COMP90038 Algorithms and Complexity

Lecture 1: Introduction
(with thanks to Harald Sondergaard)

Toby Murray

toby.murray@unimelb.edu.au

DMD 8.17 (Level 8, Doug McDonell Bldg)

http://people.eng.unimelb.edu.au/tobym

@tobycmurray

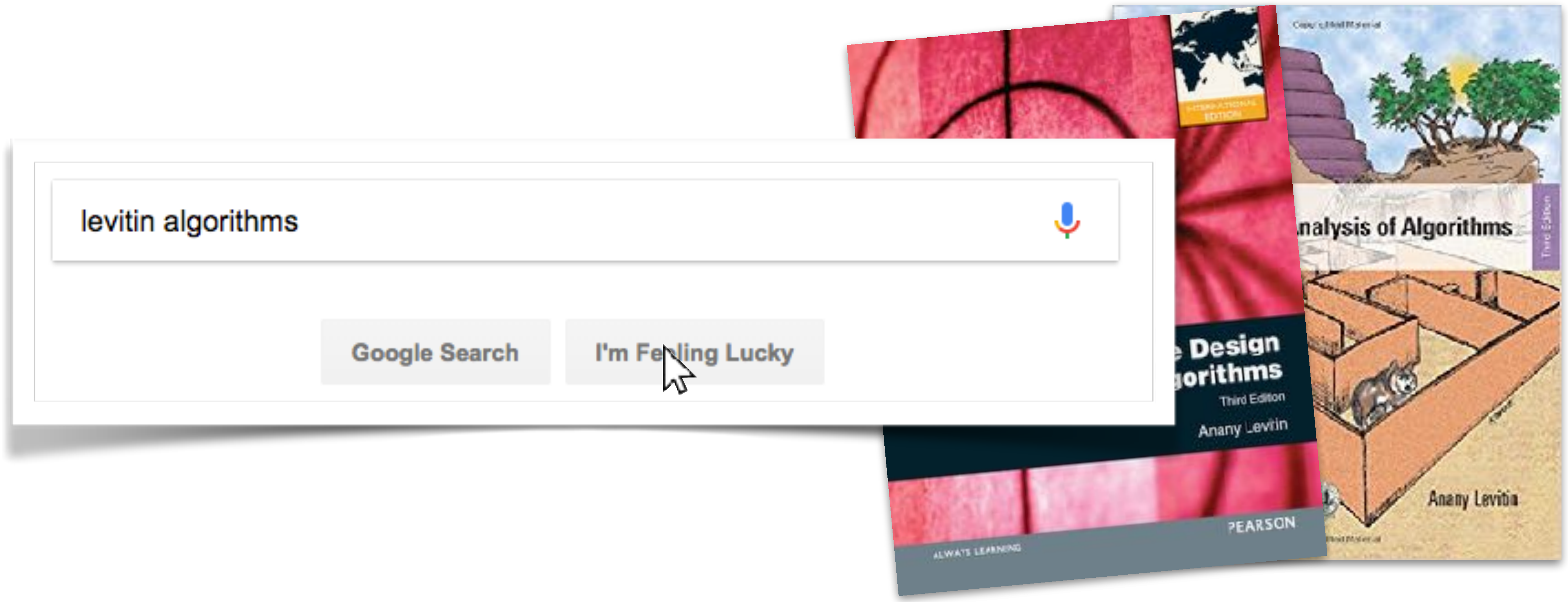
What we will learn about

THE UNIVERSITY OF MELBOURNE

- **Data structures:** e.g. stacks, queues, trees, priority queues, graphs
- **Algorithms** for various problems: e.g. sorting, searching, string manipulation, graph manipulation, …
- **Algorithm Design Techniques:** e.g. brute force, decrease-and-conquer, divide-and-conquer, dynamic programming, greedy approaches
- **Analytical and empirical assessment** of algorithms
- **Complexity classes**

Textbook

Anany Levitin. *Introduction to the Design and Analysis of Algorithms*, 3rd Edition, Pearson 2012



See also "Reading Resources" on LMS

Staff

Toby Murray
Course Coordinator
Lecturer

Andres Munoz Acosta
Lecturer

Pan Lianglu
Head Tutor

Tutors:
Annie Zhou, Assaf Dekel, Damayanthi Herath,
Nikolas Makasis, Oscar Correa Guerrero, Partha De,
Sameendra Samarawickrama, Yankun Qiu, Zheyu Ji

LMS, Lectures, Tutorials

- Primary on-line resource for the subject
- **Announcements**, lecture slides, recordings, tutorials, assignments, **weekly quizzes**, **discussion board**
- **Tutorials** start in week 2

