ALGORITHMS & DATA STRUCTURES SET08122

LECTURE 02: *ALGORITHMS & COMPLEXITY*

(& MAYBE A LITTLE COMPUTABILITY)

Dr Simon Wells
s.wells@napier.ac.uk
http://www.simonwells.org

TL/DR

- Not every problem is solvable using a computer. Computer Science is all about working out what the characteristics and performance of problems that are (not) solvable by computers.

Edinburgh Napier UNIVERSITY

At the end of this lecture you will be able to:

- Inspect code & roughly determine the order of complexity of its computations
- Describe the features & function of the Turing Machine
- Understand some of the limits of computation

OVERVIEW

- Algorithms
- An introduction to complexity (time & space)
- Some practical methods for determining complexity
- A tiny digression into computability

ALGORITHMS?
WHAT ARE THEY?
WHAT ARE THEY FOR?

Edinburgh Napier
UNIVERSITY

A LIST OF INSTRUCTIONS THAT CAN BE FOLLOWED TO SOLVE A PROBLEM.

ALGORITHMS

- This is one of those areas where there is overlap between computers, computer science, and mathematics
- An *unambiguous* specification of how to solve a (class of) problems
- We have algorithms for
  - Calculating results
  - Data processing

    (two ways of using algorithms that we're used to)
  - But also *Artificial Intelligence* - uses algorithms (Path-finding, Machine Learning, Neural Nets)

ALGORITHMS

- An effective method expressed within a finite amount of space & time using a well-defined formal language for calculating a function
- Start in an initial state (with an input [might be none])
- Instructions describe a computation
- When executed there are a finite number of well-defined successive states that eventually produce an *output* and the computation terminates at a final ending state
- NB. Transitions between states need not be deterministic
  - If you're interested take a look at **Bloom Filters** which incorporate a degree of randomness within the algorithm - It's a *probabilistic* data structure. Useful if amount of data requires an impractical amount of memory

