Apéndice 1. Unidades de medida y sistemas de numeración

Ap1.1. bytes, kilobytes, megabytes...

En informática, la unidad básica de información es el **byte**. En la práctica, podemos pensar que un byte es el equivalente a una **letra**. Si un cierto texto está formado por 2000 letras, podemos esperar que ocupe unos 2000 bytes de espacio en nuestro disco.

Eso sí, suele ocurrir que realmente un texto de 2000 letras que se guarde en el ordenador ocupe más de 2000 bytes, porque se suele incluir información adicional sobre los tipos de letra que se han utilizado, cursivas, negritas, márgenes y formato de página, etc.

Un byte se queda corto a la hora de manejar textos o datos algo más largos, con lo que se recurre a un múltiplo suyo, el **kilobyte**, que se suele abreviar **Kb** o K.

En teoría, el prefijo kilo querría decir "mil", luego un kilobyte debería ser 1000 bytes, pero en los ordenadores conviene buscar por comodidad una potencia de 2 (pronto veremos por qué), por lo que se usa 2^{10} =1024. Así, la equivalencia exacta es 1 Kb = 1024 bytes.

Los Kb eran unidades típicas para medir la memoria de ordenadores: 640 Kb fue mucho tiempo la memoria habitual en los primeros IBM PC y equipos similares. Por otra parte, una página mecanografiada suele ocupar entre 2 K (cerca de 2000 letras) y 4 Kb.

Cuando se manejan datos más extensos, necesitamos otro múltiplo, el **megabyte** o Mb, que es 1000 Kb (en realidad 1024 Kb) o algo más de un millón de bytes. Por ejemplo, en un diskette cabían 1.44 Mb, y en un Compact Disc para ordenador (CD-ROM) se pueden almacenar hasta 700 Mb.

Para unidades de almacenamiento de gran capacidad, su tamaño no se suele medir en megabytes, sino en el múltiplo siguiente: en **gigabytes**, con la correspondencia 1 Gb = 1024 Mb. Así, es habitual que un equipo actual tenga una memoria RAM entre 4 y 8 Gb, así como un disco duro de entre 500 y 2.000 Mb (2 **Terabytes**).

Todo esto se puede resumir así:

Unidad	Equivalencia	Valores posibles
Byte	-	0 a 255 (para guardar 1 letra)
Kilobyte (K o Kb)	1024 bytes	Aprox. media página mecanografiada
Megabyte (Mb)	1024 Kb	-
Gigabyte (Gb)	1024 Mb	-
Terabyte (Tb)	1024 Gb	-

Ejercicios propuestos:

(Ap1.1.1) ¿Cuántas letras se podrían almacenar en una agenda electrónica que tenga 32 Kb de capacidad?

(Ap1.1.2) Si suponemos que una canción típica en formato MP3 ocupa cerca de 3.500 Kb, ¿cuántas se podrían guardar en un reproductor MP3 que tenga 256 Mb de capacidad?

(Ap1.1.3) ¿Cuántos diskettes de 1,44 Mb harían falta para hacer una copia de seguridad de un ordenador que tiene un disco duro de 6,4 Gb? ¿Y si usamos compact disc de 700 Mb, cuántos necesitaríamos?

(Ap1.1.4) ¿A cuantos CD de 700 Mb equivale la capacidad de almacenamiento de un DVD de 4,7 Gb? ¿Y la de uno de 8,5 Gb?

Ap1.2. Unidades de medida empleadas en informática (2): los bits

Dentro del ordenador, la información se debe almacenar realmente de alguna forma que a él le resulte "cómoda" de manejar. Como la memoria del ordenador se basa en componentes electrónicos, la unidad básica de información será que una posición de memoria esté usada o no (totalmente llena o totalmente vacía), lo que se representa como un 1 o un 0. Esta unidad recibe el nombre de **bit**.