Assessment

- **Quizzes**: one each week. **Due by Tuesday of following week.**
- **You MUST complete at least 8 quizzes to sit the final exam.**
- **2 Assignments** due around Week 6 and Week 11, worth 30% together
- **3-hour Final Exam:** worth 70%
- **To pass the subject:**
    - complete at least 8 of the Week 2 — 12 quizzes
    - obtain at least 35/70 in the final exam
    - obtain at least 50/100 overall

Time Commitment

- Expect to spend:
  - 34 hours in lectures + tutes
  - 30 hours on assignments
  - 24 hours reading and reviewing
  - 24 hours on tute prep
  - 8 hours on quizzes and discussion
- On **average**: 10 hours per week
- Commitment is **worth it**
  - What you learn here will form the foundations of a career in software, IT, computational science, engineering, etc.

Assumed Knowledge

- There are **two diagnostic quizzes** for Week 1
  - not compulsory, but to help you work out how well you know the assumed background knowledge
  - **Mathematics** (sets, relations, functions, recurrence relations)
  - **Programming** (arrays, records, linked lists, dictionaries, functions, procedures, formal and actual parameters, parameter passing, return-values, pointers / references
- See the **Reading Resources** page on the LMS which has some pointers to online resources to help with this background knowledge

How to Succeed

- **Understand the material** (by doing), don’t just memorise it
- If you fall behind, try to catch up as soon as possible
- **Don’t procrastinate**, start early
- Attempt the tutorial questions **every week**, **before** you attend the cute
- **Use the LMS discussion board**
  - **Ask questions**
  - **Answer others’ questions**
- **We are all on the same learning journey and have the same goal!**

Introduction

- Introduce yourself to your neighbours
- Tell them where you are from, what degree you are doing, which programming languages you know, what music you listen to, whatever

