Text encoding

We have covered how to represent numbers in binary; in this section we'll explore representations of **text** as bits. By 'text', we mean alphabets and other writing systems — used everywhere from status updates and text messages to email and digital books.

Figure 1: @pigworker on Twitter

Beginnings

To start with, we can propose a way of mapping letters and other characters (punctuation, space, etc.) to numbers. For example, let **A** be represented as the number 0, **B** as 1, **C** as 2, and so on. There are 26 letters in the English alphabet, so **Z** is 25, and we'd need a total of 5 bits. (2^5 is 32, so we'd even have a few numbers left over for punctuation.)

Exercise: using the scheme outlined above, decode the word represented by the bits `00010 00000 10011`

If our text messages need to distinguish between upper- and lower-case letters, we'll need more than 5 bits. Upper-case A–Z is 26 characters, lower-case a–z is another 26, so that's a total of 52. 2^6 is 64, so 6 bits would cover it and again have a few available for punctuation.

But what about including **numbers** in our text? If we want to send the text message "amazon has a 20% discount on textbooks," we can't really represent that '20' as 10100 in binary, because that would conflict with the representation of the letter 'U'.

Instead, we need to add space for the standard ten numerals as characters. Including those with upper- and lower-case letters means we need at least 62 characters. Technically that fits in 6 bits, but we'd have very little room for punctuation and the character representing a space. So for practical purposes, we're up to 7 bits per character. 2^7 is 128, so now there is a good deal of room for other symbols.

As an aside, there could be a way to "reuse" alphabetic representations as numerals. We'd just have to precede them with a marker that means "this is a number," or else require the recipient to guess from context. This is the situation in Braille, a writing

system for the visually impaired that's based on 6-bit characters. (Each of six locations can be raised or not.) The Braille character for 'A' is the same as the number '1'.

Fixed vs. variable-width

The simple encodings I proposed in the previous section are based on a **fixed** number of bits per character — whether it is 5, 6, or 7. One way to illustrate that is as a **tree** — see this file:

- 5-bit fixed encoding (PDF)

Trees are a commonly-used data structure in computer science, but they are a little different than the organic trees to which they refer. First of all, we usually draw trees with the **root** at the top, and they grow down the page. Each time a circle splits into two paths, we call that a **branch.** The tree ends at the bottom with a row of **leaves.**

This particular tree is a **binary tree,** meaning that every **node** is either a leaf, or a branch with exactly two **children.** The nice thing about a binary tree is that paths from root to leaf correspond exactly to binary numbers. Just think of zero as going **left** in the tree, and one as going **right.** Then, the number `01101` (for example) corresponds to left-right-right-left-right, which lands on the leaf marked `N`. Decode the message in binary written beneath the tree.

You can tell the previous tree is fixed-width because **every** path from root to leaf is exactly 5 transitions. Now compare that to a variable-bit tree, in this file:

- Variable-bit Huffman encoding (PDF)

In this case, different letters can have very different numbers of bits representing them. For example, `E` is the shortest path, representing just 3 bits. `X` is a very long path, representing 10 bits. Decode the word given in binary in the upper right of the page.

On the handout, the word is printed with spaces between the letters, but actually they're not necessary. The bits `11100001001` can be decoded even though I haven't emphasized where one character ends and the next begins. You simply follow the path in the tree until you land on a leaf. Then, start again at the top for the next bit.

This particular variable-width tree is crafted so that the overall effect is that it **compresses** English text. This works because more commonly used letters are represented with proportionally shorted bit strings. For example, let's compare the encodings using both trees of a sequence of words:

```
word:     fixed encoding:             variable encoding:
THE       100110011100100 15 bits     11100001001     11 bits
```

```
GRASS     001101000100000             11010000001100
          1001010010      25 bits     01000100        22 bits
IS        0100010010      10 bits     01110100         8 bits
GREEN     001101000100100             1101000000001
          0010001101      25 bits     0010110         20 bits
SAID      100100000001000             010011000111
          00011           20 bits     11011           17 bits
QUUX      100001010010100             1111100001
          10111           20 bits     111111111111
                                      1111100010      32 bits

total:                   115 bits                   110 bits
```

With the fixed encoding, every character is exactly 5 bits, and so the whole sequence of words is 115 bits. (We're not counting encoding the spaces between words for this exercise.)

