en particular a los polacos y a los ingleses para romper la seguridad del sistema.

Hubo otros factores que contribuyeron en el criptoanálisis, el primer mensaje de todos

los días, por ejemplo, era un reporte del clima, en un estricto formato que empezaba con la palabra “wetter” (clima en alemán). Suficiente información: el clima era evidente para todos, así que ya se tenía una idea del contenido del mensaje, además ya se sabía que empezaba con la palabra “wetter”. Moraleja: nunca hay que cifrar información no relevante y es mejor no ser metódico en el formato de los mensajes. Afortunadamente los alemanes cometieron errores como estos, que le permitieron a los ingleses de Bletchley Park (Alan Turing entre ellos), descifrar los mensajes de enigma con ayuda de una de las primeras computadoras. El mundo sería probablemente muy diferente hoy en día de no ser por eso.

6.4 DES

En 1973 la oficina de estándares norteamericana (NBS, *National Bureau of Standards*, actualmente NIST, *National Institute of Standards and Technology*) lanzó una convocatoria pública para recibir propuestas de algoritmos criptográficos con la intención de establecer un estándar en la materia. Para ese entonces ya había una gran cantidad de mecanismos que proveían de seguridad a los sistemas de cómputo, pero esos mecanismos no estaban regulados, no era raro que fueran incompatibles entre sí, dado que eran provistos por cada fabricante de equipo de cómputo, además no había medios para garantizar el nivel de seguridad que era proporcionado, en síntesis, era un caos y sería peor si no se establecía una normatividad y un patrón de referencia.

Por supuesto la oficina de estándares impuso requisitos a los posibles concursantes, el algoritmo debía cumplir con las siguientes condiciones:

- Proveer de un alto nivel de seguridad, que sería analizado por la agencia nacional de seguridad (NSA).
- Debía estar completamente especificado y ser fácil de entender. Para evitar que los implementadores le añadieran “cualidades” o, por no entenderlo bien, cometieran errores que debilitaran la seguridad.
- La seguridad debía residir en la clave y no en el supuesto de mantener secreto el algoritmo.
- Debía poder ser puesto a disposición de todo usuario.
- Debía ser eficiente y barato al implementarse en dispositivos electrónicos.
- Debía poder ser validado, es decir, si alguien dice haberlo implementado, debe haber mecanismos que certifiquen que esto es cierto y que nada falta o sobra en la implementación.

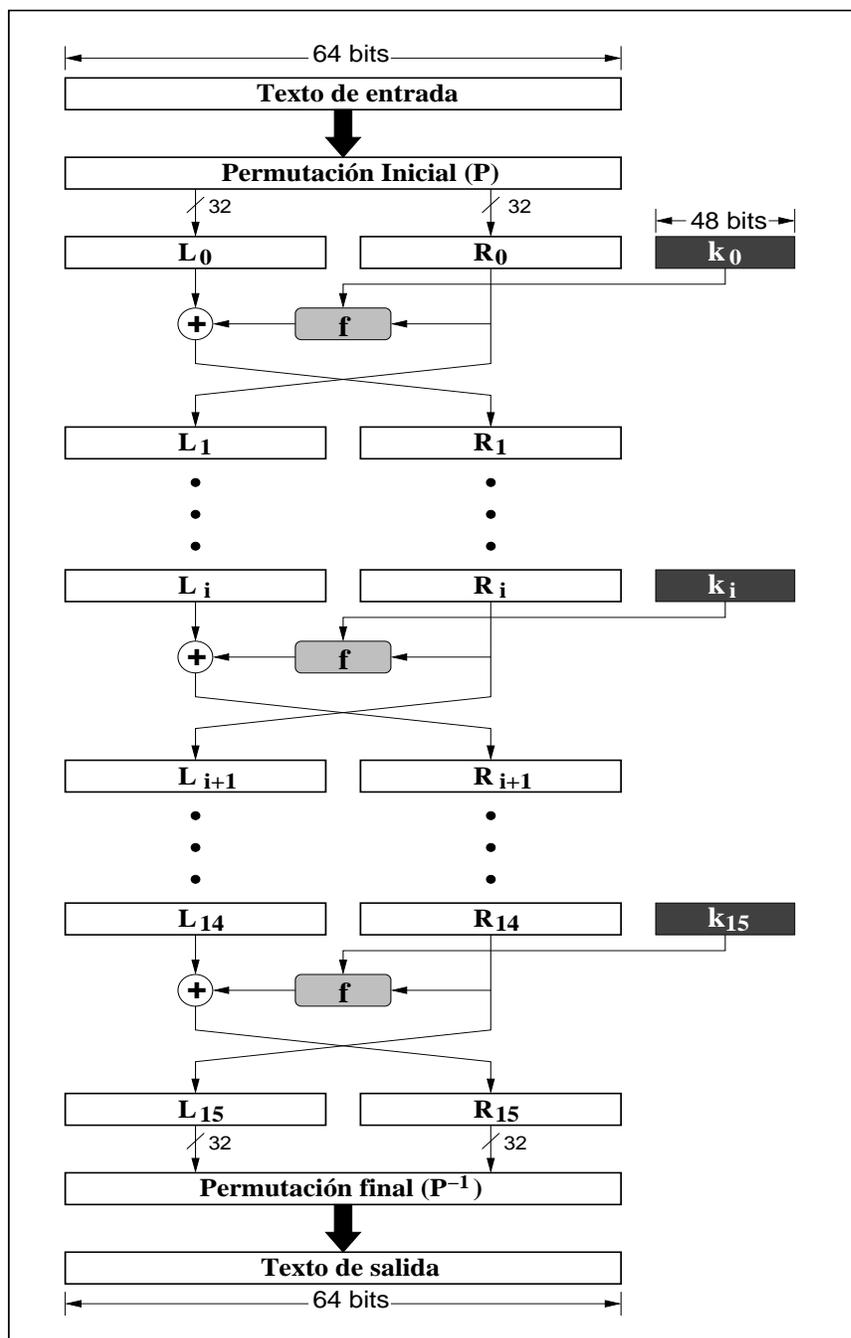


Figura 6.3: Representación esquemática del estándar de cifrado de datos (DES).

- Adaptable para diversas aplicaciones, es decir, no debe ser un algoritmo que suponga algo acerca de los datos a cifrar o del entorno en que será usado.

Nadie se presentó a la convocatoria, así que en 1974 se lanzó una nueva convocatoria pública en la que se presentó un algoritmo que estaba siendo desarrollado por un grupo de trabajo de dos laboratorios de IBM, *Lucifer*. En 1975 se publicaron el algoritmo y las garantías de no-exclusividad sobre su uso por parte de IBM. Se solicitaron entonces los comentarios de la comunidad de usuarios potenciales con el fin de que se analizara su utilidad y seguridad. Luego de hacerle depuraciones varias se adoptó como estándar federal en 1976 bajo el nombre *Data Encryption Standard* (DES).

Des cifra bloques de datos de 64 bits, regresa un bloque del mismo tamaño ya cifrado. Para cifrar utiliza una palabra clave, que es también un número de 64 bits, aunque solo se usan 56 de ellos, el bit más significativo de cada byte se considera como el bit de verificación de paridad de los restantes siete, así que una vez verificado el octeto este bit es desechado.