Un bit es demasiado pequeño para un uso normal (recordemos: sólo puede tener dos valores: 0 ó 1), por lo que se usa un conjunto de ellos, 8 bits, que forman un **byte**. Las matemáticas elementales (combinatoria) nos dicen que si agrupamos los bits de 8 en 8, tenemos 256 posibilidades distintas (variaciones con repetición de 2 elementos tomados de 8 en 8: $VR_{2,8}$):

```
00000000
00000001
00000010
00000011
```

```
00000100
...
11111110
11111111
```

Por tanto, si en vez de tomar los bits de 1 en 1 (que resulta cómodo para el ordenador, pero no para nosotros) los utilizamos en grupos de 8 (lo que se conoce como un byte), nos encontramos con 256 posibilidades distintas, que ya son más que suficientes para almacenar una letra, o un signo de puntuación, o una cifra numérica o algún otro símbolo.

Por ejemplo, se podría decir que cada vez que encontremos la secuencia 00000010 la interpretaremos como una letra A, y la combinación 00000011 como una letra B, y así sucesivamente (aunque existe un estándar distinto, que comentaremos un poco más adelante).

También existe una correspondencia entre cada grupo de bits y un número del 0 al 255: si usamos el sistema binario de numeración (que aprenderemos dentro de muy poco), en vez del sistema decimal, tenemos que:

```
0000 0000 (binario) = 0 (decimal)
0000 0001 (binario) = 1 (decimal)
0000 0010 (binario) = 2 (decimal)
0000 0011 (binario) = 3 (decimal)
...
1111 1110 (binario) = 254 (decimal)
1111 1111 (binario) = 255 (decimal)
```

En la práctica, existe un código estándar, el **código ASCII** (American Standard Code for Information Interchange, código estándar americano para intercambio de información), que relaciona cada letra, número o símbolo con una cifra del 0 al 255 (realmente, con una secuencia de 8 bits): la "a" es el número 97, la "b" el 98, la "A" el 65, la "B", el 32, el "0" el 48, el "1" el 49, etc. Así se tiene una forma muy cómoda de almacenar la información en ordenadores, ya que cada letra ocupará exactamente un byte (8 bits: 8 posiciones elementales de memoria). En el próximo apéndice tendrás un extracto de este código.

Aun así, hay un inconveniente con el código ASCII: sólo los primeros 127 números son estándar. Eso quiere decir que si escribimos un texto en un ordenador y lo llevamos a otro, las letras básicas (A a la Z, 0 al 9 y algunos símbolos) no cambiarán, pero las letras internacionales (como la Ñ o las vocales con acentos) puede que no aparezcan correctamente, porque se les asignan números que no son estándar para todos los ordenadores. Por eso, existen estándares más modernos, como UTF-8, que comentaremos en el siguiente apartado.

Nota: Eso de que realmente el ordenador trabaja con ceros y unos, por lo que le resulta más fácil manejar los números que son potencia de 2 que los números que no lo son, es lo que explica que el prefijo *kilo* no quiera decir "exactamente mil", sino que se usa la potencia de 2 más cercana: 2^{10} =1024. Por eso, la equivalencia exacta es **1 K = 1024 bytes**.

Ejercicios propuestos:
(Ap1.2.1) Un número entero largo de 64 bits, ¿cuántos bytes ocupa?
(Ap1.2.2) En una conexión de red de 100 Mbits/segundo, ¿cuánto tiempo tardaría en enviar 630 Mbytes de datos?

Apéndice 2. El código ASCII

El nombre del código ASCII viene de "American Standard Code for Information Interchange", que se podría traducir como Código Estándar Americano para Intercambio de Información.

La idea de este código es que se pueda compartir información entre distintos ordenadores o sistemas informáticos. Para ello, se hace corresponder una letra o carácter a cada secuencia de varios bits.

El código ASCII estándar es de 7 bits, lo que hace que cada grupo de bits desde el 0000000 hasta el 1111111 (0 a 127 en decimal) corresponda siempre a la misma letra.

Por ejemplo, en cualquier ordenador que use código ASCII, la secuencia de bits 1000001 (65 en decimal) corresponderá a la letra "A" (a, en mayúsculas).

Los códigos estándar "visibles" van del 32 al 127. Los códigos del 0 al 31 son códigos de control: por ejemplo, el caracter 7 (BEL) hace sonar un pitido, el caracter 13 (CR) avanza de línea, el carácter 12 (FF) expulsa una página en la impresora (y borra la pantalla en algunos ordenadores), etc.

