COMP1521 25T3 — ~~UNICODE~~ Text Encoding (Including UNICODE)

https://www.cse.unsw.edu.au/~cs1521/25T3/

Zeroth: Attribution

The inspiration for this lecture is:
There's No Such Thing As Plain Text by Dylan Beattie

Firstly: Some Revision

- Positive integers are represented in raw binary
  - eg 100_{10} = `1100100`$_2$
  - eg 364_{10} = `101101100`$_2$
- Positive and Negative integers are represented in 2's complement
  - eg 100_{10} = `00000000000000000000000001100100`$_2$
  - eg -745_{10} = `11111111111111111111110100010111`$_2$
- Floating point numbers are represented in IEEE 754
  - eg 3.14159_{10} = `01000000010010010000111111010000`$_2$

What's missing?

text!

Arguably, text is the most important data type in computing

- convert objects to text is called serialization
  - JSON, XML, YAML, etc

Strings are a sequence of characters
So we need is a way to encode characters
Once we can encode characters, we can use an array to represent a string

- This is how we represent text in C
- Other languages use more complex structures than arrays, but the basics are the same
- Modern computers use UNICODE to represent text but how did we get here

Secondly: A Western-centric Timeline

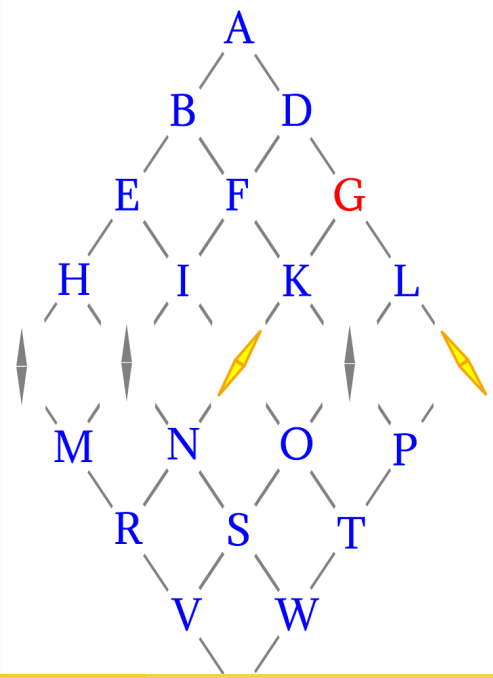
- 1828: First electronic Telegraph system (Pavel Schilling)
- 1837: Cooke and Wheatstone Telegraph
- 1844: Morse Code
- 1897: First radio transmission
- *many other encoding schemes that we won't cover*
- 1943: First (modern) computer (Colossus)
- 1963: ASCII
- 1970s: Extended ASCII
- 1963: EBCDIC
- 1987: Unicode

Secondly: The Timeline (disclaimer)

Note, this timeline (and lecture) is very western-centric
There are many *many* other encoding schemes that we won't cover

East Asian languages specifically have very interesting encoding schemes
as they have a very different way of representing language as text
resulting in huge alphabet sizes

Cooke and Wheatstone Telegraph: 1830s



Cooke and Wheatstone Telegraph: The Good and The Bad

- Only has 20 possible characters
  - The letters C, J, Q, U, X and Z are omitted
- Five needles were used to represent the 20 characters
  - One needle was deflected left and one right,
    the intersection of the two needles represented the character
- Technical limitations required this represent
  - The entire system formed a single circuit
  - one needle represented the positive voltage the other the negative
- Each needle needed it's own wire
  - connecting 2 telegraphs 1 km apart requires 5 km of wire
- Later on a sixth needle was added as a common ground
  allowing only one needle (plus the ground) to be deflected
  - This was used to encode digits 0-9
- As wires were expensive, when a wire was broken,
  instead of replacing the wire, a new encoding was used that needed less wires

Cooke and Wheatstone Telegraph: An Example

The Cooke and Wheatstone Telegraph can be represented as a 5-trit ternary encoding - a trit is the base 3 equivalent of a bit