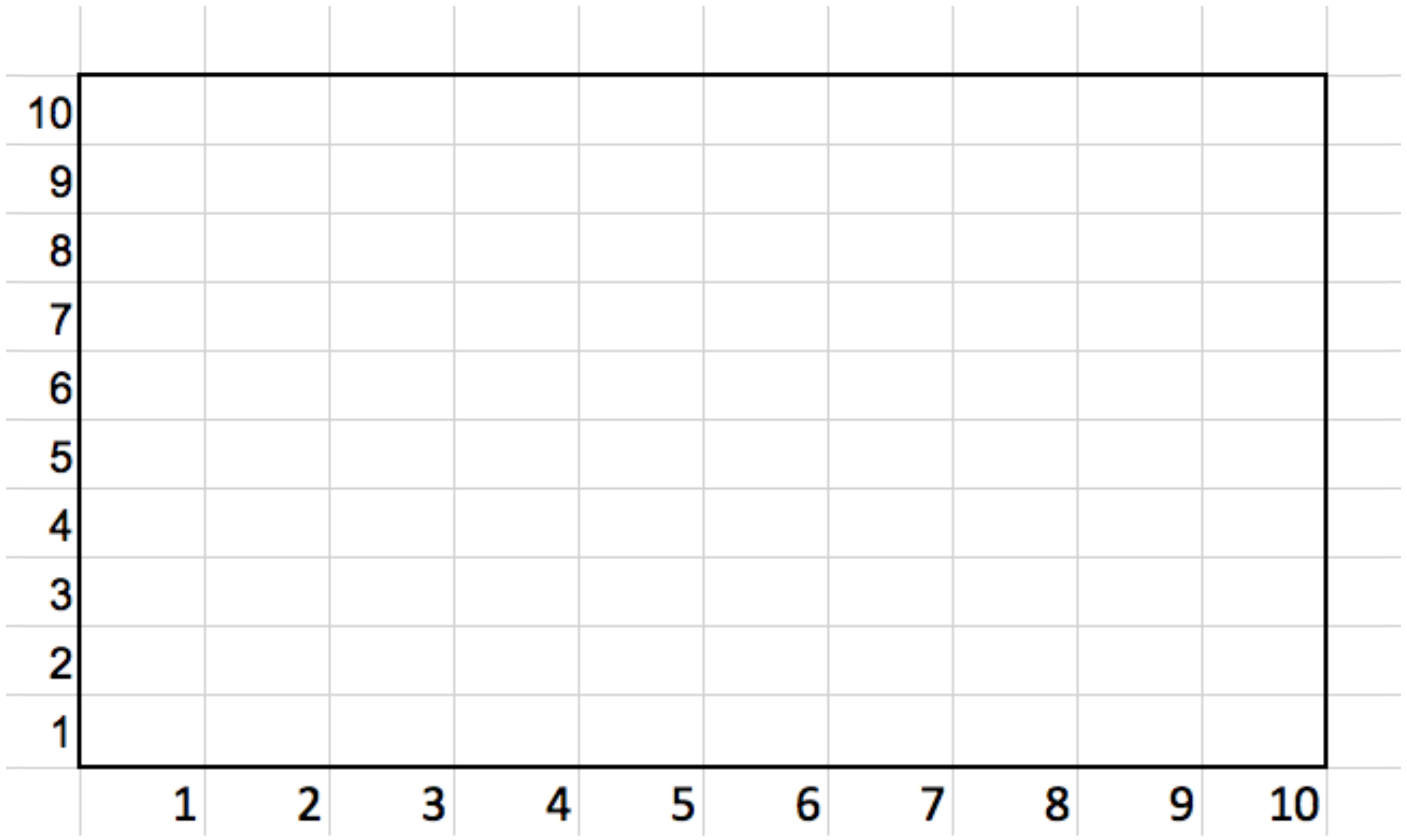
A Maze Problem

- A maze (or labyrinth) is contained in a 10 x 10 rectangle, with rows and columns numbered from 1 to 10
- It can be traversed along rows and columns, moving: up, down, left, right
- The starting point is (1,1); the goal point is (10,10)
- These points are obstacles that you cannot pass through:

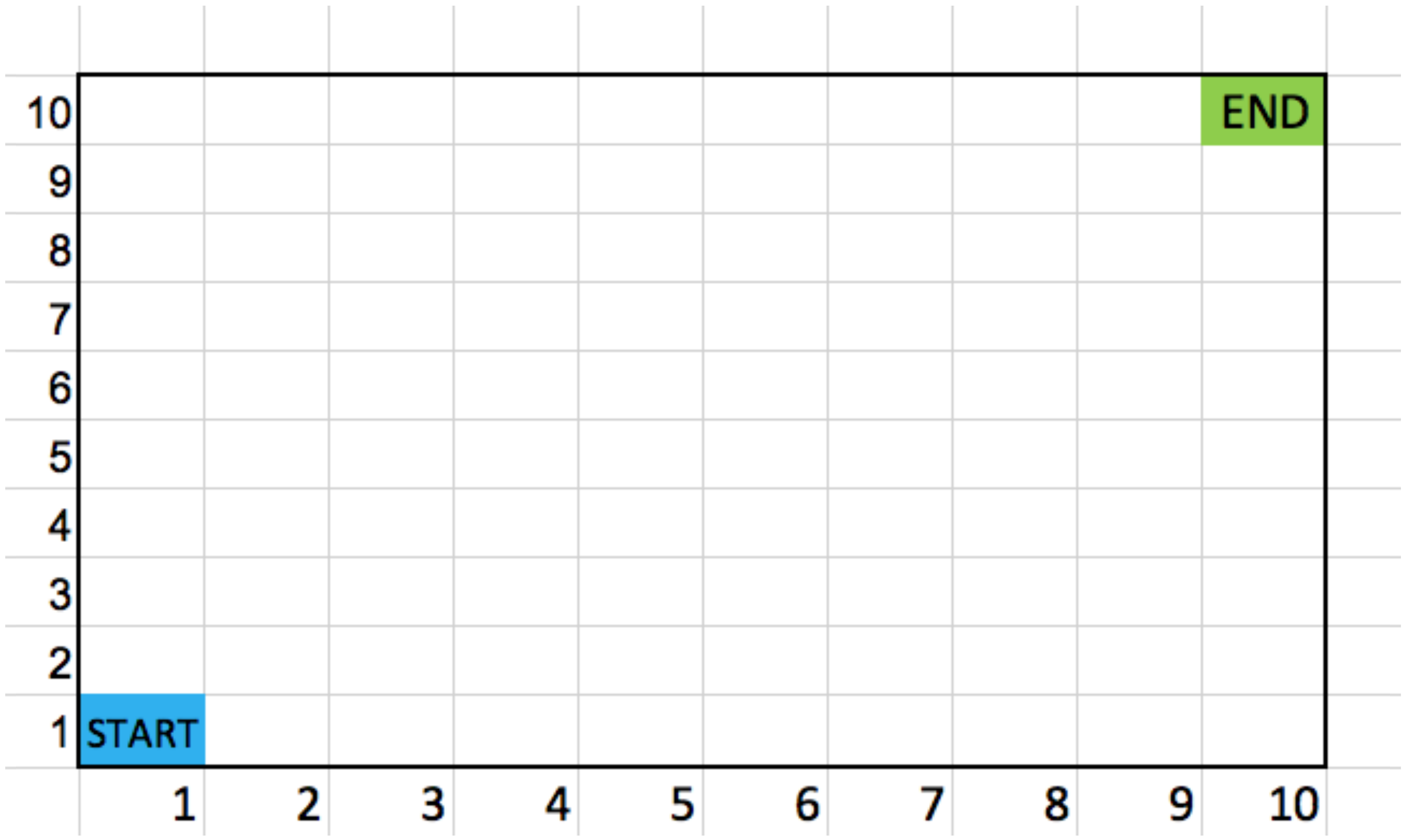
  (3,2) (6,6) (2,8) (5,9) (8,4) (2,4) (6,3) (9,3) (1,9) (3,7) (4,2) (7,8) (2,2) (4,5) (5,6) (10,5) (6,2) (6,10) (7,5) (7,9) (8,1) (5,7) (4,4) (8,7) (9,2) (10,9) (2,6)
- Find a path through the maze