Estos códigos ASCII estándar son:

	0	1	2	3	4	5	6	7	8	9
000	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
010	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
020	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
030	RS	US	SP	!	"	#	$	%	&	'
040	(	)	*	+	,	-	.	/	0	1
050	2	3	4	5	6	7	8	9	:	;
060	<	=	>	?	@	A	B	C	D	E
070	F	G	H	I	J	K	L	M	N	O
080	P	Q	R	S	T	U	V	W	X	Y
090	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~	·		

Hoy en día, casi todos los ordenadores incluyen un código ASCII extendido de 8 bits, que permite 256 símbolos (del 0 al 255), lo que permite que se puedan usar también vocales acentuadas, eñes, y otros símbolos para dibujar recuadros, por ejemplo, aunque estos símbolos que van del número 128 al 255 no son estándar,

de modo que puede que otro ordenador no los interprete correctamente si el sistema operativo que utiliza es distinto.

Una alternativa más moderna es el estándar UTF-8, que permite usar caracteres de más de 8 bits (típicamente 16 bits, que daría lugar a 65.536 símbolos distintos), lo que permite que la transferencia de información entre distintos sistemas no tenga problemas de distintas interpretaciones.

Ejercicios propuestos:

(Ap2.1) Crea un programa en C# que muestre una tabla ASCII básica, desde el carácter 32 hasta el 127, en filas de 16 caracteres cada una.

Apéndice 3. Sistemas de numeración.

Ap3.1. Sistema binario

Nosotros normalmente utilizamos el **sistema decimal** de numeración: todos los números se expresan a partir de potencias de 10, pero normalmente lo hacemos sin pensar.

Por ejemplo, el número 3.254 se podría desglosar como:

$$3.254 = 3 \cdot 1000 + 2 \cdot 100 + 5 \cdot 10 + 4 \cdot 1$$

o más detallado todavía:

$$254 = 3 \cdot 10^3 + 2 \cdot 10^2 + 5 \cdot 10^1 + 4 \cdot 10^0$$

Para los ordenadores no es cómodo contar hasta 10. Como partimos de "casillas de memoria" que están completamente vacías (0) o completamente llenas (1), sólo les es realmente cómodo contar con 2 cifras: 0 y 1.

Por eso, dentro del ordenador cualquier número se deberá almacenar como ceros y unos, y entonces los números se deberán desglosar en potencias de 2 (el llamado "**sistema binario**"):

$$13 = 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1$$

o, más detallado:

$$13 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

de modo que el número decimal 13 se escribirá en binario como 1101.

En general, **convertir** un número binario al sistema decimal es fácil: lo expresamos como suma de potencias de 2 y sumamos:

$$0110\ 1101 \text{ (binario)} =$$
$$0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 =$$
$$0 \cdot 128 + 1 \cdot 64 + 1 \cdot 32 + 0 \cdot 16 + 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 =$$
$$109 \text{ (decimal)}$$

Convertir un número de decimal a binario resulta algo menos intuitivo. Una forma sencilla es ir dividiendo entre las potencias de 2, y coger todos los cocientes de las divisiones:

```
109 / 128 = 0 (resto: 109)
109 / 64 = 1 (resto: 45)
45 / 32 = 1 (resto: 13)
13 / 16 = 0 (resto: 13)
13 / 8 = 1 (resto: 5)
5 / 4 = 1 (resto: 1)
1 / 2 = 0 (resto: 1)
1 / 1 = 1 (hemos terminado).
```

Si "juntamos" los cocientes que hemos obtenido, aparece el número binario que buscábamos:

```
109 decimal = 0110 1101 binario
```

(Nota: es frecuente separar los números binarios en grupos de 4 cifras -medio byte- para mayor legibilidad, como yo he hecho en el ejemplo anterior; a un grupo de 4 bits se le llama **nibble**).

Otra forma sencilla de convertir de decimal a binario es dividir consecutivamente entre 2 y coger los restos que hemos obtenido, pero en orden inverso:

```
109 / 2 = 54, resto 1
54 / 2 = 27, resto 0
27 / 2 = 13, resto 1
13 /2 = 6, resto 1
6 / 2 = 3, resto 0
3 / 2 = 1, resto 1
1 / 2 = 0, resto 1
(y ya hemos terminado)
```

Si leemos esos restos de abajo a arriba, obtenemos el número binario: 1101101 (7 cifras, si queremos completarlo a 8 cifras rellenamos con ceros por la izquierda: 01101101).

