Concepts of Computer Science

Data Representation

Chapter Goals

- Lecture 1:
  - Data on a finite machine
  - Representation of whole numbers
  - Representation and arithmetic on negative numbers
- Lecture 2:
  - Representing real numbers (floating point)
- Lecture 3:
  - Using numbers to represent sound, text and colours
  - Issues with compression and storage of data

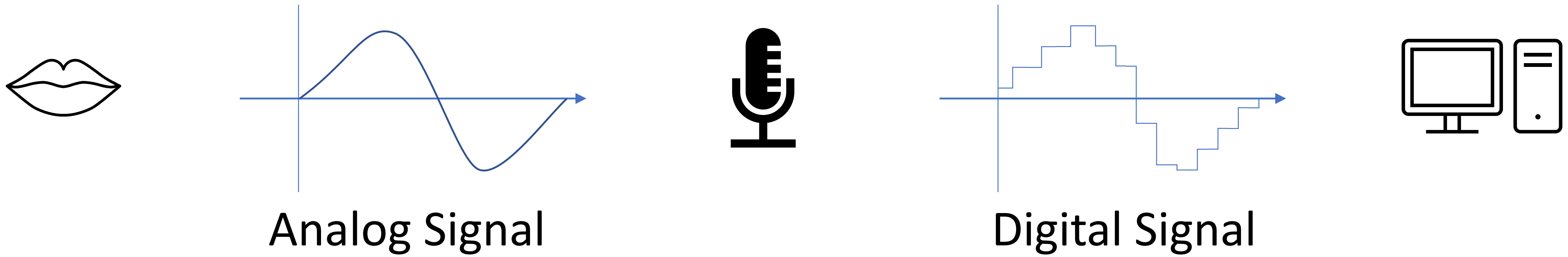
- Lecture 1:
  - Data on a finite machine
  - Representation of whole numbers
  - Representation and arithmetic on negative numbers

Finite vs. Infinite

- A computer can only process a finite amount of data in finite time.
- How do we represent an infinite world and compute on that representation?
- We have to represent enough information to satisfy our goals.
- Too little information and the result may not be good enough. Too much information and the computation may be too expensive.

Analog vs. Digital

- We often need to **digitise** our **analog data**
- Sometimes this is okay (converting integers for example), but sometimes this leads to an approximation of the original:



Binary Representations

- If **one** bit can represent 2 things (0,1)…
- **Two** bits can represent 4 things (00,01,10,11)…
- **Three** bits can represent…? (000, 001, 010, 100,…, 111)
- **Four** bits? **Eight** bits? **Sixty-four** bits?

- For each extra bit, the number of things we can represent **doubles**. With *n* bits, we can represent 2^n different things…

How many *things*

n	2^n	
1	2	Two
2	4	Four
3	8	Eight
4	16	Sixteen
8	256	Two Hundred and Fifty Six
16	65536	~ Sixty Five Thousand
32	4,294,967,296	~ Four Billion
64	18446744073709551616	~ Eighteen Quintillion

How many *things do we need?*

Representing the Natural Numbers

- **Simple**: Interpret the bit pattern as a binary number.
- 4 bits lets us represent the numbers 0-15:

Bit Pattern	Number	Bit Pattern	Number
0000	0	1000	8
0001	1	1001	9
0010	2	1010	10
0011	3	1011	11
0100	4	1100	12
0101	5	1101	13
0110	6	1110	14
0111	7	1111	15

Representing Negative Numbers

- First idea: Lets use the **first bit** to denote whether the number is positive (0) or negative (1)!
- This is called Sign-Magnitude representation.
  - You have a sign bit (the first bit)
  - The remaining bits denote the magnitude (or value)

Why might this be a problem?

Sign-Magnitude Representation

- First idea: Lets use the **first bit** to denote whether the number is positive (0) or negative (1)!

Bit Pattern	Number	Bit Pattern	Number
0000	0	**1**000	-0
0001	1	**1**001	-1
0010	2	**1**010	-2
0011	3	**1**011	-3
0100	4	**1**100	-4
0101	5	**1**101	-5
0110	6	**1**110	-6
0111	7	**1**111	-7

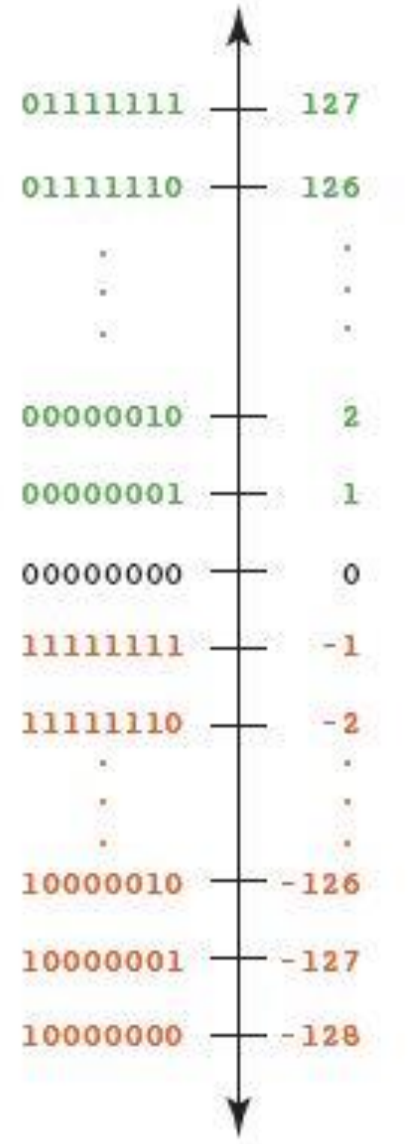
Representing Negative Numbers

- Revised idea: We’ll still have the first bit denote positive and negative, but we will let the numbers **wrap around**.
- Start counting from 0, but when we use up the *n-1* positions we resume counting from **greatest magnitude negative number**.
- Known as Two’s Complement, or Modulo Arithmetic

Two's Complement

- Revised idea: We'll still have the first bit denote positive and negative, but we will let the numbers **wrap around**.

Bit Pattern	Number	Bit Pattern	Number
0000	0	**1**000	-8
0001	1	**1**001	-7
0010	2	**1**010	-6
0011	3	**1**011	-5
0100	4	**1**100	-4
0101	5	**1**101	-3
0110	6	**1**110	-2
0111	7	**1**111	-1



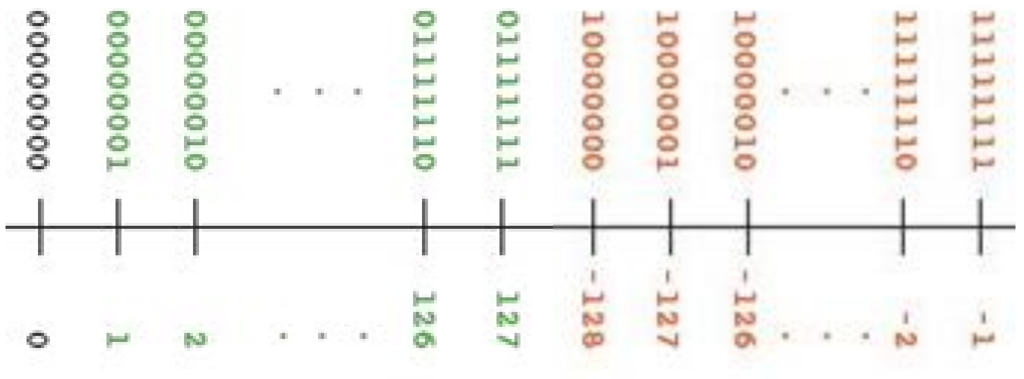
Two’s Complement:

A mapping where the negative values are mapped to the upper half of the possibly representable numbers.

To calculate the negative value of a value:

Negative(x) = $2^k - x$

(in binary we can just use flip and +1)



Ranges with ints of different sizes

- In computing, a range limited (i.e. fixed bit-length) integer is known as an **int**.
- Depending on how it is used, we can have **signed** and **unsigned** ints.

