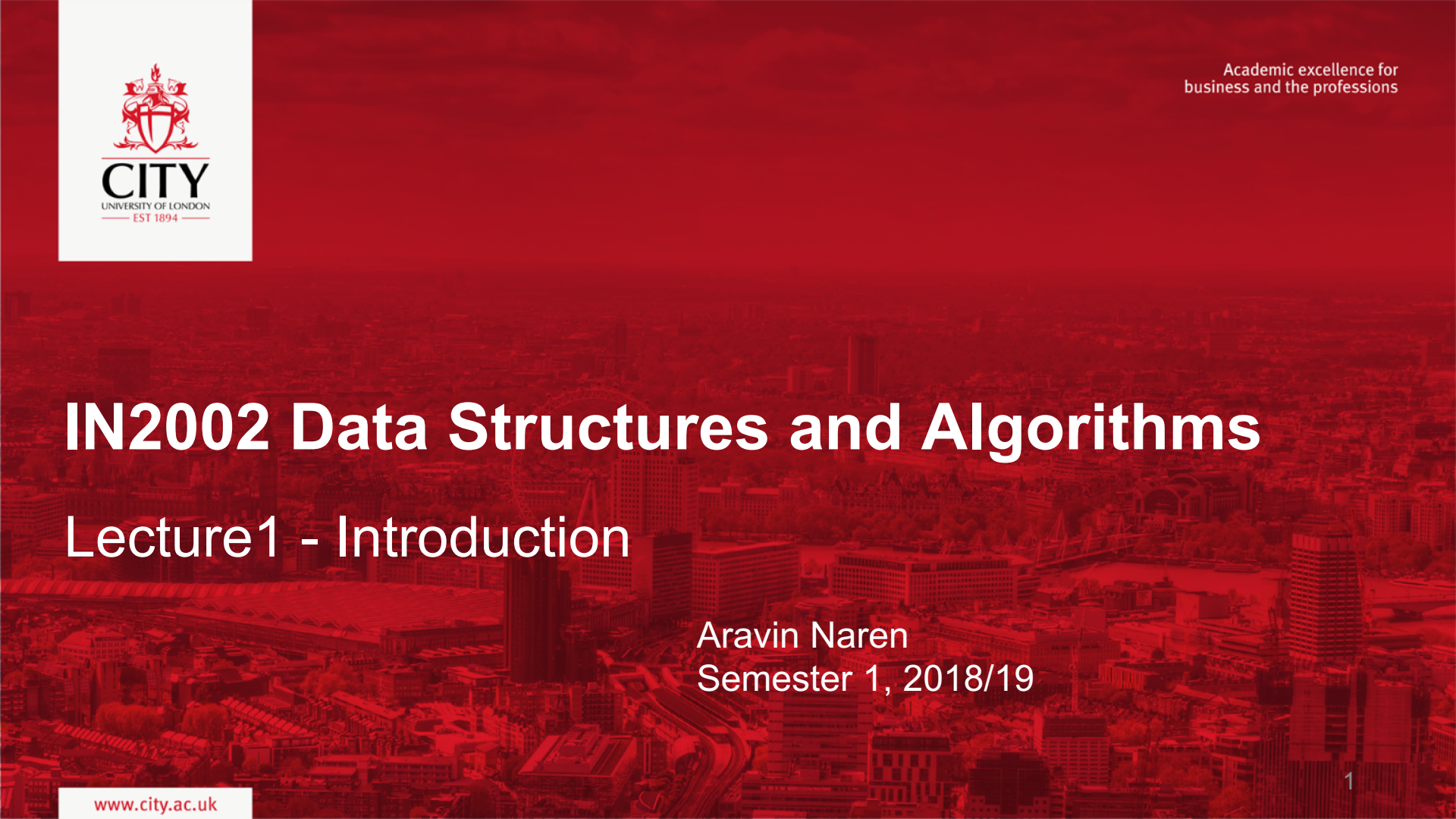


IN2002 Data Structures and Algorithms

Lecture1 - Introduction

Aravin Naren
Semester 1, 2018/19

Credit where credit is due

- The slides and material for this module have been evolving over the years. Some are brand new and some are adapted/adopted from previous years and/or previous module leaders at City.

