CS 45, Lecture 17

Encoding

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Spring 2023

Outline

Contents

1 Overview 1

2 Encoding Numbers 1

3 Text Encoding 4
3.1 English Text 4
3.2 Text Around the World 5
3.3 Unifying all the Codes 7

4 Audio Encoding 10

5 Image Encoding 11

6 Video Encoding 13

7 Conclusion 14

1 Overview

What is Encoding?

- Computers are great at numbers.
- Computers are terrible at everything else.
- We need to turn everything else into numbers for computers to deal with them.

Actually, that's not strictly true. Computers are terrible at numbers as well; the only thing they can *really* handle is truth values (true or false).

2 Encoding Numbers

Encoding Numbers

Numbers themselves can be encoded many ways:

roman: XLV $= 50 - 10 + 5$

binary 101101_2 $= 1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^2$

octal: 55_8 $= 5 * 8^1 + 5$

decimal: 45 $= 4 * 10^1 + 5 * 10^0$

hexadecimal: $\texttt{2d}_{16}$ $= 2 * 16^1 + 13$

For computers, we want a way that works well with digital logic circuits (i.e., using only 0 or 1).

Human Representation

Instead of specifying the base every time we write a number, computer scientists often use different notations as shorthand:

binary `0b101101`

octal: `0o55` or `055`

decimal: `45`

hex: `0x2d`

Binary

Binary looks great for computers (it's just zero or one), but there's a few problems:

- It's unwieldy for humans: `0b1010000001110` vs 5134.
- It's hard to convert to decimal (but easy to convert to hexadecimal or octal).
- There's no "maximum" number, so there's no maximum number of bits to store "one number".
- There's no easy way to represent negative numbers (like -01100001_2).
- It can't represent non-integer numbers (e.g., what's $\frac{01_2}{10_2}$?)

In practice, computer scientists often represent numbers as hexadecimal for concision or decimal for convenience, even though the underlying representation is binary.

Sized Integers

The size problem is solved by defining different "sizes" of numbers, with different numbers of bits:

nibble 4 bits

byte 8 bits

short 16 bits

int 32 bits

long 64 bits

Different CPUs have different "default" sizes; this size is often called "word". For example, 64-bit machines (most computers today) have 64-bit words and 32-bit "halfwords".

These particular names aren't set in stone; sometimes they have different meanings based on context. On some old computers, an `int` would be 16 bits and a `long` would be 32 bits. However, most of the time nowadays, they're used in the way described here.

Processing numbers bigger than a "word" on a computer is usually slow, while processing numbers smaller than a word is fast.

Signed Integers

The negative numbers problem is solved by using TWO'S COMPLEMENT encoding, wherein a negative number is produced by:

1. Starting with the positive version of that number.
2. Inverting every bit (0 to 1, 1 to 0).
3. Add 1 to the number.

This means 31 (as a byte) becomes `0b00011111`, but -31 becomes `0b11100001`. The uppermost bit can be thought of as a “sign bit”.

To interpret a number, you need to know if it is SIGNED or UNSIGNED; `0b11100001` could be -31 or 225.

Integer Ranges

The range of numbers we can represent depends on the encoding we use:

Width	Unsigned	Signed
8	$[0, 255]$	$[-128, 127]$
16	$[0, 65535]$	$[-32768, 32767]$
32	$[0, 4294967295]$	$[-2147483648, 2147483647]$
64	$[0, 2^{64} - 1]$	$[-2^{63}, 2^{63} - 1]$

Overflow and Underflow

Overflow/Underflow

A huge number of real-world bugs come from integer OVERFLOW and UNDERFLOW.

```
// unsigned byte overflow:
(uint8_t)(255 + 1 == 0);
// unsigned byte underflow:
(uint8_t)(0 - 1 == 255);
// signed byte overflow):
(int8_t)(127 + 1 == -128);
// signed byte underflow:
(int8_t)(-128 - 1 == 127);
```

Endianness

When a single number is represented by multiple bytes, there are two valid ways to order those bytes: with the “big end” first or the “little end” first.