	Range	
Number of Bits	**Unsigned int**	**(signed) int**
8 bits	0 – 255	-128 – 127
16 bits	0 – 65535	-32768 – 32767
32 bits	0 – 4294967295	-2147483648 – 2147483647
64 bits	0 – 18446744073709551615	-9223372036854775808 – 9223372036854775807

Negating a Two’s Complement value

- To get the negative of a number in Two’s Complement, we:
  - Invert the bits (i.e. swap 1s for 0s and vice-versa)
  - Add 1.

E.g. Negate 5 in 4-bit Two’s Complement	
5_{10} =	0101_2
Invert the bits =	1010_2
Add 1 to the result =	1011_2
-5_{10} =	1011_2

E.g. Negate -2 in 4-bit Two’s Complement	
-2_{10} =	1110_2
Invert the bits =	0001_2
Add 1 to the result =	0010_2
2_{10} =	0010_2

Arithmetic with Two’s Complement

- Addition: add the numbers which represent the number of interest, and then throw away any carried digits beyond.

$$\begin{array}{rr} & 2 \\ + & 3 \\ \hline & 5 \end{array} \Longrightarrow \begin{array}{rcccc} & 0 & 0 & 1 & 0 \\ + & 0 & 0 & 1 & 1 \\ \hline & 0 & 1 & 0 & 1 \end{array}$$

Arithmetic with Two’s Complement

- Addition: add the numbers which represent the number of interest, and then throw away any carried digits beyond.

$$\begin{array}{cr} & 5 \\ + & -6 \\ \hline & -1 \end{array} \quad \Longrightarrow \quad \begin{array}{ccccc} & 0 & 1 & 0 & 1 \\ + & 1 & 0 & 1 & 0 \\ \hline & 1 & 1 & 1 & 1 \end{array}$$

Arithmetic with Two's Complement

- Addition: add the numbers which represent the number of interest, and then throw away any carried digits beyond.

$$\begin{array}{r} -3 \\ +\quad -2 \\ \hline -5 \end{array} \quad \Rightarrow \quad \begin{array}{rcccc} \cancel{1} & 1 & & & \\ & 1 & 1 & 0 & 1 \\ + & 1 & 1 & 1 & 0 \\ \hline & 1 & 0 & 1 & 1 \end{array}$$

Arithmetic with Two’s Complement

- Subtraction: Use identity a – b = a + (-b)
- As we know how to negate a value, and we know how to apply addition, we can now do subtraction easily!

$$\begin{array}{rr} & 2 \\ - & 3 \\ \hline & -1 \end{array} \quad \Longrightarrow \quad \begin{array}{rcccc} & 0 & 0 & 1 & 0 \\ - & 0 & 0 & 1 & 1 \\ \hline & & & & \end{array} \quad \Longrightarrow \quad \begin{array}{rcccc} & 0 & 0 & 1 & 0 \\ + & 1 & 1 & 0 & 1 \\ \hline & 1 & 1 & 1 & 1 \end{array}$$

Arithmetic with Two’s Complement

- Subtraction: Use identity a – b = a + (-b)

Example of 2 – 3:

In 4-bit Two’s Complement:	**0010 – 0011**
Negate the second number:	**0010 + 1101**
Add the two together:	**1111**
Result is:	**-1**

Integer Overflow

- So far we’ve only discussed calculations where the result is within the range of numbers that can be represented with our given scheme.

- What happens if we go beyond this?
  - If 4-bit Two’s Complement can represent the range -8 to 7, what happens if we do 5+5? -5 – 5?

- The result exceeds the representable range of numbers.

Integer Overflow

- Example: 5 + 5 in 4-bit Two’s Complement:

$$\begin{array}{ccccc} & 1 & & 1 & \\ & 0 & 1 & 0 & 1 \\ + & 0 & 1 & 0 & 1 \\ \hline & 1 & 0 & 1 & 0 \end{array}$$

- In our 4-bit Two’s Complement scheme, the result here is -6, not 10!

Integer Overflow

- Example: -5 - 5 in 4-bit Two’s Complement:
- Change this to -5 + -5:

$$\begin{array}{ccccc} \cancel{1} & & 1 & 1 & \\ & 1 & 0 & 1 & 1 \\ + & 1 & 0 & 1 & 1 \\ \hline & 0 & 1 & 1 & 0 \end{array}$$

- In our 4-bit Two’s Complement scheme, the result here is 6, not -10!

Integer Overflow

- The problem here is that the value has **overflowed** beyond our representable range.
- It doesn’t matter if the value goes too high, or too negative, both result in “overflow”.
- Programmers must ensure that the result is in range, or they must handle an overflow error in some other way

- Lecture 2:
  - Representing real numbers (floating point)

Real Numbers in decimal

- A number with a whole part and a fractional part (either of which may be zero):
  - 17.0
  - 3.333333…
  - 1.41421356…
  - 0.002

- Note that fractional part may be **infinite**.
- In decimal, positions to the right of the point are tenths, hundredths, thousandths, etc.: 10^{-1}, 10^{-2}, 10^{-3}…

Real Numbers in binary

- The same happens in binary.
- Positions to the right of the **radix point** are:
  - Halves (2^{-1})
  - Quarters (2^{-2})
  - Eighths (2^{-3})
  - …

The name of the "point" is actually the 'Radix Point'.

We tend to call it a "decimal point", but that's because we often work in decimal.

…	**2^2**	**2^1**	**2^0**	**.**	**2^{-1}**	**2^{-2}**	**…**
	Fours	Twos	Ones	.	Halves	Quarters	

Real Numbers in binary

- To convert the value we can use the calculations we have seen before, taking into account the new negative positions in the fractional portion of the number.

- Example:

10.0101_2 =

2^1	2^0		2^{-1}	2^{-2}	2^{-3}	2^{-4}
1	0	.	0	1	0	1

$= 2 + 0 + 0 + 0.25 + 0 + 0.0625$

$= 2.3125_{10}$

Floating Point representation

- Now we have a complicated number (whole and fractional part)

- A Real Number can be defined by the formula:

$$\text{sign} \cdot \text{mantissa} \cdot \text{base}^{\text{exponent}}$$

- This representation is called **floating point**, because the radix point “floats”.

- We will assume that both the mantissa and the exponent fits within some pre-agreed number of bits.

Floating Point representation

$$\text{sign} \cdot \text{mantissa} \cdot \text{base}^{\text{exponent}}$$

Sign: +1 or -1, represented as 0 and 1, respectively. (think back to Two's Complement). This is a single bit.

Mantissa: Significant digits of the number being represented. This is an integer.

Base: Base used. Often pre-determined (binary), so no need to store.

Exponent: Scales our number by shifting the position of the floating radix point. This is an integer.

Floating Point example

To represent π = 3.14159265358979... with a finite number of bits:

Round the number to some number of significant digits, say 3.14159

Rewrite as floating point:

$$+1 \cdot 314159 \cdot 10{-}5$$

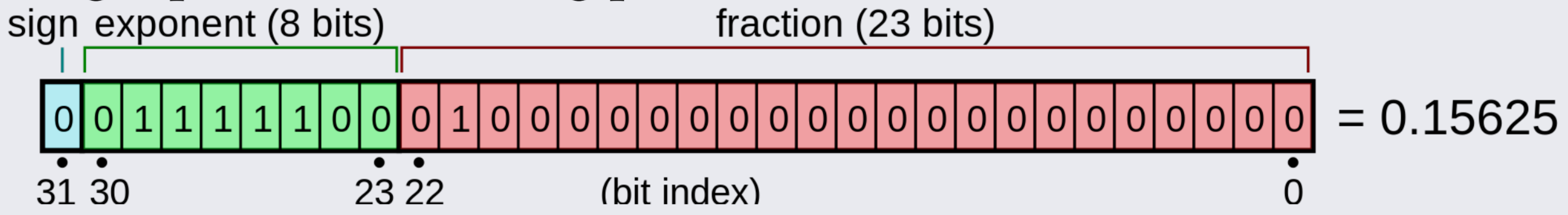
Represent +1, 314159 and -5 as integers, as in previous slides

IEEE 754 Standard for Floating Point

- Most current processors follow the IEEE 754 standard for floating point values:

Float

Single precision floating point.

