Foundations of Computing

Digital Representation

Ekaterina Vylomova

Summer Term 2023

THE UNIVERSITY OF MELBOURNE

February, 15th: On this day...(random facts)

- **1214** – During the Anglo-French War (1213–1214), an English invasion force led by John, King of England, lands at La Rochelle in France.
- **1870** – Stevens Institute of Technology is founded in New Jersey, US, and offers the first Bachelor of Engineering degree in mechanical engineering.
- **1923** – Greece becomes the last European country to adopt the Gregorian calendar.
- **1946** – ENIAC, the first electronic general-purpose computer, is formally dedicated at the University of Pennsylvania in Philadelphia.
- **2001** – The first draft of the complete human genome is published in Nature.

Reminders

- Worksheets 14–15 are due Feb, 17 (Fri) 11:59pm
- **Project 2 is due Feb, 17 11:59pm**
- **The Final Exam will be Feb, 20th 10am on Grok (online!)**
- Practice exam is available in "Assignments" and past (pre-2020) exams are provided in the "Modules"
- MST results and solutions will be available Feb 15th
- Consultations (Please attend!):
  Links: `https://canvas.lms.unimelb.edu.au/courses/157330/pages/getting-help`

Lecture Agenda

- Last lecture:
  - HTML
  - Introduction to algorithms
  - Properties and families of algorithms
- This lecture:
  - Computational counting
  - Digital representation of text

Lecture Outline

Computational Counting I

- Conventionally, we are used to representing numbers in decimal (base 10) format:

$$\begin{aligned} 2021_{10} &= 2 \times 1000 + 0 \times 100 + 2 \times 10 + 1 \times 1 \\ &= 2 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 1 \times 10^0 \end{aligned}$$

Computational Counting II

- Computers internally represent numbers in binary (base 2) format:

$$
\begin{aligned}
10001_2 &= 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
&= 1 \times 16 + 0 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1 \\
&= 17
\end{aligned}
$$

Computational Counting III

- A single-digit binary number (and the basic unit of storage/computational) is known as a "bit" (short for binary digit)
- Bits are generally processed as vectors of 8 bits (= a "byte" or "octet") or larger
- A convenient representation for bit sequences is "hexadecimal" (base 16); one byte = 8 bits = two "hex" digits (why?)
- The 16 hexadecimal digits:

Hex:	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Dec:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bin:	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111

Computational Counting IV

- Converting from bin(ary) to hex(adecimal):

$$\begin{array}{ccc}
 & 00110110_2 & \\
 & \Downarrow & \\
0011_2 & & 0110_2 \\
\Downarrow & & \Downarrow \\
11_2 & & 110_2 \\
\Downarrow & & \Downarrow \\
3_{16} & & 6_{16} \\
 & \Downarrow & \\
 & 36_{16} &
\end{array}$$

Computational Counting V

- To indicate what “base” a (non-decimal) number is in, Python uses the following prefixes:
  - binary (base 2): `0b`
  - octal (base 8): `0o`
  - hexadecimal (base 16): `0x`

```
>>> 0b11001 == 0o31 == 25 == 0x19
True
```

Computational Counting VI

- It is also possible to convert a (decimal) `int` to a string representation in one of the other bases using `bin`, `oct` and `hex`, resp.:

```
>>> bin(25)
'0b11001'
>>> oct(25)
'0o31'
>>> hex(25)
'0x19'
```

Computational Counting VII

... and to convert from any base to a decimal integer using `int` (over a string argument, with an optional second argument specifying the base):

```
>>> int('11001', 2)
25
>>> int('31', 8)
25
>>> int('19', 16)
25
```

Computational Counting VIII

- NB, for memory and storage, sizes are generally reported in bytes ("B") vs. network speeds which are reported in bits ("b")
- Because of the size/speed of modern-day computers/networks, numbers are usually reported in kilo-, mega-, giga-, etc. units:
  - kilo ("k") $= 10^3 = 1000 \approx 2^{10}$
  - mega ("M") $= 10^6 = 1,000,000 \approx 2^{20}$
  - giga ("G") $= 10^9 = 1,000,000,000 \approx 2^{30}$
  - tera ("T") $= 10^{12} = 1,000,000,000,000 \approx 2^{40}$
  - peta ("P") $= 10^{15} = 1,000,000,000,000,000 \approx 2^{50}$