¿Y se pueden hacer operaciones con números binarios? Sí, casi igual que en decimal:

```
0·0 = 0      0·1 = 0      1·0 = 0      1·1 = 1
0+0 = 0      0+1 = 1      1+0 = 1      1+1 = 10   (en decimal: 2)
```

Ejercicios propuestos:

(Ap3.1.1) Expresa en sistema binario los números decimales 17, 101, 83, 45.

(Ap3.1.2) Expresa en sistema decimal los números binarios de 8 bits: 01100110, 10110010, 11111111, 00101101

(Ap3.1.3) Suma los números 01100110+10110010, 11111111+00101101. Comprueba el resultado sumando los números decimales obtenidos en el ejercicio anterior.

(Ap3.1.4) Multiplica los números binarios de 4 bits 0100·1011, 1001·0011. Comprueba el resultado convirtiéndolos a decimal.

(Ap3.1.5) Crea un programa en C# que convierta a binario los números (en base 10) que introduzca el usuario.

(Ap3.1.6) Crea un programa en C# que convierta a base 10 los números binarios que introduzca el usuario.

Ap3.2. Sistema octal

Hemos visto que el sistema de numeración más cercano a como se guarda la información dentro del ordenador es el sistema binario. Pero los números expresados en este sistema de numeración "ocupan mucho". Por ejemplo, el número 254 se expresa en binario como 11111110 (8 cifras en vez de 3).

Por eso, se han buscado otros sistemas de numeración que resulten más "compactos" que el sistema binario cuando haya que expresar cifras medianamente grandes, pero que a la vez mantengan con éste una correspondencia algo más sencilla que el sistema decimal. Los más usados son el sistema octal y, sobre todo, el hexadecimal.

El sistema octal de numeración trabaja en base 8. La forma de convertir de decimal a binario será, como siempre dividir entre las potencias de la base. Por ejemplo:

```
254 (decimal) ->
254 / 64 = 3 (resto: 62)
62 / 8 = 7 (resto: 6)
6 / 1 = 6 (se terminó)
```

de modo que

$$254 = 3 \cdot 8^2 + 7 \cdot 8^1 + 6 \cdot 8^0$$

o bien

```
254 (decimal) = 376 (octal)
```

Hemos conseguido otra correspondencia que, si bien nos resulta a nosotros más incómoda que usar el sistema decimal, al menos es más compacta: el número 254 ocupa 3 cifras en decimal, y también 3 cifras en octal, frente a las 8 cifras que necesitaba en sistema binario.

Pero además existe una correspondencia muy sencilla entre el sistema octal y el sistema binario: si agrupamos los bits de 3 en 3, el paso de **binario a octal** es rapidísimo

```
254 (decimal) = 011 111 110 (binario)

011 (binario ) = 3 (decimal y octal)
111 (binario ) = 7 (decimal y octal)
110 (binario ) = 6 (decimal y octal)
```

de modo que

```
254 (decimal) = 011 111 110 (binario) = 376 (octal)
```

El paso desde el **octal al binario** y al decimal también es sencillo. Por ejemplo, el número 423 (octal) sería 423 (octal) = 100 010 011 (binario)

o bien

```
423 (octal) = 4 · 64 + 2 · 8 + 3 · 1 =  275 (decimal)
```

De cualquier modo, el sistema octal no es el que más se utiliza en la práctica, sino el hexadecimal…

Ejercicios propuestos:
(Ap3.2.1) Expresar en sistema octal los números decimales 17, 101, 83, 45.
(Ap3.2.2) Expresar en sistema octal los números binarios de 8 bits: 01100110, 10110010, 11111111, 00101101
(Ap3.2.3) Expresar en el sistema binario los números octales 171, 243, 105, 45.
(Ap3.2.4) Expresar en el sistema decimal los números octales 162, 76, 241, 102.
(Ap3.1.5) Crea un programa en C# que convierta a octal los números (en base 10) que introduzca el usuario.
(Ap3.1.6) Crea un programa en C# que convierta a base 10 los números octales que introduzca el usuario.

Ap3.3. Sistema hexadecimal

El sistema octal tiene un inconveniente: se agrupan los bits de 3 en 3, por lo que convertir de binario a octal y viceversa es muy sencillo, pero un byte está formado por 8 bits, que no es múltiplo de 3.