Contrast that with the variable encoding. Nearly every word has a shorter representation. The one exception is "QUUX", which of course isn't really a word in English. But it represents the case of a word with infrequently-used letters, and the encoding of that one word increased substantially in size from 20 to 32 bits. On the whole, the second tree still compresses as long as you are mostly using English words with high-frequency letters.

- Letter frequency keyboard histogram

ASCII

This brings us to the most popular and influential of the fixed-bit codes. It's called ASCII (pronounced *"ass-key"*), which stands for American Standard Code for Information Interchange. It was developed in the early 1960s, and includes a 7-bit mapping of upper- and lower-case letters, numerals, a variety of symbols, and "control characters." You can see a table of all of them at `http://www.asciitable.com/`

The control characters are in the range 0–31 (base ten). They don't have a visual representation, but instead direct the display device in particular ways. Many of them are now obsolete, but perhaps the most important one is $10_{10} = A_{16} = 0001010_2$, which is the "new line" character. Whenever you press enter to go to the next line, this character is inserted in your document.

The character 32 is a space, and 33-63 hold mostly punctuation. The numerals are at positions 48 through 57. These are easy to recognize in binary: they all start with 011 and then the lower four bits match the numeral. So you can tell at a glance that $0110101_2 = 35_{16}$ is the numeral '5'.

The range 64–95 is mostly uppercase characters, and 96–127 is mostly lowercase. (Both ranges include a few more punctuation characters and brackets.) These num-

bers correspond to bit strings starting with 10 for uppercase and 11 for lowercase. The remaining 5 bits give the position of the letter in the alphabet. So $10\ 01011_2 = 4B_{16}$ is the eleventh letter (uppercase 'K') and $11\ 01011_2 = 6B_{16}$ is the corresponding lowercase 'k'.

Babel

ASCII worked relatively well for the English-speaking world, but other nations and cultures have needs for different symbols, accents, alphabets, and other characters. It's impossible to write **niño** or **café** in ASCII, or the Polish name **Michał,** and it's hopeless for the Greek word **□λήθεια,** or the Chinese **□.**

Computer architectures eventually settled on eight bits as the smallest addressable chunk of memory, known as a **byte.** Since ASCII was 7 bits, it became possible to use that eighth bit to indicate an extra 128 characters.

This led to a wide variety of incompatible 8-bit encodings for various languages. They mostly agreed in being compatible with ASCII for the first 128 characters, but beyond that it was chaos. It's all described in the different parts of this specification:

- ISO 8859 specification

That is, ISO 8859-1 was for Western European languages, 8859-2 for Central European, 8859-4 for North European, 8859-5 for Cyrillic alphabet, 8859-7 for Greek, etc. Sending documents between these language groups was difficult, and it was impossible to create a single document containing multiple languages from incompatible encodings.

As one small example, let's take the character at position $EC_{16} = 236_{10}$. All these encodings disagree about what it should be:

- ISO 8859-1: **ì** — LATIN SMALL LETTER I WITH GRAVE
- ISO 8859-2: **ě** — LATIN SMALL LETTER E WITH CARON
- ISO 8859-4: **ė** — LATIN SMALL LETTER E WITH DOT ABOVE
- ISO 8859-5: **ь** — CYRILLIC SMALL LETTER SOFT SIGN
- ISO 8859-7: **μ** — GREEK SMALL LETTER MU
- Mac OS Roman: **Ï** — LATIN CAPITAL LETTER I WITH DIAERESIS
- IBM PC: **∞** — INFINITY

You can still see the remnants of this old incompatible encoding system in your browser's menu. Most web pages today will be in Unicode — we'll get to that in a moment — but the browser still supports these mostly-obsolete encodings, so it can show you web pages written using them. Notice that even for the same language, there are often several choices of encodings available.