La operación de DES se divide en etapas, dieciséis, para ser exactos. Cada etapa es casi idéntica a todas las demás. En general en cada etapa se hace lo siguiente:

1. La clave de 56 bits se divide en dos tramos de 28 bits cada uno, cada tramo pasa a un desplazador (de hecho debía llamarse rotor, pero no hay que confundirlo con los de enigma). El desplazador toma los 28 bits de entrada y los recorre a la izquierda uno o dos bits dependiendo de la etapa en la que nos encontremos. El bit o bits que salgan despedidos por el extremo izquierdo “regresan” por el lado derecho. En las etapas 1, 2, 9 y 16 el desplazamiento es de un bit, en el resto es de dos bits.
2. Ambas mitades de la clave pasan a un dispositivo de permutación y compresión, este, como todos los demás dispositivos son siempre iguales en cada etapa. Lo que se hace es, en esencia, alterar el orden de los 56 bits recibidos y quitar algunos de ellos, los que se quitan son (empezando de 1): 9, 18, 22, 25, 35, 38, 43 y 54; ocho bits en total, lo que nos da $56 - 8 = 48$ bits de salida.
3. Por otra parte los 64 bits de mensaje son también divididos a la mitad: la parte alta en la etapa i -ésima L_i , y la parte baja R_i , cada una tiene entonces 32 bits. La parte baja se para por un dispositivo que permuta los 32 bits que recibe y además repite algunos de ellos, obteniendo un total de 48 bits, es decir repite 16 bits, la mitad de los que recibe. De hecho los bits duplicados son el 1, 4, 5, 8, 9, 12, 13, 16, 17, 20, 21, 24, 25, 28, 29 y 32, la secuencia es pues 32, 1, 2, 3, 4, 5, 4, 5, 6, 7, 8, 9, 8, 9, ..., 31, 32, 1.
4. La salida de la etapa 3 se introduce a un XOR junto con lo obtenido en el paso 2. La salida del XOR se introduce luego en un dispositivo de sustitución.

- (a) El dispositivo está hecho de ocho diferentes dispositivos menores llamados *S-Box* o cajas de sustitución. Cada S-Box recibe por tanto seis bits de entrada (48 bits de salida del XOR entre ocho cajas), el primero y último de los seis que entran a cada caja se usan como índices de renglón de una matriz construida dentro de la caja, los cuatro bits de en medio funcionan como el índice de la columna de dicha matriz. Cada S-Box tiene una matriz diferente, fija, pero en cada etapa de DES se usan las mismas ocho cajas en el mismo orden. Usando los índices se extrae el contenido de la matriz en el lugar indicado, este es un número de cuatro bits, dado que salen cuatro bits por cada una de las ocho cajas tenemos un total de $4 \times 8 = 32$ bits de salida.
5. La salida de las ocho S-Box es introducida luego a un dispositivo de permutación, esta no es expansiva no compresiva, es una permutación en todo el sentido matemático del término: biyectiva, entran 32 bits y salen los mismos 32 bits pero desordenados.
6. La salida de la permutación anterior es introducida a un XOR con la parte alta del mensaje. El resultado (32 bits) constituye la nueva parte baja del “mensaje” que entra a la etapa siguiente de DES.
7. La parte alta de entrada a la siguiente etapa es la parte baja de la anterior.

Adicionalmente a las 16 etapas casi idénticas de DES, antes de la primera etapa se pasa el mensaje por una permutación inicial, la que por cierto es también una permutación real. Al final, luego de la última etapa, se pasa el resultado por una permutación, que por cierto es la inversa de la permutación inicial. El objetivo de estas permutaciones es hacer que DES se comporte como enigma. Si se cifra un mensaje con DES usando una cierta clave y luego el mensaje cifrado es reintroducido a DES con la clave invertida¹ el resultado es el mensaje descifrado original.

El objetivo de DES es hacer que cada bit del bloque de salida este estadísticamente correlacionado con todos y cada uno de los bits de entrada, es decir, cada bit de la salida está en función de TODOS los bits de entrada.

Desde su introducción el estándar DES ha sido revisado cada cinco años, como fué establecido en el estándar mismo. Desde la primera revisión se le pretendía eliminar como estándar, por considerarlo susceptible de ser roto en un futuro cercano, antes de la siguiente revisión, sin embargo, a falta de algo mejor, se la he confirmado como estándar hasta la fecha. En 1993 Michael Wiener diseño una computadora capaz de romper DES por fuerza bruta (buscando exhaustivamente la clave) en un promedio de 3 horas y media, el costo

¹No es realmente la clave invertida en el sentido usual de invertir bits, es realmente la clave desplazada de tal forma que la clave que entraba a la etapa i ahora entre a la $17 - i$.

de tal computadora en 1993 hubiera sido de un millón de dolares, aproximadamente. Hay que señalar que el hardware ha bajado su precio desde entonces y seguirá bajando y que un millón de dolares puede no ser mucho dinero en cierto contexto. El mecanismo más famoso de criptoanálisis de DES, que ha puesto en tela de juicio la longitud de la clave, por ejemplo, es el *criptoanálisis diferencial* algo que escapa del alcance de este texto. El lector interesado puede acudir a libros de criptología o criptoanálisis.

Criptografía de llave pública

7.1 Funciones de un solo sentido

Probablemente todos hemos armado algún rompecabezas durante nuestra vida y luego de armarlo, generalmente, con todo el dolor de nuestro corazón, lo hemos desarmado para guardarlo en su caja porque estorbaba en la mesa. Desarmarlo es cosa fácil, pero armarlo no lo es en general. Este es un ejemplo de un proceso que es fácil de llevar a cabo en una dirección pero difícil de revertir. Podríamos pensar en la función que mapea cada pieza en su posición en el rompecabezas y estaríamos hablando entonces de una función biyectiva, por tanto invertible, pero cuya inversa es difícil de obtener. Esta noción es de gran utilidad en criptografía, a las funciones que son fáciles de calcular en un sentido pero difíciles de calcular a la inversa se les suele llamar *funciones de un solo sentido* o *one-way functions*.

Hablando en términos estrictamente matemáticos tales cosas no existen, una función biyectiva es invertible y ya. Sin embargo, hablando en términos computacionales, donde

generalmente no basta con decir si algo existe o no, sino que además hay que calcularlo, sí es posible pensar en tales funciones y, mejor aún, usarlas. Podríamos pensar, por ejemplo, en escribir un mensaje coherente en las piezas de un rompecabezas armado y luego dárselo desarmado a alguien, por supuesto que descifrar el mensaje en tales circunstancias no será fácil, este es un ejemplo trivial de que las funciones de un solo sentido pueden ser útiles en criptografía.

Pero que tal si numeramos las piezas del rompecabezas y hacemos un patrón de como deben ir acomodadas, entonces será trivial armar el rompecabezas en poco tiempo, muy poco tiempo comparando con el que tomaría armarlo sin ayuda del patrón. Ahora estamos hablando de funciones de un solo sentido en las que, si se da una clave, obtener la inversa, que era difícil, se vuelve fácil. A este tipo particular de funciones de un solo sentido se les llama funciones de puerta de trampa (*trapdoor functions*). Una puerta de trampa es fácil de abrir si se conoce el secreto para abrirla.

Conocemos buenos ejemplos de funciones de un solo sentido. Sabemos, por ejemplo, que es fácil generar números primos, sabemos también que es fácil multiplicar un par (o más) de números primos, pero si se nos da un número cualquiera es difícil, computacionalmente hablando, obtener los factores primos que lo determinan¹. Otro caso es el de la exponenciación en un campo finito² es fácil elevar un número a una potencia, pero dados un par de números a y b no es fácil determinar a qué exponente x hay que elevar a para obtener b (i.e. determinar x tal que $a^x = b$). Cabe hacer énfasis en que esta última función es de un solo sentido en campos finitos. Todos sabemos que en \mathbb{R} es fácil en ambos sentidos, el inverso de la exponenciación es el logaritmo, pero cuando se trabaja en un campo finito el problema se complica, en ese caso se habla del *logaritmo discreto*.

7.2 Factorización

¿Cuáles son los factores primos de 1386? Tratamos de ver que números primos dividen a 1386, así que el algoritmo más ingenuo que podemos plantear es el mostrado en la figura 7.1, que consiste esencialmente en probar con cada número primo a ver si divide al número en cuestión, comenzando por 2, el primer primo. Tratamos entonces de dividir entre cada número primo hasta que ya no sea posible dividir entre él y entonces nos pasamos al siguiente. Nos detenemos cuando obtenemos de cociente un número primo justamente. En la

¹El teorema fundamental de la aritmética establece que cualquier número natural puede escribirse de manera única (salvo el orden) como el producto de números primos.