What this module is about

An abstract approach to programming as:

- organising information (*data structures*)
- performing computation (*algorithms*)

Essential for computing professionals:

- knowledge of common *data structures* and *algorithms*
- knowledge of when and how to use them
- ability to analyse and develop them

Context

Builds on concepts from…

- Computation and Reasoning
  - Algorithms
  - Complexity analysis
- Systems Architecture
  - How the operating systems handle programs execution
- Programming
  - Describing algorithms
  - Executing algorithms

Provides the basis for…

- Any work relying on the creation and/or modification of programs

Learning Objectives

- Explain the workings of standard data structures and algorithms
- Analyse the space and time complexity of algorithms
- Identify the most important abstract data types and the ways in which they may be implemented
- Select appropriate data structures and algorithms for particular practical situations
- Describe an implementation using plain English or pseudocode
- Devise appropriate algorithms to address specific problems

How to reach the learning goals

- Work continuously
- Attend the lectures (arrive on time!)
- Go to the tutorials and do the exercises
- Use interactive examples
- Program the algorithms yourself
- Work through all the exercises
- Read the relevant sections in the textbook
- Do additional exercises

Course Text

Weiss, Mark Allen: *Data Structures & Problem Solving using Java (4th edition).* Pearson Addison-Wesley, 2014

…or, alternatively,

Drozdek, Adam: Data Structures and Algorithms in Java (4th edition). Cengage 2013

Earlier editions will do fine, and other books on data structures and algorithms will help, too.

Organisation

Module Lecturer: Aravin Naren <aravin.naren.1@city.ac.uk>

Office hours: A302 (College Building)

usually Thu 15:00-16:50

check on-line (via Moodle) for potential changes

Lectures: 2 hours / week

Thursdays; 9:00 - 10:50 in A130

Tutorial/Lab: 1 hour / week

Leaving 120 hours for reading, coursework preparation, etc…

Material on Moodle, with discussion board for mutual help

Exercises

Exercise sheets will be available on Moodle every week. The exercises will be the topic of the tutorials.

On even weeks will include a programming exercise.

Doing the exercises is **essential** in preparing for the assessments.

Questions & Discussion

Discussion forum on Moodle

- We read it frequently and reply/comment as appropriate

Asking specific questions

- Moodle, tutorials, lectures, TAs surgeries and module leader's office hours

Note that Moodle will also be used for announcements

Assessment

Formative:

- 3 Multiple Choice Quizzes in weeks 3, 5 and 7

Summative:

- Oral exam (viva) late in the term based on lectures and tutorials, worth 30% of overall marks
- Written open books exam in January, worth 70% of overall marks

Feedback

Tutorials

- Formative feedback, but you must do the exercises first

Quizzes

- Formative feedback online via Moodle

Viva

- Immediately, face to face

Exam

- Observations on general performance via Moodle

2016-2017 Year Results

Overall

- 135 fully enrolled by exam time
- 92 passed 1st attempt (68%)
- 26 passed resit (another 19%)
- 4 failed resit (3% of students)
- 14 did not attempt the resit (another 10%)

2017-2018 Year Results

Overall

- 149 fully enrolled by exam time
- 130 passed 1st attempt (87%)

Some input from previous students

Advice for new students

- Practice on examples (tutorials and exercises) and seek feedback

Advantages of the module

- Real world applicability
- Thought provoking
- Connection to other modules
- Important for job interviews

Changes over the years

- Changed some slides & reshuffled some of the material
- Made the link to programming more explicit
- Made MCQs formative
- Changed the exam format

Rules of Engagement

- Follow the student charter
- Do NOT disturb others during the lectures
  - If you are more than FIVE minutes late to the lecture, then wait for the break to come in
  - Don’t use your mobile phones, Ipods, etc.
    - Keep them off or silent mode
  - Don’t hold parallel conversations during the lectures
- Do give us constructive feedback off line as we are moving along
- We will follow the University rules
- We will strive to give you the best learning experience possible

Synopsis

- Revisit intro to algorithms, pseudocode, time and space complexity
- Recursion
- Pointers
- Abstract data types
  - Priority queues, heaps, stacks, queues
  - Linked lists
    - Singly linked lists, doubly linked lists and circular lists
- Hash tables
- Advanced trees
- Graphs