How did you go?

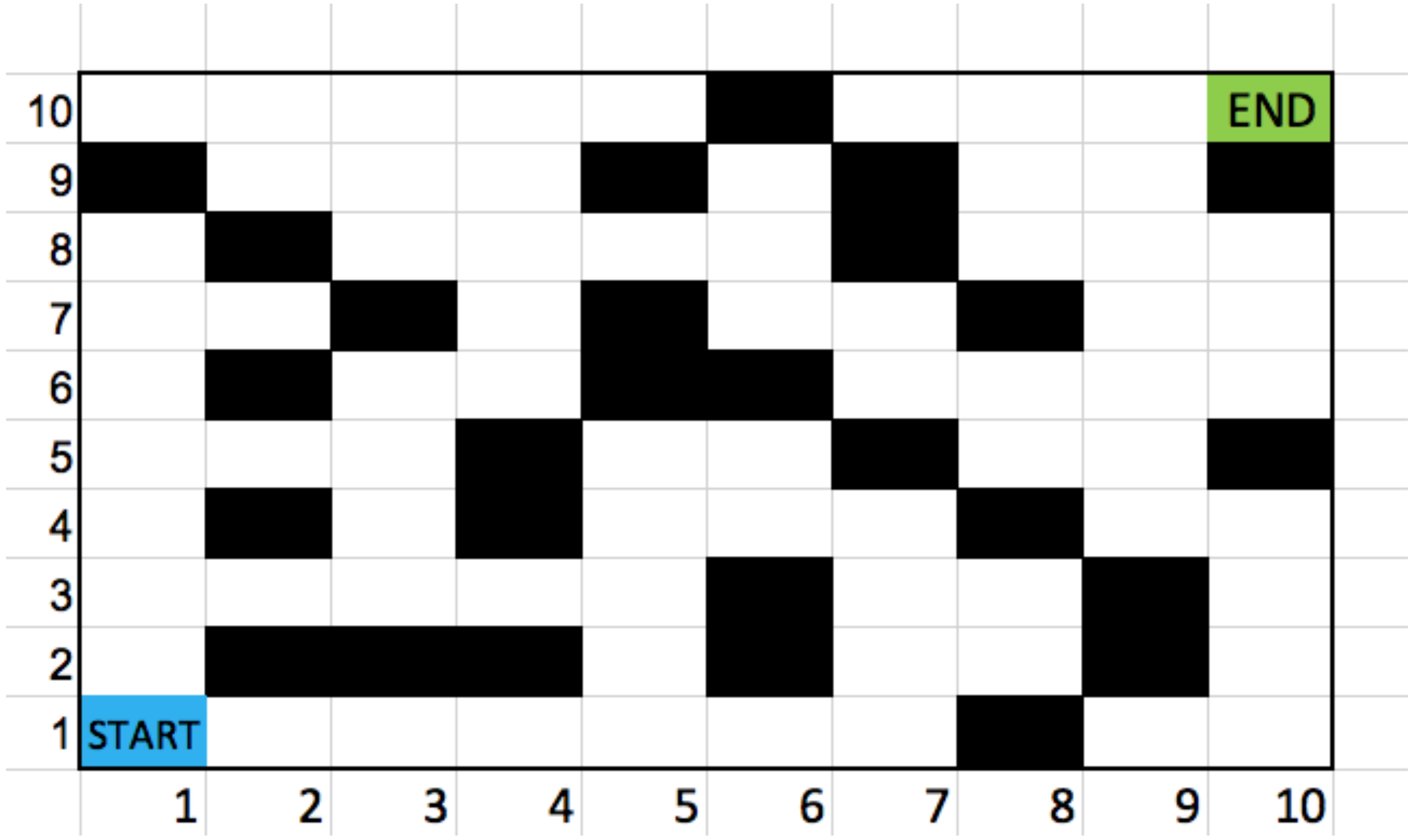
How did you go?