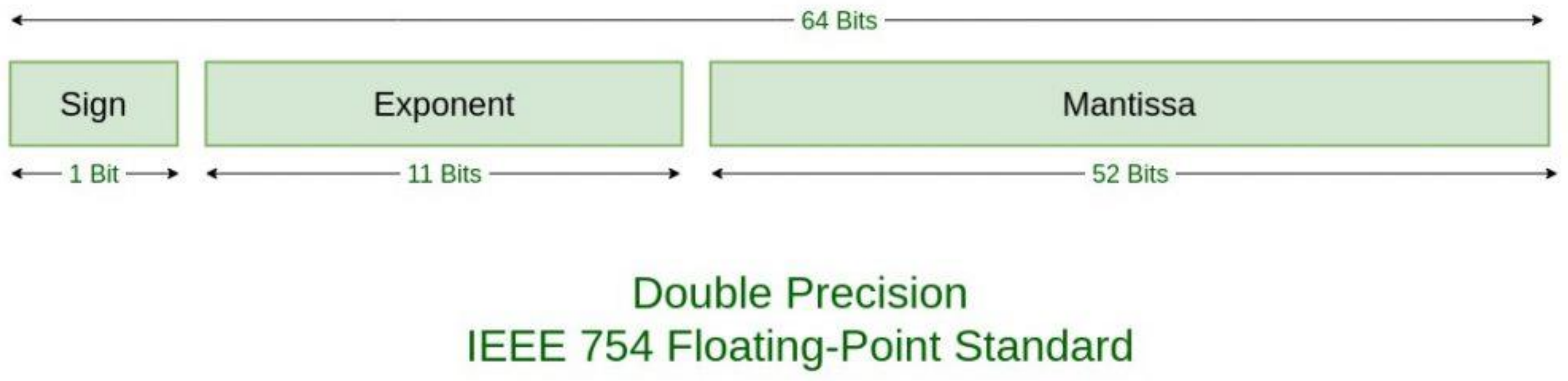


Number encoded is

$$(-1)^{b_{31}} \cdot (1.b_{22}b_{21}\ldots b_0)_2 \cdot 2^{(b_{30}b_{29}\ldots b_{23})_2 - 127}$$

IEEE 754 Standard for Floating Point

- There are other standards on number representation. For example the following gives higher precision floating point values than the Single Precision on the last slide:



https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/

Base conversion with real numbers

- Remember our method for calculating an integer number with positional notation:

```
While (value is not zero)
    Divide value by the new base
    Store remainder
    Replace value with the quotient
Read remainders in reverse order
```

- A similar method holds for the fraction component in a real number...

Base conversion with real numbers

- To calculate the value of the fractional part, we now want to scale up (due to the negative exponent):

```
While (value is not zero OR precision is reached)
    Multiply value by the new base
    Store the whole part
    Replace value with the fractional part of the result
Read whole parts in order
```

Example: Convert 3.625_{10} to binary:

Whole part (3), use **division** and **remainder** method:

3 / 2 = 1, remainder: 1
1 / 2 = 0, remainder: 1

Read remainders **in reverse**: 11

Fractional part (0.625), use **multiply** and **whole** method:

0.625 * 2 = 1, fraction: 0.25
0.25 * 2 = 0, fraction: 0.5
0.5 * 2 = 1, fraction: 0

Read wholes **in order**: 101

Therefore: $3.625_{10} = 11.101_2$

Example: Convert 0.3_{10} to binary:

Whole part (0):

0 / 2 = 0, remainder: 0

Read remainders **in reverse**: 0

Fractional part (0.3):

0.3 * 2 = 0, fraction: 0.6
0.6 * 2 = 1, fraction: 0.2
0.2 * 2 = 0, fraction: 0.4
0.4 * 2 = 0, fraction: 0.8
0.8 * 2 = 1, fraction: 0.6
0.6 * 2 = 1, fraction: 0.2
…

Read wholes **in order**: 010011…

Therefore: $0.625_{10} = 0.0100110011\ldots_2$

Scientific notation

A form of floating-point representation in which the decimal point is kept to the right of the leftmost digit

12001.32708 becomes 1.200132708E+4 in scientific notation (E+4 is how computers display x10^4)

What is 123.332 in scientific notation?

What is 0.0034 in scientific notation?

Representing Text

- Simple Idea:

  Assign a binary bit pattern per character you want to represent. Text is then stored as a sequence of bit patterns.

- Not so simple implementation:
  - There are many characters (different alphabets, mathematical symbols, emojis etc.)
  - Some characters are modifications of others: e, è, é, ë
  - Direction of the character sequence can matter: Arabic, Hebrew etc.

Character set standards

ASCII (1960’s) 7-bit encoding.

EBCDIC (1960’s) 8-bit encoding.

ISO 8859 (1987) 8-bit encodings.

- ISO 8859-1 also known as Latin-1.

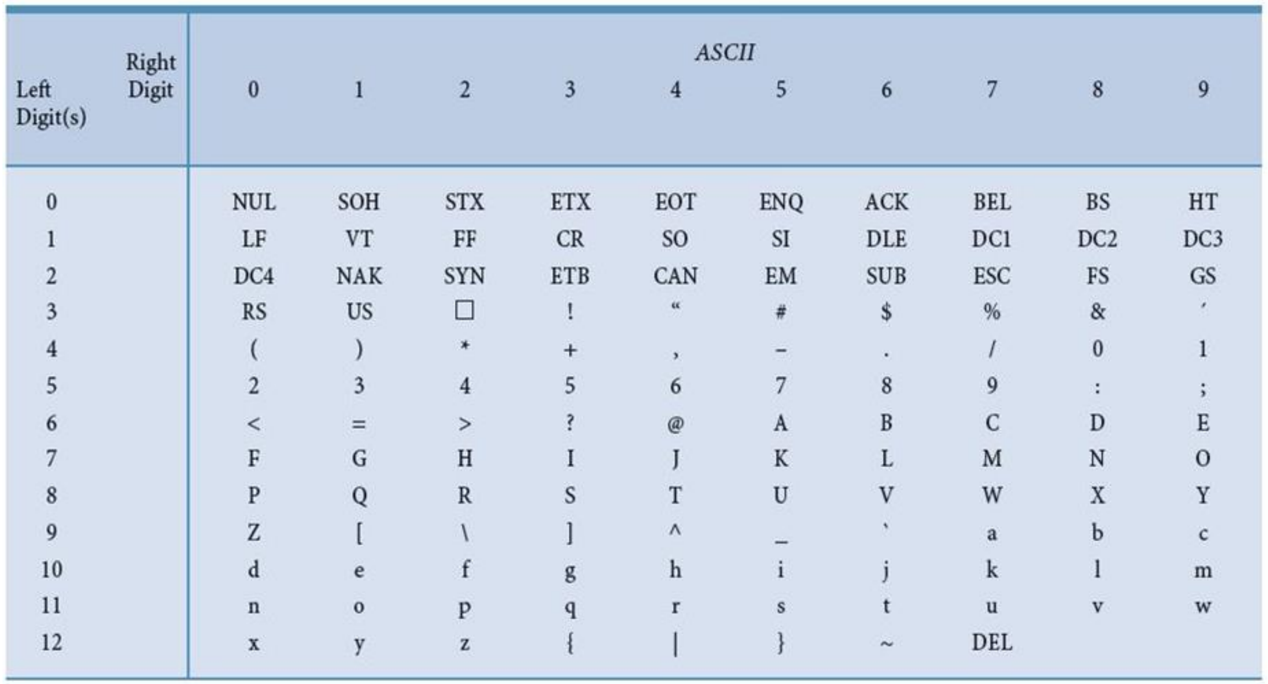
Unicode (1993–) 143,859 characters covering 154 modern and historic scripts.

- UTF-8
- UTF-16
- UTF-32

ASCII

- Originally 7 bits, providing 128 unique characters.
- Later extended to use 8 bits. Now each character is a byte.
- First 32 characters aren’t normal characters, what do they do?