Week 1: Basic Concepts

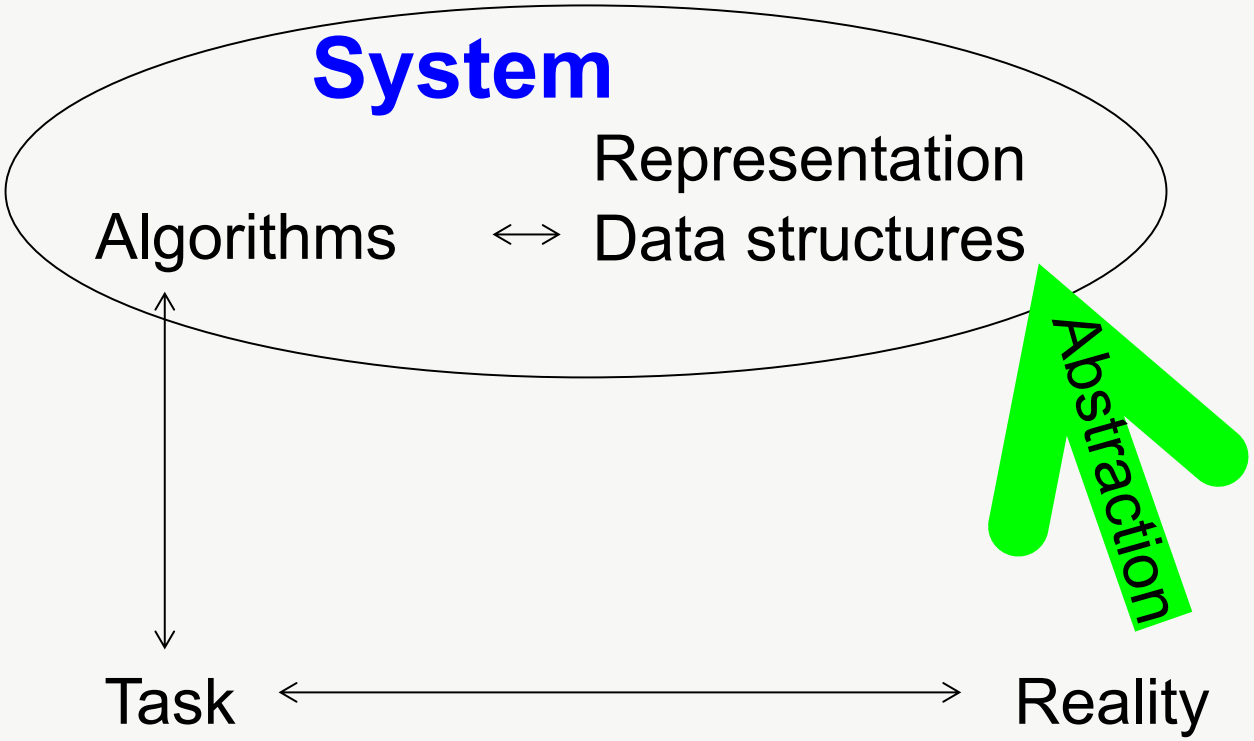
Basic Concepts

- What are Algorithms and Data Structures?

- Describing an algorithm
- Properties of algorithms

- Data structure: static arrays
- Algorithm type: search

Data Structures and Algorithms*



* See Computation & Reasoning lecture notes

Algorithms in Computing

Mathematical description of actions on numbers or symbols.

- can be executed by a machine
- are easily accessible to analysis

We will look at the abstract properties of algorithms, understanding how they work and using a bit of mathematics to determine how they behave.

Describing an Algorithm

- As unstructured text
- As structured text
- As a flow chart
- As pseudocode
- As Java code (or any programming language)

In this module we will mainly use pseudocode and Java.

Pseudocode Syntax*

Title with function/method name and arguments

Sequence control in capitals:
– WHILE
– IF/THEN/ELSE

Indent to indicate scope

Arrow for assignment (can be typed “<-”)

Function foo(array):
bar ← 0, i ← 1
WHILE i <= length of array
bar ← bar+ array[i]
i ← i + 1
Return bar

* See Computation & Reasoning lecture notes

Another Example

Title with function/method name and arguments

Sequence control in capitals:
– WHILE
– IF/THEN/ELSE

Indentation indicates scope

Arrow for assignment (can be typed as “<-”)

Function foo2(array):
bar ← 0, i ← 1
WHILE i <= length of array
IF array[i] = 0 THEN
bar ← bar + 1
ELSE
array[i] ← 0
i ← i + 1
Return bar

Data Structures

Data structures are ways of organising data.

