Bit-by-Bit

Step 1

In the digital age, people are typically comfortable with the notion of "bits" as being "1s and 0s" as well as with the notion of "bytes" being the fundamental unit of memory on a computer, but the connection between the two is not always obvious.

A **bit** is the basic unit of information in computing and can have only one of two values. We typically represent these two values as either a 0 or a 1, but they can be interpreted as logical values (true/false, yes/no), algebraic signs (+/−), activation states (on/off), or any other two-valued attribute.

A **byte** is a unit of digital information, and it is just a sequence of some number of bits. The size of the byte has historically been hardware dependent and no definitive standards existed that mandated the size, but for the purposes of this course as well as almost all computer applications, you can assume that a byte is specifically a sequence of **8 bits**. Note that, with modern computers, a byte is the smallest unit that can be stored. In other words, a file can be 1 byte, 2 bytes, 3 bytes, etc., but a file *cannot* be 1.5 bytes.

Step 2

EXERCISE BREAK: How many distinct symbols could be represented with 1 byte? (Write the answer as an integer, not in scientific notation)

To solve this problem please visit https://stepik.org/lesson/27076/step/2

Step 3

EXERCISE BREAK: How many distinct symbols could be represented with 4 bytes? (Write the answer as an integer, not in scientific notation)

To solve this problem please visit https://stepik.org/lesson/27076/step/3

Step 4

As was mentioned previously, 1 byte is the smallest unit of storing memory in modern computers. Thus, every single file on a computer is simply a sequence of bytes, and by extension, a sequence of 8-bit sequences. Yes, *every* file on a computer is just a sequence of 8-bit chunks. Text, images, videos, audio, literally *all* filetypes.

As one can infer, if everything is represented as some sequence of bytes, there must be some mapping that is done to represent useful information as a sequence of 1s and 0s. **ASCII**, abbreviated from **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange, is a character-encoding scheme where each possible byte is mapped to a specific "symbol" (I say "symbol" in quotes because not all ASCII characters are meaningful to humans).

Step 5

Below is a mapping of each of the possible bytes to their corresponding ASCII characters:

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

128	Ç	144	É	160	á	176	░	192	└	208	╨	224	α	240	≡
129	ü	145	æ	161	í	177	▒	193	┴	209	╤	225	ß	241	±
130	é	146	Æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
131	â	147	ô	163	ú	179	│	195	├	211	╙	227	π	243	≤
132	ä	148	ö	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
133	à	149	ò	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
134	å	150	û	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
135	ç	151	ù	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
137	ë	153	Ö	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙
138	è	154	Ü	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
139	ï	155	¢	171	½	187	╗	203	╦	219	█	235	δ	251	√
140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	^{n}
141	ì	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	^{2}
142	Ä	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■
143	Å	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	

Source: www.LookupTables.com

Step 6

If we think of decimal numbers in terms of their binary representations (i.e., as a sequence of bits), we can perform various bitwise operations on them. Before talking about bitwise operations, though, let's first (re-)familiarize ourselves with how to think of numbers in binary.

Recall from elementary school that, in the decimal system, numbers are organized into columns: the rightmost column is the "**ones**" column, then to the left of it is the "**tens**" column, then the "**hundreds**" column, etc. We then learned that "**decimal**" meant "**base 10**" and that the rightmost column of a decimal number actually represents $\mathbf{10^0}$, the column to the left of it represents $\mathbf{10^1}$, then $\mathbf{10^2}$, etc. For example, the number **729** can be thought of as $\mathbf{(10^2 \times 7) + (10^1 \times 2) + (10^0 \times 9)}$.

Binary numbers work in the same way, but instead of being "base 10" (which is the case for "decimal"), they are "**base 2**". Thus, the rightmost column represents $\mathbf{2^0}$, the column to the left of it represents $\mathbf{2^1}$, then $\mathbf{2^2}$, etc. For example, the number **101** can be thought of as $\mathbf{(2^2 \times 1) + (2^1 \times 0) + (2^0 \times 1)}$. In other words, **101** in binary is equal to **5** in decimal.

Step 7

EXERCISE BREAK: Convert the binary number **101010** to decimal.

To solve this problem please visit https://stepik.org/lesson/27076/step/7

Step 8

Binary addition and subtraction work just like decimal addition and subtraction: align the digits of the two numbers and simply add column by column, carrying the 1 if a given column overflows. In decimal addition, we carry the one when a given column wraps around from a value of 9 to a value of 0 (because in decimal, the valid digits are 0-9, so incrementing 9 wraps around to 0). Identically, in binary addition, we carry the one when a given column wraps around from a value of 1 to a value of 0 (because in binary, the valid digits are 0-1, so incrementing 1 wraps around to 0).

Below is an example of the addition of two binary numbers:

a:	0	1	0	1
b:	0	1	1	0
a+b:	1	0	1	1

Note that the right column's addition was simply 1 + 0 = 1. Then, the next column to the left was simply 0 + 1 = 1. Then the next column was 1 + 1 = 10, so we put a value of 0 in that column and carry the 1. Lastly, the leftmost column was simply 0 + 0 = 0, plus the 1 we carried over, so we put a value of 1.