²Recuérdese nuestros conocidos \mathbb{Z}_p con p un número primo, que de hecho son casos especiales de campos finitos. En general para todo primo p y todo entero positivo n existe un campo finito de tamaño p^n y es único salvo isomorfismo. p es llamada la *característica* del campo.

```

FACTORIZACIÓN( $N$ )
1   $p \leftarrow 2$ 
2   $c \leftarrow N$ 
3  while  $\neg(esPrimo(c))$  do
4      while  $\neg(p \mid c)$  do
5           $p \leftarrow siguientePrimo(p)$ 
6      endwhile
7       $p \rightarrow factores$ 
8       $c \leftarrow c/p$ 
9  endwhile
10  $c \rightarrow factores$ 
11 end

```

Figura 7.1: Algoritmo simple para factorizar. $esPrimo(x)$ regresa *verdadero* si su argumento es primo, $siguientePrimo(x)$ regresa el primer número primo mayor que su argumento, $factores$ es una lista donde se van guardando los factores primos del número N , $p \mid c$ significa p divide a c y el símbolo \neg denota la negación lógica.

tabla 7.1 se muestra el proceso, finalmente la expresión buscada es: $1386 = 2 \times 3^2 \times 7 \times 11$

Cabe mencionar que tras la función booleana $esPrimo(x)$ usada en el algoritmo de la figura 7.1 hay un proceso bastante tardado. La función debe decidir si su argumento es un número primo o no, es decir si a su vez tiene factores primos (es compuesto) o no, parece que regresamos al punto de partida. Hay diversos mecanismos que permiten decidir si un número n es primo o no. Si es par, por ejemplo, ya sabemos que no es (porque lo divide 2), si no lo divide ningún número (de hecho ningún número impar, usando lo anterior) menor que \sqrt{n} sabemos que si lo es.³

En general lo que suele hacerse en criptografía cuando se desea obtener un número primo, es utilizar *pruebas de primalidad* que proporcionan un cierto grado de certidumbre, arbitrariamente decidido por el usuario, acerca de si un número dado n es primo o no aprovechándose de que los primos poseen propiedades que, en general, es raro que posean el resto de los números. Por ejemplo todo primo n satisface que, para toda b tal que $\text{mcd}(b, n) = 1$ ocurre:

$$b^{n-1} \equiv 1 \pmod{n} \quad (7.2.1)$$

³Si todos los factores primos de n fueran mayores que \sqrt{n} , y n no es primo, entonces sería el producto de dos números, p y q , ambos mayores que \sqrt{n} . Pero entonces $n = pq > \sqrt{n}\sqrt{n} = n$, lo cual es una contradicción.

Si n no es primo puede ser que satisfaga 7.2.1 pero es raro⁴. Si aplicamos repetidamente la prueba a un cierto número n con diferentes valores de b entonces incrementamos nuestra certeza de que n es primo.

Existen diversas pruebas de primalidad, como la de *Solovay-Strassen*, *Lehmann*, *Rabin-Miller* o *Cohen-Lenstra*. Puede profundizarse en el tema consultando [22] o [11].

Evidentemente existen algoritmos mejores para factorizar enteros, pero no mucho mejores. Los algoritmos con complejidad probada son exponenciales y de los que parecen tener complejidad menor, esta aún no se ha demostrado. En síntesis, de todas maneras se tardan mucho para enteros “grandes”, donde “grande” significa de cientos de dígitos. Los ordenes de complejidad de los algoritmos buenos al parecer son una función exponencial con argumento que involucra productos de potencias fraccionarias del logaritmo del tamaño del número. Muchos de estos algoritmos están someramente descritos en las páginas de Web [25, 26], entre ellos el algoritmo de la criba de campo numérico (*Number Field Sieve*) que es probablemente el mejor algoritmo de factorización en general, es decir sin consideraciones de casos especiales, sin suponer nada acerca del número a factorizar. Existen algoritmos muy buenos para factorizar si el número de entrada tiene factores primos pequeños, por ejemplo, cosa que no ocurre con los que se utilizan en criptografía.

En 1977 se publicó en la columna de Martin Gardner en *Scientific American* un número de 129 dígitos (uno de los conocidos retos de RSA) y en 1994 un equipo de voluntarios coordinado por Michael Graff y Paul Leyland intercambiando información por correo electrónico logró factorizarlo utilizando el algoritmo polinomial múltiple de la criba cuadrática (MPQS) de A.K. Lenstra y Manasse tardándose ocho meses. En el proyecto participaron un total de 1600 máquinas trabajando autónomamente. Esto da una buena idea de lo tardado que puede ser factorizar un número grande. Es posible encontrar más información acerca de este proyecto y otros en [27].

En desarrollos puramente teóricos, existe un algoritmo, el de Shor, que en una computadora cuántica tendría una complejidad del orden de $(\ln(N))^3$. Acerca de esto puede hallarse información en [14].

7.3 El logaritmo discreto

En un campo finito de los que solemos utilizar (\mathbb{Z}_p con p primo) también llamados Campos de Galois ($GF(p)$) es fácil elevar un número a una potencia, por ejemplo $2^8 = 256$

⁴A los números que satisfacen 7.2.1 sin ser primos se les denomina números de Carmichael, en 1994 se probó que hay una infinidad de ellos.

Número	Divisor	Cociente
1386	2	693
693	2	No se puede
693	3	231
231	3	77
77	3	No se puede
77	5	No se puede
77	7	11 (primo)

Tabla 7.1: Factorización de 1386.

en \mathbb{Z}_{11} sería:

$$2^8 = 3 \pmod{11}$$

porque $256 \equiv 3 \pmod{11}$.

Pero si se nos dice ¿A qué potencia hay que elevar 2 para obtener 3 en \mathbb{Z}_{11} ? ya no es tan fácil responder, habría que probar con diferentes potencias. El problema es tanto más difícil cuanto más grande sea el tamaño del campo finito⁵.

Por cierto, en \mathbb{R} el problema, además de ser fácil tiene solución siempre que el argumento y la base del logaritmo sean positivos, pero en un campo finito puede ni siquiera tenerla. Por ejemplo: no existe x tal que $3^x = 7$ en \mathbb{Z}_{13} .

Igual que en el caso anterior existen algoritmos más o menos rápidos para calcular el logaritmo discreto en \mathbb{Z}_p si $p - 1$ sólo tiene factores primos pequeños, así que en criptografía se usan campos en los que $p - 1$ tenga al menos un factor primo grande.

Los problemas de factorización y de logaritmo discreto están relacionados. Se ha demostrado que si es resuelto el problema del logaritmo discreto entonces también se resuelve el de la factorización. Al parecer no ocurre exactamente al revés: el problema del logaritmo discreto, según evidencia experimental, es ligeramente más complicado que el de la factorización.

Puede hallarse más información en [29, 2].

⁵De hecho el problema del logaritmo discreto no requiere que la estructura del conjunto sea un campo, basta con que sea un grupo (véase [28]). En criptografía es donde se le refiere a un campo. Por supuesto en un grupo $G(C, *)$ la potencia n -ésima de $a \in C$, a^n , se define como $a * \dots * a$, n veces.

7.4 Mensajes y números

Bueno, ahora sabemos que hay procesos que es fácil hacer en un sentido pero que es difícil invertir y estos procesos son funciones que operan sobre números. Pero si lo que deseamos es cifrar mensajes hechos de letras y números ¿cómo hacemos para transformar un mensaje en un número y así poder usar esas funciones de un solo sentido para cifrarlo?

Dado que en particular en una computadora el mensaje debe ser almacenado carácter por carácter mediante el código binario usado por la computadora para representar información, sea ASCII o algún otro, es posible pensar en utilizar esta representación como el número que representa al mensaje. Así por ejemplo el mensaje “HOLA” podría verse como el número 484F4C41 en hexadecimal dado que el código ASCII de la “H” es 48, el de la “O” es 4F y así sucesivamente.