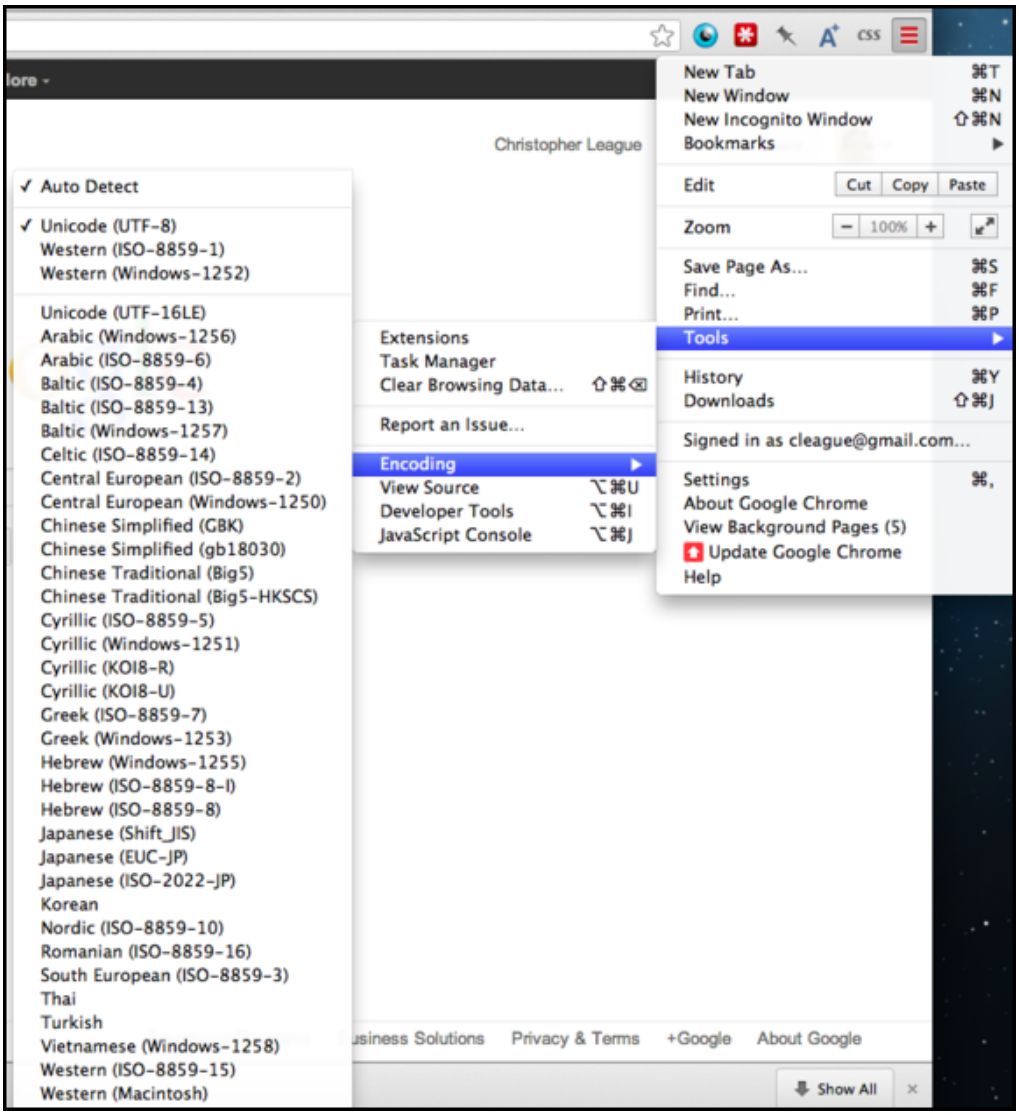
New Tab ⌘T
New Window ⌘N
New Incognito Window ⇧⌘N
Bookmarks
Edit Cut Copy Paste
Zoom − 100% +
Save Page As... ⌘S
Find... ⌘F
Print... ⌘P
Tools
History ⌘Y
Downloads ⇧⌘J
Signed in as cleague@gmail.com...
Settings ⌘,
About Google Chrome
View Background Pages (5)
Update Google Chrome
Help
Extensions
Task Manager
Clear Browsing Data... ⇧⌘⌫
Report an Issue...
Encoding
View Source ⌥⌘U
Developer Tools ⌥⌘I
JavaScript Console ⌥⌘J
Christopher League
✓ Auto Detect
✓ Unicode (UTF-8)
Western (ISO-8859-1)
Western (Windows-1252)
Unicode (UTF-16LE)
Arabic (Windows-1256)
Arabic (ISO-8859-6)
Baltic (ISO-8859-4)
Baltic (ISO-8859-13)
Baltic (Windows-1257)
Celtic (ISO-8859-14)
Central European (ISO-8859-2)
Central European (Windows-1250)
Chinese Simplified (GBK)
Chinese Simplified (gb18030)
Chinese Traditional (Big5)
Chinese Traditional (Big5-HKSCS)
Cyrillic (ISO-8859-5)
Cyrillic (Windows-1251)
Cyrillic (KOI8-R)
Cyrillic (KOI8-U)
Greek (ISO-8859-7)
Greek (Windows-1253)
Hebrew (Windows-1255)
Hebrew (ISO-8859-8-I)
Hebrew (ISO-8859-8)
Japanese (Shift_JIS)
Japanese (EUC-JP)
Japanese (ISO-2022-JP)
Korean
Nordic (ISO-8859-10)
Romanian (ISO-8859-16)
South European (ISO-8859-3)
Thai
Turkish
Vietnamese (Windows-1258)
Western (ISO-8859-15)
Western (Macintosh)
Privacy & Terms
+Google
About Google
Show All