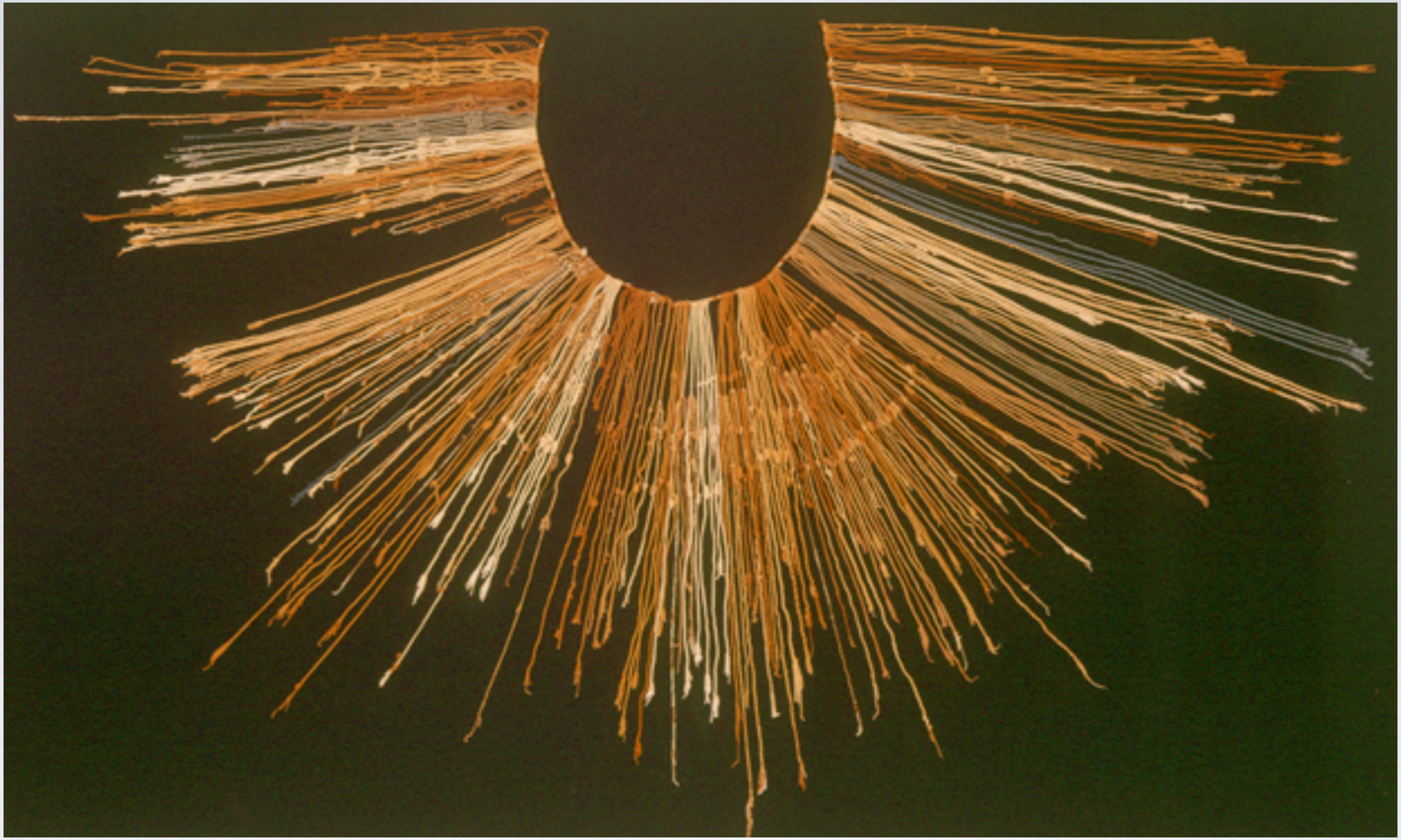
Edinburgh Napier
UNIVERSITY

EFFECTIVE METHODS

- Finite number of exact finite instructions
- When applied to a problem from its class:
  - It always terminates after a finite number of steps
  - It always produces a correct answer
- Note: This is getting us to the edge of hard computer science questions like "what is computable?"
- *In principle, a person could do the work by hand…*
  - *only need to follow the algorithm rigorously*