They should

- represent all relevant information
- use little memory
- support efficient algorithms

A simple data structure: a static array

- Built into most programming languages
- A finite set of elements
- The elements are in a fixed sequence
- Elements can be addressed by

indices (Pseudocode, Mathematics, Pascal) or offsets (Java, C, C++)

	26	3	7	13	9	5	17	4
index	1	2	3	4	5	6	7	8
offset	0	1	2	3	4	5	6	7

Devising programmes to solve problems

… is a combination of identifying the most suitable

- data structure to contain the data
- algorithm to accomplish the task

- Not all combinations of data structures and algorithms work
- A given combination of data structure and algorithm may be ideal for one situation, yet not the best for another

An algorithm example

What does *foo* do? *

```
Function foo(array):
bar ← 0, i ← 1
WHILE i <= length of array
    bar ← bar+ array[i]
    i ← i + 1
Return bar
```

* From slide 23

Another algorithm example

What does *foo2* do? *

```
Function foo2(array):
bar ← 0, i ← 1
WHILE i <= length of array
    IF array[i] = 0 THEN
        bar ← bar + 1
    ELSE
        array[i] ← 0
    i ← i + 1
Return bar
```

* From slide 23

Known algorithm examples

- Selection sort
- Insertion sort
- Quicksort
- Mergesort
- Binary search

➢ Why were these examples explored?

* Seen on Computation & Reasoning

Computational Complexity

Computational Complexity

Description of the resources needed by an algorithm:

- Complexity in
  - (running) time or
  - (memory) space
- Best, worst and average cases
- Representation by growth

Formalism for big-O

DEFINITION: *f(n)* is in *O(g(n))* if and only if *c* and *N>0* exist such that for all *n* > *N*, $f(n) \leq c \cdot g(n)$

c is a constant factor, and *N* is an index of asymptotic behaviour.

Example: $f(n)=2n^2 + 3n + 1$ is in $O(n^2)$:

N=2 and *c=4*, or *N=4* and *c=3*, etc.

Property: if *f1(n)* and *f2(n)* are both in *O(g(n))*, so is *f1(n)+f2(n)*

Proof (sketch): Let c_1, c_2 be the constant factors for *f1 and f2, then with* $c_{1+2} = c_1 + c_2$ *apply big-O definition.*

Implications…

- Big-O is an upper bound, actual values may be less
- If $g(n) \leq f(n)$ then $g(n)$ is in $O(f(n))$
- Saying that an algorithm *a* is in $O(g(n))$ means that it grows no faster than $g(n)$.

Note: We often omit the word 'in' and say *a* is $O(g(n))$.

Properties of big-O

Rule 1: Scaling by a constant: $O(c \cdot f(n)) = O(f(n))$

Rule 2: Addition:

if $f(n)$ is $O(g(n))$ *(always applies to one part)*

then $O(f(n)) + O(g(n)) = O(g(n))$ (greater term dominates)

Rule 3: Multiplication:

$O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$

Rule 4: Logarithms:

$O(\log_a n) = O(\log_b n)$ (we usually just write $O(\log n)$)

Example calculations

- Orders of sequential algorithm phases are added:

e.g. an $O(n^2)$ phase followed by an $O(n \log(n))$ phase is

$$O(n^2) + O(n \log(n)) = O(n^2)$$

  - ($n \log(n)$) is $O(n^2)$, apply Rule 2)

- $3n \log(n) + 100n + 50\log(n) + 23$ is …

Example calculations (2)

- Orders of nested loops are multiplied:

  e.g. an outer loop is executed $O(n^2)$ times and contains an inner loop that is executed $O(n)$ times (its content is O(1)):