Step 9

EXERCISE BREAK: Add the binary numbers **0101** and **0101** (enter the result as a binary number)

To solve this problem please visit https://stepik.org/lesson/27076/step/9

Step 10

Aside from addition, there are many **bitwise operations** one can do on numbers. These bitwise operations treat the numbers as their binary and perform the relevant operation bit-by-bit. The following bitwise operations will be explained using single-bit examples, but when performed on numbers with more than one bit, you simply go through the numbers' bits column-by-column, performing the single-bit examples independently on each column independently. It will help to follow the logic of the operations by thinking of the bits in terms of `1 = TRUE` and `0 = FALSE`.

- **`Bitwise AND (&):`** `1 & 1 = 1` `0 & 1 = 0` `1 & 0 = 0` `0 & 0 = 0`
- **`Bitwise OR (|):`** `1 | 1 = 1` `0 | 1 = 1` `1 | 0 = 1` `0 | 0 = 0`
- **`Bitwise XOR (^):`** `1 ^ 1 = 0` `0 ^ 1 = 1` `1 ^ 0 = 1` `0 ^ 0 = 0`
- **`Bitwise NOT (~):`** `~1 = 0` `~0 = 1`

In addition to these single-bit operations, there are bit-shifting operations that can be done on binary numbers. The **left bit-shift operator (<<)** shifts each bit of the binary number left by the specified number of columns, and the **right bit-shift operator (>>)** shifts each bit of the binary number right by the specified number of columns. For example, below we shift two 8-bit numbers:

- `00001000 << 2 = 00100000`
- `00001000 >> 2 = 00000010`

When bit-shifting, one should imagine the 1s as "information" and the 0s as "empty space". If a 1 gets "pushed over the edge", it is simply lost (or "cleared"). Also, as can be seen in the example above, when we shift left, the the columns on the right side of the number are filled with "empty space" (0s, shown in red), and when we shift right, the columns on the left side of the number are also filled with "empty space" (0s, shown in red).

Step 11

Below are some examples of performing the previously described bitwise operators. Note that, even though the numbers are initially represented as decimal numbers, the operations treat them as their binary representations (because, in reality, the numbers are represented in binary on the computer and are simply displayed to us in decimal, or hex, or whatever representation we choose).

```
unsigned char a = 5, b = 67;
```

a:	0	0	0	0	0	1	0	1
b:	0	1	0	0	0	0	1	1
Bitwise AND: a & b:	0	0	0	0	0	0	0	1
Bitwise OR: a | b:	0	1	0	0	0	1	1	1
Bitwise XOR: a ^ b:	0	1	0	0	0	1	1	0
Bitwise NOT: ~a:	1	1	1	1	1	0	1	0
Left Shift: a << 2:	0	0	0	1	0	1	0	0
Right Shift: a >> 2:	0	0	0	0	0	0	0	1

Just like with any other type of math, bitwise operations can only really be learned through practice. As such, we will be testing your skills in the next few steps.

Step 12

EXERCISE BREAK: What is the result of the following bitwise expression? Assume the numbers are unsigned 8-bit numbers. Enter your answer as a decimal number (not binary)

```
7 | (~125 << 3)
```

To solve this problem please visit https://stepik.org/lesson/27076/step/12

Step 13

Note that, previously, we specified the datatype `unsigned char` to represent a byte in C++. In many programming languages (such as in C++), datatypes can be either *signed* or *unsigned*. In all of the previous examples in this lesson, we exclusively dealt with *unsigned* values: the smallest value is 0, and all other values are positive. For example, in C++, an `unsigned int` is 4 bytes (i.e., 32 bits), meaning the smallest possible value is 00000000 00000000 00000000 00000000 (which has a value of 0), and the largest possible value is 11111111 11111111 11111111 11111111 (which has a value of $2^{32}-1$). In other words, with an *unsigned* datatype, all of the bits are used to represent *magnitude* of the number.

In a *signed* datatype, one bit is reserved to represent the *sign* of the number (typically, 0 = positive and 1 = negative), and the remaining bits represent the *magnitude* of the number. Typically, the "sign bit" is the left-most bit of the number (as it is in C++). Because one of the bits is reserved to represent the *sign* of the number, if a given signed datatype has a size of *n* bits, the smallest value it can hold is -2^{n} and the largest value it can hold is $2^{n-1}-1$. For example, in C++, a `signed int` is 4 bytes (i.e., 32 bits, just like an `unsigned int`), but because the leftmost bit represents the *sign* of the number, the remaining 31 bits are used to represent *magnitude*, meaning the smallest value that can be represented is -2^{31} and the largest value that can be represented is $2^{31}-1$.

Depending on the programming language, performing bitwise operations on a signed datatype performs the given operation on the $n-1$ *magnitude* bits without modifying the *sign* bit (such that the sign of the number does not change). As a result, programming languages that have this feature often have signed *and* unsigned versions of each of the bitwise operations. In general, be wary when performing bitwise operations on a signed datatype, and be sure to reference the relevant documentation to see how the bitwise operations work on signed datatypes in the language you are using.

Step 14

EXERCISE BREAK: What is the **largest** integer a C++ **`unsigned char`** can represent, given that it has a size of 1 byte (i.e., 8 bits)?

To solve this problem please visit https://stepik.org/lesson/27076/step/14

Step 15

EXERCISE BREAK: What is the **largest** integer a C++ **`signed char`** can represent, given that it has a size of 1 byte (i.e., 8 bits)?

To solve this problem please visit https://stepik.org/lesson/27076/step/15

Step 16

EXERCISE BREAK: What is the **smallest** integer a C++ **`signed char`** can represent, given that it has a size of 1 byte (i.e., 8 bits)?

To solve this problem please visit https://stepik.org/lesson/27076/step/16