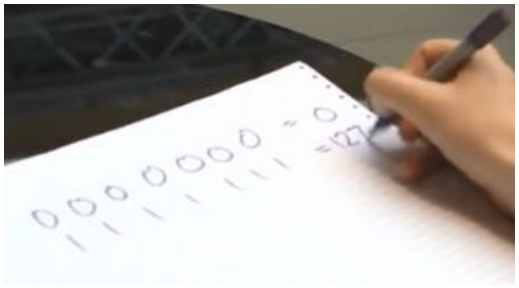
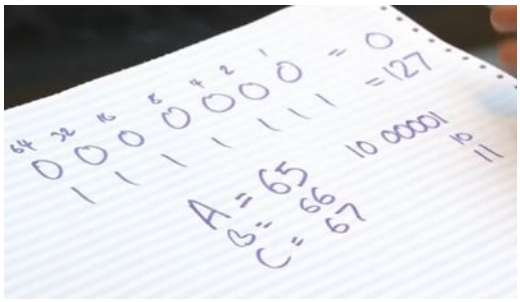
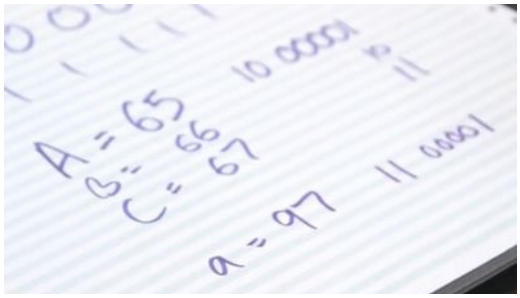
Characters, symbols and the Unicode "miracle" [1]







1. **UTF-8** is perhaps the best hack, the best single thing that is used that can be written down on the back of a napkin and that's how it was put together, it was… The first draft of UTF-8 was written on the back of a napkin in a diner, and it's just such an elegant hack that solves so many problems that I absolutely love it.

   Back in the 1960's, we had teleprinters, we had simple devices: you type a key and it sends some numbers and the same letter comes out the other side. But there needs to be a standard, so in the mid 1960's America at least settled on **ASCII**, which is the ***American standard code for information interchange*** and it's a 7-bit binary system.

2. So each letter you type in gets converted into 7 binary numbers and sent over the wire. Now that means you can have numbers from **0** to **127**. They sort of moved the first 32 for control codes and less important stuff for writing, things like "go down a line" or "backspace", and then they made the rest characters, they added some numbers, some punctuation marks.

   They did a really clever thing, which is that:

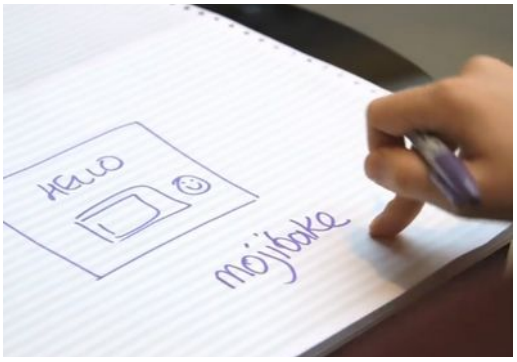
   - they made '**A**': **65** which, in binary – by 1, 2, 4, 8, 16, 32, 64 in binary – 65 is `1000001`;
   - which means that '**B**' is **66**, which means you've got '2' in binary just here;
   - '**C**': **67** – '3' in binary.

3. So you can look at a 7-bit binary character and just knock off first two digits and know what its position in the alphabet is. Even cleverer than that, they started lowercase 32 later, which means that lowercase '**a**' is **97** – `1100001` – anything which doesn't fit into that is probably a *space* which, conveniently, will be 'all zeros', or some kind of punctuation mark. Brilliant, cleaver, wonderful, great way of doing things! And that became the standard at least in the English-speaking world.

4. As ***for the rest of the world***, a few of them did versions of that, but you start getting into other alphabets, into languages that don't really use alphabets at all. They all came up with their own encoding, which is fine… And then along came computers and over time, things changed: we moved to 8-bit computers, so we now have a whole extra number at the start – just to confuse matters – which means we can go to 256! We can have twice as many characters, and of course everyone settled on the same standard for this as that would make perfect sense... No, none of them did!

1 https://www.youtube.com/watch?v=MijmeoH9LT4

- All the Nordic countries started putting Norwegian characters and Finnish characters in there.
- Japan just doesn't use ASCII at all, Japan goes and creates its own multibyte encoding, with more letters and more characters and more binary numbers going to each individual character.



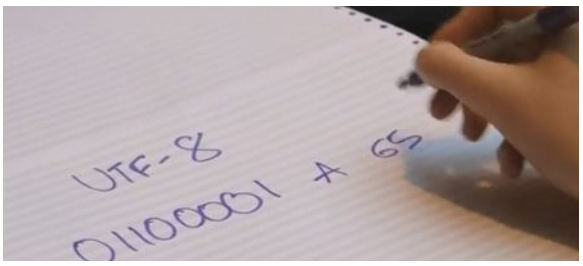
6. All these things are massively incompatible: Japan actually has three or four different encodings, all of which are completely incompatible with each other. So you send a document from one old-school Japanese computer to another: it will come out so garbled that there is even a word in Japanese for "garbled characters" which is – I'm probably mispronouncing this, but it's – "mojibake".

   It's a bit of a nightmare, but it's not bad because, how often does someone in London have to send a document to a completely incompatible unknown computer at another company in Japan? In those days, it's rare: you printed off and you faxed it.