Figure 2:

Unicode

To deal with this problem of incompatible encodings across different language groups, the Unicode Consortium was founded with the amazing and noble goal of developing **one** encoding that would contain **every** character and symbol used in **every** language on the planet.

You can get a sense of the variety and scope of this goal by browsing the code charts on the Unicode web site:

- Code charts

Each one is a PDF file that pertains to a particular region, language, or symbol system. In total, it's close to a hundred thousand characters.

The code charts give a distinct number to every possible character, but there is still the issue of how to encode those numbers as bits. Most of the numbers fit in 16 bits, which is why they are expressed as four hexadecimal digits in the code charts (such as **1F30** for an accented Greek iota: ἰ). But 2^{16} is 65,536 and we said there were closer to 100,000 characters, so obviously 16 bits is not enough. Most of the time Unicode is represented as a multi-byte (variable) encoding called **UTF-8.** The original ASCII characters are still represented as just one byte, but setting the eighth bit enables a clever mechanism that indicates how many bytes follow. Here is a nice explanation of Unicode and UTF-8 by Tom Scott on Computerphile:

Nowadays, Unicode works just about everywhere, and almost all new content uses it. There is still an occasional problem of whether or not your computer has the correct fonts installed that contain all the characters you need. Sometimes you will see a box show up in place of an unsupported character. Here is the same text displayed on three different systems:

nirvāṇa
$\aleph_0 \ll \aleph_1 \leq |2^{\aleph_0}|$
ἐτεῇ δὲ οὐδὲν ἴδμεν · ἐν βυθῷ γὰρ ἡ ἀλήθεια.
Stanisław
☸ Die Thôrin denkt sich schôn in schônen Kleidern seyn.
Хромая судьба
$\forall x \in \mathbb{R}: \exists n \in \mathbb{N}: x < n$

Figure 3:

The one above shows every character perfectly. The one below is missing a few characters.

nirvā□ a
□ 0 ≪ □ 1 □ □2□ₒ□
ἐτεῇ δὲ οὐδὲν ἴδμεν · ἐν βυθῷ γὰρ ἡ ἀλήθεια.
Stanisław
✵ Die Thôrin denkt sich schôn in schônen Kleidern seyn.
Хромая судьба
∀x∈ℝ: ∃n∈ℕ: x < n