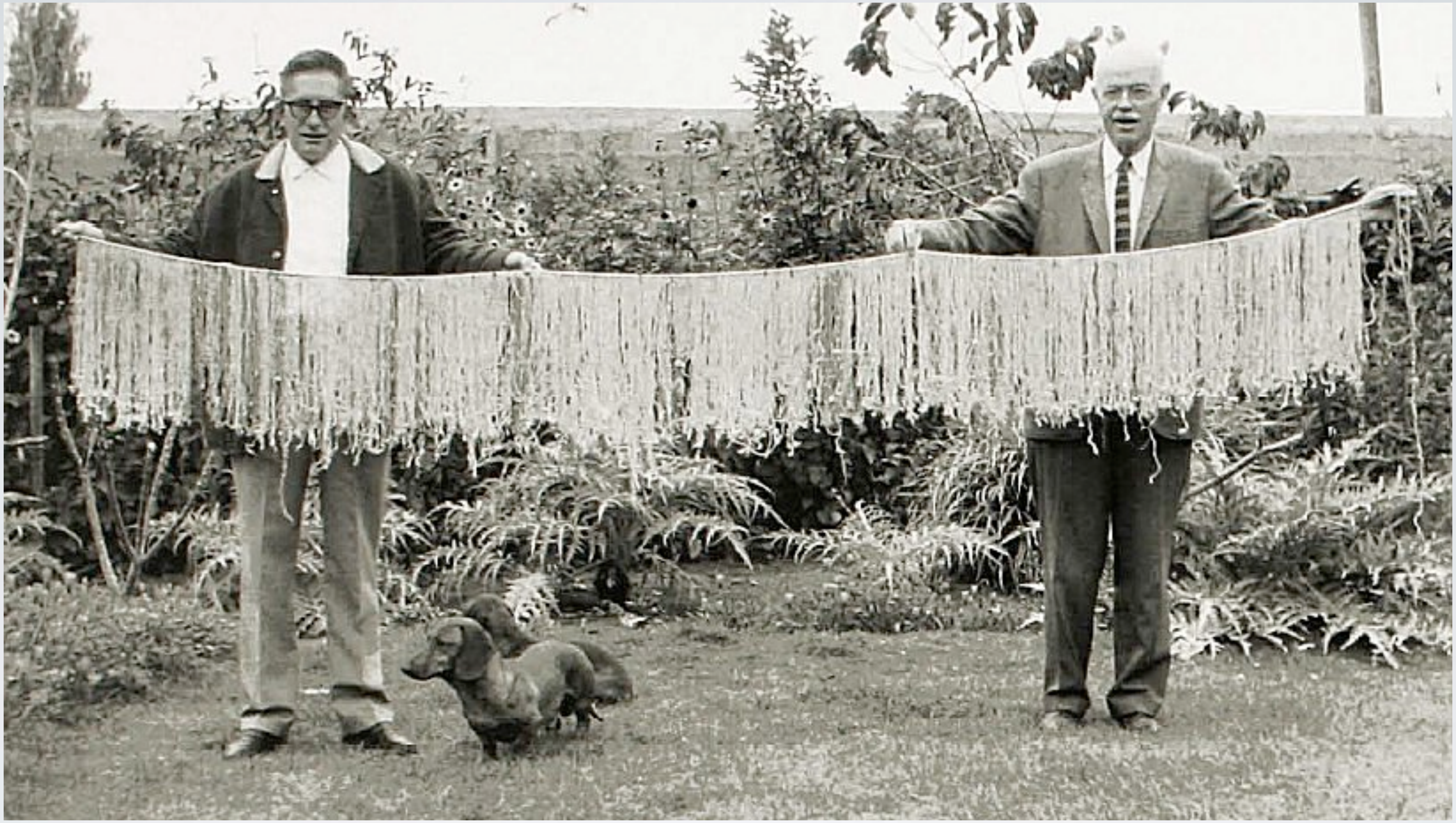
HISTORY

- Still some discussion of formal definition of *algorithm* and what it means to be computable
  - Concept been around for centuries (at least back to Euclid)
  - Hilbert (1928) Entscheidungsproblem (decision problem)
  - Logicians refined the problems, e.g. Godel, Herbrand, Kleene ('30s)
  - Alonzo Church (1936) Lambda Calculus
  - Alan Turing Turning



QUIPUS

Recording information with knotted ropes



QUIPUS CAN GET QUITE BIG

CLASSIFYING ALGORITHMS

BIG OH NOTATION

- Big Oh is just fancy sounding words for insight & practices that many progressional developers know & use (often without realising it)
- Big Oh refers to the "order" associated with the performance, i.e. the degree of complexity , so O(n) is read "The order of *n*"
- O really refers to the Order function
- A function's Big Oh notation is generally determined by how it responds to different inputs
  - e.g. How much slower is this function if we give it 1,000,000 items instead of 1 item?
- Essentially we are *approximating* orders of magnitude
  - i.e. Does the algorithm run in constant time, linear time, quadratic time, logarithmic time?
- This lets us predict how a given algorithm will perform for a given input size

OTHER NOTATIONS

- Big Oh gives the upper bound
- Big Ω (Omega) gives the lower bound
- There is Big ϴ (Theta) notation to asymptotically bound the growth to within constant factors above and below
- Important because a single notation doesn't always give the full story
- Each notation can also be used to reason about best, worst, & average cases

CALCULATING #1

- We take measurements of how an algorithm performs
- Graph the results (where *n* the number of items corresponds to the x axis)
- Match the curve to known performance curves
- Dealing with worst case scenarios (Can chart the upper & lower bounds which yields a)
- We graph the n in O(n) where *n* corresponds to x axis

CALCULATING #2

```
def count_ones(a_list):
    total = 0                    Constant time O(1)
    for element in a_list:       Linear time O(n)]
        if element == 1:         Constant time O(1)
            total += 1           Constant time O(1)
    return total
```

- By counting/inspecting operations:

O(1)+O(n) * (O(1) + O(1))

Reduces to O(2n)+O(1)