Otra posibilidad es la utilizada en [11]. Ver a cada carácter de un número como un dígito en base 26 (si sólo utilizamos letras y sólo utilizamos el alfabeto de 26 letras) y ver entonces un mensaje como un número escrito en esa base. Así que la “A” es nuestro cero, luego “B” vale 1, “C” vale 2 y así sucesivamente hasta llegar a “Z” que vale 25. En este esquema el mensaje “HOLA” sería el número:

$$7 \times 26^3 + 14 \times 26^2 + 11 \times 26^1 + 0 \times 26^0 = 132782$$

dado que “H”=7, “O”=14, “L”=11 y “A”=0. Nótese que el valor de nuestros caracteres en este esquema es la diferencia entre el valor que poseían en el esquema anterior y el valor de la “A” también en el esquema anterior.

Ahora bien, dado un mensaje largo, de varias cuartillas por ejemplo, si usamos cualquiera de los esquemas mencionados obtendremos un número gigantesco, ¿como hacemos para manipularlo? Para responder esta pregunta hay que hacer varias consideraciones. Por lo mencionado en las secciones anteriores y lo que veremos más adelante las funciones de un solo sentido que se suelen usar en criptografía involucran números grandes, de cientos de dígitos, así que nuestros mensajes pueden ser también muy grandes. Aun así en varias cuartillas hay muchos cientos de caracteres, podríamos rebasar fácilmente cualquier límite que se nos imponga. Lo que se hace en este caso es fragmentar el mensaje en varios números del tamaño que se este utilizando y listo.

Una vez superado el problema de la representación queda aun el de la manipulación de estos grandes números y para resolverlo se echa mano de las computadoras y de bibliotecas de software especializado en la manipulación de grandes números.

7.5 Idea de la criptografía de llave pública

Como ya dijimos, el hecho de que se requiera un sistema criptográfico presupone la existencia de tres cosas, a saber:

- El enemigo, el que no deseamos que se entere de ciertas cosas.
- Los amigos, aquella persona o personas con las que queremos comunicarnos sin que nadie se entere de lo que nos decimos.
- Un medio de comunicación inseguro, susceptible de ser observado por el enemigo.

Los antiguos mecanismos de cifrado de mensajes basaban su seguridad en mantener oculta (al menos) la clave usada para cifrar los datos. Las partes a comunicarse no podían darse el lujo de decirse la clave a través del canal, dado que este es inseguro, así que debían ponerse de acuerdo por adelantado. En sistemas como el de Vigenère por ejemplo, el enemigo puede conocer el algoritmo, pero no debe conocer la palabra clave usada para cifrar, que es la misma que se usa para descifrar. Los sistemas en los que una sola clave sirve tanto para cifrar como para descifrar se denominan *simétricos*.

El problema de ponerse de acuerdo en una clave se resolvería si hubiera de hecho dos claves, una que cualquiera puede conocer y que por tanto puede decirse a través del canal de comunicación, y otra que sólo sirve para descifrar, que sólo conoce su propietario y, algo muy importante, que no puede ser obtenida fácilmente a partir de la clave para cifrar. Podemos entonces imaginar un esquema como este: A quiere comunicarse secretamente con B y le avisa a B de su deseo, B le envía a A una clave que puede usar para cifrar aquellos mensajes secretos que quiere que sólo conozca B , sólo B conoce una segunda clave capaz de descifrar los mensajes dirigidos a él, nadie, incluyendo a A mismo, es capaz de encontrar en un tiempo razonable la clave secreta que tiene B para descifrar mensajes a pesar de conocer la clave para cifrarlos.

A un sistema criptográfico en el que las claves para cifrar y para descifrar son diferentes se le llama *asimétrico*, la existencia de estos es condición necesaria (pero no suficiente) para lograr lo expuesto en el párrafo anterior. Si el sistema, además de ser asimétrico, posee la cualidad de que una de las claves no proporciona información relevante para obtener la otra, entonces ya reunimos la condición necesaria y suficiente para lograrlo y el problema de distribuir la clave entre los usuarios queda resuelto.

En 1976 un par de ingenieros eléctricos de la Universidad de Stanford, Whitfield Diffie y Martin Hellman, junto con uno de sus estudiantes Ralph Merkle (un estudiante de licenciatura de Berkeley), publicaron un artículo titulado *New Directions in Cryptography* en el

que exponían un mecanismo mediante el cual es posible que dos personas (en general dos entes) pueden ponerse de acuerdo en una palabra clave secreta comunicándose a gritos entre una multitud que no debe conocer su secreto (es decir comunicándose mediante un canal inseguro) y que puede conocer el algoritmo utilizado.

La idea general de los sistemas criptográficos de llave pública es, como ya se mencionó, que existen dos claves: una pública, para cifrar y otra secreta, para descifrar. Es posible pensar en un directorio donde se encuentran las claves de cifrado para varias personas, esas claves están relacionadas íntimamente con las claves para descifrar, cada clave para descifrar es conocida únicamente por la persona a la que le pertenece. Con este esquema cualquier persona A puede enviar un mensaje cifrado a otra B usando la clave pública de cifrado para B , pero solo B puede descifrar el mensaje. El hecho de conocer la clave de cifrado no facilita (al menos no grandemente) la obtención de la clave de descifrado.

En síntesis un sistema de llave pública posee dos propiedades:

1. Cada persona en el sistema posee los medios necesarios para cifrar y descifrar mensajes. Cifrar los destinados a cualquier otra persona y descifrar los que vayan dirigidos a ella.
2. Los medios para descifrar los mensajes de una persona no son obtenibles en un tiempo razonable por cualquier otra persona.

Al parecer la *Agencia de Seguridad Nacional (NSA)* norteamericana ya tenía conocimiento de sistemas criptográficos de llave pública desde 1966, diez años antes de Diffie-Hellman, aunque por supuesto no ha ofrecido pruebas de ello. También el servicio criptográfico británico descubrió por su cuenta la criptografía de llave pública antes de su descubrimiento oficial, lo que se mantuvo en secreto hasta hace poco.

Adicionalmente los sistemas criptográficos de llave pública pueden utilizarse para certificar que un mensaje dado haya sido enviado por un usuario particular del sistema. Supóngase que alguien, A , desea enviar un mensaje de tal forma que, una vez recibido, al destinatario no le quepa duda de que fue A quien lo envió y además nadie que atrape el mensaje pueda obtener información suficiente como para hacerse pasar por A y enviar mensajes a su nombre en el futuro. En un esquema de llave pública bastaría que A añadiera al mensaje información que solo puede ser obtenida usando su llave privada y que, procesándola usando la llave pública de A , haga evidente que solo A pudo enviar el mensaje. A un esquema de certificación de este tipo se le denomina *firma digital*, más adelante veremos como pueden usarse algunos esquemas criptográficos de llave pública para implementar firmas digitales.

$x \in \mathbb{Z}_{11}$	Expresión	Valor nominal
1	2^{10}	1024
2	2^1	2
3	2^8	256
4	2^2	4
5	2^4	16
6	2^9	512
7	2^7	128
8	2^3	8
9	2^6	64
10	2^5	32

Tabla 7.2: Los elementos de \mathbb{Z}_{11} expresados como potencias de 2. Los elementos de la primera columna son los de la última módulo 11.

7.6 Intercambio de llaves (Diffie-Hellman)

Imaginemos que queremos usar un sistema criptográfico simétrico. Hay una sola palabra clave que es utilizada para cifrar y descifrar y esta no debe conocerla el enemigo. Así que ambas partes deben acordar una clave sin que nadie más se entere de ella. El canal de comunicación es inseguro, es escuchado por todo mundo, así que las partes tienen una de dos opciones:

1. Ponerse de acuerdo por adelantado: enviarse la clave a través de un mensajero autorizado o citarse en algún lugar para ponerse de acuerdo sin que nadie escuche.
2. Ponerse de acuerdo usando el canal: de tal manera que nadie pueda obtener la clave (o al menos obtenerla en un tiempo razonable) a partir de la información que es enviada a través del canal por ambas partes.

El esquema de intercambio de llaves de Diffie-Hellman hace posible la segunda alternativa.