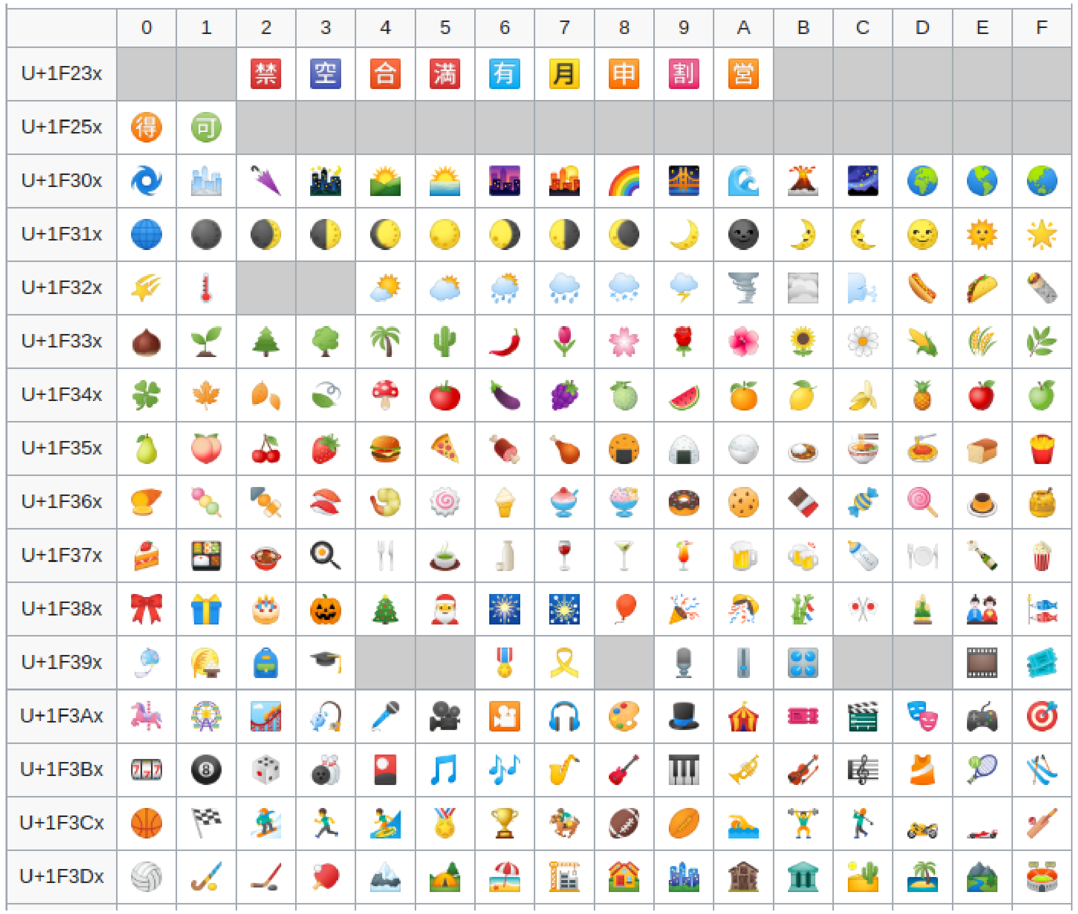
- ASCII doesn’t cover advanced international requirements



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

Unicode

- More bits used:
  - Commonly: 16 bits
  - Sometimes: 8 or 32 bits
- First 256 correspond to the extended ASCII set
- UTF-16 allows 65536 different characters



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
U+1F23x			🈲	🈳	🈴	🈵	🈶	🈷	🈸	🈹	🈺					
U+1F25x	🉐	🉑														
U+1F30x	🌀	🌁	🌂	🌃	🌄	🌅	🌆	🌇	🌈	🌉	🌊	🌋	🌌	🌍	🌎	🌏
U+1F31x	🌐	🌑	🌒	🌓	🌔	🌕	🌖	🌗	🌘	🌙	🌚	🌛	🌜	🌝	🌞	🌟
U+1F32x	🌠	🌡			🌤	🌥	🌦	🌧	🌨	🌩	🌪	🌫	🌬	🌭	🌮	🌯
U+1F33x	🌰	🌱	🌲	🌳	🌴	🌵	🌶	🌷	🌸	🌹	🌺	🌻	🌼	🌽	🌾	🌿
U+1F34x	🍀	🍁	🍂	🍃	🍄	🍅	🍆	🍇	🍈	🍉	🍊	🍋	🍌	🍍	🍎	🍏
U+1F35x	🍐	🍑	🍒	🍓	🍔	🍕	🍖	🍗	🍘	🍙	🍚	🍛	🍜	🍝	🍞	🍟
U+1F36x	🍠	🍡	🍢	🍣	🍤	🍥	🍦	🍧	🍨	🍩	🍪	🍫	🍬	🍭	🍮	🍯
U+1F37x	🍰	🍱	🍲	🍳	🍴	🍵	🍶	🍷	🍸	🍹	🍺	🍻	🍼	🍽	🍾	🍿
U+1F38x	🎀	🎁	🎂	🎃	🎄	🎅	🎆	🎇	🎈	🎉	🎊	🎋	🎌	🎍	🎎	🎏
U+1F39x	🎐	🎑	🎒	🎓			🎖	🎗		🎙	🎚	🎛			🎞	🎟
U+1F3Ax	🎠	🎡	🎢	🎣	🎤	🎥	🎦	🎧	🎨	🎩	🎪	🎫	🎬	🎭	🎮	🎯
U+1F3Bx	🎰	🎱	🎲	🎳	🎴	🎵	🎶	🎷	🎸	🎹	🎺	🎻	🎼	🎽	🎾	🎿
U+1F3Cx	🏀	🏁	🏂	🏃	🏄	🏅	🏆	🏇	🏈	🏉	🏊	🏋	🏌	🏍	🏎	🏏
U+1F3Dx	🏐	🏑	🏒	🏓	🏔	🏕	🏖	🏗	🏘	🏙	🏚	🏛	🏜	🏝	🏞	🏟

https://en.wikipedia.org/wiki/Emoji#Unicode_blocks

Some Unicode examples

Character Symbol	**Hexadecimal value**	**Name**
Ä	C4	Latin Capital Letter a with Diaeresis
Д	414	Cyrillic Capital Letter De
आ	906	Devanagari Letter Aa
㐮	342E	Ideograph: to help; to assist, to achieve, to rise; to raise
𓀊	1300A	Egyptian Hieroglyph A008; rejoice, celebrate, to be jubilant
😁	1F601	Grinning Face with Smiling Eyes Emoji

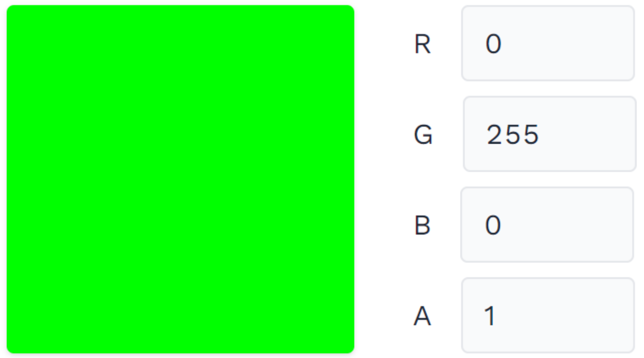
Representing colours

- What is colour? Essentially it is an attribute caused by things reflecting or emitting certain wavelengths of light.

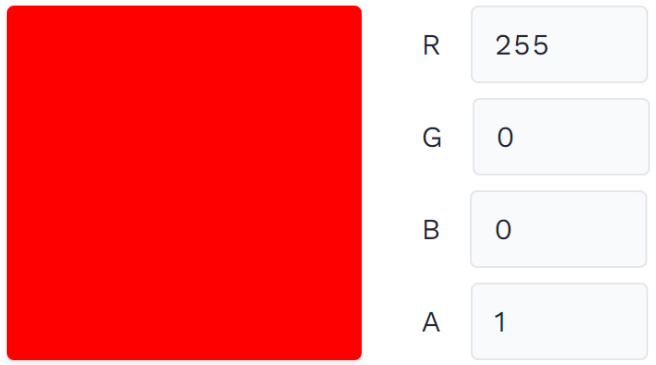
- We perceive colour with **photoreceptors** in our eyes, these cells respond to particular wavelengths and our brain interprets this as a colour.