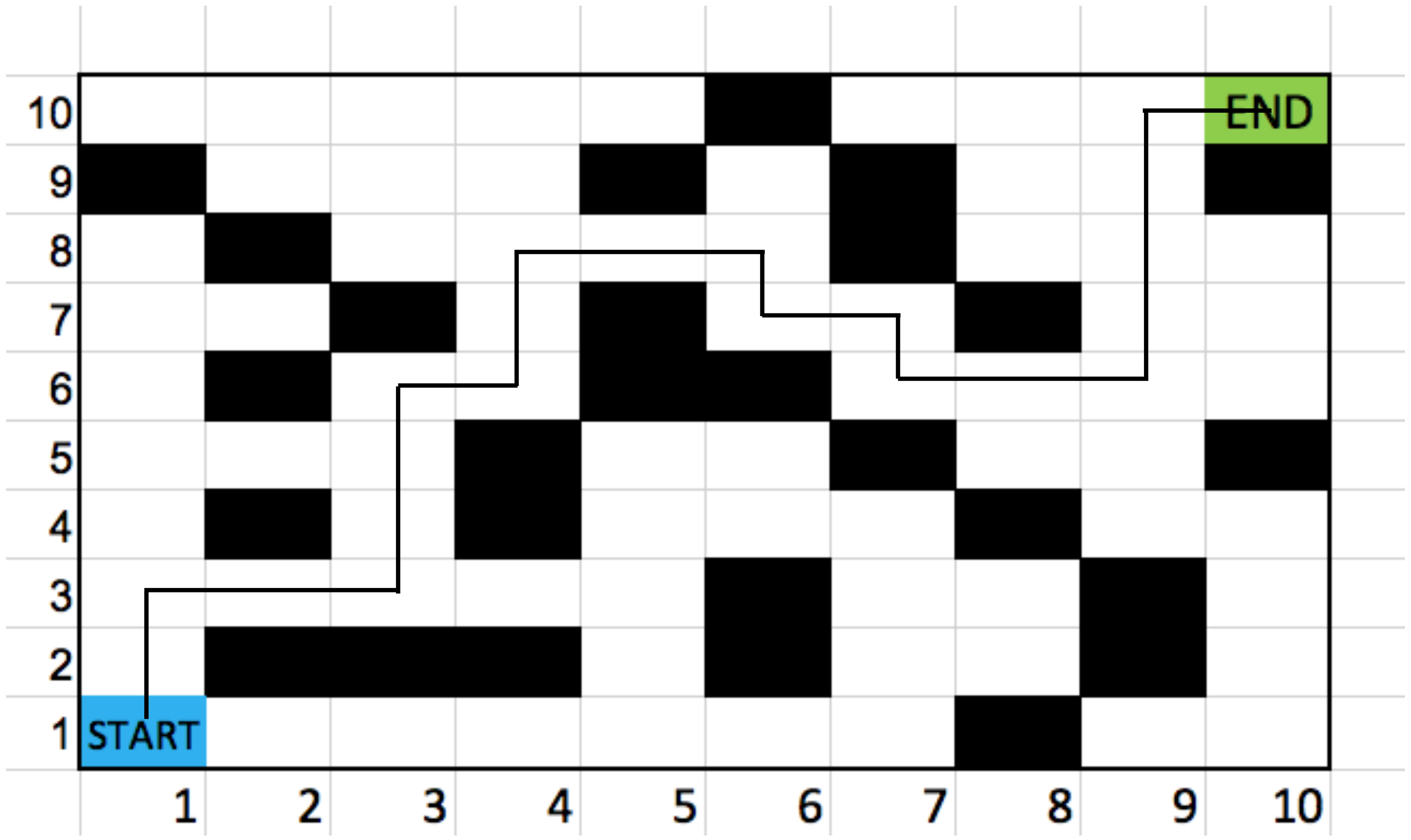
How did you go?



How did you go?



How did you go?