`0xAABBCCDD =`

Offset	Big Endian	Little Endian
0	0xAA	0xDD
1	0xBB	0xCC
2	0xCC	0xBB
3	0xDD	0xAA

Most CPUs are little-endian, but most network protocols are big-endian. The way we write numbers (e.g., `0xAABBCCDD`) is big-endian.

Floating Point

Real numbers (i.e., non-integers) are represented using “floating-point” arithmetic.

A number like 1701.47 could be written as 170147×10^{-2}.

The IEEE 754 Standard specifies how computers would store 170147 and -2 so they can do math easily.

Floating point numbers are inherently approximations, and prone to inaccuracies:

```
0.1 + 0.2 == 0.3 // false
0.1 + 0.2 // 0.30000000000000004
```

Sometimes 32-bit floating point numbers are called `float` and 64-bit floating point numbers are called `double`.

3 Text Encoding

Now that we know how to represent numbers, we can use those numbers to represent more complicated things.

3.1 English Text

Text

Let's say we want to save this file:

```
Hi!
```

One idea might be to do a simple substitution: A=1, B=2, C=3, etc. Problems:

- We need to handle both upper- and lower-case letters.
- We need to handle punctuation.
- We need to handle numbers.
- We need to handle special things like "space" and "enter"/"return".

ASCII

In the 1960s, Bell Labs created the American Standard Code for Information Interchange, which deals with all this:

```
Hi!
```

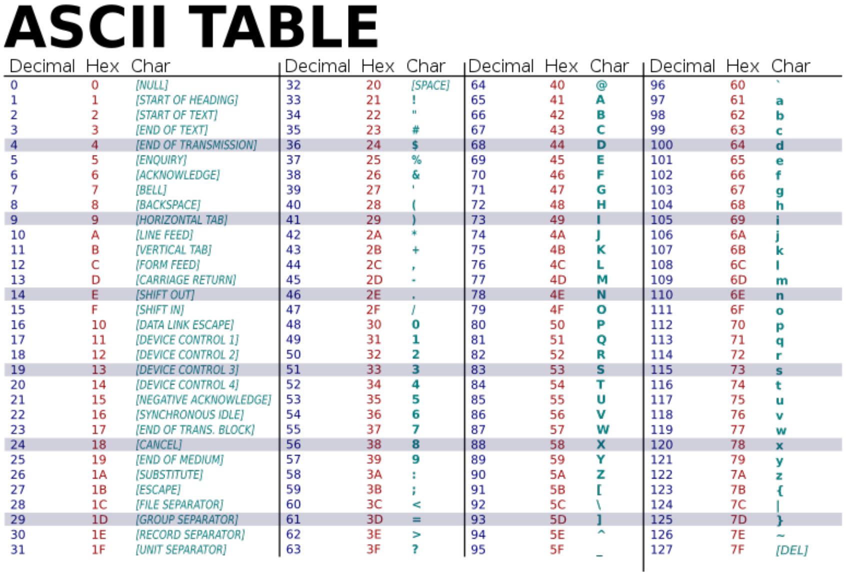
becomes

```
48 69 21
```

This is a 7-bit encoding (each letter/symbol takes 7 bits of data), but is often treated as an 8-bit encoding for convenience.

This encoding became super popular and is used everywhere.

ASCII Table



ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Image Credit: https://simple.wikipedia.org/wiki/ASCII