En un campo finito \mathbb{Z}_p (p primo por supuesto) un elemento g se dice que es *generador* o *primitivo módulo p* si para cualquier elemento $x \in \mathbb{Z}_p^* = \mathbb{Z}_p \setminus \{0\}$ existe un número n tal que $x = g^n$, es decir $g^n \equiv x \pmod{p}$. Por ejemplo en \mathbb{Z}_{11} , 2 es generador módulo 11 porque todo elemento de $\mathbb{Z}_{11}^* = \{1, \dots, 10\}$ puede escribirse como una potencia de 2 (véase tabla 7.2). Si un elemento a no es primitivo al elevarlo a diferentes potencias se obtienen *algunos* elementos de \mathbb{Z}_p , es decir el conjunto de posibles resultados de a^n es un subconjunto propio de \mathbb{Z}_p , pero si es primitivo el conjunto de posibles resultados de a^n es exactamente todo \mathbb{Z}_p .

Sean Alicia y Bartolo⁶ (A y B respectivamente) las dos partes que desean obtener una clave secreta. El algoritmo de Diffie-Hellman es el siguiente:

1. Alicia y Bartolo se ponen de acuerdo en un par de números: un primo grande p y un primitivo en \mathbb{Z}_p al que llamaremos g . Se pueden poner de acuerdo a gritos.
2. Alicia elige un entero aleatorio grande $\alpha < p - 1$ y le envía a Bartolo

$$X = g^\alpha \pmod{p}$$

3. Bartolo elige un entero aleatorio grande $\beta < p - 1$ y le envía a Alicia

$$Y = g^\beta \pmod{p}$$

4. Alicia calcula $K = Y^\alpha \pmod{p}$.
5. Bartolo calcula $K' = X^\beta \pmod{p}$.

Resulta que $K = K' = g^{\alpha\beta} \pmod{p}$. Nadie que haya estado escuchando la conversación entre Alicia y Bartolo puede calcular K . Sólo conoce p , g , X y Y , para calcular K tendría que poder hacer alguna de las siguientes cosas:

1. Obtener el logaritmo discreto de X o Y en base g módulo p para obtener α y β respectivamente y poder calcular $g^{\alpha\beta} \pmod{p}$.
2. Calcular $g^{\alpha\beta} \pmod{p}$ de alguna otra manera diferente a la opción anterior.

La conjetura de Diffie-Hellman es que 2 no puede hacerse⁷, es decir, forzosamente se tiene que optar por 1. Como sabemos, esto es en general difícil. Para asegurarse de que lo sea realmente debe elegirse p grande y de tal forma que $\frac{p-1}{2}$ sea también primo, g puede elegirse arbitrariamente (de hecho no es necesario que sea primitivo, pero es mejor. ¿Puedes explicar por qué?).

7.7 Algunos preliminares

Ya sabemos que $a \equiv b \pmod{m}$ significa que $m \mid (a - b)$. Recordemos también el siguiente hecho evidente.

⁶Generalmente en los libros y artículos de criptografía se usan *Alice* y *Bob*

⁷Hasta la fecha no se ha probado que 2 no pueda hacerse sin 1, pero no hay nada que asegure que no puede hacerse.

Teorema 7.1 Sean a, b y c en \mathbb{Z} . Si $a \mid b$ y $b \mid c$ entonces $a \mid c$.

Dem.: Como $a \mid b$ entonces es posible escribir $b = ax$ para algún $x \in \mathbb{Z}$. Además $b \mid c$ entonces $c = by$ para algún $y \in \mathbb{Z}$. Sea $z = xy \in \mathbb{Z}$. Entonces $c = axy = az$ por lo que $a \mid c$.
□

Tomando esto en consideración pensemos en el inverso multiplicativo de un elemento en un conjunto \mathbb{Z}_m . Encontrar el inverso de un número $a \in \mathbb{Z}_m$ consiste en hallar $x \in \mathbb{Z}_m$ tal que $ax = 1$ tomando el producto módulo m . Es decir:

$$ax \equiv 1 \pmod{m} \quad (7.7.2)$$

Esta ecuación tiene solución, y es única, si y sólo si a y m son primos relativos, es decir $\text{mcd}(a, m) = 1$ (mcd es *máximo común divisor*). Es fácil demostrarlo.

Teorema 7.2 Un elemento a de \mathbb{Z}_m posee inverso multiplicativo sii $\text{mcd}(a, m) = 1$.

Dem.: Supongamos que:

1. $\text{mcd}(a, m) = d > 1$.
2. Existe $b \in \mathbb{Z}_m$ tal que $ab \equiv 1 \pmod{m}$.

1 significa que:

$$d \mid a \quad (7.7.3)$$

y

$$d \mid m \quad (7.7.4)$$

con $d > 1$.

Por otra parte 2 significa que

$$m \mid (ab - 1) \quad (7.7.5)$$

Así que por 7.7.4 y 7.7.5 usando el teorema 7.1 tenemos que:

$$d \mid (ab - 1) \quad (7.7.6)$$

y por 7.7.3:

$$d \mid ab \quad (7.7.7)$$

```

MCD( $a, b$ )
1  if  $b = 0$  then
2    return  $a$ 
3  else
4    return  $MCD(b, a \pmod{b})$ 
5  endif

```

Figura 7.2: Versión recursiva del algoritmo de Euclides para obtener el máximo común divisor de dos números a y b .

Así que para que se cumplan simultáneamente 7.7.6 y 7.7.7 debe ocurrir que:

$$d \mid 1$$

lo que, por supuesto, es imposible.

El regreso es también fácil y será evidente luego de revisar, más adelante, el algoritmo de Euclides extendido.

□

Por supuesto si estamos en \mathbb{Z}_p , con p primo, entonces todos los elementos tienen inverso multiplicativo porque cualquier número $a \in \mathbb{Z}_p$ distinto de cero está en el conjunto $\{1, \dots, p-1\}$ y resultará primo relativo a p , que es primo.

Recordemos ahora el algoritmo de Euclides para encontrar el máximo común divisor de dos números. El algoritmo, que nos fue planteado en los cursos elementales de álgebra es mostrado en la figura 7.2 en versión recursiva.

Observemos un ejemplo de ejecución de este algoritmo. En la tabla 7.3 se muestran los valores de a y b en las llamadas recursivas que se efectúan para encontrar el mcd de 97 y 77, que por cierto son primos relativos y por tanto $\text{mcd}(97, 77) = 1$.

Por la manera en que se efectúa la llamada en la línea 4 del algoritmo (fig. 7.2) vemos que en la tabla 7.3 tenemos en la llamada i como valor de a (llamémosle a_i en adelante) el valor que tenía b en la llamada anterior (llamémosle b_{i-1}). Por otra parte el argumento que ocupará el valor de b en la llamada que ocurre en la línea 4 es $a \pmod{b}$, el residuo que resulta de dividir a por b . Así que podemos escribir $a_i = q_i b_i + b_{i+1}$, es decir el siguiente valor de b será el residuo de dividir la a actual entre la b actual. Pero el valor actual de a es en realidad el que tenía b en la llamada anterior como habíamos observado, así que:

$$b_{i-1} = q_i b_i + b_{i+1} \tag{7.7.9}$$

llamada	a	b	cociente
0	97	77	1
1	77	20	3
2	20	17	1
3	17	3	5
4	3	2	1
5	2	1	2
6	1	0	

Tabla 7.3: Valores de a y b , argumentos del algoritmo de Euclides en las llamadas recursivas sucesivas para encontrar $\text{mcd}(97, 77)$.

En esencia la ecuación 7.7.9 dice que cualquier residuo (los valores de b) resultado del proceso, puede escribirse como combinación lineal de sus dos predecesores, solo tenemos que reescribir la ecuación como $b_{i+1} = -q_i b_i + b_{i-1}$. Pero si esto es posible entonces podemos irnos hacia atrás escribiendo cada residuo de la expresión como combinación lineal de sus predecesores hasta llegar a los mismísimos a y b . Como el último residuo es $\text{mcd}(a, b)$ esto quiere decir que el máximo común divisor de dos números puede escribirse como combinación lineal de ellos. En particular si los dos números a y b eran primos relativos entonces:

$$1 = \alpha a + \beta b \quad (7.7.10)$$

Ahora bien, si estamos en \mathbb{Z}_b ¿quien es α ? la expresión 7.7.10 dice que $\alpha a \equiv 1 \pmod{b}$ así que α resulta ser el inverso multiplicativo de a en \mathbb{Z}_b .