Representing colours

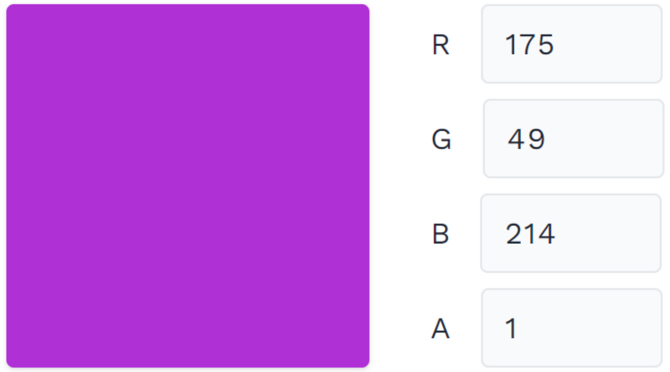
- One way represent a colour is to encode the intensity of **red**, **green**, and **blue** (RGB) frequencies:



R	0
G	255
B	0
A	1



R	255
G	0
B	0
A	1



R	175
G	49
B	214
A	1

Give it a try [RGBA Color Picker - https://rgbacolorpicker.com/](https://rgbacolorpicker.com/)

RGB colours

- Assuming that we use 8 bits per channel, then we can encode colours as triples of numbers (red, green, blue) in the range 0-255.
  - 0 represents no presence, 255 represents maximum presence.

- Examples:
  - (0,255,0) is full green, but no red or blue. This looks green.
  - (255,255,255) is full red, green and blue. This looks white .
  - (150,75,0) is some red, less green, and no blue. This looks brown.

- Hexadecimal is often used for colours, for example the brown colour above can be written as #964B00.

Other colour encodings

- Apart from RGB encoded colours there are many other schemes:
  - **HSV** (hue, saturation, value) and related models.
  - **CMY** (cyan, magenta, yellow).
  - **CMYK** (cyan, magenta, yellow, black). Popular in printing process.

- **Colour depth**
  - Number of bits used to encode colour can vary. 8 bits is standard, but some situations use more bits. More bits, more colours representable.
  - Humans can perceive ~10 million different colours, and 24 bits would give us the ability to represent over 16.7 million.

Digitized images and graphics

- Need to represent shapes and colours within an image.
- Requires discretization of the observation in 2D space, and in colour.
- **Pixels**
  - Dots of colour in image (or display device)
- **Resolution**
  - Number of pixels in image (or device)

Vector vs Raster Graphics

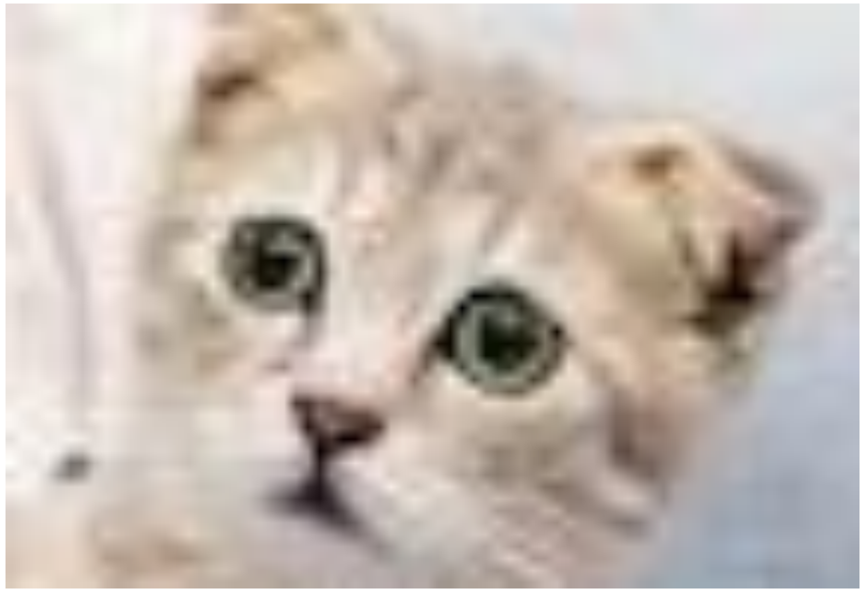
- **Raster Graphics**
  - Treat image as collection of pixels, encoding the colour for each.
  - Can be larger in size. Resizing leads to pixelation.
  - Most common formats: BMP, GIF, PNG, and JPEG

- **Vector Graphics**
  - Treat image as collection of mathematically defined geometric objects, in the image. Describing direction, length, thickness.
  - Re-sizing is easy, little pixelation.
  - Common formats: SVG

JPEG Images
(Joint Photographic Experts Group)

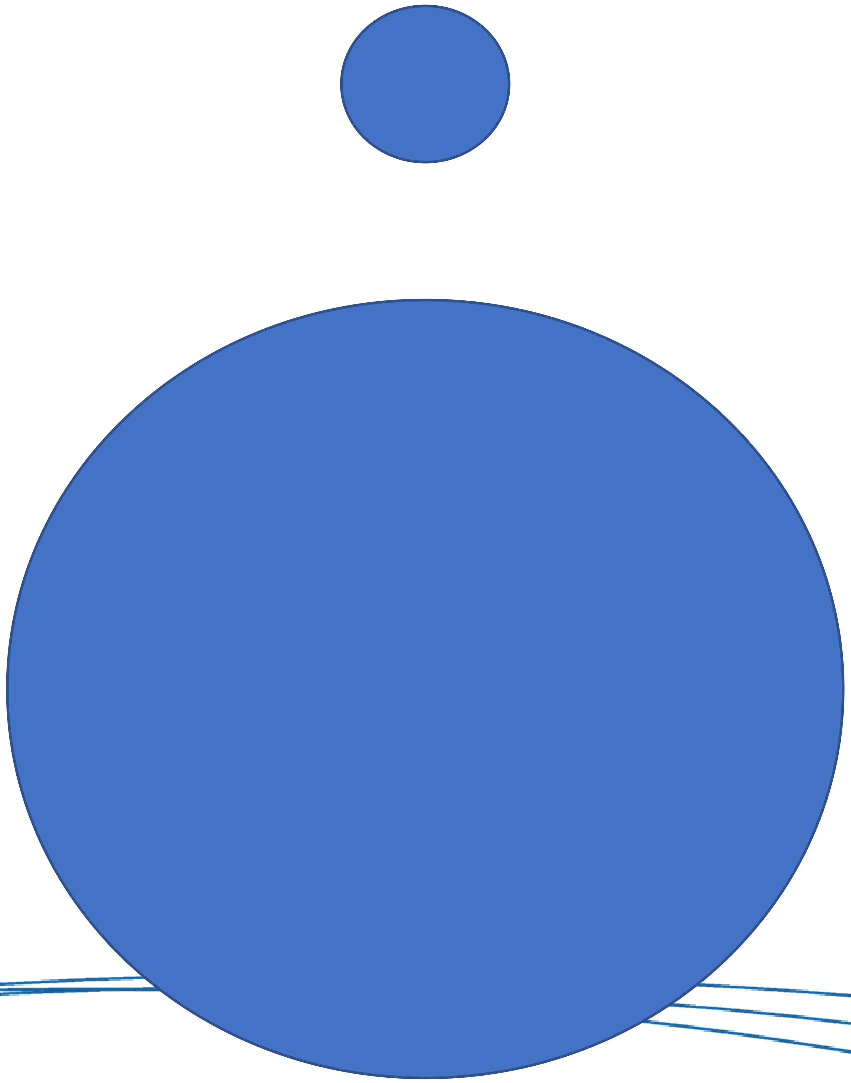
- JPEG is actually a compression scheme, looking to approximate the pixel-wise colours within the image.
- It averages colour hues over short distances, lowering the need to store every single pixel value.
- **Why:** Human vision tends to blur colours together within small areas, so no need to be perfect.
- **How:** Transform from the spatial domain to the frequency domain, then discard high frequency component. Argh, maths!