Only care about biggest terms

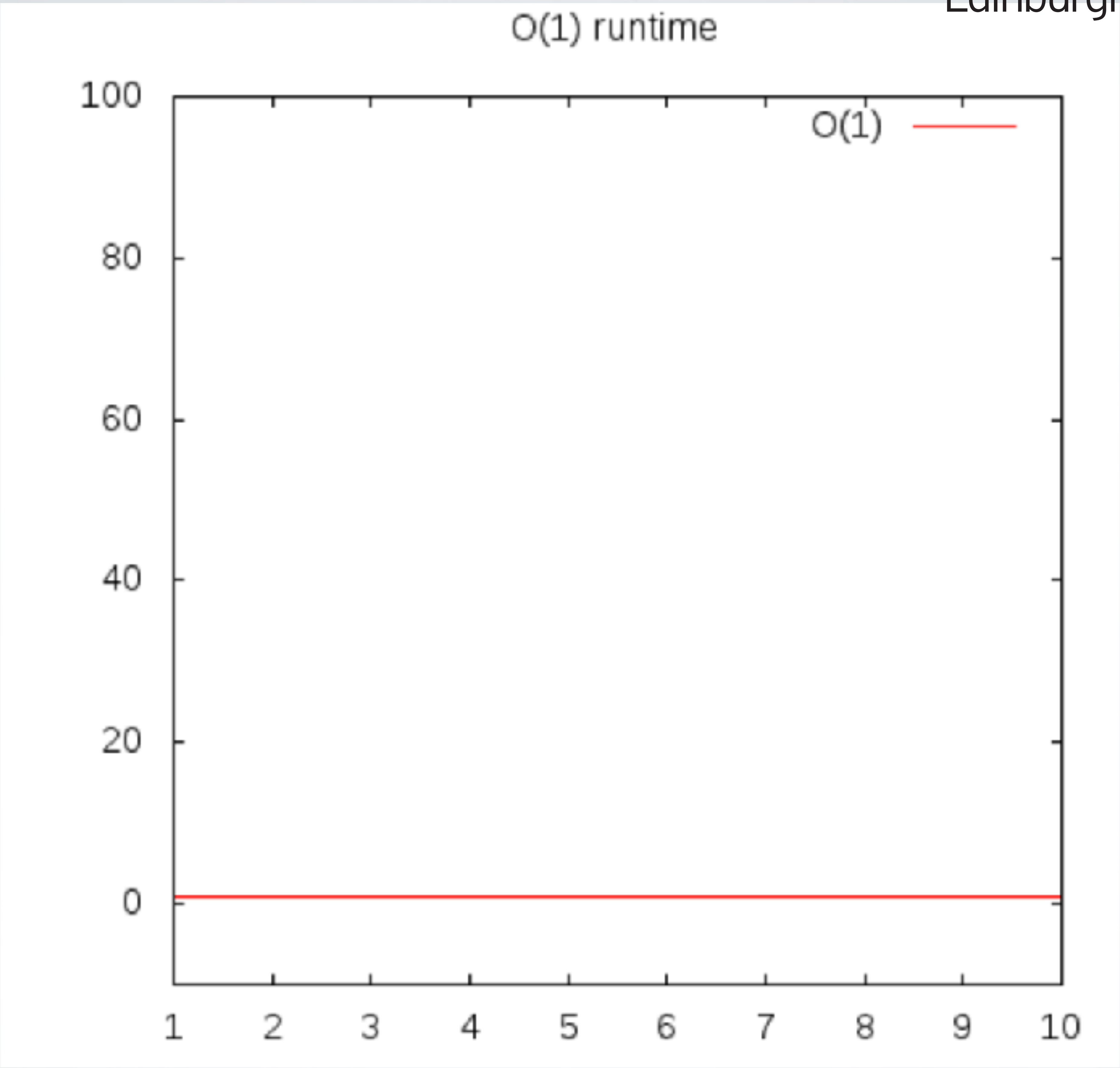
O(2n) isn't much different to O(n)

Count operations, simplify, drop multipliers

CONSTANT TIME

- An algorithm runs in constant time if it requires the same amount of time regardless of input size
- Big Oh Notation/Complexity is O(1)
- No matter how big the input will always take the same amount of time
- Example: Access any element of an array, push & pop to a fixed size stack, Enqueue to & dequeue from a fixed size queue

```
def is_none(item):
    return item is None
```