`"Hello"` would be:

Letter	Nedles	Ternary	Decimal
H	+-...	12000_3	135_{10}
E	+.-..	10200_3	99_{10}
L	...+-	00012_3	5_{10}
L	...+-	00012_3	5_{10}
O	..-+.	00210_3	21_{10}

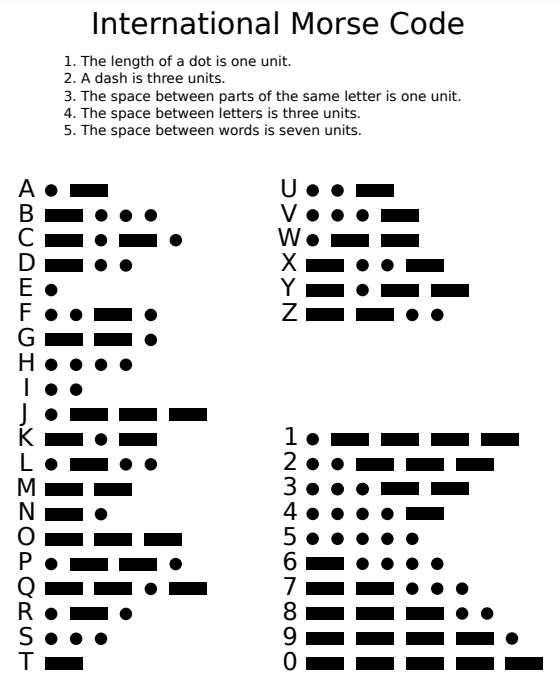
A 5-trit ternary encoding has a maximum of 3^5 = 243 possible values
But only 20 are used, giving a 8.23% efficiency

Cooke and Wheatstone Telegraph: The lesson

Character encoding is done by a lookup table

- Not a mathematical expression (like 2's complement or IEEE 754)

Morse Code: 1844



Morse Code: An Example

Can be represented as binary encoding:

`"Hello"` would be:

Letter	Morse Code	Binary	Decimal
H	....	0000_2	0_{10}
E	.	0_2	0_{10}
L	.-..	0100_2	4_{10}
L	.-..	0100_2	4_{10}
O	—	111_2	7_{10}

But both *H* and *E* are represented by the same decimal value **0**?

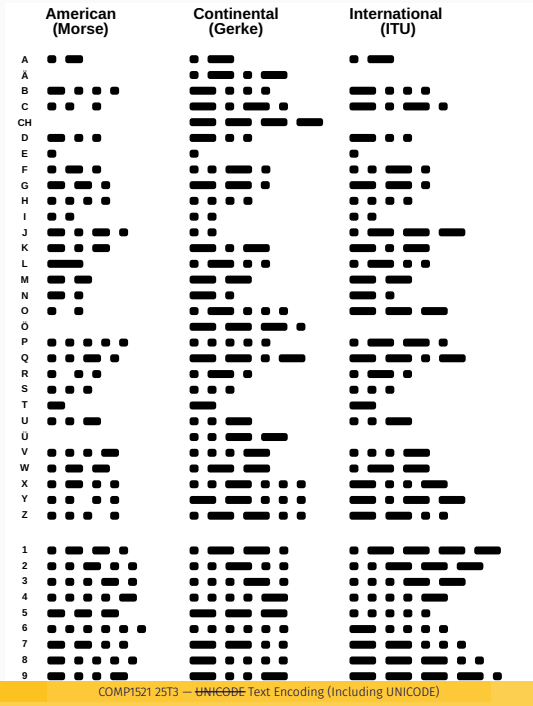
Morse Code is a variable length encoding
So one 0 and four 0s are different values