JPEG pixelation



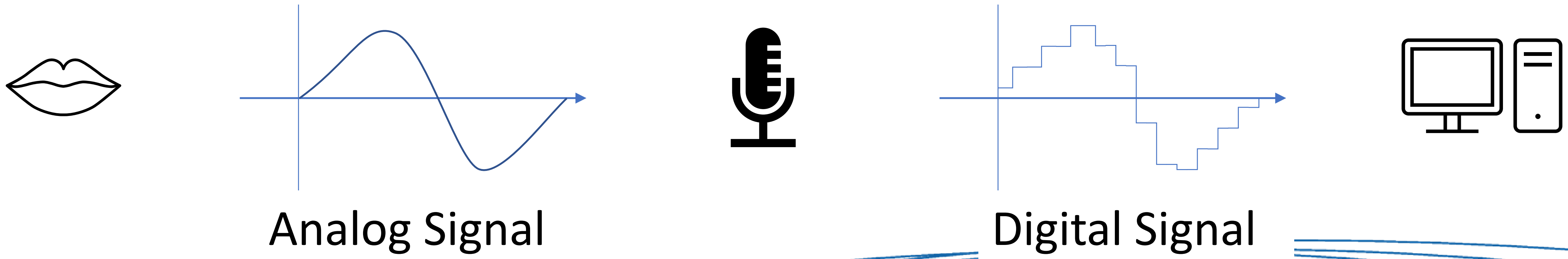
SVG

- As the circles are mathematically defined, we can make them larger/smaller and pixelation does not occur.
- Sometimes the representation of a vector graphic is smaller in size compared to a raster version, as not every pixel is stored.
- However, requiring a mathematical definition for the objects means that it doesn't work well for representing natural photographs.



Representing Sound

- Sound is the perception of pressure changes within the air (roughly)
- Convert pressure to digital signal with membrane (speaker / mic)
- Record air pressure on the membrane, sampled at periodic intervals. This is the **sampling frequency**

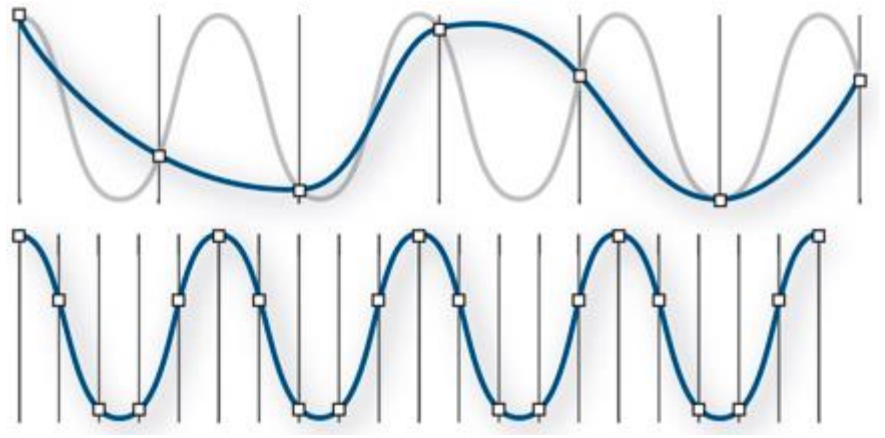


Representing Sound

- The frequency at which we sample the air pressure allows us to more faithfully represent the original continuous frequency.

- Humans can hear up to a frequency of 20 kHz

- We often sample at higher frequencies:
  - ~11kHz – AM Radio
  - ~44kHz – CD quality
  - ~48kHz – DVD quality
  - 96kHz – Blu-ray quality



Digitizing audio in Audition (adobe.com)

- Lecture 3:
  - Issues with compression and storage of data

Representing data and storing it

- All of the *things* we've discussed so far are represented as binary values on the machine.

- Some of these representations can be quite large:
  - One RGB colour requires us to store 24 bits (8 bits for R, G, and B).
  - A High-def image has a resolution of 1920 x 1080, or 2,073,600 pixels.
  - That would be just under 50 million bits to store a single picture.
  - Works out to roughly 6 mega-bytes.

Data Compression

- **Data compression**
  - Reduction in the amount of space needed to store a piece of data or the bandwidth to transmit it
- **Compression ratio**
  - The size of the compressed data divided by the size of the original data

What does a small ratio mean? A large ratio?

$$\text{compression\_ratio} = \frac{length(\text{compressed})}{length(\text{original})}$$

Data Compression

A data compression technique can be:

- **lossless**, which means the data can be retrieved without any loss of original information
- **lossy**, which means some information may be lost in the process of compression

When might you want one over the other?

Data Compression

- The examples we will use in this lecture will predominantly focus on text data, however remember that all of this information is represented on the machine as **bit strings**.

- This means you should think about how these methods may apply in other situations (i.e. different types of data rather than text), and how they may work on the bit string level.

Run-Length Encoding

- In some kinds of files, a single value may be repeated many times in a long sequence.

- Can you think of any example scenarios?

- Why repeat these values, when we can have them once and then say how many times to repeat it. We can encode the **run length** of the value.

Run-Length Encoding

- Replace the repeated sequence with:
  - A flag character (such as * or \)
  - The value to repeat
  - The number of times to repeat it.

Why?

- Only need to do this with a **sequence of more than 3** values

- This may not happen to often in natural text, but it could happen quite often with binary values!

Run-Length Encoding

Example

Original text	aaaaabbbccccccc+!!!!!!
Encoded text	*a5bbb*c7+*!6

Compression ratio is $\frac{13}{22} = 0.59$.

Example

Encoded text	a*b5*c4de!
Decoded text	abbbbbccccde!

Keyword Encoding

- Repeating single values may not be very common.
- What about finding frequently used patterns and replacing those with a single character?
- All we would need is a list of the common patterns and the single character that they were replaced with.

Keyword Encoding Example

Original message:

The shop sells fruit and bread and drinks and clothes and vegetables

“and” is used a lot. If we encode the keyword *and* by & we get:

The shop sells fruit & bread & drinks & clothes & vegetables.

The original message has 69 characters and the compressed has 61 characters. The compression rate is 61/69 = ~0.88

Keyword Encoding

- To perform encoding and decoding we need the mapping table. This can often be document or application specific.
- In our previous example, we would need to record that “and” is replaced with “&” and back.
- Obviously storing this table takes up space too, so this can affect how effective the compression is. (we’ll ignore this in our calculations for the moment)

Variable Length Bit String Encoding

- Remember that these values have an underlying bit string representation:
    - Text characters may be Extended ASCII (8 bits) or perhaps UTF-16

- **Observation**:

  The letter ‘e’ is far more common than the letter ‘q’

- **Idea**:

  Can we change the number of bits used to represent a value, making more frequent values use less bits than uncommon ones?

Values could be represented by **variable length bit strings**.

The character “e” in Unicode:

0000000001100101

The character “x” in Unicode:

0000000001111000

The “Pile of Poo” 💩 emoji in Unicode:

11111010010101001

Is this fair?

Variable Length Bit String Encoding

- To allow us to use variable bit-lengths, we need to know where one encoded value stops and the next begins.

- As long as we can ensure that *no encoding is a prefix of another*, then we can encode and decode these values safely.

- How can we ensure that property?

Huffman Tree

- One way to do this is by constructing a **binary tree**, and placing the characters on the leaves.
- The encoding is the path from root to leaf, and this means each character has a unique path with no other character as a prefix.
- This method is known as the **Huffman Tree.**

Huffman Tree Algorithm

- Start with the leaves containing the frequencies.

- While more than one node left do
  - Combine the two lowest frequency nodes into new subtree.
  - Remove those two nodes from consideration.
  - Add the root of the subtree with the combined frequency.