Edinburgh Napier
UNIVERSITY

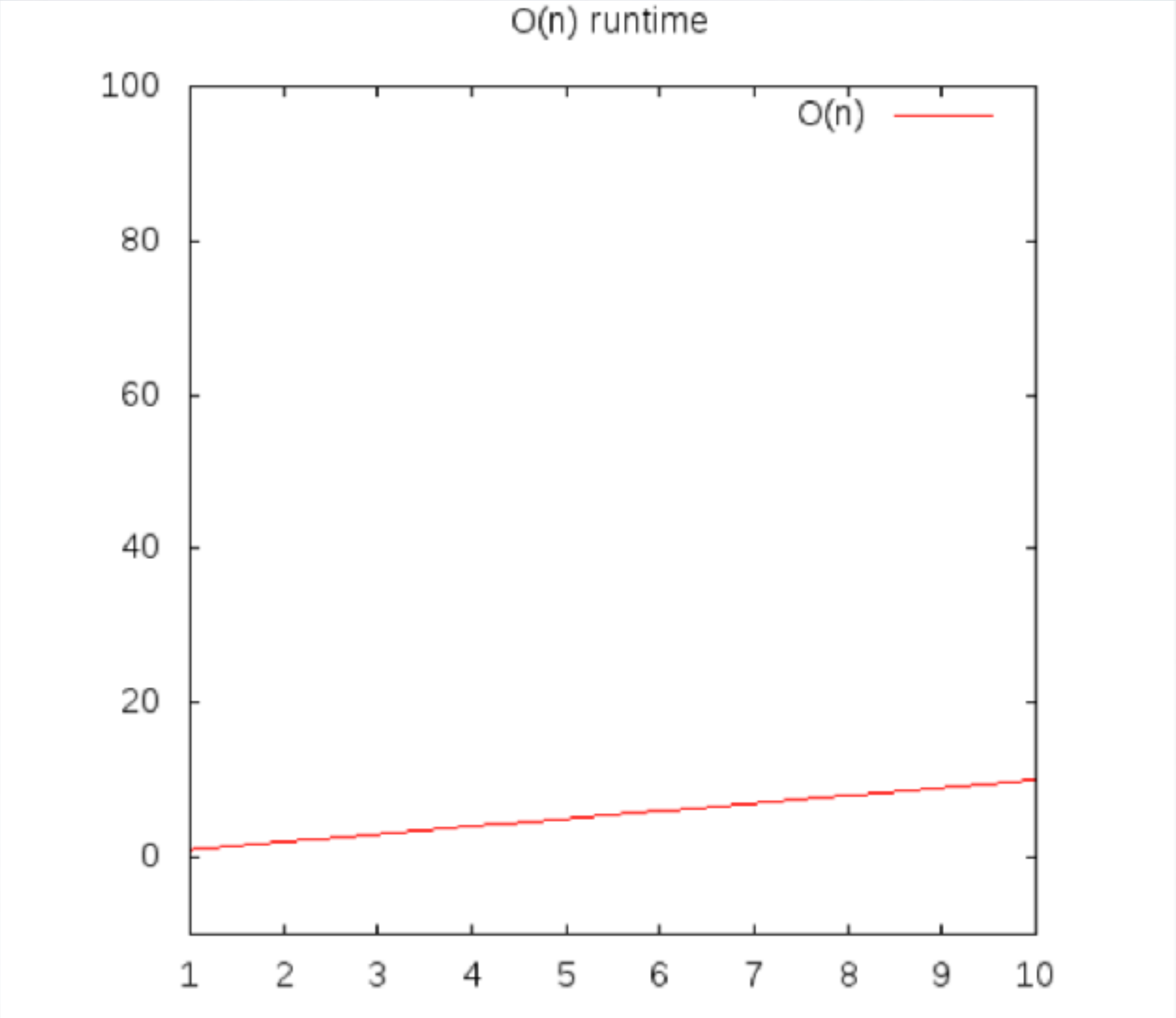
LINEAR TIME

- An algorithm runs in linear time if the time it takes to execute is directly proportional to input size
- Complexity is **O(n)**
- **Examples:**
  - **Array:** Linear search, Traversal, Find minimum
  - **ArrayList:** Contains
  - **Queue:** Contains