Al algoritmo de Euclides modificado para obtener la combinación lineal que expresa el mcd se le suele llamar el *algoritmo de Euclides extendido* y en particular es útil para calcular inversos multiplicativos como hemos visto. Ahora es posible echar un nuevo vistazo al teorema 7.2 y pensar en la demostración de “regreso”. Un tratamiento bonito de esto puede hallarse en [4].

Otros dos resultados que nos serán útiles son los siguientes.

Teorema 7.3 (*Teorema de Fermat*). *Sea p un número primo y a un entero. Si $\text{mcd}(p, a) = 1$ (p y a son primos relativos) entonces:*

$$a^{p-1} \equiv 1 \pmod{p}$$

Otra manera de enunciarlo restringiendo el conjunto de posibles valores de a es: Si p es primo entonces

$$a^{p-1} \equiv 1 \pmod{p}$$

para toda a en \mathbb{Z}_p^*

\mathbb{Z}_n^* denota el subconjunto de elementos en \mathbb{Z}_n relativamente primos con n . Por ejemplo $\mathbb{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$. Como en el teorema p es primo entonces de hecho $\mathbb{Z}_p^* = \mathbb{Z}_p$.

Teorema 7.4 *Si n_1, n_2, \dots, n_k son primos relativos por parejas y $n = n_1 n_2 \dots n_k$ entonces, para cualesquiera enteros x y a y toda $i \in \{1, 2, \dots, k\}$ ocurre:*

$$x \equiv a \pmod{n_i}$$

si y sólo si:

$$x \equiv a \pmod{n}$$

este teorema es una consecuencia del conocido *teorema chino del residuo*.

7.8 Transmisión de mensajes (Massey-Omura)

Un mecanismo de cifrado de mensajes similar al intercambio de llaves de Diffie-Hellman es el de Massey-Omura. El algoritmo es el siguiente:

1. Alicia y Bartolo se ponen de acuerdo en un entero primo grande p .
2. Alicia selecciona aleatoriamente un entero e_A entre 0 y $p-1$ tal que $\text{mcd}(e_A, p-1) = 1$ y usando el algoritmo de Euclides calcula su inverso d_A , es decir:

$$e_A d_A \equiv 1 \pmod{(p-1)} \tag{7.8.11}$$

3. Bartolo selecciona aleatoriamente un entero e_B entre 0 y $p-1$ tal que $\text{mcd}(e_B, p-1) = 1$ y usando el algoritmo de Euclides calcula su inverso d_B , es decir:

$$e_B d_B \equiv 1 \pmod{(p-1)} \tag{7.8.12}$$

4. Alicia, quien posee el mensaje a cifrar M , envía a Bartolo:

$$M_A = M^{e_A} \tag{7.8.13}$$

5. Bartolo recibe el mensaje cifrado y calcula

$$M_{A,B} = M_A^{e_B} \quad (7.8.14)$$

lo que envía de vuelta a Alicia.

6. Alicia recibe $M_{A,B}$ y calcula:

$$M' = M_{A,B}^{d_A} \quad (7.8.15)$$

lo que envía a Bartolo.

7. Bartolo calcula:

$$M'' = M'^{d_B} \quad (7.8.16)$$

que resulta ser el mensaje original M .

Veamos por qué esto funciona. De las expresiones 7.8.16 y 7.8.15:

$$M'' = M_{A,B}^{d_A d_B}$$

por 7.8.14 y usando 7.8.12:

$$M'' = (M_A^{e_B})^{d_A d_B} = (M_A^{e_B d_B})^{d_A} = M_A^{d_A} \pmod{p}$$

ahora usando 7.8.13 y 7.8.11:

$$M'' = (M^{e_A})^{d_A} = M^{e_A d_A} = M \pmod{p}$$

7.9 Cifrado y firma (ElGamal)

En 1984 Taher ElGamal⁸ propuso un esquema que permite tanto el cifrado de mensajes como la implementación de firmas digitales. El procedimiento común a ambos usos es el siguiente:

1. Elegir los siguientes números:

- Un primo p (grande). Esto determina el campo finito donde se efectuarán las operaciones.

⁸Cuyo nombre es frecuente encontrar como *Elgamal* (corrompiendo la sintaxis árabe). Fue consultor de Netscape en el desarrollo del protocolo SSL, antes trabajo en *RSA Data Security* y actualmente tiene su propia compañía llamada *Securify*.

- Un número aleatorio $g < p$, generador en \mathbb{Z}_p .
 - Cada usuario u elige un número aleatorio $x_u < p$
2. Calcular

$$y_u = g^{x_u} \pmod{p} \quad (7.9.20)$$

3. Pueden hacerse públicos y_u , g y p . Debe mantenerse en secreto x_u . De hecho p y g son públicos y comunes para todos los usuarios del sistema de llave pública (todo el directorio de personas y llaves), y_u es particular de cada usuario.

7.9.1 Cifrado de mensajes

Imaginemos que Alicia (A) desea enviar un mensaje (M) a Bartolo (B) usando el esquema de ElGamal. Alicia conoce g , p y y_B y procede a hacer lo siguiente:

1. Elegir k aleatoriamente de tal forma que sea primo relativo a $p - 1$, k se mantendrá en secreto.

2. Calcular

$$a = g^k \pmod{p} \quad (7.9.21)$$

y

$$b = y_B^k M \pmod{p} \quad (7.9.22)$$

3. El mensaje cifrado es la pareja a , b .

La expresión 7.9.22 implica que

$$\frac{b}{a^{x_B}} \equiv \frac{y_B^k M}{a^{x_B}} \pmod{p}$$

Usando 7.9.20 obtenemos:

$$\frac{y_B^k M}{a^{x_B}} \equiv \frac{g^{k x_B} M}{a^{x_B}} \pmod{p} \quad (7.9.24)$$

Por otra parte, de la expresión 7.9.21 tenemos:

$$a^{x_B} \equiv g^{k x_B} \pmod{p}$$

Usando esto en 7.9.24 tenemos finalmente:

$$\frac{y_B^k M}{a^{x_B}} \equiv \frac{g^{kx_B} M}{a^{x_B}} \equiv \frac{g^{kx_B} M}{g^{kx_B}} \equiv M \pmod{p} \quad (7.9.26)$$

Así que para descifrar el mensaje se calcula

$$M = \frac{b}{a^{x_B}} \pmod{p}$$

que puede efectuar solo quien posea la clave secreta x_B . Así que este esquema de cifrado se basa en la hipótesis de que el logaritmo discreto es difícil de calcular.

7.9.2 Firma digital

Sea M el mensaje que se desea firmar y A el usuario que desea firmarlo para enviarlo a B .

1. A elige un número aleatorio k primo relativo a $p - 1$ que mantendrá en secreto.
2. Calcula $a = g^k \pmod{p}$.
3. Encontrar (usando el algoritmo de Euclides extendido) b tal que $M = (x_A a + kb) \pmod{(p - 1)}$
4. El mensaje M es firmado con a y b .

Para confirmar que la firma es válida B procede a calcular: $y_A^a a^b \pmod{p}$ cuyo valor debe coincidir con: $g^M \pmod{p}$.

¿Quién hubiera podido enviar M , a y b de tal forma que usando a y b de la manera descrita se obtiene M ? aparentemente solo quien conoce x_A y ese debe ser A mismo, así que recibir la terna M , a y b certifica que M fue enviado por A .

7.10 RSA

En 1977 tres científicos de la computación adscritos al MIT, Ron Rivest, Adi Shamir y Leonard Adleman, desarrollaron un algoritmo de cifrado y firma digital conocido como RSA, las iniciales de los apellidos de los creadores. El algoritmo es el siguiente:

1. Elegir dos números primos grandes p y q aleatoriamente.
2. Calcular $n = pq$.
3. Elegir una llave de cifrado e tal que sea primo relativo con $(p-1)(q-1)$.
4. Calcular d , la llave de descifrado, de tal forma que:

$$ed \equiv 1 \pmod{(p-1)(q-1)} \quad (7.10.28)$$

lo que puede hacerse, como hemos visto, usando el algoritmo extendido de Euclides. Básicamente hay que calcular el inverso de e módulo $(p-1)(q-1)$, que existe porque e y $(p-1)(q-1)$ son primos relativos.