Example

Consider the text to encode:

CABAEACABCDEAEEFCEFF

Analyse the frequencies of each character:

Character	**A**	**B**	**C**	**D**	**E**	**F**
Frequency	**5**	**2**	**4**	**1**	**5**	**3**

Clearly A and E should have shortest encoding

Example

Method: Build the tree from the bottom up

- Find 2 rows (sub-trees) with lowest frequency and join

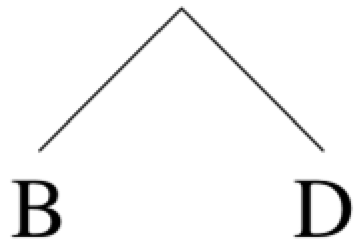
Character	Count
A	5
B	2
C	4
D	1
E	5
F	3

Example

Method: Build the tree from the bottom up

- Find 2 rows (sub-trees) with lowest frequency and join

Character	Count
A	5
B + D	3
C	4
E	5
F	3

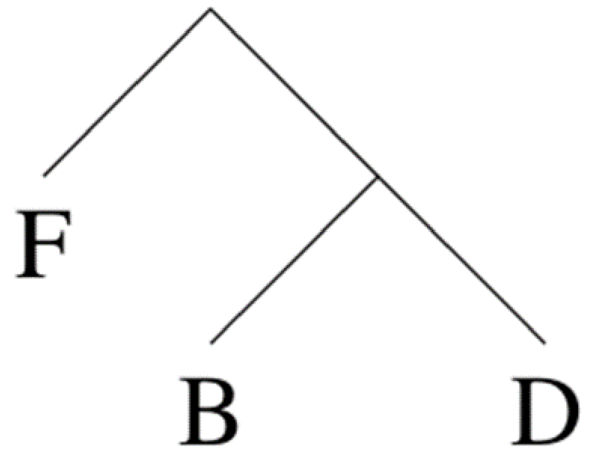


Example

Method: Build the tree from the bottom up

- Find 2 rows (sub-trees) with lowest frequency and join

Character	Count
A	5
F + (B + D)	6
C	4
E	5

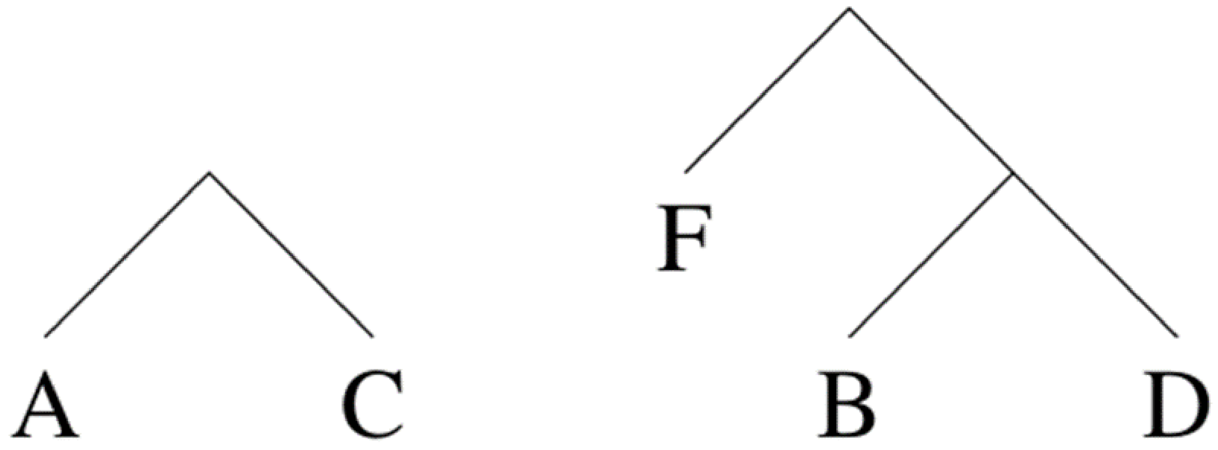


Example

Method: Build the tree from the bottom up

- Find 2 rows (sub-trees) with lowest frequency and join

Character	Count
A + C	9
F + (B + D)	6
E	5

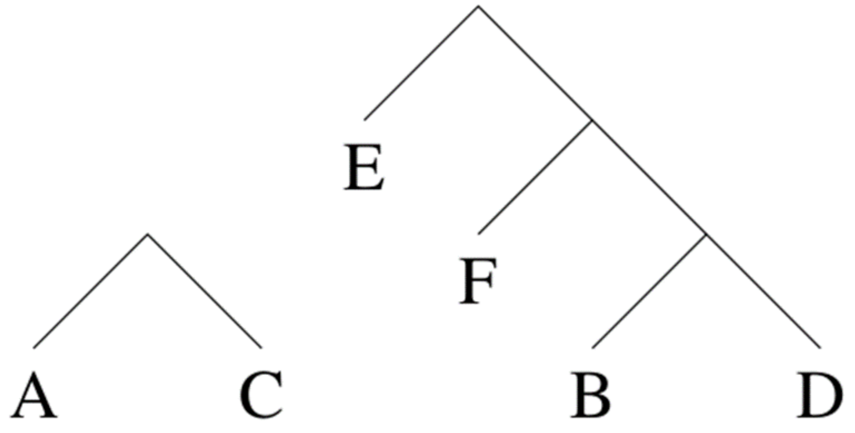


Example

Method: Build the tree from the bottom up

- Find 2 rows (sub-trees) with lowest frequency and join

Character	Count
A + C	9
E+ (F + (B + D))	11

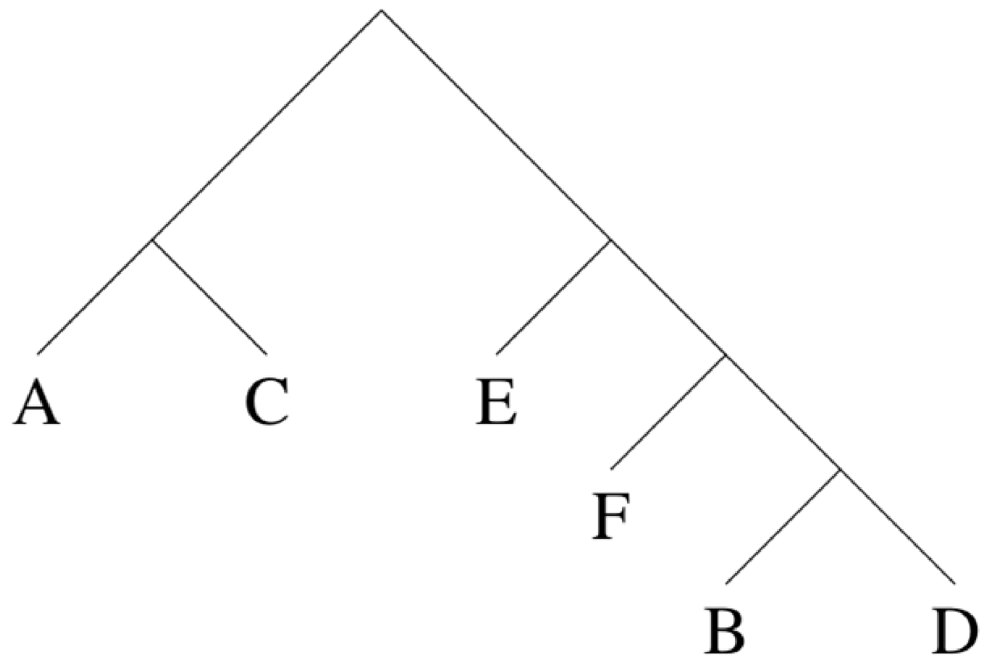


Example

Method: Build the tree from the bottom up

- Find 2 rows (sub-trees) with lowest frequency and join

Character	Count
(A + C) + (E+ (F + (B + D)))	20

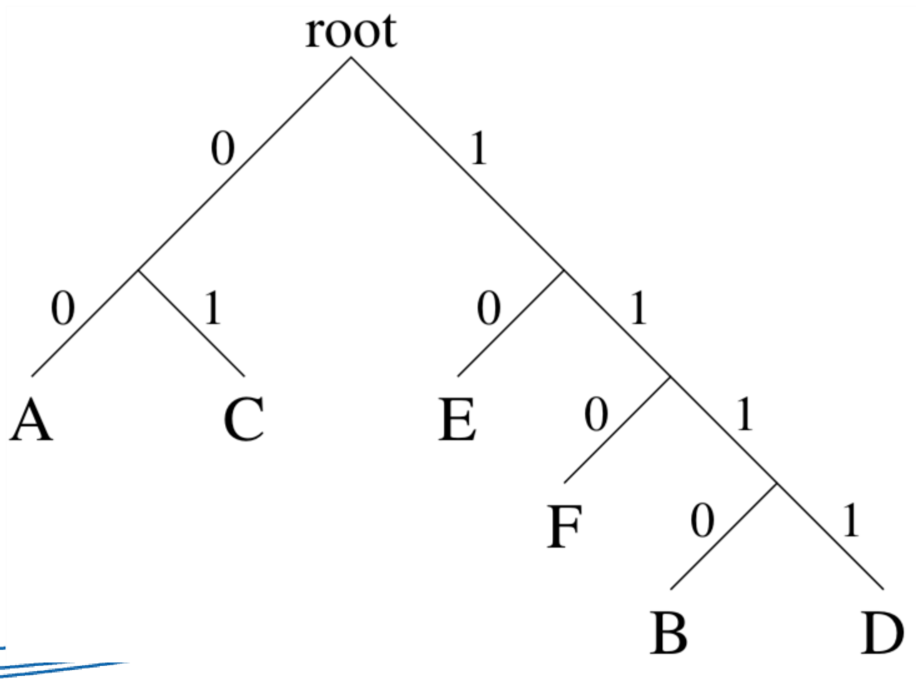


Example

Method:

- Now label **left** branches with 0 and **right** branches with 1

Character	Count
(A + C) + (E+ (F + (B + D)))	20

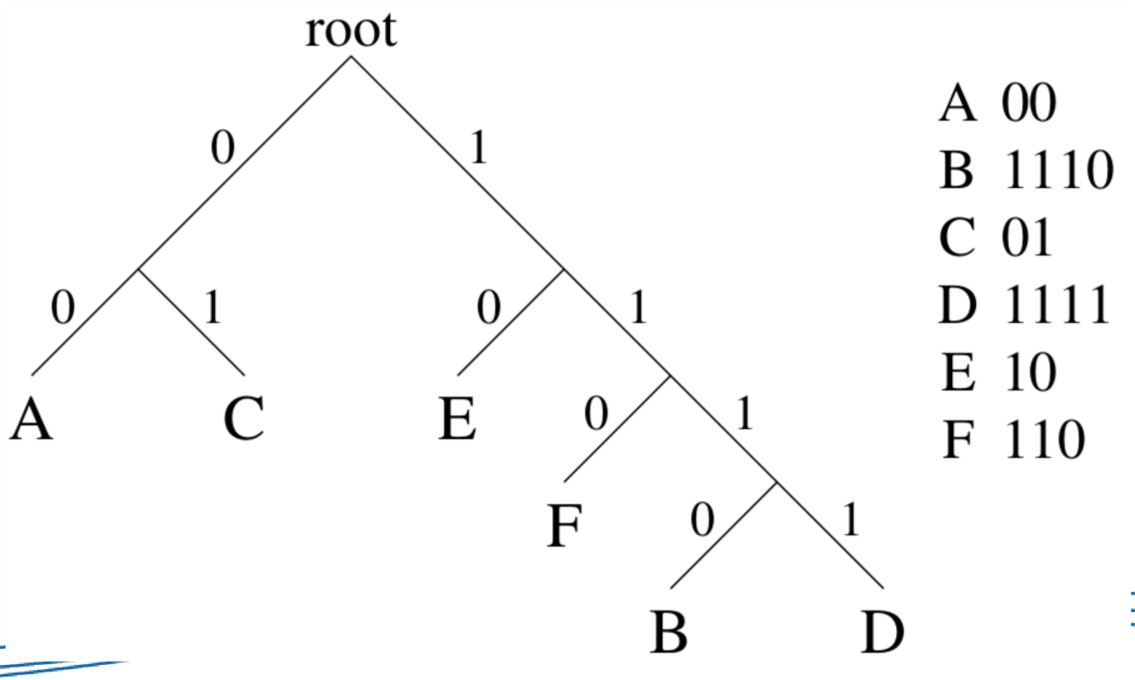


Example

Method:

- **Path from root to leaf** gives bit string encoding for a given character

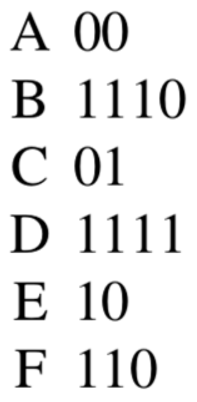
Character	Count
(A + C) + (E+ (F + (B + D)))	20



Example

Method:

- Generate the encoded message by reading left to right, replacing the character with its mapped bit-string:



A	00
B	1110
C	01
D	1111
E	10
F	110

CABAEACABCDEAEEFCEFF

becomes

0100111000100001001110011111100010101100110110110

Variable vs Fixed Length

- The encoded message may look longer than the original.

- But remember that each character in the original message was represented with a fixed-length bit string.
  - E.g. If it was Extended-ASCII then it was 8 bits per character. Now it is less than 5 per character.

- The original message has 20 * 8 bits and the compressed has 49 bits. The compression rate is 49/160 = ~0.30 ☺
- Even better if it was in UTF-16: 49/320 = ~0.15!

Example

Decode:

- Read from left to right, checking for matches
- Swap match with mapped character
- Resume reading

A	00
B	1110
C	01
D	1111
E	10
F	110

0100111000100001001110011111100010101100110110110

Example

Decode:

- Read from left to right, checking for matches
- Swap match with mapped character
- Resume reading

A	00
B	1110
C	01
D	1111
E	10
F	110

0100111000100001001110011111100010101100110110110

Example

Decode:

- Read from left to right, checking for matches
- Swap match with mapped character
- Resume reading

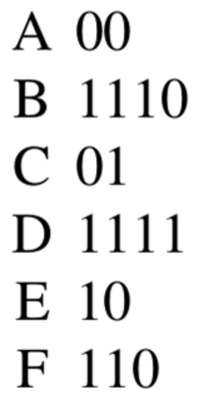
A	00
B	1110
C	01
D	1111
E	10
F	110

0100111000100001001110011111100010101100110110110

Example

Decode:

- Read from left to right, checking for matches
- Swap match with mapped character
- Resume reading



A	00
B	1110
C	01
D	1111
E	10
F	110

<u>C</u>00111000100001001110011111100010101100110110110

Example

Decode:

- Read from left to right, checking for matches
- Swap match with mapped character
- Resume reading

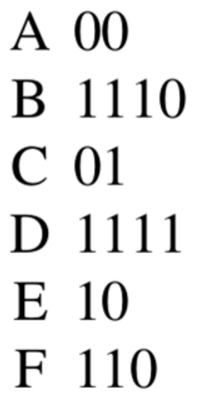
A	00
B	1110
C	01
D	1111
E	10
F	110

C00111000100001001110011111100010101100110110110

Example

Decode:

- Read from left to right, checking for matches
- Swap match with mapped character
- Resume reading



A	00
B	1110
C	01
D	1111
E	10
F	110

C00111000100001001110011111100010101100110110110

Example

Decode:

- Read from left to right, checking for matches
- Swap match with mapped character
- Resume reading

A	00
B	1110
C	01
D	1111
E	10
F	110

C<u>A</u>11100010000100111001111100010101100110110110

Example

Decode:

- Read from left to right, checking for matches
- Swap match with mapped character
- Resume reading

A	00
B	1110
C	01
D	1111
E	10
F	110

CA111000100001001110011111100010101100110110110

Example

Decode:

- Read from left to right, checking for matches
- Swap match with mapped character
- Resume reading

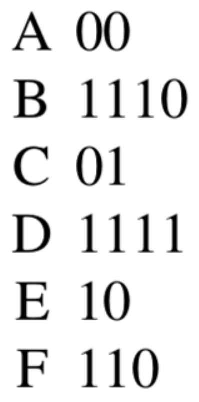
A	00
B	1110
C	01
D	1111
E	10
F	110

CA111000100001001110011111100010101100110110110

Example

Decode:

- Read from left to right, checking for matches
- Swap match with mapped character
- Resume reading



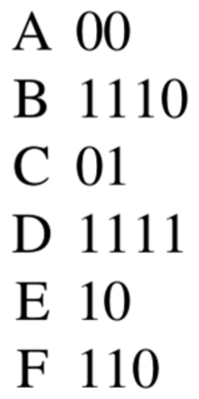
A	00
B	1110
C	01
D	1111
E	10
F	110

CA111000100001001110011111100010101100110110110

Example

Decode:

- Read from left to right, checking for matches
- Swap match with mapped character
- Resume reading



A	00
B	1110
C	01
D	1111
E	10
F	110

CA$\underline{\mathbf{1110}}$**000100001001110011111100010101100110110110**

Example

Decode:

- Read from left to right, checking for matches
- Swap match with mapped character
- Resume reading

A	00
B	1110
C	01
D	1111
E	10
F	110

CAB00100001001110011111100010101100110110110