Module Introduction

210CT

2015/16

1 Module Introduction

1.1 Introduction to the module

This module builds on the concepts and principles outlined in the Programming module at level 1.

The aim of this module is to give you additional insight into programming techniques, in the context of data structures and algorithms.

The subject of complexity analysis for algorithms and operations upon data structures will be introduced and developed within a practical context.

Advanced language features such as the use of event driven programming and concurrency will also be introduced in this module.

1.2 In plain English

Year 1 programming was all about basics, learning syntax and some simple practical examples.

Lots of problems turn out to be more interesting than that. Some problems are simple to describe but turn out to be very difficult to solve.

- Example 1: The travelling salesman problem
- Example 2: Sorting
- Example 3: This number is the product of two primes: 1,522,605,027,922,533,360,535,618,378,132,6
  718,068,114,961,380,688,657,908, 494,580,122,963,258,952, 897,654,000,350,692,006,139
  - Which two?
    - 3797522793694367392280887275544562785456553663819 9 and 4009469095092088103068373529276146838921489972406 1. It took days to factorise this 330bit number on a powerful "super" computer.

  - Why do we care?

1.3 Staff

- James Shuttleworth, `mailto:csx239@coventry.ac.uk`
- Leonid Smalov `mailto:csx211@coventry.ac.uk`
- Diana Hintea `mailto:ab8351@coventry.ac.uk`
- Kamal Bentahar `mailto:ab3735.ac.uk`
- Abdulrahman Altahhan `mailto:ab8556@coventry.ac.uk`
- Dan Goldsmith: `mailto:aa9863@coventry.ac.uk`
- Simon Billings `mailto:ab9160@coventry.ac.uk`
- Each lab has different staffing. Find the Theta availability, and contact details, of your lab tutors on the sigma timetable: `http://computing.coventry.ac.uk/sigma`

1.4 Getting Help

- Labs
- Module staff
- Theta center (Or Delta, or Sigma)
- Programming Support Centre
  - `http://computing.coventry.ac.uk/sigma`

1.5 Assessment

- Assessment will be by a portfolio of exercises and an exam
- The portfolio will be worth 50% of the module
- The exam is worth 50% of the module
- The portfolio will be made up of weekly exercises and handed-in in two parts, one half-way through the semester and one at the end

- The exact selection of exercises won't be revealed until a few weeks before the portfolio is due
- You should complete all homework exercises. The only way to learn about programming is to do it.

1.6 Registers

- There will be a register each week in the lecture.
- You must sign your name.
- Non attendance will be met with severe letter writing and tutting

1.7 Programming Language

- You can work in whatever you like
- Lectures and notes will be presented in Python and C++
  - Unless I decide there's an interesting reason to do something else
  - Feel free to ask about how things might work in another language

2 Algorithms

2.1 What is an algorithm?

- A solution to a problem
- A way of doing things
- As programmers: an algorithm is a precise description of a procedure that has enough detail for someone to implement it
- Not tied to a given language
- Often we use **pseudocode** to describe algorithms in a language-independent way

2.2 Problems

- A problem is a general description of what needs to be solved
  - Find z, where $z = x + y$ and both x and y are both integers...
- An instance of a problem is a specific case of the problem
  - Find $2 + 2$

2.3 Designing and Implementing Algorithms

- Some algorithms are used over and over
- Often they have general-purpose implementations
- Some problems are new, or new enough that the general-purpose implementations can;t be used
- You should be able to choose a suitable algorithm or be able to develop a correct and efficient solution to a range of problems by the end of the module

2.4 How do we evaluate algorithms?

- What makes an algorithm "correct"?
- An algorithm is correct only if it solves every possible instance of a problem
  - We call this the correctness condition
  - For example if out problem is "find the number that is the sum of two integers", this is not a complete solution:

```
begin
  return 5
end
```

- But: not all algorithms produce 100% perfect answers!
  - Not the same as completeness...
  - Not all problems can **have** 100% perfect answers. Example: what is the value of π

  - Some problems don't have a clear way to get to 100% perfect answers without waiting for longer than the universe has yet to live

- Running time and efficiency are also big factors
  - How do we measure this?
  - Running time is dependent on hardware
  - If we have a standard set of hardware, can we restrict all other processes that might suddenly use ram, peripherals or just CPU time and affect the answer?
  - Need to talk in more abstract terms

3 Run-time analysis example

- Input: Two strings of characters
- Output: True if the strings are anagrams on one another; False if they are not
- A problem with a yes/no (or true/false) output is referred as a decision problem
- We will consider two different algorithms to solve this problem

3.1 Algorithm 1

```
begin
  if string1 length != string2 length
    return False
  for all letters i in string1
    matched <- false
    #set a variable 'matched' with the value false
    for all letters j in string2
      if j is not marked and i = j
        mark j
        matched <- True
        break
        #exit inner loop
    if matched = false
```

```
      return False
  return True
end
```