Figure 4:

Finally, the system below is unable to display any characters except those in ASCII.

nirv□□ a
□ 0□ □ 1□ □2□ₒ□
□□□□□□ □□ □□□□□ □□□□□ □ □□ □□□□□□ □□□ □ □□□□□□□.
Stanis□aw
□ Die Th□rin denkt sich sch□n in sch□nen Kleidern seyn.
□□□□□□ □□□□□□
□x□□:□n□□: x < n

Figure 5:

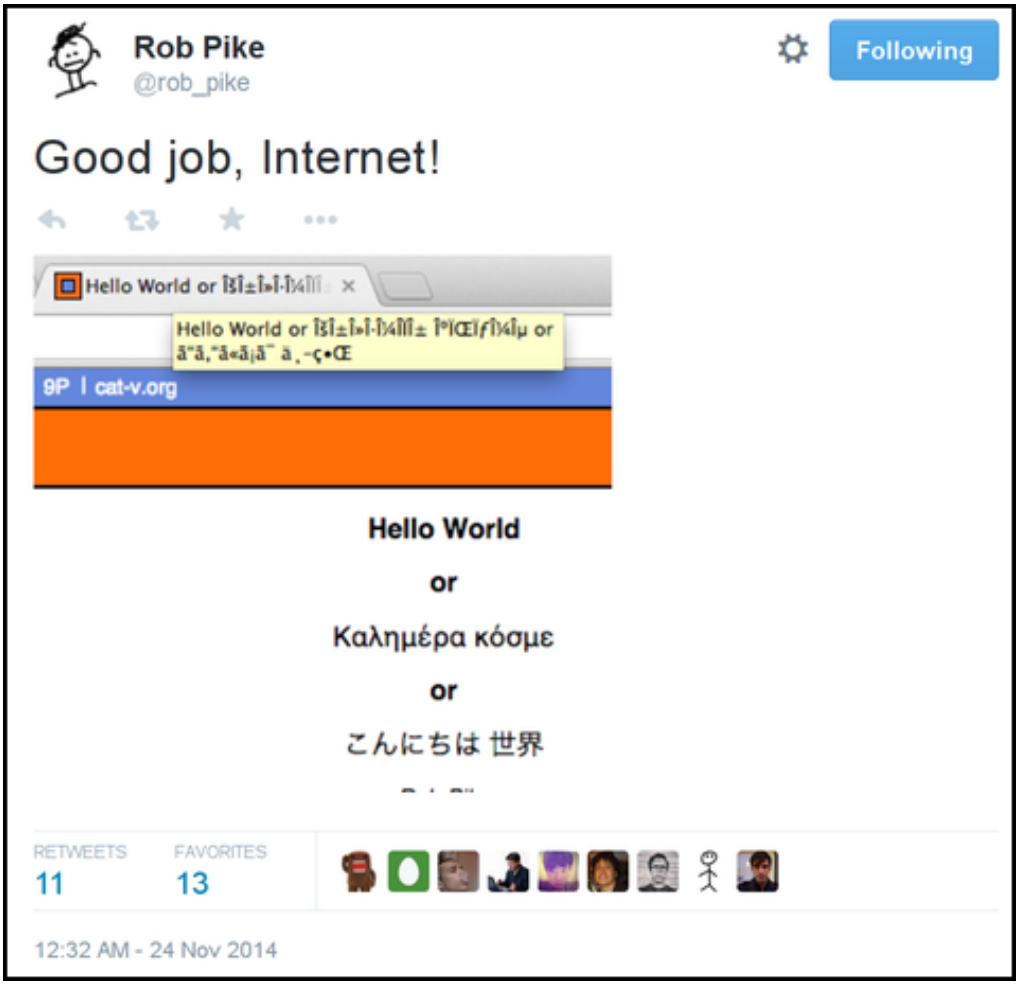

Figure 6: @rob_pike on Twitter