  $$O(n^2) \cdot O(n) = O(n^3) \qquad \text{Rule 3}$$

- $3n \log(n) \cdot 2n + 7\log(n)$ is …

Common types of complexity *

constant: $O(1)$

logarithmic: $O(\log n)$ — ***good***

linear: $O(n)$

quadratic: $O(n^2)$ — ***acceptable*** *(sometimes)*

cubic: $O(n^3)$

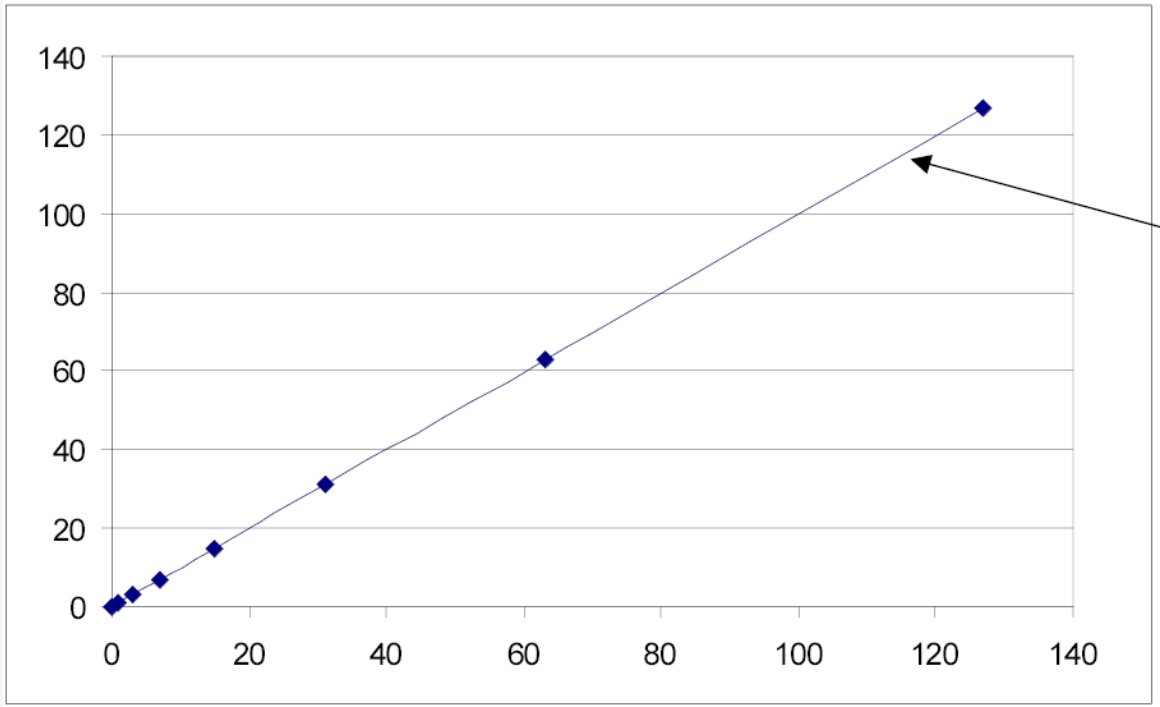
polynomial: $O(n^k)$

exponential: $O(a^n)$ — ***intractable***