Lecture Outline

Internal Representation of Characters

- Earlier in the subject, you were introduced to the notion that characters are internally just (positive) integers:

```
>>> ord('a')
97
>>> ord('ö')
246
```

- These values are the “code point” values for each character, as based on the Unicode standard

Unicode I

- Unicode is an attempt to represent all text from all languages (and much more besides) in a single standard
- Unicode is intended to support the electronic rendering of all texts and symbols in the world's languages
- Each **grapheme** is assigned a unique number or **code point**, conventionally represented as a hexidecimal number
- There is scope within unicode for both **precomposed** (e.g. *á*) and **composite** characters/**glyphs** (e.g. $\acute{} + a$)
- For (some) emoji, there is scope to change skin-tone via composite characters

Unicode II

- There are plenty of code points to go around (over 1M), to cater for the “big” orthographies
- With Unicode, different orthographies can happily co-exist in a single document
- The basic philosophy behind Unicode has been to (monotonically) add more code points for different “languages”, starting with the pre-existing encodings

How are Text Documents Represented?

- Text documents are represented as a sequence of numbers, meaning that one possible document representation would simply be the sequence of Unicode code point values of the component characters, e.g.:

```
>>> [ord(i) for i in "computing"]
[99, 111, 109, 112, 117, 116, 105, 110, 103]
```

meaning that the docu-
ment containing the single word `computing` could be “encoded” as:
  991111091121171161051101 03

... or could it?

Text Document Encoding v1 I

- The simplest form of text encoding is through fixed "precision", i.e. a fixed number of digits to represent the code point for each character, e.g. assuming that the highest code point were $10^6 - 1$, we could encode
  each code point in our document with 6 decimal digits, as follows:
  000099000111000109000112000117000
  116000105000110000103

Text Document Encoding v1 II

- Given knowledge of the precision, decoding the document would consist simply of reading off 6 digits at a time and converting them into a code point:

```
>>> prec = 6
>>> doc = "0000990001110001090001..."
>>> "".join([chr(int(doc[i:i+prec]))
... for i in range(0,len(doc),prec)])
'computing'
```

Text Document Encoding v1 III

- This is the method used for many of the popular non-Unicode character encodings, e.g. **ASCII**
  - 128 characters, based on 7-bit encoding (+ 1 redundant bit)
  - compact, but has the obvious failing that it can only encode a small number of characters
- At other end of extreme, **UTF-32** uses a 32-bit encoding to represent each Unicode code point value directly
  - can encode all Unicode characters, but bloated (documents are much bigger than they need to be)
- How to get the best of both worlds — a compact encoding, but which supports a large character set?

Lecture Outline

Take 1: Set the “Top Bit”

- **ISO-8859**
    - single-byte encoding built on top of ASCII (7-bit), to include an extra 128 characters, which is enough to represent many orthographies (Latin $\pm$ diacritics, Cyrillic, Thai, Hebrew, ...)
    - as 128 characters isn’t enough to cover all these orthographies at once, the standard occurs in 16 variants (ISO-8859-1 to ISO-8859-16) representing different orthographic combinations (e.g. ISO-8859-7 for Latin/Greek)

The Limitations of Single-Byte Encodings

- Single-byte encodings such as ISO-8859-* are great if the alphabet is small, but they aren't able to support "big" orthographies such as Chinese, Japanese and Korean (CJK)
- Also, what if we want to have Greek, Hebrew and Arabic in the same document?