Terminology: Problem

- **Algorithmic** problem is one that can be solved mechanically (i.e. by a computer program)
- Usually we want to find a description of a **single generic program** that can solve a bunch of similar problems
- e.g. the "maze problem" is to come up with a mechanical solution to **any** particular maze
- The maze we solved is called an **instance** of the maze problem
- A problem may have many (even infinitely many) instances
- **Algorithmic problem:** a **family** of **instances** of a general problem

Algorithmic Problems

- **Algorithmic problem:** a **family** of **instances** of a general problem
- An **algorithm** for a problem has to work for all possible instances (i.e. for all possible inputs)
- **Example:** the **sorting** problem
  - Instance is a sequence of items to sort
- **Example:** graph colouring problem
  - Instance is a graph
- **Example:** equation solving problem
  - Instance is a set of, say, linear equations

Terminology: Algorithm

- Dictionary definition: “process or rules for (esp. machine) calculation etc.”
- A finite sequence of instructions, with
  - **no ambiguity:** each step is precisely defined.
  - It should work for all (well-formed) inputs
  - It should finish in a finite (reasonable) amount time
- The (single) description of a process that will transform arbitrary input to the correct output—even when there are infinitely many possible inputs
- Like a cookbook recipe? Sort of, but more like a **general “method”**, a systematic approach that works for **any** instance

Algorithm: Euclid's

- Once, "algorithm" meant "numeric algorithm".
- Mathematicians developed many clever algorithms for solving all sorts of numeric problems
- The following algorithm calculates the **greatest common divisor** of positive integers *m* and *n*, which we write as *gcd(m,n).* The algorithm is called **Euclid's Algorithm.**
- To find *gcd(m,n):*
  - **Step 1:** if $n = 0$, return the value of *m* as the answer and stop
  - **Step 2:** Divide *m* by *n* and assign the remainder to *r*.
  - **Step 3:** Assign the value of *n* to *m*, and the value of *r* to *n*; go to Step 1.

Example: gcd

Let’s run this on some example inputs:

```
$Euclid(m,n)$ =
    while $n \neq 0$
        $r \leftarrow m \bmod n$
        $m \leftarrow n$
        $n \leftarrow r$
    return $m$
```

r	m	n
	24	60
24	60	24
12	24	12
0	12	0

Example: gcd

Let's run this on some example inputs:

```
$Euclid(m,n) =$
    while $n \neq 0$
        $r \leftarrow m \bmod n$
        $m \leftarrow n$
        $n \leftarrow r$
    return $m$
```

r	m	n
	171	7
3	7	3
1	3	1
0	1	0

Non-Numeric Algorithms

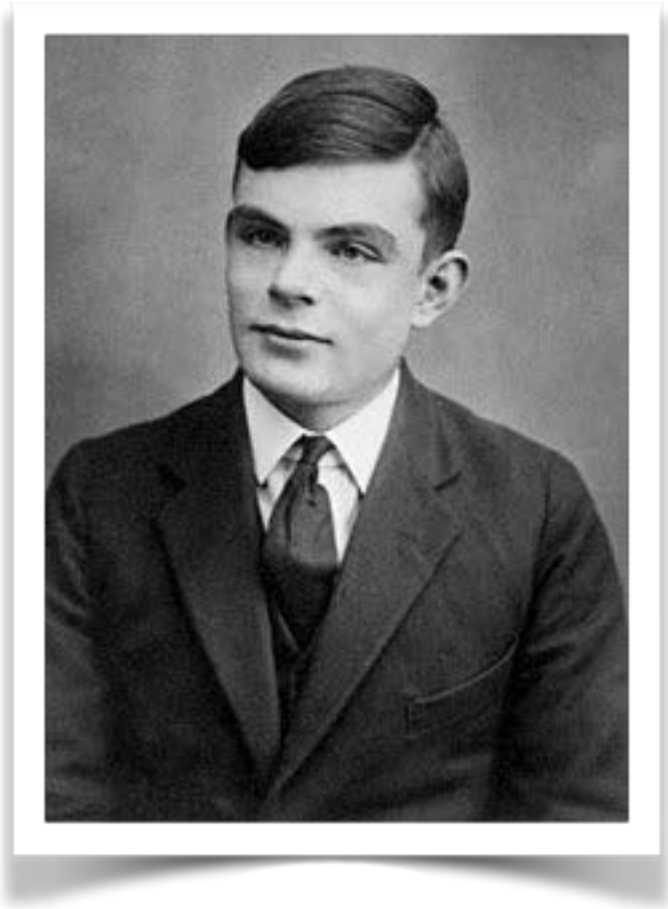
- 350 years ago, Thomas Hobbes, in discussing the possibility of automated reasoning, wrote:

  “We must not think that computations, that is, ratiocination, has place only in numbers.”

- Today, numeric algorithms are just a small part of the syllabus in an algorithms course

- (The kind of computations that Hobbes was really after was mechanised reasoning, that is, algorithms for logical formalisms, for example to decide “does this formula follow from that?”