3.2 Text Around the World

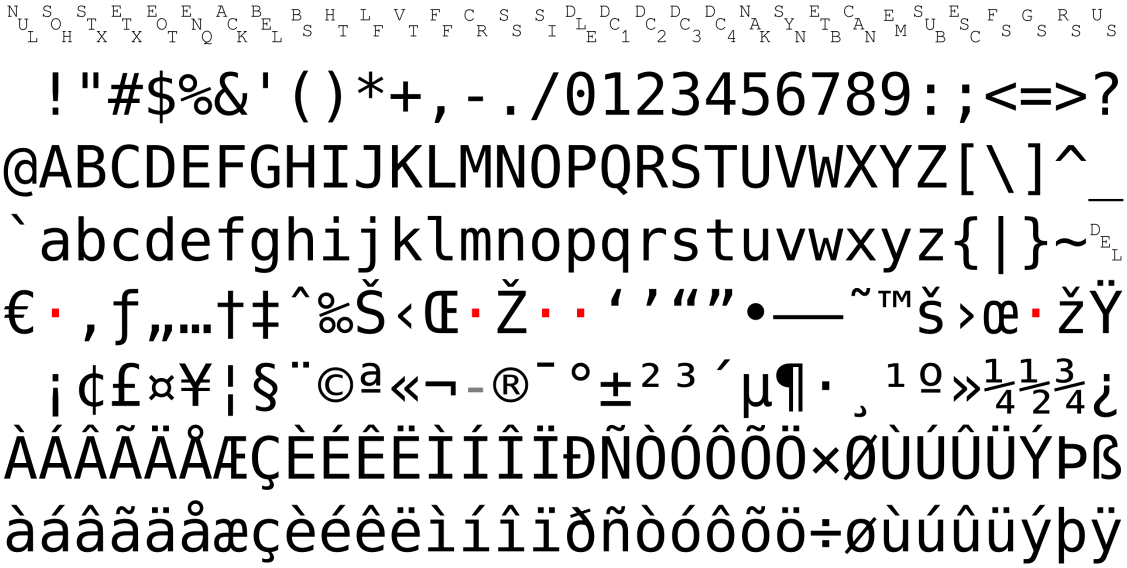
ASCII outside America

- ASCII (also called ISO 646-US) is unabashedly America-centric.
- It only supports the "basic latin alphabet" (the 26 letters in English).
- Some other countries created slight modifications of ASCII for their use:
  - ISO 646-GB adds `£` (UK English).
  - ISO 646-FR adds `à`, `ç`, `é`, `ù`, and `è` (French).
  - ISO 646-ES adds `ñ`, `Ñ`, `¡`, `¿`, and `ç` (Spanish).
- In some cases, people used the extra top bit to encode up to 256 extra characters in an 8-bit encoding, e.g., Code Page 1252.

Code Page 1252

- Code Page 1252 was historically the default text encoding on Windows.
- The subset ISO 8859-1 is also considered equivalent in most cases.
- These are the most widespread single-byte (8-bit) text encodings in the world today, but still only feature on 1.7% of websites.

Code Page 1252 Table



```
NUL SOH STX ETX EOT ENQ ACK BEL BS HT LF VT FF CR SS SI DLE DC1 DC2 DC3 DC4 NAK SYN ETB CAN EM SUB ESC FS GS RS US
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~DEL
€·‚ƒ„…†‡ˆ‰Š‹Œ·Ž··‘’“”•–—˜™š›œ·žŸ
 ¡¢£¤¥¦§¨©ª«¬-®¯°±²³´µ¶·¸¹º»¼½¾¿
ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞß
àáâãäåæçèéêëìíîïðñòóôõö÷øùúûüýþÿ
```