5. e y n son la llave pública, d es la privada. p y q no deben revelarse, aunque no se utilizarán más.

Para cifrar un mensaje M (de longitud menor a n) se calcula el mensaje cifrado C como sigue:

$$C = M^e \pmod{n} \quad (7.10.29)$$

Para descifrar, operación que sólo puede llevar a cabo el destinatario real del mensaje, se calcula:

$$M' = C^d \pmod{n} \quad (7.10.30)$$

M' resulta ser el mensaje original M .

Veamos ahora por qué esto funciona.

Por la expresión 7.10.28:

$$ed = k(p-1)(q-1) + 1 \quad (7.10.31)$$

de 7.10.30 usando 7.10.29:

$$M' = C^d \pmod{n} = (M^e)^d \pmod{n} = M^{ed} \pmod{n}$$

usando 7.10.31:

$$M' = M^{k(p-1)(q-1)+1} \pmod{n} \quad (7.10.32)$$

n es producto de p y q así que hay dos factores que debemos observar:

- Si nos fijamos en p . Dado que p es primo (sólo lo dividen p mismo y 1) hay dos posibilidades para $\text{mcd}(M, p)$

1. $\text{mcd}(M, p) = 1$ entonces se cumplen las hipótesis del teorema de Fermat (7.3) y tenemos:

$$M^{p-1} \equiv 1 \pmod{p}$$

elevando a la $k(q-1)$ ambos lados:

$$M^{k(q-1)(p-1)} \equiv 1^{k(q-1)} \pmod{p}$$

lo que es congruente con 1 módulo p , es decir:

$$M^{k(q-1)(p-1)} \equiv 1 \pmod{p}$$

Multiplicando ahora por M :

$$M \cdot M^{k(q-1)(p-1)} \equiv M \pmod{p}$$

de donde:

$$M^{k(q-1)(p-1)+1} \equiv M \pmod{p} \quad (7.10.33)$$

2. $\text{mcd}(M, p) = p$, es decir M es múltiplo de p . En este caso 7.10.33 sigue siendo verdadera dado que ambos lados son, de hecho, congruentes con cero módulo p .

- Si nos fijamos en q tendremos, por razones análogas:

$$M^{k(q-1)(p-1)+1} \equiv M \pmod{q} \quad (7.10.34)$$

Como p y q son dos primos diferentes y n es el producto de ellos, de 7.10.33 y 7.10.34 tenemos:

$$M^{k(q-1)(p-1)+1} \equiv M \pmod{n} \quad (7.10.35)$$

Así que en 7.10.32 finalmente:

$$M' = M \pmod{n}$$

Otra vez se recomienda en general usar primos p y q tales que $p-1$ y $q-1$ tengan, al menos, un factor primo grande. A los números primos que cumplen con características como esta y otras se les suele llamar *primos fuertes*. Avances recientes en factorización han hecho que ya no sea muy relevante el usar primos fuertes, pero se sigue recomendando.

RSA puede usarse también para firmas digitales. Si A desea enviar un mensaje firmado a B entonces, luego de enviar el mensaje M (cifrado o no) a B manda $M_f = M^d$. Una vez recibido esto B calcula M_f^e y debe obtener M nuevamente, si es así puede estar seguro de que quien envió el mensaje fue A y nadie más, porque nadie más conocería la llave de cifrado de A que puede ser invertida solo por la de descifrado que todos conocen.

Nótese que n es pública y es el producto de los dos primos usados para obtener la llave de descifrado (privada) a partir de la de cifrado (que también es pública), así que podría romperse la seguridad de RSA si se lograra descomponer n en sus factores primos. La seguridad de RSA está basada en el supuesto de que factorizar es una tarea difícil, cosa que no ha sido demostrada. Lo que sí está demostrado es que romper la seguridad de RSA es equivalente a factorizar el módulo $n = pq$. Es decir si factorizamos n rompemos la seguridad de RSA (lo que resulta evidente). Por otra parte si rompemos la seguridad de RSA de alguna manera, entonces obtenemos d y podemos usar esta información para obtener los factores primos de n .

Nótese que hablamos de factorizar el módulo de RSA, no de factorizar en general. Es mucho más fácil factorizar el producto de dos primos que factorizar el producto de $k > 2$ primos, así que romper RSA no es equivalente a factorizar en general.

INTERMEZZO C

Software criptográfico

C.1 La contraseña en UNIX

En 1979 Morris y Thompson escribieron el artículo [13] que definió el estándar utilizado por las contraseñas de los usuarios en sistemas UNIX. Originalmente los sistemas UNIX guardaban las contraseñas de cada usuario en un archivo. Por supuesto una falla en la seguridad del sistema ocasionaba que se obtuvieran ilegalmente todas las contraseñas de una sola vez. Luego el esquema de contraseñas se cambió, en vez de registrar la contraseña en texto plano se registraba una versión cifrada de la misma. Cuando el usuario tecleaba su contraseña esta se cifraba y el resultado se comparaba con la versión registrada, si coincidían se dejaba pasar al usuario, si no se le volvía a solicitar la contraseña. El esquema criptográfico usado era el implementado por una máquina de cifrado usada por el ejército de Estados durante la segunda guerra mundial, la M-209. Pero el esquema era muy susceptible a ataques debido a que, como suele suceder también hoy en día, las contraseñas de los usuarios suelen ser pequeñas.

Se optó por cambiar el esquema criptográfico a una variante del estándar DES, ya mencionado. Al igual que en la versión anterior, en el sistema se registra una versión cifrada de la contraseña. Cuando se dá de alta un usuario nuevo en el sistema y este introduce su contraseña por primera vez, el sistema observa la hora actual en el reloj, con base en la hora se determina un número de 12 bits de longitud, que es almacenado en los seis bits menos significativos de dos bytes (caracteres). A este número de 12 bits se le denomina normalmente *salt*. Por otra parte la contraseña del usuario está constituida de ocho caracteres, de los 64 bits contenidos en estos ocho bytes realmente se utilizan sólo los 56 bits constituidos por los siete bits menos significativos de cada byte, tal como ocurre en DES. Estos dos elementos: el *salt* y la contraseña del usuario, son la entrada requerida por el algoritmo que obtiene la versión cifrada de la contraseña. En los sistemas UNIX existe una función de biblioteca para lenguaje C llamada *crypt* que, de hecho, es la implementación del algoritmo de cifrado.

Pero..., nos falta algo. Sabemos que a DES entran dos cosas del mismo tamaño, un texto a cifrar y una llave, ambos de 64 bits. En nuestra descripción previa del algoritmo de cifrado de contraseñas de UNIX solo hemos mencionado uno de esos elementos, el *salt* es muy corto para ser el otro. En efecto, en la versión de DES usada por *unix* la contraseña tecleada por el usuario hace las veces de la clave de DES y se utiliza para cifrar un texto plano predefinido como nulo, es decir 64 bits en cero. Nótese que lo que se cifra *no* es la contraseña del usuario, sino un texto arbitrario predefinido como nulo, esto es así porque DES es reversible. Recordemos que si se introduce como texto de entrada a DES el resultado de cifrar con DES mismo un cierto texto original y se reordena la clave usada para cifrarlo, el resultado es el texto original. El hecho de que lo que la contraseña del usuario sea usada como la clave DES y no como el texto a cifrar hace que la propiedad de reversibilidad de DES ya no sea útil para que un atacante sea capaz de recuperar la contraseña del usuario a partir del resultado de DES.

