Unicode

Enter Unicode.

Unicode is a system designed to represent *every* character from every language. Unicode represents each letter, character, or ideograph as a 4-byte number. Each number represents a unique character used in at least one of the world's languages. (Not all the numbers are used, but more than 65535 of them are, so 2 bytes wouldn't be sufficient.) Characters that are used in multiple languages generally have the same number, unless there is a good etymological reason not to. Regardless, there is exactly 1 number per character, and exactly 1 character per number. Every number always means just one thing; there are no "modes" to keep track of. `U+0041` is always `'A'`, even if your language doesn't have an `'A'` in it.

On the face of it, this seems like a great idea. One encoding to rule them all. Multiple languages per document. No more "mode switching" to switch between encodings mid-stream. But right away, the obvious question should leap out at you. Four bytes? For every single character‽ That seems awfully wasteful, especially for languages like English and Spanish, which need less than one byte (256 numbers) to express every possible character. In fact, it's wasteful even for ideograph-based languages (like Chinese), which never need more than two bytes per character.

There is a Unicode encoding that uses four bytes per character. It's called UTF-32, because 32 bits = 4 bytes. UTF-32 is a straightforward encoding; it takes each Unicode character (a 4-byte number) and represents the character with that same number. This has some advantages, the most important being that you can find the `Nth` character of a string in constant time, because the `Nth` character starts at the `4×Nth` byte. It also has several disadvantages, the most obvious being that it takes four freaking bytes to store every freaking character.

Even though there are a lot of Unicode characters, it turns out that most people will never use anything beyond the first 65535. Thus, there is another Unicode encoding, called UTF-16 (because 16 bits = 2 bytes). UTF-16 encodes every character from 0–65535 as two bytes, then uses some dirty hacks if you actually need to represent the rarely-used “astral plane” Unicode characters beyond 65535. Most obvious advantage: UTF-16 is twice as space-efficient as UTF-32, because every character requires only two bytes to store instead of four bytes (except for the ones that don’t). And you can still easily find the Nth character of a string in constant time, if you assume that the string doesn’t include any astral plane characters, which is a good assumption right up until the moment that it’s not.

But there are also non-obvious disadvantages to both UTF-32 and UTF-16. Different computer systems store individual bytes in different ways. That means that the character `U+4E2D` could be stored in UTF-16 as either `4E` `2D` or `2D` `4E`, depending on whether the system is big-endian or little-endian. (For UTF-32, there are even more possible byte orderings.) As long as your documents never leave your computer, you’re safe — different applications on the same computer will all use the same byte order. But the minute you want to transfer documents between systems, perhaps on a world wide web of some sort, you’re going to need a way to indicate which order your bytes are stored. Otherwise, the receiving system has no way of knowing whether the two-byte sequence `4E` `2D` means `U+4E2D` or `U+2D4E`.

To solve *this* problem, the multi-byte Unicode encodings define a “Byte Order Mark,” which is a special non-printable character that you can include at the beginning of your document to indicate what order your bytes are in. For UTF-16, the Byte Order Mark is `U+FEFF`. If you receive a UTF-16 document that starts with the bytes `FF` `FE`, you know the byte ordering is one way; if it starts with `FE` `FF`, you know the byte ordering is reversed.

Still, UTF-16 isn’t exactly ideal, especially if you’re dealing with a lot of `ASCII` characters. If you think about it, even a Chinese web page is going to contain a lot of `ASCII` characters — all the elements and attributes surrounding the printable Chinese characters. Being able to find the `Nth` character in constant time is nice, but there’s still the nagging problem of those astral plane characters, which mean that you can’t *guarantee* that every character is exactly two bytes, so you can’t *really* find the `Nth` character in constant time unless you maintain a separate index. And boy, there sure is a lot of `ASCII` unless you maintain a separate index. And boy, there sure is a lot of `ASCII` text in the world…

Other people pondered these questions, and they came up with a solution:

UTF-8

UTF-8 is a *variable-length* encoding system for Unicode. That is, different characters take up a different number of bytes. For `ASCII` characters (A-Z, &c.) `UTF-8` uses just one byte per character. In fact, it uses the exact same bytes; the first 128 characters (0–127) in `UTF-8` are indistinguishable from `ASCII`. “Extended Latin” characters like ñ and ö end up taking two bytes. (The bytes are not simply the Unicode code point like they would be in UTF-16; there is some serious bit-twiddling involved.) Chinese characters like 中 end up taking three bytes. The rarely-used “astral plane” characters take four bytes.

Disadvantages: because each character can take a different number of bytes, finding the `Nth` character is an O(N) operation — that is, the longer the string, the longer it takes to find a specific character. Also, there is bit-twiddling involved to encode characters into bytes and decode bytes into characters.

Advantages: super-efficient encoding of common `ASCII` characters. No worse than UTF-16 for extended Latin characters. Better than UTF-32 for Chinese characters. Also (and you’ll have to trust me on this, because I’m not going to show you the math), due to the exact nature of the bit twiddling, there are no byte-ordering issues. A document encoded in `UTF-8` uses the exact same stream of bytes on any computer.