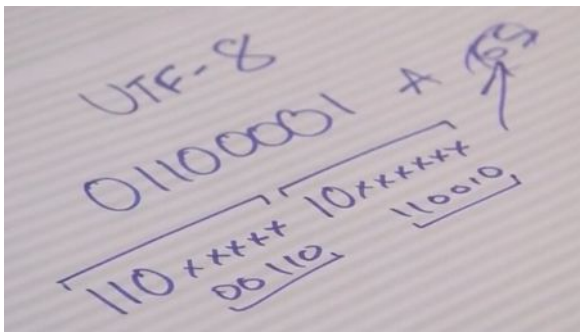
7. And then the *world wide web* hit and we have a problem! Because suddenly, documents are being sent from all around the world, all the time. So a thing is set up called the ***Unicode consortium***. In what I can only describe as a miracle, over the last couple of decades, they have hammered out a standard. *Unicode* now have a list of more than a hundred thousand characters that covers everything you could possibly want to write in any language: English alphabet, Cyrillic alphabet, Arabic alphabet, Japanese, Chinese and Korean characters.
8. What you have at the end is the *Unicode consortium* assigning 100,000+ characters to 100,000 numbers. They have not chosen binary digits, they have not chosen what they should be represented as. All they have said is:
   - that… that Arabic character there, that is number 5700 or something else…
   - and this… this linguistic symbol here, that's 10,000 or something…

   I have to simplify massively here because there are about, of course, five or six incompatible ways to do this. But what the web has more or less settled on is something called **UTF-8**.
9. There are ***a couple of problems*** with doing the obvious thing, which is saying: "OK, we are going to 100,000, that's gonna need, what… to be safe, that's gonna need 32 binary digits to encode it". They encoded the English alphabet in exactly the same way as ASCII did: '**A**' is still **65**. So if you have just a string of English text and you're encoding it at 32 bits per character, you are gonna have about 20-something… 26? Yeah! 26 - 27 zeros and then a few ones, for every single character. That is incredibly wasteful! Suddenly, every English language text file takes four times the space on disc. So:
10. - Problem 1 – You have to get rid of all the zeroes in English text.
    - Problem 2 – There are lots of old computer systems that interpret 8 zeroes in row – a *NULL* – as "this is the end of the string of characters". So if you ever send 8 zeros in a row, they just stop listening. They assume the string has ended there and it gets cut off, so you can't have 8 zeros in a row, anywhere. OK.
    - Problem number 3 – It has to be backwards compatible. You have to be able to take this *Unicode* text and chuck it into something that only understands basic ASCII, and have it more or less work for English text.

11. **UTF-8** solves all these problems and I think it is just a wonderful hack. It starts by just taking ASCII. If you have something under 128 that can just be expressed as 7 digits, you put down a `0` and then you put the same numbers that you would otherwise. So let's have that '**A**' again: there we go, that's still '**A**', that's still **65**, that is still UTF-8 valid and that's still ASCII valid. Brilliant!

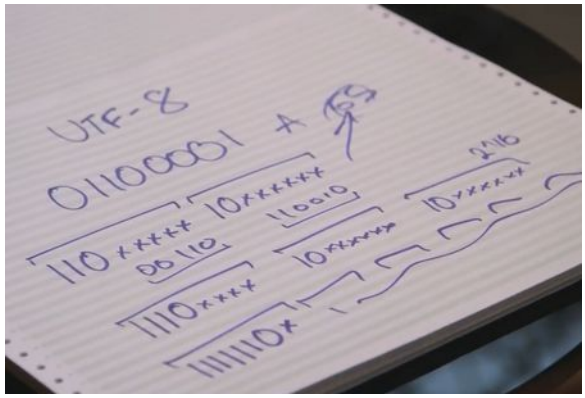


12. OK, now let's say we are going above that. Now, you need something that is going to work more or less for ASCII, or at least not break things, but still be understood.
    - So what you do is you start by writing down `110`. This means 'this is the start of a new character' and this character is going to be two bytes long: two `1`, two bytes – a byte being 8 characters[2].
    - And you say on this one, we're gonna start with `10`, which means 'this is a continuation', and at all these blank spaces – of which you have five here, and six here – you fill in the other numbers.



13. And then when you calculate it, you just take off those headers, and it understands just these as being whatever number that turns out to be (that's probably somewhere in these hundreds).

    That'll do you for the first 4,096[3]. What about above that? Well, ***above that***, you go `1110`, meaning there are three bytes in this – three `1`, three bytes – with two continuation bytes. So now you have 1, 2, 3, 4… 10… 16 spaces! You want to go above that? You can! The specification goes all the way to `1111110x` with this many continuation bytes after it[4].



14. It's a neat hack that you can explain on the back of a napkin or a bit of paper. It's ***backwards-compatible***. It avoids waste. At no point will it ever, ever, ever send 8 zeroes in a row, and really, really crucially, the one that made this win over every other system is that you can move backwards and forwards really easily. You do not have to have an index of where the characters start. If you are halfway through a string and you wanna go back one character, you just look for the previous header.
15. And that's is it, and that works, and as of a few years ago, UTF-8 beat out ASCII and everything else, as for the first time, the dominant character encoding on the web. We don't have that "mojibake" that Japanese has. We have something that nearly works and that is why it is the most beautiful hack that I can think of that is used around the world every second of everyday.

2 A byte is made of 8 bits, not 8 characters.
3 A two-bytes only encode 2048 characters in UTF-8.
4 A character has a maximum of 4 bytes, so the longest prefix of the first byte is `11110` (not `1111110`) – cf. RFC 3629, p. 4.