Set 3. Storing Text in Binary

Skill 3.01: Explain how computers utilize protocols to store text

Skill 3.01 Concepts

Computers store more than just numbers in binary. But how can binary numbers represent non-numbers such as letters and symbols?

As it turns out, all it requires is a bit of human cooperation. We must agree on **encodings**, mappings from a character to a binary number.

For example, what if we wanted to store the following symbols in binary?

☮❤😀

We can invent this simple encoding:

Binary	Symbol
01	☮
10	❤
11	😀

Let's call it the HPE encoding. It helps for encodings to have names, so that programmers know they're using the same encoding.

If a computer program needs to store the ❤ symbol in computer memory, it can store 10 instead. When the program needs to display10 to the user, it can remember the HPE encoding and display ❤ instead.

Computer programs and files often need to store multiple characters, which they can do by stringing each character's encoding together.

A program could write a file called "msg.hpe" with this data:

010111111010

☮☮😀😀❤❤

Skill 3.01 Exercise 1

Skill 3.02: Describe the ASCII system for storing text

Skill 3.02 Concepts

The HPE encoding only uses 2 bits, so that limits how many symbols it can represent. In our 2 bit encoding system, we can only store $2^2 - 1$ or 3 symbols.

However, with more bits of information, an encoding system can represent enough letters for computers to store messages, documents, and webpages.

ASCII was one of the first standardized encodings. It was invented back in the 1960s when telegraphy was the primary form of long-distance communication, but is still in use today on modern computing systems.

Teletypists would type messages on teleprinters such as this one:

The teleprinter would then use the ASCII standard to encode each typed character into binary and then store or transmit the binary data.

This page from a 1972 teleprinter manual shows the 128 ASCII codes:

USASCII code chart

b_4	b_3	b_2	b_1	Row ↓ / Column →	0 (0 0 0)	1 (0 0 1)	2 (0 1 0)	3 (0 1 1)	4 (1 0 0)	5 (1 0 1)	6 (1 1 0)	7 (1 1 1)
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

Each ASCII character is encoded in binary using 7 bits. In the chart above, the column heading indicates the first 3 bits and the row heading indicates the final 4 bits. The very first character is "NUL", encoded as 0000000.

The first 32 codes represent "control characters," characters which cause some effect besides printing a letter. "BEL" (encoded in binary 0000111) caused an audible bell or beep. "ENQ" (encoded as 0000101) represented an enquiry, a request for the receiving station to identify themselves.

The control characters were originally designed for teleprinters and telegraphy, but many have been re-purposed for modern computers and the Internet—especially "CR" and "LF". "CR" (0001101) represented a "carriage return" on teleprinters, moving the printing head to the start of the line. "LF" (0001010) represented a "line feed", moving the printing head down one line. Modern Internet protocols, such as HTTP, FTP, and SMTP, use a combination of "CR" + "LF" to represent the end of lines.

The remaining 96 ASCII characters look much more familiar.

Here are the first 8 uppercase letters:

Binary	Character
1000001	A
1000010	B
1000011	C
1000100	D
1000101	E
1000110	F
1000111	G
1001000	H

Following the ASCII standard, we can encode a four-letter message into binary:

10000111001000100001011000110

The above binary message translates to “CHEF”

Skill 3.02 Exercise 1

Skill 3.03: Identify the limitations of the ASCII system

Skill 3.03 Concepts

There are several problems with the ASCII encoding, however.

The first big problem is that ASCII only includes letters from the English alphabet and a limited set of symbols.

A language that uses less than 128 characters could come up with their own version of ASCII to encode text in just their language, but what about a text file with characters from multiple languages? ASCII couldn't encode a string like: "Hello, José, would you care for Glühwein? It costs 10 €".

And what about languages with thousands of logograms? ASCII could not encode enough logograms to cover a Chinese sentence like "我的氣墊船滿是鱔魚".

The other problem with the ASCII encoding is that it uses **7** bits to represent each character, whereas computers typically store information in bytes—units of **8** bits—and programmers don't like to waste memory.