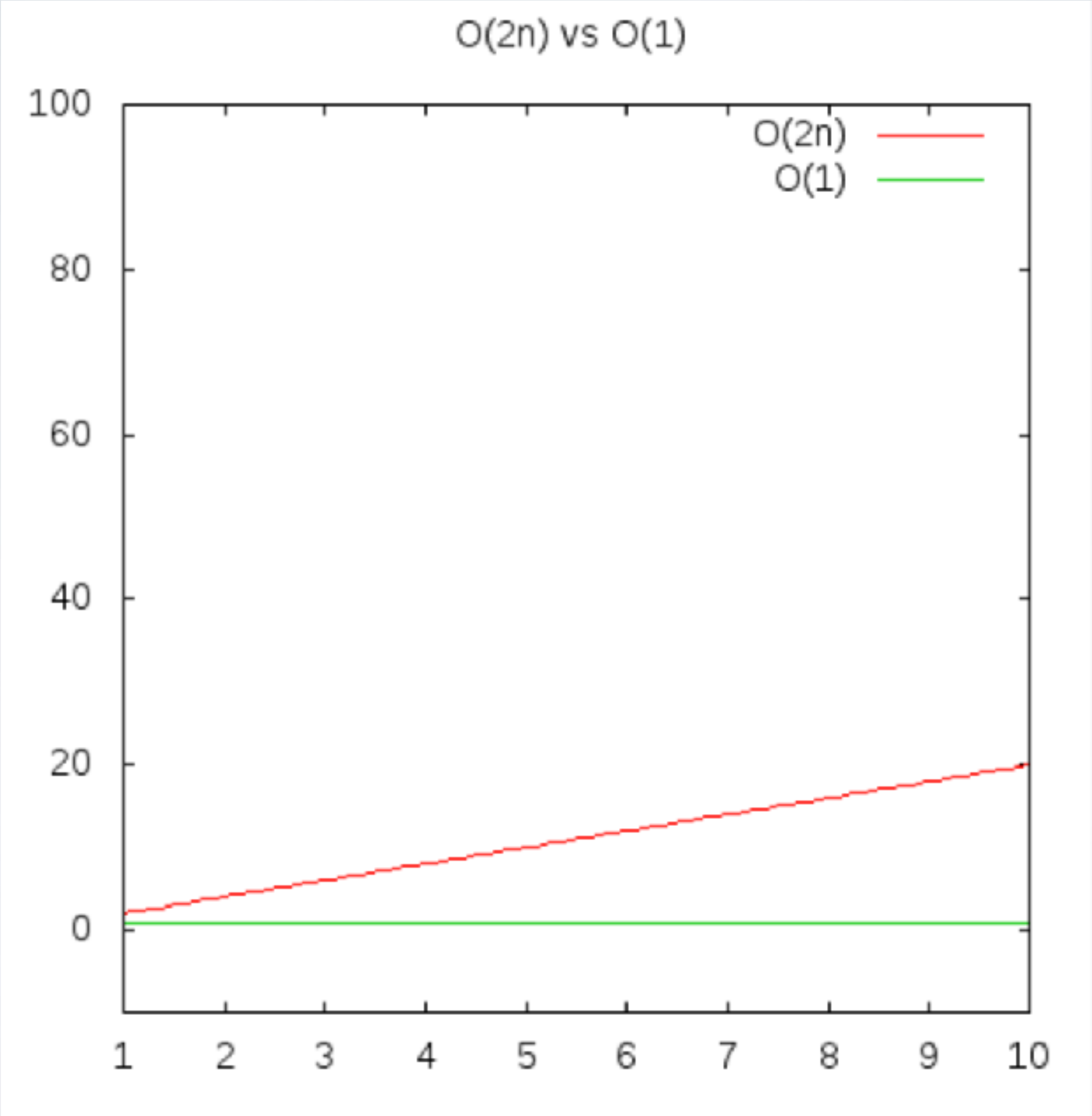
LINEAR TIME

- Call with, e.g. **item_in_list(2, [1,2,3])**
- If we graph the time it takes the function with different sized inputs (arrays) we'd see that this approximately corresponds to the number of items in the array

```
def item_in_list(to_check, the_list):
    for item in the_list:
        if to_check == item:
            return True
    return False
```