factorial: $O(n!)$

* See Computation & Reasoning lecture notes

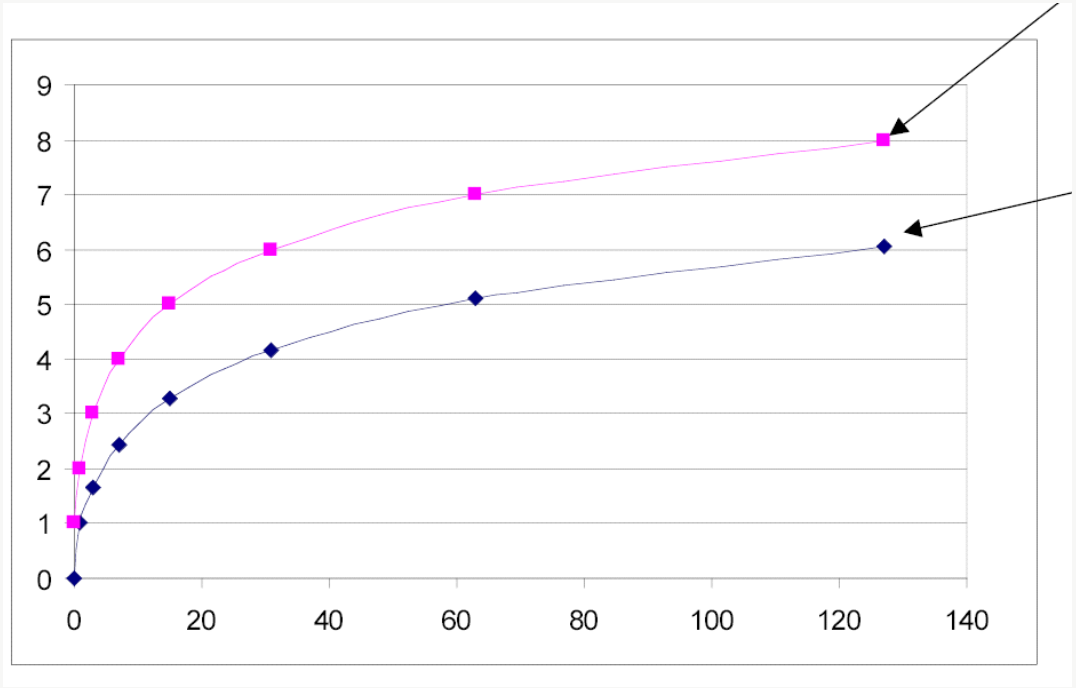
Sequential search is linear



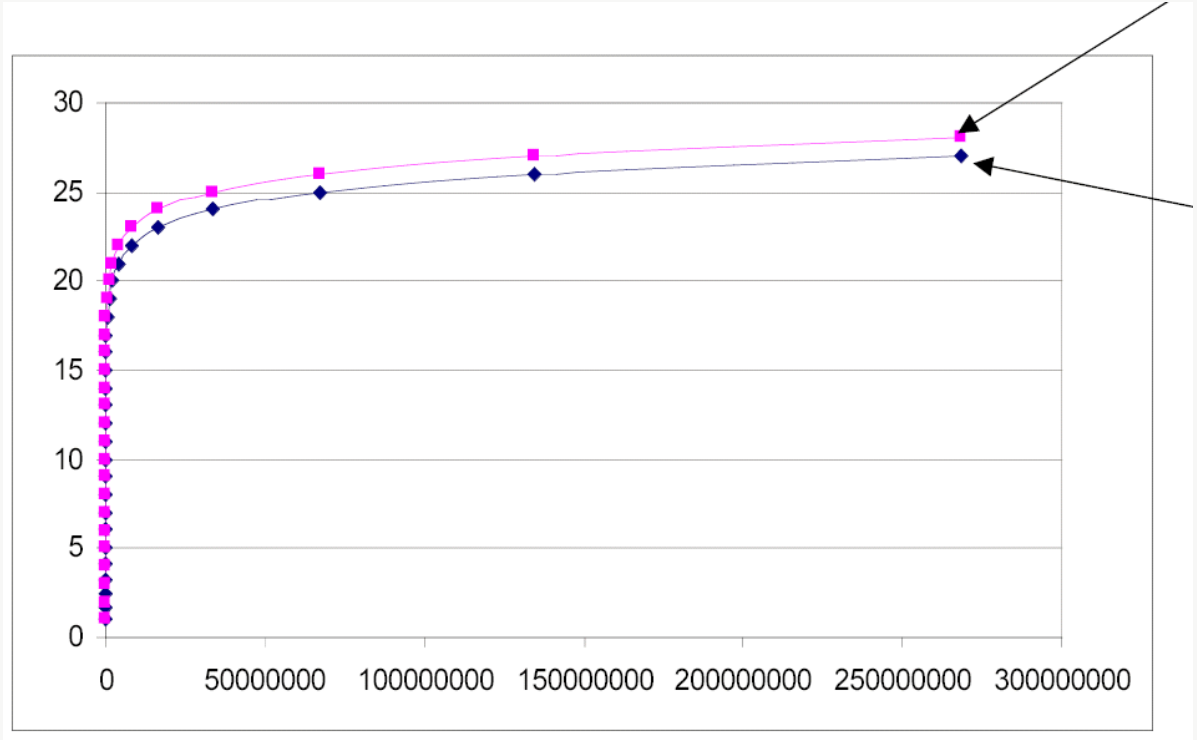
number of loops

array length

Binary search is logarithmic: O(log n)



Binary search is logarithmic: O(log n)



Note how flat the graph gets for large numbers.