Sería más cómodo poder agrupar de 4 en 4 bits, de modo que cada byte se representaría por 2 cifras. Este sistema de numeración trabajará en base 16 (2^4=16), y es lo que se conoce como sistema hexadecimal.

Pero hay una dificultad: estamos acostumbrados al sistema decimal, con números del 0 al 9, de modo que no tenemos cifras de un solo dígito para los números 10, 11, 12, 13, 14 y 15, que utilizaremos en el sistema hexadecimal. Para representar estas cifras usaremos las letras de la A a la F, así:

```
 0 (decimal) = 0 (hexadecimal)
 1 (decimal) = 1 (hexadecimal)
 2 (decimal) = 2 (hexadecimal)
 3 (decimal) = 3 (hexadecimal)
 4 (decimal) = 4 (hexadecimal)
 5 (decimal) = 5 (hexadecimal)
 6 (decimal) = 6 (hexadecimal)
 7 (decimal) = 7 (hexadecimal)
 8 (decimal) = 8 (hexadecimal)
 9 (decimal) = 9 (hexadecimal)
10 (decimal) = A (hexadecimal)
11 (decimal) = B (hexadecimal)
12 (decimal) = C (hexadecimal)
13 (decimal) = D (hexadecimal)
14 (decimal) = E (hexadecimal)
15 (decimal) = F (hexadecimal)
```

Con estas consideraciones, expresar números en el sistema hexadecimal ya no es difícil:

```
254 (decimal) ->
254 / 16 = 15 (resto: 14)
14 / 1 = 14 (se terminó)
```

de modo que

$$254 = 15 \cdot 16^1 + 14 \cdot 16^0$$

o bien

```
254 (decimal) = FE (hexadecimal)
```

Vamos a repetirlo para un convertir de **decimal a hexadecimal** número más grande:

```
54331 (decimal) ->
54331 / 4096 = 13 (resto: 1083)
1083 / 256 = 4 (resto: 59)
59 / 16 = 3 (resto: 11)
11 / 1 = 11 (se terminó)
```

de modo que

$$54331 = 13 \cdot 4096 + 4 \cdot 256 + 3 \cdot 16 + 11 \cdot 1$$

o bien

$$254 = 13 \cdot 16^{3} + 4 \cdot 16^{2} + 3 \cdot 16^{1} + 11 \cdot 16^{0}$$

es decir

```
54331 (decimal) = D43B (hexadecimal)
```

Ahora vamos a dar el paso inverso: convertir de **hexadecimal a decimal**, por ejemplo el número A2B5

$$A2B5\ (hexadecimal) = 10 \cdot 16^{3} + 2 \cdot 16^{2} + 11 \cdot 16^{1} + 5 \cdot 16^{0} = 41653$$

El paso de **hexadecimal a binario** también es (relativamente) rápido, porque cada dígito hexadecimal equivale a una secuencia de 4 bits:

```
0 (hexadecimal) =  0 (decimal) = 0000 (binario)
1 (hexadecimal) =  1 (decimal) = 0001 (binario)
2 (hexadecimal) =  2 (decimal) = 0010 (binario)
3 (hexadecimal) =  3 (decimal) = 0011 (binario)
4 (hexadecimal) =  4 (decimal) = 0100 (binario)
5 (hexadecimal) =  5 (decimal) = 0101 (binario)
6 (hexadecimal) =  6 (decimal) = 0110 (binario)
7 (hexadecimal) =  7 (decimal) = 0111 (binario)
8 (hexadecimal) =  8 (decimal) = 1000 (binario)
9 (hexadecimal) =  9 (decimal) = 1001 (binario)
A (hexadecimal) = 10 (decimal) = 1010 (binario)
B (hexadecimal) = 11 (decimal) = 1011 (binario)
C (hexadecimal) = 12 (decimal) = 1100 (binario)
D (hexadecimal) = 13 (decimal) = 1101 (binario)
E (hexadecimal) = 14 (decimal) = 1110 (binario)
```

```
F (hexadecimal) = 15 (decimal) = 1111 (binario)
```

de modo que A2B5 (hexadecimal) = 1010 0010 1011 0101 (binario)

y de igual modo, de **binario a hexadecimal** es dividir en grupos de 4 bits y hallar el valor de cada uno de ellos:

```
110010100100100101010100111 =>
0110 0101 0010 0100 1010 1010 0111 = 6524AA7
```