Image Credit: https://en.wikipedia.org/wiki/Windows-1252

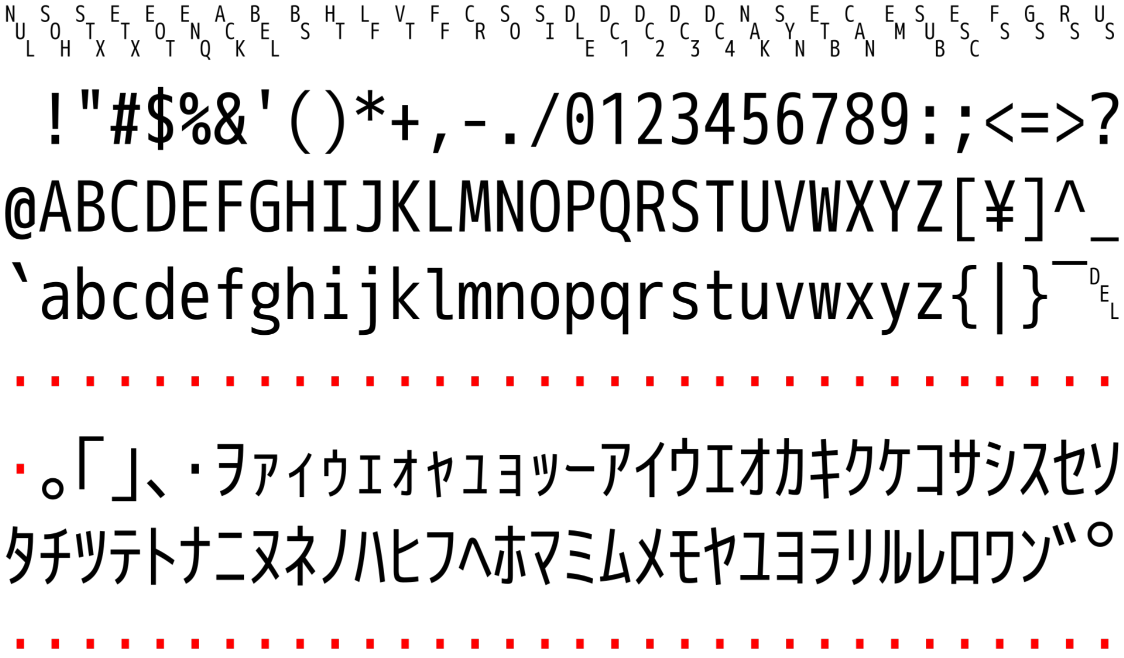
By now you've probably noticed an issue—even though we can encode a bunch of letters from different Latin-based alphabets, we still can't represent other alphabets at all (e.g., Greek, Cyrillic, Chinese, Korean).

Alternatives to ASCII

Countries which use other alphabets came up with their own ASCII/ISO 646 variants.

These generally encode all the ASCII characters for compatibility, but take advantage of the 8th bit to encode their own characters (or, often, a subset thereof).

JIS X 0201 (ISO 646-JP)



```
NUL SOH STX ETX EOT ENQ ACK BEL BS HT LF VT FF CR SO SI DLE DC1 DC2 DC3 DC4 NAK SYN ETB CAN EM SUB ESC FS GS RS US
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[¥]^_
`abcdefghijklmnopqrstuvwxyz{|}‾DEL
································
·｡｢｣､･ｦｧｨｩｪｫｬｭｮｯｰｱｲｳｴｵｶｷｸｹｺｻｼｽｾｿ
ﾀﾁﾂﾃﾄﾅﾆﾇﾈﾉﾊﾋﾌﾍﾎﾏﾐﾑﾒﾓﾔﾕﾖﾗﾘﾙﾚﾛﾜﾝﾞﾟ
································
```

This was superseded by Shift JIS, which uses 2 bytes (16 bits). Image Credit: https://en.wikipedia.org/wiki/JIS_X_0201

Custom Encodings

Some companies also came up with custom encodings for their products.

Generally these would only be usable on devices made by that company, so they tried to remain backwards-compatible with ASCII.

Over time, there grew to be thousands of incompatible character sets.

Custom Characters

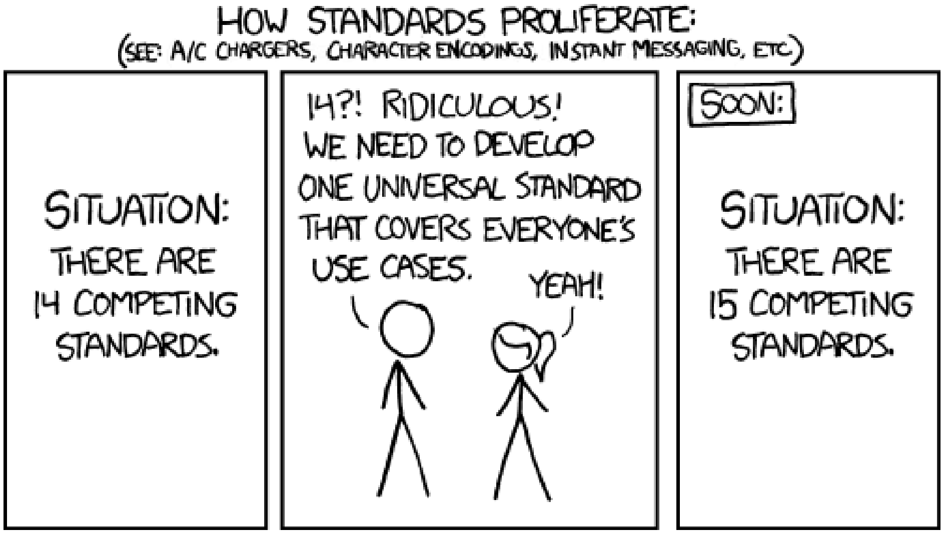
In 1997, J-Phone (a Japanese phone company) released this character set:

Image Credit: https://emojitimeline.com/the-real-original-emojis/ This was the first appearance of EMOJI in a character set.