This makes it difficult to represent in anything other than binary

Morse Code: The Good and The Bad

- Morse Code isn't actually binary because it has a *time* component
  - Unlike Cooke and Wheatstone Telegraph where all 5 trits were sent at the same time Morse Code sends each dot and dash sequentially
  - A dot and a dash are both 1 values, but they are different lengths
  - This means that Morse Code can't be represented as a binary encoding
- Morse Code has many versions
  - International Morse Code was standardized in 1848
  - Hasn't changed since it was standardized
  - Allows for memorization of the encoding
    - Makes it easy to learn
    - Makes it very fast to use by a trained operator
  - But makes it very hard to change and/or improve
- Morse Code is a variable length encoding
  - More length of each letter's encoding is proportional to the frequency of the letter
    - *E* is the most common letter in English, so it has the shortest encoding
    - *Q* is the least common letter in English, so it has the longest encoding

Morse Code: Other Encodings



Morse Code: The lesson

Standardization is good

- It allows for communication between different people in different places

Variable length encodings are very efficient

- Both in terms of the amount of data needed to represent a character and the amount of time needed to send the data
  - Variable length encoding allowed (allows) for experts to send messages at very higher speeds
- But they are more complex

ASCII: 1963

USASCII code chart

b_7 b_6 b_5 →					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
Bits b_4	b_3	b_2	b_1	Column → / Row ↓	0	1	2	3	4	5	6	7
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	~

ASCII

`American Standard Code for Information Interchange`
created by the `American Standards Association` (ASA)
who later became the `American National Standards Institute` (ANSI)
(who where the first organization to standardize the C programming language)

- 7-bit (fixed-size) encoding
  - 128 possible values
    - all of the values are used

One of the most common encodings in computing

- One of the most influential encodings in computing

ASCII: Layout

ASCII is split into “sticks” which were blocks of 16 characters

- the first 2 sticks are control characters
- the space character is the first character in the 3rd stick
  - as it is both a control character and a printable character
  - plus this made sorting stings by ASCII value much more intuitive
- for similar reasons, the next several characters are commonly used as “word separators”
- the 2-5 sticks are a usable alphabet by themselves

ASCII: Layout (cont.)

- The digits per placed in such a way that there value is `0b011` followed by the digits value in binary
  - This allows for very fast conversion between ASCII characters and binary numbers
- The Uppercase and Lowercase letters are placed in such a way that the only difference between them is the 6th bit
  - This allows for very fast case conversion and case insensitive string comparison

ASCII: Layout (what could have been)

- `<` and `>` 60 and 62, so `< + 2 = >`
- `[` and `]` 91 and 93, so `[ + 2 = ]`
- `{` and `}` 123 and 125, so `{ + 2 = }`
- `(` and `)` 40 and 41, so `( + 2 = *`

WHY!?!

ASCII: Control Characters

When ASCII was created, computers didn't have a keyboard and monitors.

Computers had teletype machines, which was a typewriter like device that could be controlled by a human (for input) or a computer (for output)

Becase they were physical devices, they had to be physically controlled thus the control characters.

ASCII: TTY

ASCII: DEL

Figure 7: punch cards

ASCII: DEL (cont.)

Punch cards were used to store data on computers (a very long time ago)

The problem was that storing data on punch cards made a physical change to the card

So deleting data from a punch card was not possible

Instead, the data was replaced with a special character called `DEL`
`DEL` is encoded as `0b0111111` (127) (all bits set)

So on a punch card, `DEL` would be represented as a hole in all 7 columns

Thus making what was previously stored on the card unrecognizable

ASCII: ^C

On a modern computer, `^C` is used to send a `SIGINT` signal to the current process

This effectively stops the current process (if it is well behaved)

But why is it `^C`?

Because on a teletype machine, `^C` is how you would input the 4th control character
`^@`, `^A`, `^B`, `^C` (this makes sense on a typewriter where the keys are in alphabetical order)

What is the 4th control character?

`ETX` (End of Text)

This tells the teletype machine to stop receiving data

Thus ending the current process

Extended ASCII (Code Pages)

ASCII works well for English with exception, e.g if you want a British pound sign (£)

It almost works for some Europen languages: Spanish, French, German, ... and doesn't work at all for most languages.