- Thanks to Theocharis Ledakis for spotting a typo

3.2 Analysis of Algorithm 1

- Two nested loops, each the size of the strings' length (call this length n)
- Iterates along the first string n times
- Each time it then iterates through the second string, stopping when it finds a matching character
- Each character is matched once and only once
- The if statement executes 1+2+...+n times
  - Therefore we have roughly $\frac{n(n+1)}{2}$ steps
    - Thanks to Zsolt Ban for pointing out that the subtraction should be an addition

3.3 Algorithm 2

```
begin
  if string1 length != string2 length
    return No
  character count array <- 0
  for all letters i in string1
    increment the number of occurrences of i #increment: add 1 to
  for all letters j in string2
    decrement the number of occurrences of j #decrement: subtract 1 from
  for all integers k in the array
    if k != 0
      return No
  return Yes
end
```

3.4 Analysis of Algorithm 2

- No nested loops, just three one after the other

- Loop to initialise character count array iterates 26 time (the number of characters in the alphabet)
- The second two loops iterate n times
- Therefore the number of steps is 26+n+n

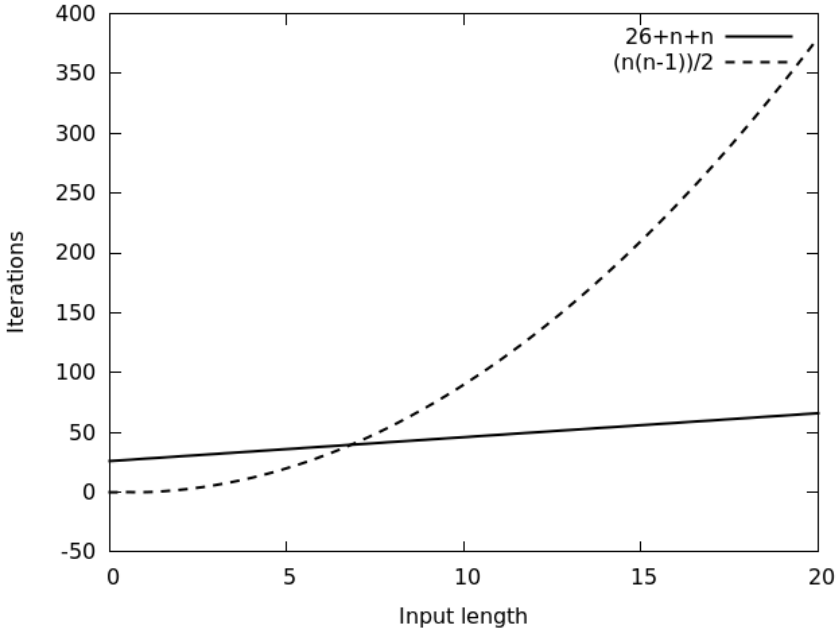
3.5 Comparison

- If the length of the strings are small, e.g. 5
  - 1st algorithm $\frac{5(6)}{2} = \frac{30}{2} = 15$ steps
  - 2nd algorithm $26 + 5 + 5 = 36$
- The 1st algorithm is faster
- But, if the length of the strings is large e.g. 200
  - 1st algorithm $\frac{200(201)}{2} = \frac{40200}{2} = 20100$ steps
  - 2nd algorithm $26 + 200 + 200 = 426$ steps

3.6 Conclusion

- Different algorithms can have dramatically different performance
- The performance difference often depends greatly on the size of the data set
- Choosing the right algorithm for the job at hand is important!

3.7 Gratuitous graph



4 Data Structures

4.1 Data Structures

- Ways of storing data, impacts upon
  - Size
  - Speed
  - Expression
- You should have had some experience with data structures
  - Strings
  - Lists
  - Dictionaries
  - Tuples

4.2 Data Structures

- Each data structure has its pros and cons, for example:
- It's efficient to add or remove elements from a list
  - But can take up to n comparisons to find an element in the first place
- Dictionaries allow us to find elements quickly
  - But can take up quite a bit more memory than a plain array
- Arrays allow us to access items directly using offset/index/address, which is very fast and takes up almost no memory other than the array contents
  - But inserting an item in the middle is very slow

4.3 Example Data Structure: the Set

- Items are stored without any order
- No duplicates
- Operations include union, intersection, difference
- Useful as the basis for many algorithms

5 Homework

5.1 Pre-Homework

- This is the "week 0" set of exercises.

5.2 Actual work

1. Write the algorithm for determining if a given string a is a substring of another string, b. For example, "cat" is a substring of "dedicated". Try to use the "pseudocode" style from the lecture, but don't worry too much as long as it's understandable. We will talk more on presenting algorithms in pseudocode later.
2. Implement the algorithm in a function called "isSubString"

3. Write a program that asks the user for two strings and uses the function you made to find out if one is a substring of the other. It shouldn't matter in what order the user types the strings.

5.3 Challenge

1. Look up the Levenshtein Distance
2. Implement it.
3. I've seen almost every implementation on the web. I will recognise one of them if you hand it in. This would be bad.

Emacs 23.3.1 (Org mode 8.0.3)