3.3 Unifying all the Codes

This whole situation is pretty well summed up by this XKCD:

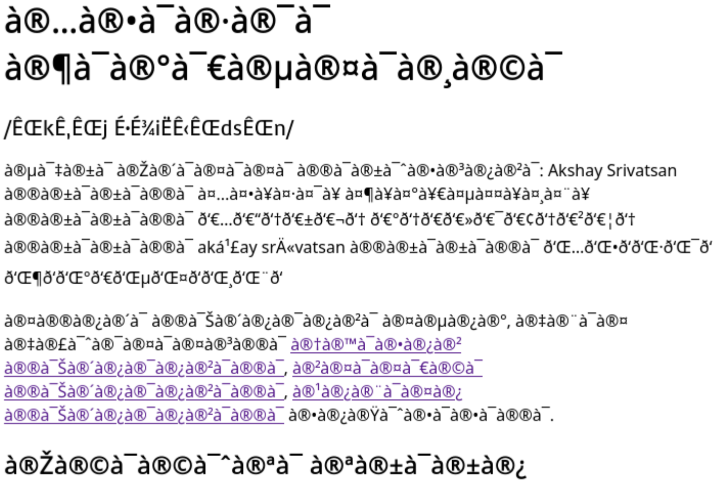
Standards



Source: xkcd 927

Encoding Chaos

Mixing up encodings is problematic:



à®…à®•à¯à®·à®¯à¯
à®¶à¯à®°à¯€à®µà®¤à¯à®¸à®©à¯

/ÊŒkÊ‚ÊŒj É·É¾iËÊ‹ÊŒdsÊŒn/

à®µà¯‡à®±à¯ à®Žà®´à¯à®¤à¯à®¤à¯ à®®à¯à®±à¯ˆà®•à®³à®¿à®²à¯: Akshay Srivatsan
à®®à®±à¯à®±à¯à®®à¯ à¤…à¤•à¥à¤·à¤¯à¥ à¤¶à¥à¤°à¥€à¤µà¤¤à¥à¤¸à¤¨à¥
à®®à®±à¯à®±à¯à®®à¯ ð‘€…ð‘€“ð‘†ð‘€±ð‘€¬ð‘† ð‘€°ð‘†ð‘€ð‘€»ð‘€¯ð‘€¢ð‘†ð‘€²ð‘€¦ð‘†
à®®à®±à¯à®±à¯à®®à¯ akáº£ay srÄ«vatsan à®®à®±à¯à®±à¯à®®à¯ ð‘Œ…ð‘Œ•ð‘ð‘Œ·ð‘Œ¯ð‘
ð‘Œ¶ð‘ð‘Œ°ð‘€ð‘Œµð‘Œ¤ð‘ð‘Œ¸ð‘Œ¨ð‘

à®¤à®®à®¿à®´à¯ à®®à¯Šà®´à®¿à®¯à®¿à®²à¯ à®¤à®µà®¿à®°, à®‡à®¨à¯à®¤
à®‡à®£à¯ˆà®¯à®¤à¯à®¤à®³à®®à¯ à®†à®™à¯à®•à®¿à®²
à®®à¯Šà®´à®¿à®¯à®¿à®²à¯à®®à¯, à®²à®¤à¯à®¤à¯€à®©à¯
à®®à¯Šà®´à®¿à®¯à®¿à®²à¯à®®à¯, à®¹à®¿à®¨à¯à®¤à®¿
à®®à¯Šà®´à®¿à®¯à®¿à®²à¯à®®à¯ à®•à®¿à®Ÿà¯ˆà®•à¯à®•à¯à®®à¯.