Ejercicios propuestos:
(Ap3.3.1) Expresa en sistema hexadecimal los números decimales 18, 131, 83, 245.
(Ap3.3.2) Expresa en sistema hexadecimal los números binarios de 8 bits: 01100110, 10110010, 11111111, 00101101
(Ap3.3.3) Expresa en el sistema binario los números hexadecimales 2F, 37, A0, 1A2.
(Ap3.3.4) Expresa en el sistema decimal los números hexadecimales 1B2, 76, E1, 2A.
(Ap3.3.5) Crea un programa en C# que convierta a hexadecimal los números (en base 10) que introduzca el usuario.
(Ap3.3.6) Crea un programa en C# que convierta a base 10 los números hexadecimales que introduzca el usuario.
(Ap3.3.7) Crea un programa en C# que convierta a hexadecimal los números binarios que introduzca el usuario.

Ap3.4. Representación interna de los enteros negativos

Para los **números enteros negativos**, existen varias formas posibles de representarlos. Las más habituales son:

Signo y magnitud: el primer bit (el de más a la izquierda) se pone a 1 si el número es negativo y se deja a 0 si es positivo. Los demás bits se calculan como ya hemos visto.

Por ejemplo, si usamos 4 bits, tendríamos

```
3 (decimal) = 0011        -3 = 1011
6 (decimal) = 0110        -6 = 1110
```

Es un método muy sencillo, pero que tiene el inconveniente de que las operaciones en las que aparecen números negativos no se comportan correctamente. Vamos a ver un ejemplo, con números de 8 bits:

```
13 (decimal) = 0000 1101  - 13 (decimal) = 1000 1101
34 (decimal) = 0010 0010  - 34 (decimal) = 1010 0010
13 + 34 = 0000 1101 + 0010 0010 = 0010 1111 = 47  (correcto)
(-13) + (-34) = 1000 1101 + 1010 0010 = 0010 1111 = 47  (INCORRECTO)
13 + (-34) =  0000 1101 + 1010 0010 = 1010 1111 = -47  (INCORRECTO)
```

Complemento a 1: se cambian los ceros por unos para expresar los números negativos.

Por ejemplo, con 4 bits

```
3 (decimal) = 0011        -3 = 1100
6 (decimal) = 0110        -6 = 1001
```

También es un método sencillo, en el que las operaciones con números negativos salen bien, y que sólo tiene como inconveniente que hay dos formas de expresar el número 0 (0000 0000 o 1111 1111), lo que complica algunos trabajos internos del ordenador.

Ejercicio propuesto: convierte los números decimales 13, 34, -13, -34 a sistema binario, usando complemento a uno para expresar los números negativos. Calcula (en binario) el resultado de las operaciones 13+34, (-13)+(-34), 13+(-34) y comprueba que los resultados que se obtienen son los correctos.

Complemento a 2: para los negativos, se cambian los ceros por unos y se suma uno al resultado.

Por ejemplo, con 4 bits

```
3 (decimal) = 0011        -3 = 1101
6 (decimal) = 0110        -6 = 1010
```

Es un método que parece algo más complicado, pero que no es difícil de seguir, con el que las operaciones con números negativos siempre se realizan de forma correcta, y no tiene problemas para expresar el número 0 (que siempre se almacena como 00000000).

Ejercicio propuesto (Ap3.4.1): Convierte los números decimales 13, 34, -13, -34 a sistema binario, usando complemento a dos para expresar los números negativos. Calcula (en binario) el resultado de las operaciones 13+34, (-13)+(-34), 13+(-34) y comprueba que los resultados que se obtienen son los correctos.

En general, todos los formatos que permiten guardar números negativos usan el primer bit para el signo. Por eso, si declaramos una variable como "unsigned", ese primer bit se puede utilizar como parte de los datos, y podemos almacenar números más grandes. Por ejemplo, un "ushort" (enterno corto sin signo) en C# podría tomar valores entre 0 y 65.535, mientras que un "short" (entero corto con signo) podría tomar valores entre +32.767 y –32.768.