En el esquema original las contraseñas “cifradas” como se ha descrito se guardan, junto con los identificadores de usuario y otra información, en el archivo de acceso irrestricto */etc/passwd*. En el campo especificado para almacenar la contraseña de cada usuario aparecen 13 caracteres: los dos que contienen el *salt* y 11 caracteres que contienen, en los seis bits menos significativos de los primeros 10 caracteres (de derecha a izquierda) y los cuatro primeros bits del onceavo byte; los 64 bits que resultan de aplicar el algoritmo de cifrado a texto nulo usando como clave la contraseña del usuario. Si la contraseña del usuario fuera usada como el texto a cifrar con una clave DES arbitraria predefinida, cualquier otro usuario del sistema podría recuperar la contraseña de cualquier otro aprovechando la propiedad de reversibilidad de DES.

Aún falta por aclarar la utilidad del *salt*. Esencialmente el *salt* es utilizado para ampliar el espacio de búsqueda de la clave de un usuario por parte de un atacante. El *salt* es

un número de 12 bits que especifica una de un total de 4096 posibles permutaciones que alteran el esquema original de DES. La permutación definida por el *salt* tiene lugar luego de la etapa de permutación expansiva de la mitad derecha del texto de entrada a cada etapa (véase la sección dedicada a DES), inmediatamente antes de que se lleve a cabo el XOR de la parte derecha del texto expandido y la clave. Esta permutación, por cierto, intercambia bits entre la primera y la tercera cuarta parte de la palabra de 48 bits que sale de la etapa de permutación expansiva, es decir intercambia bits cuyos índices están entre 1 y 12 con bits cuyos índices están entre 25 y 36. Si un atacante que trata de encontrar la contraseña de otro, consigue los 64 bits (de hecho, como sabemos, los 56 bits) que aparecen en `/etc/passwd` asociados a su víctima y quiere encontrar por fuerza bruta la contraseña que los produce, tendrá que encontrar también uno, de un total de 4096 valores, que alteran DES. Es como “agrandar” artificialmente la contraseña del usuario.

Recordemos que DES está constituido de 16 etapas casi idénticas. Pues bien, en UNIX además se aplican 25 iteraciones sucesivas de DES (formalmente hablando, de la variante de DES que describimos). Cuando se introduce la contraseña de un usuario se hace lo siguiente:

1. Se usa al login del usuario para consultar la base de datos de usuarios (originalmente `/etc/passwd`) para saber:
 - (a) El valor de *salt* (los seis bits menos significativos, de los dos primeros bytes del campo asociado a la contraseña en la base de datos, concatenados). Este es un número de 12 bits al que llamaremos *s*
 - (b) El valor que se debe obtener luego de cifrar la contraseña del usuario, que son los seis bits menos significativos de los 10 últimos bytes del campo asociado a la contraseña y los cuatro primeros bits del tercer byte. A este número de 64 bits le llamaremos *c*.
2. Se define como texto a cifrar, t_1 , el número cero escrito en 64 bits.
3. Desde $i = 1$ y hasta que $i = 25$ inclusive repetir:
 - (a) Se introducen al algoritmo modificado de DES: el texto a cifrar, t_i , el número *c* como clave y el número *s* como añadido a la permutación de extensión de cada etapa. El resultado obtenido por las 16 etapas de DES modificado es un texto cifrado de 64 bits al que llamaremos r_i
 - (b) Sea $t_{i+1} = r_i$
4. Se compara r_{25} con *c*. Si ambos son iguales el usuario es validado y se le permite el acceso al sistema, de otro modo se le niega el acceso.

En la mayoría de los sistemas UNIX modernos la base de datos de usuario ya no posee acceso irrestricto, generalmente solo puede ser leída por el superusuario del sistema (*root*), lo que comúnmente se llama *contraseñas ocultas* (*shadow passwords*). Además se han inventado otras variantes de algoritmos de cifrado (algunos sistemas usan MD5 como esquema de verificación en vez de DES) y se han incrementado las longitudes de las llaves (contraseñas), algo que, por cierto puede no resultar bueno (véase por ejemplo lo dicho al respecto en [10]).

Bibliografía

- [1] Arazi, Benjamin, *A Commonsense Approach to the Theory of Error Correcting Codes*, MIT press, 1988, Computer Systems Series.
- [2] Arnaud, Denis, *Security Status and Issues, Electronic Commerce on the Internet*, <http://denis.arnaud.free.fr/perso/reports.html>, 1995.
- [3] Baylis, John, *Error-Correcting Codes, A Mathematical Introduction*, Chapman & Hall Mathematics, 1998.
- [4] Bressoud, David y Stan Wagon, *A Course in Computational Number Theory*, Key College Publishing-Springer Verlag, 2000.
- [5] Burrows M. y D. J. Wheeler, "A Block-sorting Lossless Data Compression Algorithm", *Systems Research Center Report 124*, Digital Equipment Corporation, 10 de mayo de 1994,
<http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-124.html>.
- [6] Gailly, Jean-Loup (editor), *Compression FAQ Part 2: Introduction to Data Compression Techniques*, septiembre de 1999,
<http://www.faqs.org/faqs/compression-faq>.
- [7] Hamming, Richard, *Coding and Information Theory*, Prentice Hall, 1980.

- [8] Held, Gilbert, *Data Compression, Techniques and Applications, Hardware and Software Considerations*, 2a. ed., John Wiley & Sons, 1987. (con software elaborado por Thomas Marshall).
- [9] Hill, Raymond, *A First Course in Coding Theory*, Oxford University Press, 1986, Oxford Applied Mathematics and Computing Sciences Series.
- [10] Jackson, Michael, *Linux Shadow Password HOWTO*, Linux Documentation Project, 1996.
- [11] Koblitz, Neal, *A Course in Number Theory and Cryptography*, 2a ed., Springer Verlag, 1994, Graduate Texts in Mathematics.
- [12] McIlroy, M. D., "The number of 1's in binary integers: bounds and extremal properties", *SIAM Journal of Computing*, 3, 255, 1974.
- [13] Morris, Robert y Ken Thompson, "Password security: A case history", *Communications of the ACM*, 22(11), 1979, pp. 594-597.
- [14] Muthukrishnan, Ashok, *Quantum Factoring and Search Algorithms*, <http://www.optics.rochester.edu:8080/users/stroud/talks/muthukrishnan992/>, 1999.
- [15] Nelson, Mark, "Data Compression with the Burrows-Wheeler Transform", *Dr. Dobbs Journal*, septiembre 1996, <http://dogma.net/markn/articles/bwt/bwt.htm>.
- [16] Nelson, Mark y Jean-Loup Gailly, *The Data Compression Book*, 2a ed., M & T Books, 1995.
- [17] Reed, I. S. y G. Solomon, "Polynomial Codes over Certain Finite Fields", *SIAM, Journal of Applied Mathematics*, vol. 8, junio de 1960, pp. 300-304.
- [18] Roman, Steven, *Introduction to Coding and Information Theory*, Springer Verlag, 1996, *Undergraduate Texts in Mathematics*.
- [19] Roman, Steven, *Coding and Information Theory*, Springer Verlag, 1992, *Graduate Texts in Mathematics*.
- [20] Rosing, Michael, *Implementing Elliptic Curve Cryptography*, Manning, 1999.
- [21] Santa Cruz, Diego, T. Ebrahimi y C. Christopoulos, "The JPEG 2000 Image Coding Standard", *Dr. Dobb's Journal*, 323, abril de 2001, pp. 46-54.
- [22] Schneier, Bruce, *Applied Cryptography*, 2a ed., John Wiley & Sons, 1996.

-
- [23] Ziv, J. y A. Lempel, "A universal algorithm for sequential data compression", *IEEE Transactions on Information Theory*, 23(3), mayo 1977, pp. 337-343.
- [24] Ziv, J. y A. Lempel, "Compression of individual sequences via variable-rate coding", *IEEE Transactions on Information Theory*, 23(5), septiembre 1978, pp. 530-536.
- [25] <http://www.frenchfries.net/paul/factoring/theory/>
- [26] <http://www.iks-jena.de/mitarb/lutz/security/cryptfaq/q48.html>
- [27] <http://www.npac.syr.edu/factoring/overview.html>
- [28] <http://www.rsasecurity.com/rsalabs/faq/A-3.html>
- [29] <http://www.rsasecurity.com/rsalabs/faq/2-3-7.html>