à®Žà®©à¯à®©à¯ˆà®ªà¯ à®ªà®±à¯à®±à®¿

This is what happens when a text file is saved with one encoding but opened with another. Back when the internet was young, this was a huge problem—people would use whatever encoding made sense locally to save their webpages, but others on the internet (possibly on the other side of the planet) might use a different encoding to open it. The result was garbled nonsense, and for a long time browsers included tools to try different encodings to see what actually worked.

Unification

- In 1988, three engineers—Joe Becker, Lee Collins, and Mark Davis—proposed a unifying encoding that would supersede all the existing character sets.
- They wanted an encoding that was:
  - universal (to every language)
  - uniform (fixed-width), and

    - unique (no ambiguity)
- They called this encoding UNICODE.
- In 1991, the Unicode Consortium was founded to develop this encoding further.
- In 1992, the Unicode 1.0 was released.

Info from: https://unicode.org/

Representing every Character

- Unicode defines hundreds of thousands of characters.
- Eventually, Unicode wants to encode *every* character used for human writing.
- Each character is assigned a CODE POINT, a number uniquely identifying it.
- Related characters (e.g., letters in the same alphabet) are grouped together into "code planes".
- To uniquely represent all of these characters using a fixed-width encoding, each character would need a lot of bits… but this would be inefficient.

UTF-16 and UTF-32

- Unicode used to have fewer than $2^{16} = 65356$ characters in it (i.e., it had a single "plane").
- At that point, it made sense to use a fixed 16-bit representation for each code point.
- This representation is called the 16-bit Unicode Transformation Format, or UTF-16.
- After Unicode 3.0, there were far too many code points to fit in 16 bits.
- The logical next step, UTF-32, was considered wasteful (every code point would then take 4 bytes).

UTF-16 and UTF-32 may be big-endian or little-endian, adding even more chaos into this.

Ken Thompson Saves the Day: UTF-8

- Ken Thompson (the UNIX guy) and Rob Pike developed a new variable-length encoding for Unicode called UTF-8.
- Under this encoding, more frequently used symbols (like the Latin alphabet) get represented by shorter sequences, while less frequently used ones (like hieroglyphics) get longer sequences.
- Each character is now represented by both a CODE POINT and a UTF-8 BYTE SEQUENCE.
- This encoding is the most popular encoding today, and used on all major operating systems.

There's no concept of endianness for UTF-8 by definition; there is only one correct order for any sequence of bytes.

Line Endings

- Since UTF-8 is backwards-compatible with ASCII, it inherits an issue from ASCII: line endings.
- ASCII was originally designed to be compatible with typewriters.
- On typewriters "carriage return" (`0x0D`) moves the paper to the right, while "line feed" (`0x0A`) moves the paper up.
- MS-DOS (and later Windows) preserved this exactly: a newline was `0x0D 0x0A`.
- Unix decided to save a byte: a newline was just `0x0A`.

- This distinction exists to this day; some people standardized on “DOS line endings” (the web, email, etc.) and some on “UNIX line endings” (git, compilers, etc.).

Generally, internet protocols tend to prefer DOS line endings while command-line tools prefer UNIX line endings. Internet *documents* themselves might use either one, but the protocols (e.g., the SMTP email headers you set in assignment 6) mandate DOS endings.

4 Audio Encoding

Audio

- Just like text, there are many ways to encode audio.
- At the most basic level, audio is an array of “loudness” over time.
- Each array element corresponds to some amount of time, determined by the SAMPLE RATE. A common sample rate for audio is 44100 samples per second (44.1 kHz).
- Each array element has a certain size, corresponding to its accuracy. Eight-bit audio sounds worse than 16-bit audio.
- As most humans have two ears, we tend to prefer STEREO audio, where there are left and right CHANNELS; each channel is a separate array.
- When these arrays are stored raw, they’re called PULSE-CODE MODULATION files (or PCM) files. It’s common to store these in files with a `.wav` file extension.