O(n) runtime
O(n)
100
80
60
40
20
0
1
2
3
4
5
6
7
8
9
10


O(2n) vs O(1)
O(2n)
O(1)
100
80
60
40
20
0
1
2
3
4
5
6
7
8
9
10

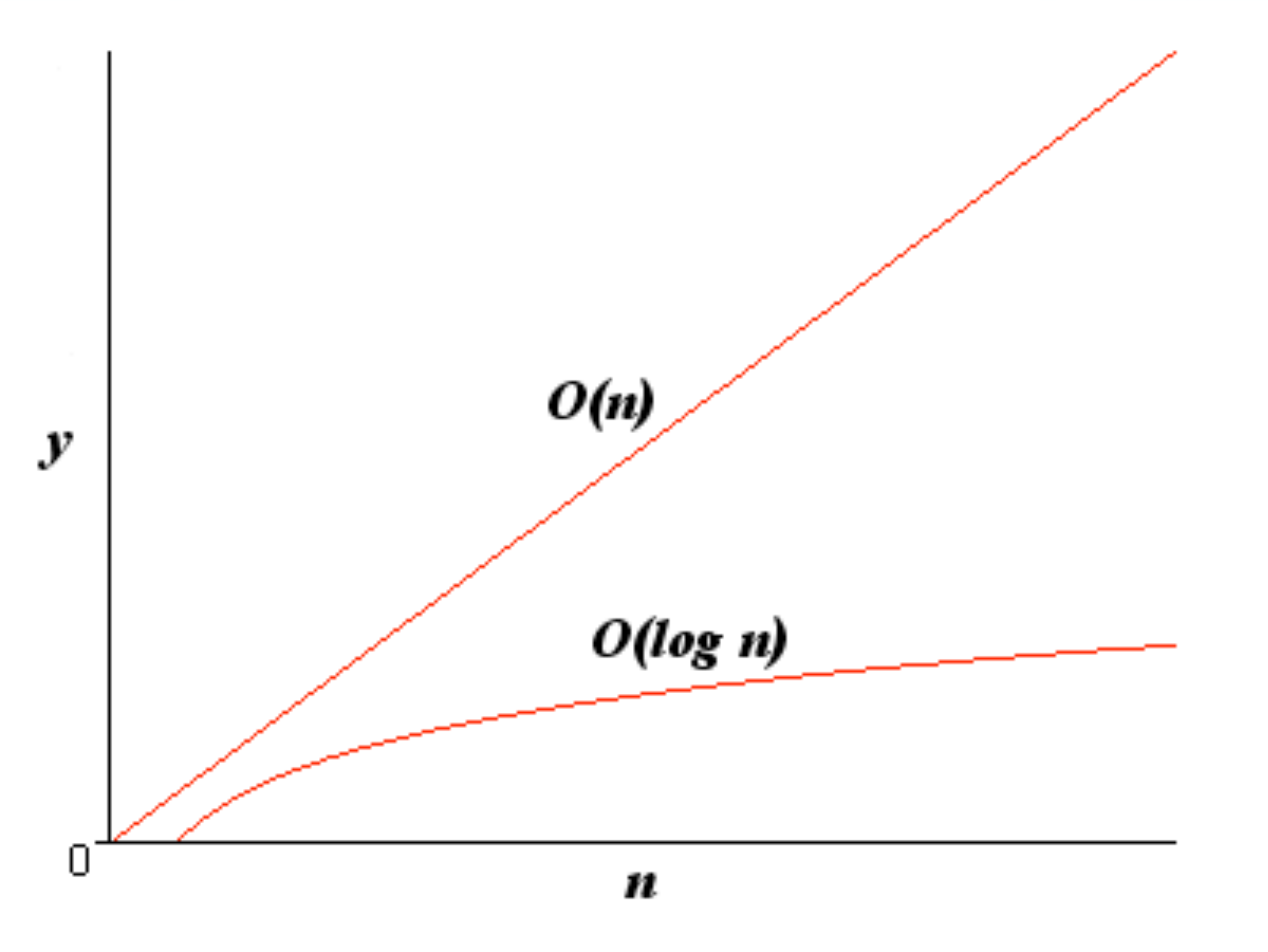
LOGARITHMIC TIME

- If the execution time is proportional to the logarithm of the input size

- A common attribute of algorithms with logarithmic running times is that there is often a choice of new element on which to perform an action & only one needs to be chosen