Take 2: Variable-width Encodings

- Encode the code points using a variable number of “code units” of fixed size
- The most popular variable-width Unicode encodings are:
  - **UTF-8**: code unit = 8 bits (1 byte)
  - **UTF-16**: code unit = 16 bits (2 bytes)

UTF-8

- UTF-8 is an 8-bit, variable-width encoding, which is compatible with ASCII

Code range (hex)	UTF-8 (bin)
0x000000–0x00007F	0xxxxxxx
0x000080–0x0007FF	110xxxxx 10xxxxxx
0x000800–0x00FFFF	1110xxxx 10xxxxxx 10xxxxxx
0x010000–0x10FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

(i.e. a UTF-8 document containing all code points in the range 0x00–0x7F is identical to an ASCII document)

Features of UTF-8

- Superset of ASCII
- Standard encoding within XML and for HTML5 (and modern operating systems)
- The byte sequence for a given code point never occurs as part of a longer sequence
- Character boundaries are easily locatable

UTF-16

- Similar to UTF-8, except that the code unit is 16 bits rather than 8 bits
- Variable width, as can't fit all 1m+ code points into 16 bits; similar variable-width trick to UTF-8 for higher code points (with the maximum code point length being two code units = 32 bits)
- Not used very widely, but can lead to more compact representations for languages using code points in the 0x000800–0x00FFFF range (which map onto 3 bytes in UTF-8, but 2 bytes in UTF-16)

Sorting in Different Languages I

- Sorting is a solved problem under Unicode, right? ... well, no, not for all languages
- Case of Korean:
    - individual Hangul characters are made up of (up to) three “jamo”, in the form of the initial consonant (one of 16), vowel (one of 21), and final consonant (one of 28)
    - there is a sort order within each of these jamo sets, with the initial consonant being the primary key, the vowel the secondary key, and the final consonant the tertiary key
    - in Unicode, the first Hangul character is at position 44032_{10}
    - as such, the Unicode code point/sort order is defined by:

```
(((initial * 21 + vowel) * 28) + final) + 44032
```

Sorting in Different Languages II

- Case of Japanese:
  - actual sort order in standard dictionaries = pronunciation, then radical/stroke count ... with the complication that most kanji characters have at least two pronunciations, so you have to know how to pronounce the word to know how to sort it(!)
  - in Unicode, the code point order is a hodge-podge in practice, in part because of being backwards compatible with pre-existing standards, and partly because of “unihan” (a unified set of ideographs from Japanese, Korean, and Chinese)
- Case of Chinese (traditional or simplified):
  - actual sort order in standard dictionaries can be based on pronunciation (e.g. pinyin) or stroke order
  - in Unicode, once again, the actual code point order is a mess

Declaring Character Encodings

- As we have seen, it is not necessarily immediately evident which encoding a given document is encoded in (e.g. consider the ISO-8859-* encodings)
- The primary ways of dealing with this are:
  1. manually specify the "character encoding" in the document, e.g. in the case of HTML:

```
<meta http-equiv="Content-Type" content="text/html; charset=iso8859-8" />
```

  2. automatically detect the character encoding, wrt compatibility with encoding standards, user preferences, statistical models, ...

Python and Unicode I

- Mostly in Python3, you don't have to worry about character encodings, as strings are natively in Unicode (unlike Python2, which where the `str` type was ASCII only, and there was a separate `unicode` type for Unicode strings)
- The only place to be a little bit careful is with file I/O, where if the file you are opening/writing to is not in utf-8, you need to specific the encoding, e.g.

```
>>> fp = open("greek.txt", encoding="iso-8859-7")
>>> doc = fp.read()
```

Python and Unicode II

- You can enter Unicode characters either via direct entry, via their code point, or via the Unicode character name:

```
>>> "\U0001F415" == "\N{DOG}"
True
```

- You can encode/decode a string to/from a given encoding using the `encode()`/`decode()` methods:

Python and Unicode III

```
>>> dog_utf8 = "\N{DOG}".encode('utf-8')
>>> print(dog_utf8)
b'\xf0\x9f\x90\x95'
>>> dog_utf8 == "\N{DOG}"
False
>>> dog_utf8.decode("utf-8") == "\N{DOG}"
True
```

Lecture Summary

- What are bits and bytes?
- What are binary, octal, and hexadecimal numbers, and how do they relate to decimal numbers?
- What is Unicode, and what problems does it solve?
- What are Unicode code points and code units?
- What is the "ISO-8859" character encoding family, and how does it extend ASCII?
- What is the relationship between encodings such as `utf-8` and Unicode?
- What is the difference between utf-8 and utf-16?