The solution for languages were ASCII almost works: was to use the 8th bit to extend the encoding.

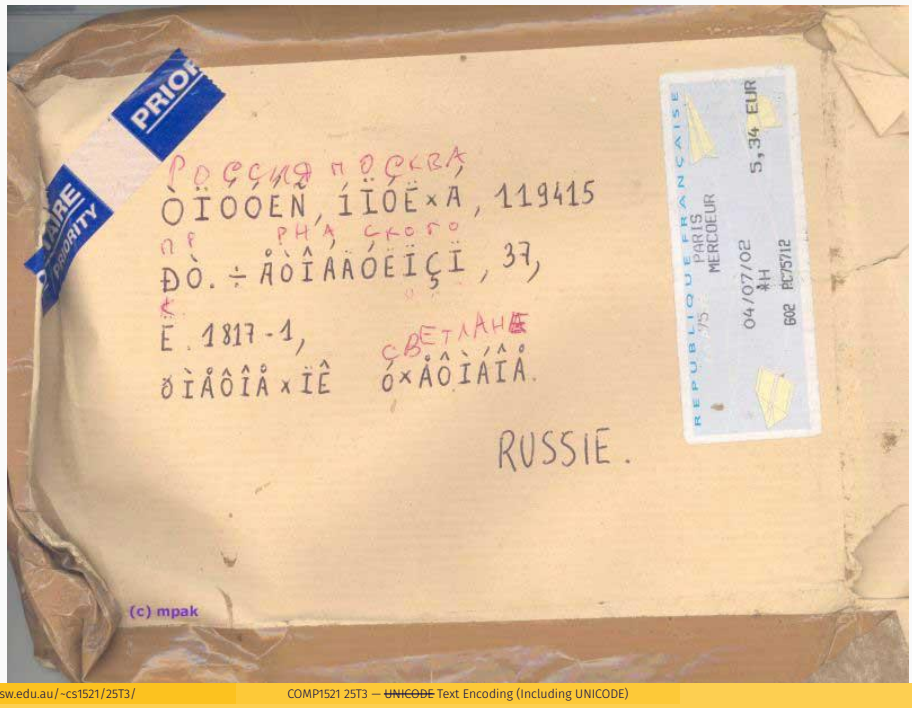
Extended ASCII

The issue with EASCII is that it is not standardized, there are many different encodings
that are all in the category of "Extended ASCII"

KOI-8: Russian encoding ISO 8859-1 (aka Latin-1): Western European encoding Code page 899: DOS mathematical symbols etc...

(wikipedia lists 100s of different Code Pages)

The Mojibake Story



The Mojibake Story

- a Russian student emailed her French friend asking for a Harry Potter book
- her French friend posted the book to the address in the email
- but her e-mail client did not understand the encoding of the Russian (Cyrillic) characters
- coders called garbled (incorrect) decoding of characters: “Mojibake”
- miraculously the Russian postal service figured out the Mojibake

EBCDIC

EBCDIC

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x	NUL	SOH	STX	ETX	SEL	HT	RNL	DEL	GE	SPS	RPT	VT	FF	CR	SO	SI
1x	DLE	DC1	DC2	DC3	RES/ENP	NL	BS	POC	CAN	EM	UBS	CU1	IFS	IGS	IRS	IUS/ITB
2x	DS	SOS	FS	WUS	BYP/INP	LF	ETB	ESC	SA	SFE	SM/SW	CSP	MFA	ENQ	ACK	BEL
3x			SYN	IR	PP	TRN	NBS	EOT	SBS	IT	RFF	CU3	DC4	NAK		SUB
4x	SP										¢	.	<	(	+	|
5x	&										!	$	*	)	;	¬
6x	-	/									¦	,	%	_	>	?
7x										`	:	#	@	'	=	"
8x		a	b	c	d	e	f	g	h	i						±
9x		j	k	l	m	n	o	p	q	r						
Ax		~	s	t	u	v	w	x	y	z						
Bx	^										[	]				
Cx	{	A	B	C	D	E	F	G	H	I						
Dx	}	J	K	L	M	N	O	P	Q	R						
Ex	\		S	T	U	V	W	X	Y	Z						

