COMP10001 Foundations of Computing
Algorithms (cont.); Digital Representation

Semester 1, 2019

Tim Baldwin, Nic Geard, Farah Khan, and Marion Zalk

THE UNIVERSITY OF MELBOURNE

— VERSION: 1576, DATE: MAY 20, 2019 —

Reminders/Announcements

- Project 3 due Thursday 30/5
- All Grok worksheets done!
- Last "content" lectures this week; next week will be all about exam preparation, and Project 3/subject wrap-up
- Last guest lecture this Friday, by Dr. Anna Phan of IBM Research on quantum computing
- SES open for subject feedback

Lecture Agenda

- Last lecture:
  - Properties and families of algorithms
- This lecture:
  - Properties and families of algorithms (cont.)
  - Computational counting
  - Digital representation of text

Lecture Outline

More Divide & Conquer

- A more interesting example of divide & conquer:

  Given a list of integers, calculate the maximum sum of a contiguous sublist of elements in the list

```
>>> lst = [5, 3, -1]
>>> maxsubsum(lst)
8
```

- The brute-force solution simply calculates the sum of each (non-empty) sublist, and calculates the maximum among them

More Divide & Conquer

```
def sublist_sum_bf(lst):
  """Brute force."""
  max_so_far = (lst[0], 0, 0)

  for s in range(len(lst)):
    for e in range(s, len(lst)):
        subsum = sum(lst[s:e+1])
        if subsum > max_so_far[0]:
            max_so_far = (subsum, s, e)
  return max_so_far

print(sublist_sum_bf([3,-1,-2,2]))
```

More Divide & Conquer

- The divide-and-conquer approach work as follows:
  - Assume `maxsubsum(i-1)` is the maximum sum for the sublist ending at $i-1$ (inclusive)
  - The maximum sum for the sublist `lst[:i+1]` is `max(lst[i],lst[i]+maxsubsum(i-1))`
- The recursive version will have issues with the limit on recursion depth, so implement iteratively

More Divide & Conquer

```
def dc_maxsubsum(lst):
    # base case (list of length 1)
    assert len(lst) > 0
    max_sum_i = [lst[0]]

    for i in range(1,len(lst)):
        # start new subsequence
        c1 = lst[i]

        # or extend subsequence
        c2 = max_sum_i[-1]+lst[i]

        # now take the max of those two
        max_sum_i.append(max(c1, c2))

    # calculate overall max
    return max(max_sum_i)
```


More Divide & Conquer

- How run-time efficient are the respective implementations (brute-force vs. divide-and-conquer)? (best case vs. worst case vs. average)?
- How storage efficient are the respective implementations (brute-force vs. divide-and-conquer)? (best case vs. worst case vs. average)?

Simulation

- Strategy:
  - Randomly generate a large amount of data to predict an overall trend
  - Use multiple runs to verify the stability of an answer
  - Used in applications where it is possible to describe individual properties of a system, but hard/impossible to capture the interactions between them
- Applications:
  - Weather forecasting
  - Movement of planets
  - Prediction of share markets

Simulation: A Game of Chance

- Gambling game:
  - You bet \$1, and roll two dice
  - If the total is between 8 and 11, you win \$2
  - If the total is 12, you win \$6
  - Otherwise, you lose
- Is it worth playing?
  - Start with a \$5 float and play to \$0 or \$20
  - How many games do you win on average?

Monte Carlo Simulation

- Method:
  - iteratively test a model using random numbers as inputs
  - problem is complex and/or involves uncertain parameters
  - a simulation typically has at least 10,000 evaluations
  - approximate solution to problem that is not (readily) analytically solvable
- Game of chance:
  - should a casino offer this game?

Heuristic Search

- Strategy:
  - Search via a cheap, approximate solution which works *reasonably* well *most* of the time ... but where there is no proof of how close to optimal the proposed solution is
- Examples:
  - For finding a closest neighbor many Location-based Services use Euclidean distances as they are easy to compute
  - There is no guarantee they are equal to road-network distances though
  - So whatever you find, is just "possibly" a good solution

COMP10001 Foundations of Computing Week 11, Lecture 1 (21/5/2019)

Lecture Outline

1. Algorithm Families (cont.)
   - Divide and Conquer
   - Simulation
   - Heuristic Search
2. Computational Counting
3. Character Encoding and Multilingual Text

COMP10001 Foundations of Computing Week 11, Lecture 1 (21/5/2019)

Computational Counting

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

$$\begin{aligned} 2019_{10} &= 2 \times 1000 + 0 \times 100 + 1 \times 10 + 9 \times 1 \\ &= 2 \times 10^3 + 0 \times 10^2 + 1 \times 10^1 + 9 \times 10^0 \end{aligned}$$

COMP10001 Foundations of Computing Week 11, Lecture 1 (21/5/2019)

Computational Counting

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

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

COMP10001 Foundations of Computing Week 11, Lecture 1 (21/5/2019)

Computational Counting

- 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

COMP10001 Foundations of Computing Week 11, Lecture 1 (21/5/2019)

Computational Counting

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

$$00110110_2$$
$$\Downarrow$$
$$0011_2 \quad 0110_2$$
$$\Downarrow \qquad \Downarrow$$
$$11_2 \quad 110_2$$
$$\Downarrow \qquad \Downarrow$$
$$3_{16} \quad 6_{16}$$
$$\Downarrow$$
$$36_{16}$$

Computational Counting

- 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

- It is also possible to "cast" 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

... and to cast 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

- 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

1. Algorithm Families (cont.)
   - Divide and Conquer
   - Simulation
   - Heuristic Search
2. Computational Counting
3. Character Encoding and Multilingual Text

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

- 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. ´ + *a*)
- For (some) emoji, there is scope to change skin-tone via composite characters, e.g. 👦 + 🏾 = 👦🏾

Unicode

- 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 document containing the single word `computing` could be "encoded" as:

99111109112117116105110103

... or could it?

Text Document Encoding v1

- 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

- 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 = "000099000111000109000l..."
>>> "".join([chr(int(doc[i:i+prec]))
... for i in range(0,len(doc),prec)])
'computing'
```


Text Document Encoding v1

- 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 Summary

- What are each of: divide and conquer, simulation, and heuristic search?
- What are bits and bytes?
- What are binary, octal, and hexadecimal numbers, and how do they relate to decimal numbers?
- How does Python distinguish between binary, octal, decimal, and hexadecimal numbers? How can we convert between them?
- What is Unicode, and what problems does it solve?