When the earliest computers first started using ASCII to encode characters, different computers would come up with various ways to utilize the final bit. For example, HP computers used the eighth bit to represent characters used in European countries (e.g. "£" and "Ü"), TRS-80 computers used the bit for colored graphics, and Atari computers used the bit for inverted white-on-black versions of the first 128 characters.

The result? An "ASCII" file created in one application might look like gobbledy gook when opened in another "ASCII"-compatible application.

Computers needed a new encoding, an encoding based on 8-bit bytes that could represent all the languages of the world.

Skill 3.03 Exercise 1

Skill 3.04: Describe the Unicode character set

Skill 3.04 Concepts

To address the limitations of the ASCII system, in 1987, a group of computer engineers came up with Unicode, a universal character set which assigns each character a "code point" (a hexadecimal number) and a name.

For example, the character "ą" is assigned to "U+0105" and named "Latin Small Letter A with Ogonek". There's a character that looks like "ą" in 13 languages, such as Polish and Lithuanian. Thus, according to Unicode, the "ą" in the Polish word "robią" and the "ą" in the Lithuanian word "aslą" are both the same character. Unicode saves space by unifying characters across languages.

But there are still quite a few characters to encode. The Unicode character set started with 7,129 named characters in 1991 and has grown to 137,929 named characters in 2019. The majority of those characters describe logograms from Chinese, Japanese, and Korean, such as "U+6728" which refers to "木". It also includes over 1,200 emoji symbols ("U+1F389" = "🎉").

Skill 3.05: Describe the UTF-8 system for storing text

Skill 3.05 Concepts

In 1992, computer scientists invented UTF-8, an encoding that is compatible with ASCII encoding but also solves its problems.

UTF-8 can describe every character from the Unicode standard using either 1, 2, 3, or 4 bytes.

When a computer program is reading a UTF-8 text file, it knows how many bytes represent the next character based on how many `1` bits it finds at the beginning of the byte.

Number of bytes	Byte 1	Byte 2	Byte 3	Byte 4
1	`0xxxxxxx`			
2	`110xxxxx`	`10xxxxxx`		
3	`1110xxxx`	`10xxxxxx`	`10xxxxxx`	
4	`11110xxx`	`10xxxxxx`	`10xxxxxx`	`10xxxxxx`

If there are no 1 bits in the prefix (if the first bit is a 0), that indicates a character represented by a single byte. The remaining 7 bits of the byte are used to represent the original 128 ASCII characters. That means a sequence of 8-bit ASCII characters is also a valid UTF-8 sequence.

Two bytes beginning with 110 are used to encode the rest of the characters from Latin-script languages (e.g. Spanish, German) plus other languages such as Greek, Hebrew, and Arabic. Three bytes starting with 1110 encode most of the characters for Asian languages (e.g. Chinese, Japanese, Korean). Four bytes starting with 11110 encode everything else, from rarely used historical scripts to the increasingly commonly used emoji symbols.

Below is an example of how UTF-8 can applied to encode characters.

According to the UTF-8 standard, 3 characters are stored in the 8 bytes shown.

01001001 11110000 10011111 10010010 10011001 11100010 10010011 10001010

The first byte is 01001001. It starts with a 0 so we know this single byte represents a character. We don't need to actually determine which character; we just need to understand that the first byte represents one character.

The next byte is 11110000 starts with four 1 bits which means this byte is the start of 4-byte sequence representing a character:

11110000 10011111 10010010 10011001

The sixth byte is 11100010. It starts with three 1 bits, so this byte is the start of a 3-byte sequence:

11100010 10010011 10001010

Thus, in UTF-8, the 8 bytes represent 3 characters.

Below are the characters they represent:

Binary	Character
01001001	I
11110000 10011111 10010010 10011001	💙
11100010 10010011 10001010	Ⓤ

The UTF-8 encoding standard is now the dominant encoding of HTML files on the web, accounting for 94.5% of webpages as of December 2019.

Generally, a good encoding is one that can represent the maximum amount of information with the least number of bits. UTF-8 is a great example of that, since it can encode common English letters with just 1 byte but is flexible enough to encode thousands of letters with additional bytes.

Skill 3.04 Exercise 1