EBCDIC

- alternative 6-bit encoding developed by IBM
- widely used for decades on IBM's very successful mainframes
- use slowly disappearing

UNICODE

UNICODE is maintained by the `Unicode Consortium`
The goal of UNICODE is to create a single encoding that can represent all of the characters in all of the languages in the world.

There are currently 149,251 characters in UNICODE

UNICODE: Layout

Because UNICODE is so large, it has a very structured layout to try and make it more intuitive

The Unicode Standard defines a *codespace*, (ie "The encoding")

The Unicode codespace ranges from `0x0000` to `0x10FFFF`

Where each hex value represents a *code point* (ie a character)

giving a total of 1,114,112 code points, (293,168 are currently assigned)
approximately 25%

UNICODE: Layout (cont.)

These 1.1 million code points are split into 17 *planes*

The zeroth plane 0x0000 - 0xFFFF is the *Basic Multilingual Plane* (BMP)
which contains the vast majority of characters for almost all modern languages

The first plane 0x10000 - 0x1FFFF is the *Supplementary Multilingual Plane* (SMP)

The second plane 0x20000 - 0x2FFFF is the *Supplementary Ideographic Plane* (SIP)

The third plane 0x30000 - 0x3FFFF is the *Tertiary Ideographic Plane* (TIP)

The fourth through 13th planes 0x40000 - 0xDFFFF are *Unassigned Planes*

The 14th plane 0xE0000 - 0xEFFFF is the *Supplementary Special-purpose Plane* (SSP)

And the last two planes 0xF0000 - 0x10FFFF are the *Private Use Planes* (SPUA-A/B)
which are used for private use thus they are assigned but not to any specific character

UNICODE: Layout (cont.) (cont.)

Within each plane, the code points are split into *blocks*

Blocks are not a standard size, but are always multiples of 16 and usually multiples of 128

Blocks are used to roughly group characters by their purpose

The first plane contains the following blocks:

- Basic Latin (0x0000 - 0x007F)
- Latin-1 Supplement (0x0080 - 0x00FF)
- Latin Extended-A (0x0100 - 0x017F)
- Latin Extended-B (0x0180 - 0x024F)
- ...
- Greek and Coptic (0x0370 - 0x03FF)
- ...
- Mongolian (0x1800 - 0x18AF)
- ...
- Symbols and Punctuation (0x2000 - 0x206F)
-

UNICODE: Layout (cont.) (cont.) (cont.)

The second plane mostly contains historical characters
as well as notational symbols

- Hieroglyphs ?
- musical symbols ?
- Emoji ?

The third plane is almost entirely used by the CJK characters

The fourth plane is mostly unused but contains additional CJK characters

The 14th plane contains a few misc characters

UNICODE: Layout (cont.) (cont.) (cont.) (cont.)

Every UNICODE character also has a major and minor category

The major category is one of the following:

- Letter
- Mark
- Number
- Punctuation
- Symbol
- Separator
- Other

And the minor category depending on the major category.

The largest category is `Letter - other` which contains 131,612 out of 149,251 characters almost 90%

- This is essentially because all of the CJK characters are in this category and there are just so many of them compared to any other category

UTF-32

How do we store UNICODE characters?

The easiest way is to use the smallest power of 2 that can represent all of the code points in UNICODE.

As the code points range from `0x0000` to `0x10FFFF` we need at least 21 bits to represent them.

So we can use 32 bits to represent a single character.

UTF-32 is a fixed width encoding that uses 32 bits to represent each character.