Computability



- 2012: Alan Turing's 100th birthday
- When Turing was born, "computer" meant a human who was employed to do tedious numerical tasks
- Turing's legacy: "Turing machine", "Church-Turing thesis", "Turing reduction", "Turing test", "Turing Award".
- One of his great accomplishments was to put the concept of an **algorithm** on firm foundations and to establish that certain important problems **do not have algorithmic solutions**

Abstract Complexity

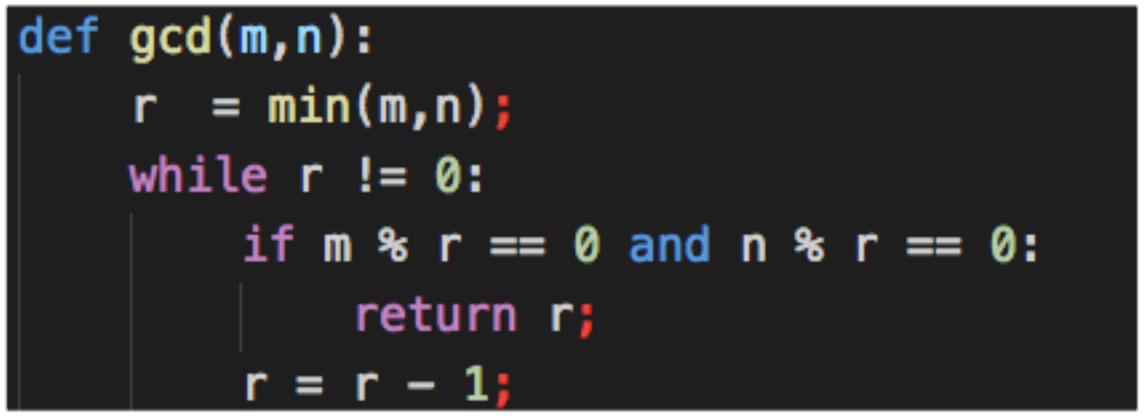
- In this course, we are interested only in problems that have algorithmic solutions
- However, amongst those, there are many that provably do not have **efficient** solutions
- Towards the end of the subject, we briefly discuss **complexity theory**—this theory is concerned with the inherent “hardness” of **problems**

Why Study Algorithms?

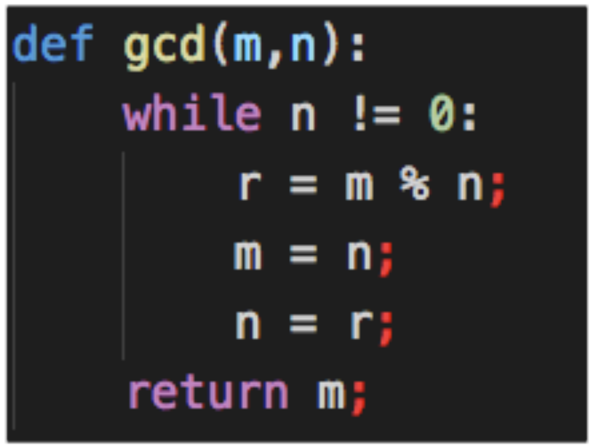
- Computer science is increasingly an enabler for other disciplines, providing useful tools for these
- Algorithmic thinking is relevant in the life sciences, in engineering, in linguistics, in chemistry, etc.
- Today computers allow us to solve problems whose size and complexity is vastly greater than what could be done a century ago
- The use of computers has changed the focus of algorithmic study completely, because algorithms that work for a human (small scale) usually do not work well for a computer (big scale)

Algorithm Complexity

Two implementations of gcd:



```
def gcd(m,n):
    r  = min(m,n);
    while r != 0:
        if m % r == 0 and n % r == 0:
            return r;
        r = r - 1;
```



```
def gcd(m,n):
    while n != 0:
        r = m % n;
        m = n;
        n = r;
    return m;
```

Time to compute gcd(2147483647,2147483646)

3 minutes 19 seconds   0.033 seconds

We would like to **predict** how long an algorithm will take to run as the size of its input increases.