Audio Codecs

- PCM data takes up a lot of space, and audio is usually predictable, so there are a lot of encoders/decoders (codecs) to store audio more efficiently:
  - MPEG-1 AUDIO LAYER III (MP3)
  - FREE LOSSLESS AUDIO CODEC (FLAC)
  - ADVANCED AUDIO CODING (AAC)
  - OPUS
- Some of these codecs are LOSSY, meaning they discard some data they consider unnecessary from the original audio.
- Some of these codecs are LOSSLESS, meaning they can be reversed 100% accurately to the original audio.

Lossy protocols provide smaller files, at the cost of some audio quality. Lossless protocols provide the best audio quality, at the cost of larger files. Which one is best depends on the particular use case. Note that most lossy protocols do let you configure the *amount* of loss, so you can use them to find a middle ground if you so desire. However, if you reencode an audio file through lossy encodings enough times, you will eventually end up with an inaudible file.

Audio Containers

- While a codec determines how raw audio data is represented as numbers, a CONTAINER format determines how that encoded audio is saved on disk.
- While some containers and codecs are often found together, it’s generally possible to mix-and-match a codec and a container format.
- The container stores information like song titles and artist names.

- The container may include multiple channels of audio.

5 Image Encoding

Image Formats

- Images are slightly simpler than audio, since each container format generally contains a specific codec.
- There are two kinds of images: raster and vector.
- RASTER images are essentially a grid of pixels, each of which has a specific color.
  - Photographs
- VECTOR images are a set of instructions to draw an image.
  - Drawings, Logos, Diagrams
- Vector images can be rendered at any size on-demand, while raster images have a fixed size.

Vector images are also more efficiently stored than raster images. In general, it's a good idea to use vector images whenever possible. The main case where raster images are useful is for photographs, since cameras can only produce raster images. Raster images may also be useful for speed, since vector images need to be converted to raster before they can be displayed on a computer monitor.

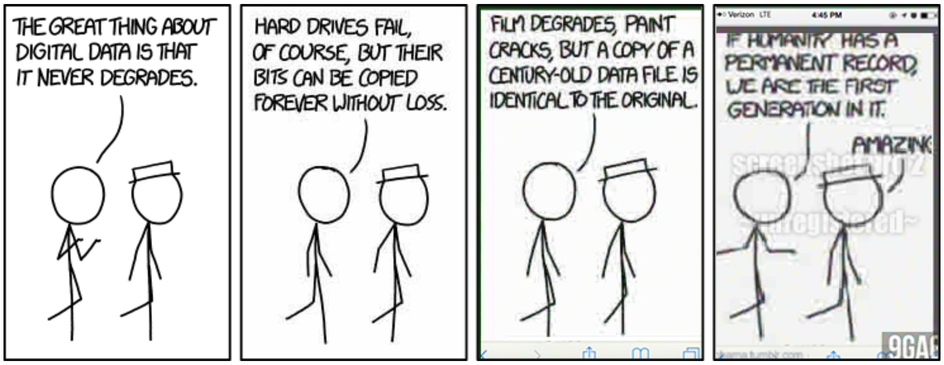
Raster Formats

Common raster image formats include:

- Joint Photographic Experts Group (JPEG)
- Graphics Interchange Format (GIF)
- Portable Network Graphics (PNG)
- High Efficiency Image File Format (HEIF/AVIF)
  - High Efficiency Video Coding (HEVC)
  - AOMedia Video 1 (AV1)
- Tagged Image File Format (TIFF)
- Windows Bitmap (BMP)

Once again, there is a lossy vs. lossless distinction.

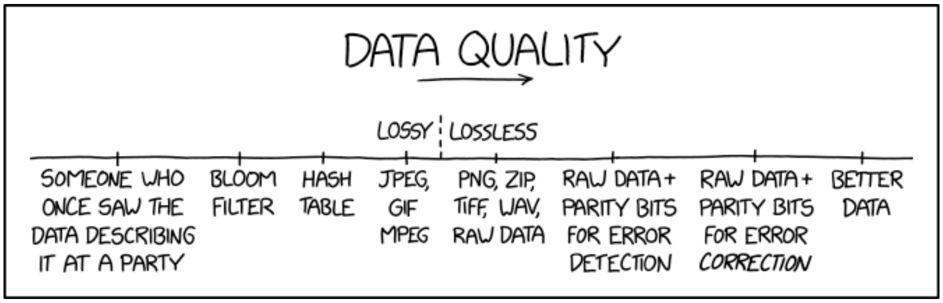
Lossy Compression



Source: xkcd 1683

And, while we're on the topic:

Data Quality



Source: xkcd 2739

Vector Formats

Common vector image formats include:

- Scalable Vector Graphics (SVG)
- Gerber (for printed circuit board designs)

Vector formats are also common for 3D objects due to their relatively small size:

- Wavefront OBJ
- Blender
- various Autodesk formats

Raw Images

- High-end cameras can "shoot in raw", which captures the raw pixel data coming from the image sensor.
- These images contain ridiculous amounts of data, most of which isn't useful.
- Photographers need to "develop" RAW images into a final image.

Color Spaces

- While light forms a continuous spectrum of colors, humans can usually only see the intensities of three wavelengths.
- These wavelengths roughly correspond to red, green, and blue, so we can try to parametrize a color by how much red, green, and blue (RGB) need to be mixed to make it.
- The exact way human brains reconstruct colors from these three wavelengths is complicated, so there are various COLOR SPACES which attempt to approximate it.
- There are other ways to parametrize colors; for example, hue, saturation, and value/brightness (HSV).

A particular image format may require a specific color representation, or it may support multiple color spaces and require the software reading it to translate as appropriate.

Even More Options

- As if different color spaces weren't bad enough, there are different representations of each color space.
- RGB (red, green, blue) is pretty common, but sometimes BGR (blue, green, red) or other permutations are also used.
- Sometimes you'll see RGBA or ARGB ("A" stands for "alpha", which is transparency).
- RGB888 uses 8 bits per color (24 bits total), RGB565 uses 5 or 6 per color (16 bits total), RGB332 uses 2 or 3 (8 total).
- 24-bit color is called "true color", 16-bit color is called "high color", and anything higher than 24 is called "deep color".

High Dynamic Range

- Human eyes have a very high "dynamic range"; they can resolve detail even when some parts of an image are very bright and some are very dark.
- Cameras don't have this, so they fake it by taking photos with different exposure settings and merging them together.
- Only some codecs support HDR, including HEIC and AVIF.

Resolution

The "size" of an image in pixels is called its resolution.

Common resolutions include:

HD 1280×780

Full HD 1280×1080

Full HD 1920×1080

4K Ultra HD 3840×2160

8K Ultra HD 7680×4320

Square power-of-two resolutions are also common (e.g., 16×16, 32×32, 2048×2048), especially for logos.

6 Video Encoding

Video Codecs

Like audio, there are several different video codecs in use:

- High Efficiency Video Coding (H.265)
- Advanced Video Coding (H.264)
- VP8 and VP9
- Theora

Uncompressed video is almost entirely useless outside of professional use, since the files are ridiculously large. Video is compressed using lossy algorithms.

Video Containers

There are also several container formats:

- Matroska (MKV)
- MPEG-4 (MP4)
- QuickTime (MOV)
- Audio Video Interleave (AVI)
- WebM
- Ogg

Video containers also double as audio containers, since most videos have associated audio.

Frame Rate

In addition to all the parameters from both images and audio, video also has an additional parameter: the frame rate.

The frame rate of a video controls how many images are shown per second.

High frame rates look smoother but require more data.

Typical frame rates are 24, 30, and 60 frames per second.

Reencoding Video

- In order to do almost any operation to a video, it needs to be reencoded.
- Reencoding a video is slow; usually it's about 1:1 with the length of the video.
- Reencoding a 90-minute lecture video takes about 90 minutes on my laptop.
- Reencoding also effectively recompresses a video, losing information every time if the encoding is lossy.

7 Conclusion

Useful media tools

`file`: identifies the format of many files

`iconv`: converts between text encodings

FFmpeg: converting (or identifying) audio and video

ImageMagick: editing images

Pandoc: converts documents between formats

Review

- There are lots of ways to encode any given piece of media.
- The appropriate encoding for a specific purpose depends on a lot of factors.
- Different people/software/systems may expect different encodings, and run into problems when given data in the wrong encoding.

- It's important to keep encoding in mind when working with media, and generally not mix together things with different encodings.