Simply take the UNICODE code point and store it in 32 bits.

UTF-32: Example

A -> U+0041 -> 0b00000000000000000000000001000001

€ -> U+20AC -> 0b00000000000000000010000010101100

? -> U+5B57 -> 0b00000000000000000101101101010111

? -> U+1F600 -> 0b00000000000000011111011000000000

[TAG DIGIT TWO] -> U+E0032 -> 0b00000000000011100000000000110010

U+XXXX is the representation of a raw UNICODE code point
code points are always at least 4 hex digits.

The 5th digit is the plane number
or the 0th plane (BMP) if there is no 5th digit

UTF-32: is very very inefficient

even if we are representing the character U+10FFFF (the largest code point)
there would still be 11 wasted bits

And the vast majority of characters used are in the 0th plane (BMP)
only using 16 bits to represent them, giving 16 wasted bits per character

And the vast majority of characters used in the BMP are in the first block (ASCII)
using only 7 bits to represent them giving 25 wasted bits per character

UTF-8

Taking a lesson from morse code, we can use a variable width encoding
the more common the character, the less bits we need to represent it.

Unicode already puts the most common characters near the beginning
so the goal of UTF-8 is to store the fewest number of leading 0s

UTF-8 (cont.)

This is the layout of UTF-8

#bytes	*#bits*	*Byte 1*	*Byte 2*	*Byte 3*	*Byte 4*
1	7	`0xxxxxxx`	-	-	-
2	11	`110xxxxx`	`10xxxxxx`	-	-
3	16	`1110xxxx`	`10xxxxxx`	`10xxxxxx`	-
4	21	`11110xxx`	`10xxxxxx`	`10xxxxxx`	`10xxxxxx`

A single UTF-8 character can be anywhere from 1 to 4 bytes long

The entire ASCII character set can be represented in 1 byte with zero wasted bits

The entire BMP can be represented in 3 bytes, being 8 bits more efficient than UTF-32

and the entire UNICODE character set can be represented in 4 bytes/ using exactly the same number of bits as UTF-32 in the worst case

UTF-8: Example

A -> U+0041 -> 0b01000001 -> 0x41

€ -> U+20AC -> 0b10000010101100 -> 0b10 000010 101100 -> 0b11100010 10000010 10101100 -> 0xE282AC

? -> U+5B57 -> 0b101101101010111 -> 0b101 101101 010111 -> 0b11100101 10101101 10010111 -> 0xE5AD97

? -> U+1F600 -> 0b111110110000000000 -> 0b 11111 011000 000000 -> 0b11110000 10011111 10011000 10000000 -> 0xF09F9880

UTF-8: Example (cont.)

[TAG DIGIT TWO]

U+E0032

to **UTF-32**

UTF-32 is just a raw 32 bit representation of the code point

```
0xE0032

0b00000000000011100000000000110010
```


UTF-8: Example (cont.) (cont.)

to **UTF-8**

remove leading 0s from the UTF-32 encoding

```
0b 11100000000000110010
```

split into 6 bit chunks from right to left

```
0b 11 100000 000000 110010
```

match with the appropriate multi-byte encoding (in this case there are 4 chunks)

```
0b 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
0b       11   100000   000000   110010
```

replace the **x** with the appropriate bits

```
0b 11110011 10100000 10000000 10110010
```

translate to hex

```
0b 1111 0011 1010 0000 1000 0000 1011 0010
```

UTF-16

UTF-16 is a variable width encoding that uses 16 bits to represent each character.

It's a strange hybrid of UTF-8 and UTF-32

Part of the BMP is reserved for "Surrogates"

Surrogates are used by UTF-16 to represent characters outside of the BMP

UTF-16 is mainly used by Windows and Java and Javascript

UTF-16 also requires a "BOM" (Byte Order Mark) to determine the endianness

UTF-1, UTF-7, UTF-EBCDIC

Lesser used encodings

Writing C Code that uses UNICODE

DEMO