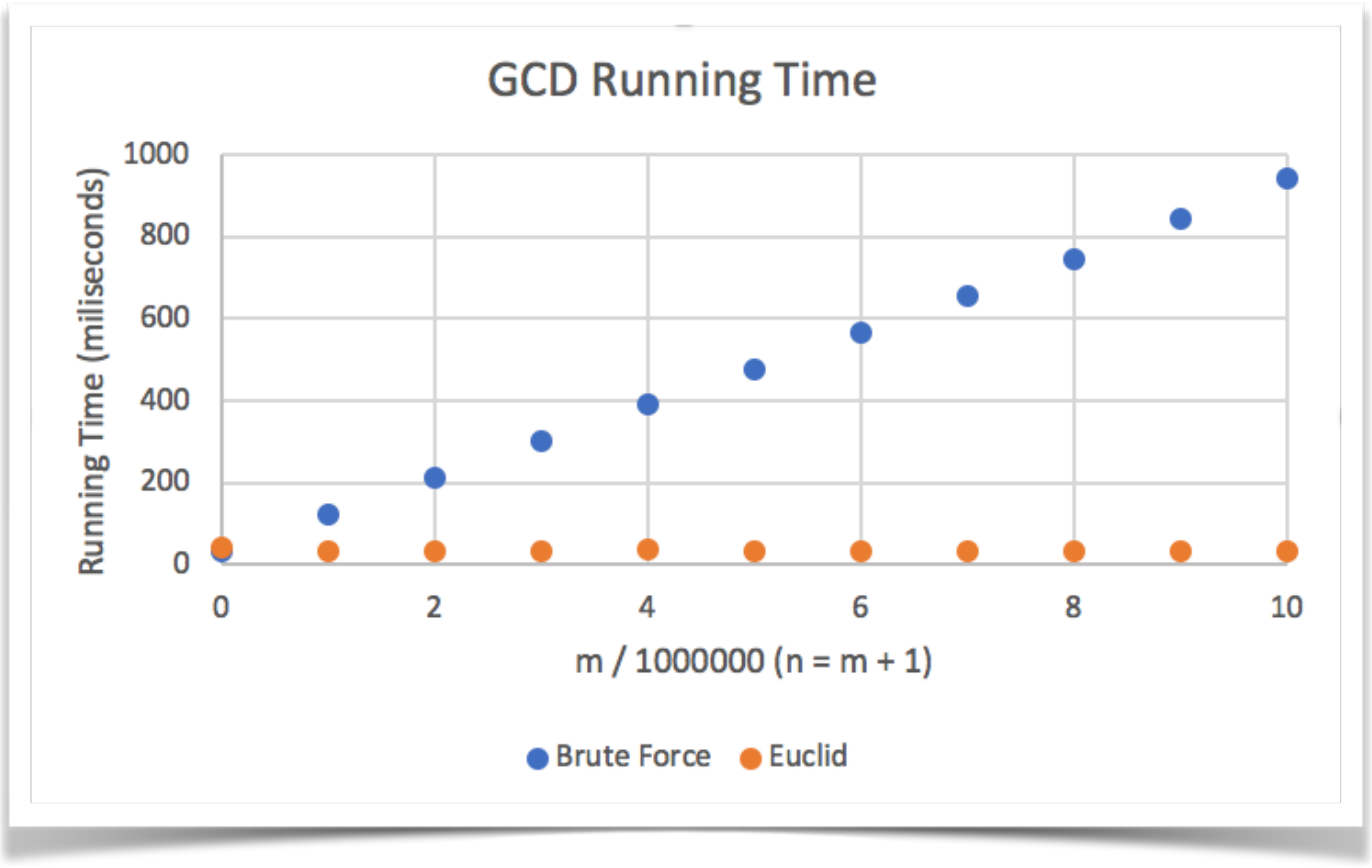
Studying Algorithms and their Complexity

- To collect useful problem solving tools
- To learn, from examples, strategies for solving computational problems
- To be able to write robust programs whose behaviour we can reason about
- To develop analytical skills
- To learn about the inherent difficulty of some types of problems

Problem Solving Steps

1. Understand the problem
2. Decide on the computational means (sequential vs parallel, exact vs. approximate)
3. Decide on the method to use (the algorithm design technique or strategy)
4. Design the necessary data structures and algorithm
5. Check for correctness, trace example input
6. Evaluate analytically (time, space, worst case, average case)
7. Code it
8. Evaluate it empirically

Empirical Evaluation of gcd



What we will study

- Algorithm analysis
- Important algorithms for various problems, namely:
  - Sorting
  - Searching
  - String processing
  - Graph algorithms
- Approaches to algorithm design:
  - Brute force brute force gcd
  - Decrease and conquer Euclid's Algorithm
  - Divide and conquer
  - Transform and Conquer
  - …

Study Tips

- **Before** the lecture, as a minimum make sure you read the introductory section of the relevant chapter
- Always read (and work) with paper and pencil handy; run algorithms by hand
- Always have a go at the tutorial exercises; this subject is very much about learning-by-doing
- **After** the lecture, reread and consolidate your notes
- Identify areas not understood, use the LMS discussion board
- Rewrite your notes if that helps

Things to do First

- Take the two Week 1 diagnostic quizzes on the LMS
- Get the textbook, read Chapter 1, skim Chapter 2
- Make sure you have a unimelb account
- Visit COMP90038 LMS page, check the weekly schedule, any new announcements
- Use LMS discussion board, e.g. if you're interested in forming a study group with like-minded people, the Discussion Board is a useful place to say so