Finding the order of complexity

```
Function binary_search(array, key)
lo← 1, hi ← length of array
WHILE lo ≤ hi
        mid ← (lo + hi) / 2
        IF key < array[mid] THEN
                hi = mid - 1
        ELSE
                IF key > array[mid] THEN
                        lo = mid + 1
                ELSE
                        Return mid
Return -1
```

Examples of code complexity analysis

```
Function foo3(array):
sum ← 0, c ← 1
WHILE c <= length of array
        c1 ← 1
        WHILE c1 <= length of array
                sum ← sum + array[c] *
array[c1]
                c1 ← c1 + 1
        c ← c + 1
Return sum
```

Other examples of complexity code analysis *

Function foo(array):
bar ← 0, i ← 1
WHILE i <= length of array
    bar ← bar+ array[i]
    i ← i + 1
Return bar

* From slides 22 and 27

Other examples of complexity code analysis *

```
Function foo2(array):
bar ← 0, i ← 1
WHILE i <= length of array
    IF array[i] = 0 THEN
        bar ← bar + 1
    ELSE
        array[i] ← 0
    i ← i + 1
Return bar
```

* From slides 23 and 28

Some algorithms and their complexity

- Binary search: O(log n) time, O(1) space **fast**
- Linear search: O(n) time, O(1) space
- Multiplication of n-bit integers: O(n log n log log n) **doable**
- Matrix multiplication for n*n matrices (std): $O(n^3)$
- Travelling salesman: Best known algorithms take $O(2^n)$ time **intractable**
- All permutations of a sequence: O(n!)

Well known complexity classes

P: computable in polynomial time

- these problems are considered tractable (although there are limits in practice).

NP: computable in polynomial time on a non-deterministic (infinitely parallel) machine

- if we can take a solution and test it in polynomial time, we can find any solution by testing all possible solutions in parallel

EXPTIME: exponential time

- these problems are known to be really hard.

NP-Complete Problems

Class of combinatorial problems that are equally hard; e.g.:

- Travelling salesman problem:

  Shortest tour visiting all cities in a set

- Knapsack:

  Given a set of items with a given value and weight each,
  find the set with the highest value within a weight limit

- Satisfiability:

  Find the assignment of values to make a logical proposition true

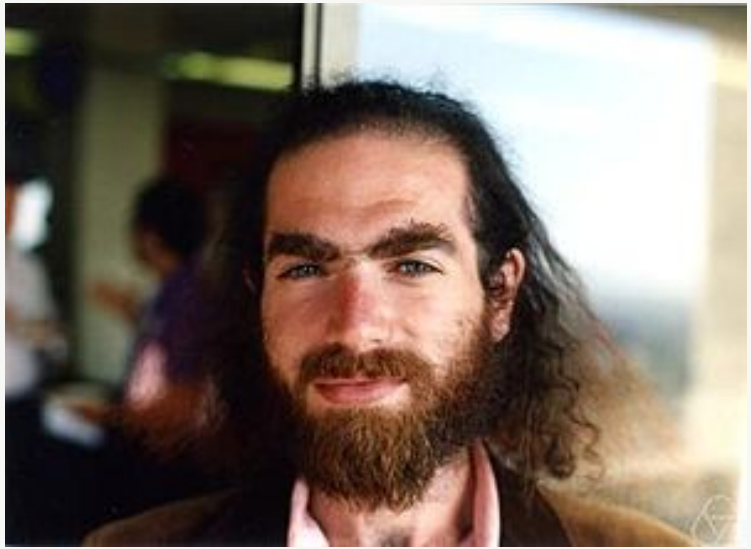
Any NP problem can be reduced to any of these, i.e. if we have a solution to one of them we have one to all NP.

Famous Question: Is P = NP?

One of the 6 Millennium Prize Problems
US $1 million offered for a solution by
Clay Mathematics Institute, Massachusetts.

If there is an polynomial solution to an NP-complete problem, then P = NP.
No solution found yet, but also no proof that it is impossible.

Most scientists believe P != NP. Otherwise some very hard problems will become easy (possibly including some cryptographic code-breaking). Some solutions have been presented, but none has stood up to scrutiny yet.



Grigori Perelman

Reading

- For visualisations check Linear and Binary Search on: https://www.cs.usfca.edu/~galles/visualization/Search.html

- Weiss: Chapter 5

- Drozdek: Chapter 2

Next week: Recursion & Abstract Data Types

City, University of London
Northampton Square
London
EC1V 0HB
United Kingdom

T: +44 (0)20 7040 8406
E: ug-compsci@city.ac.uk
www.city.ac.uk/department-computer-science