- Example: Binary Search

- Classical "divide & conquer" scenarios, e.g. looking up someone in the phonebook

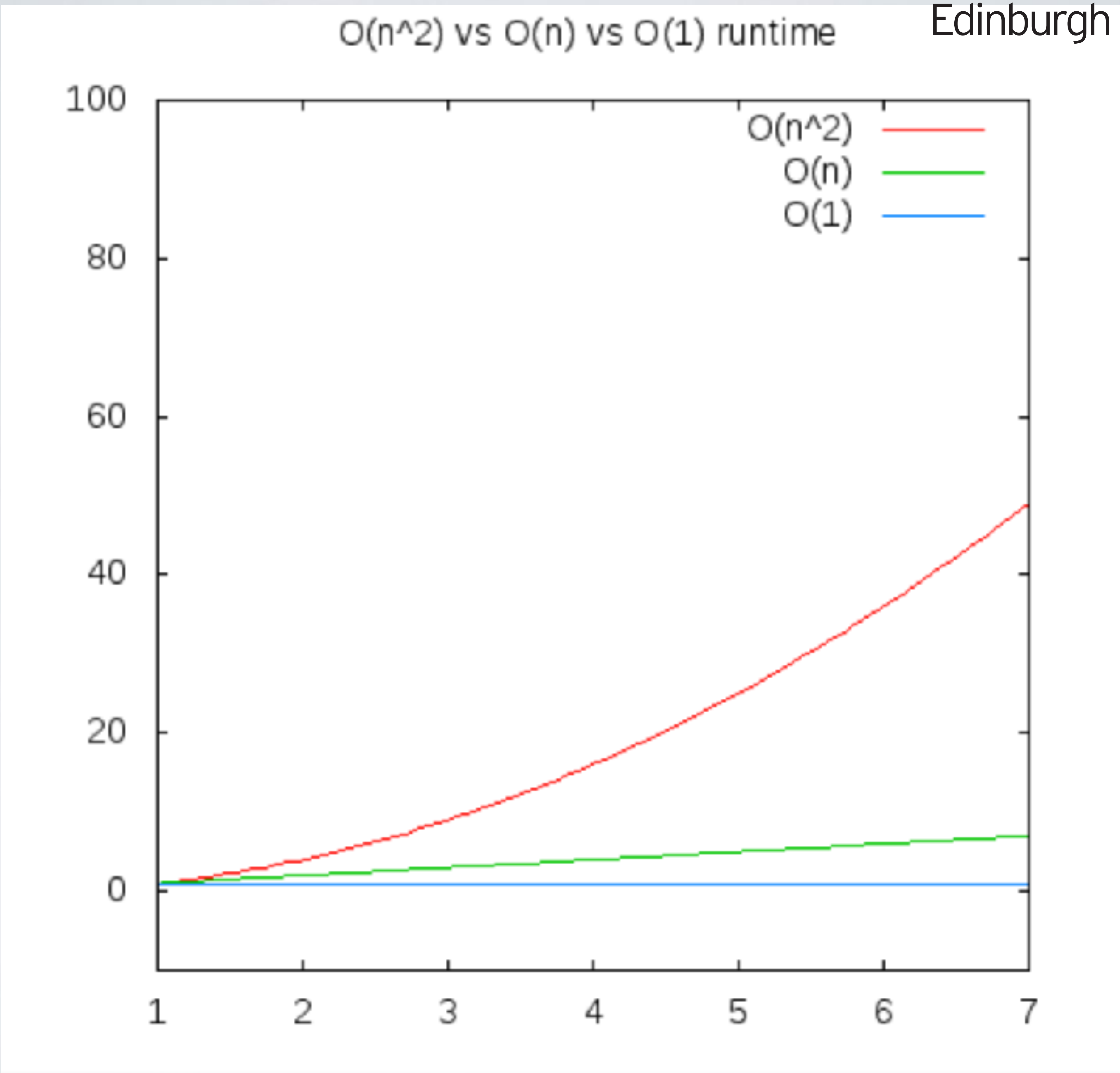


y
O(n)
O(log n)
0
n

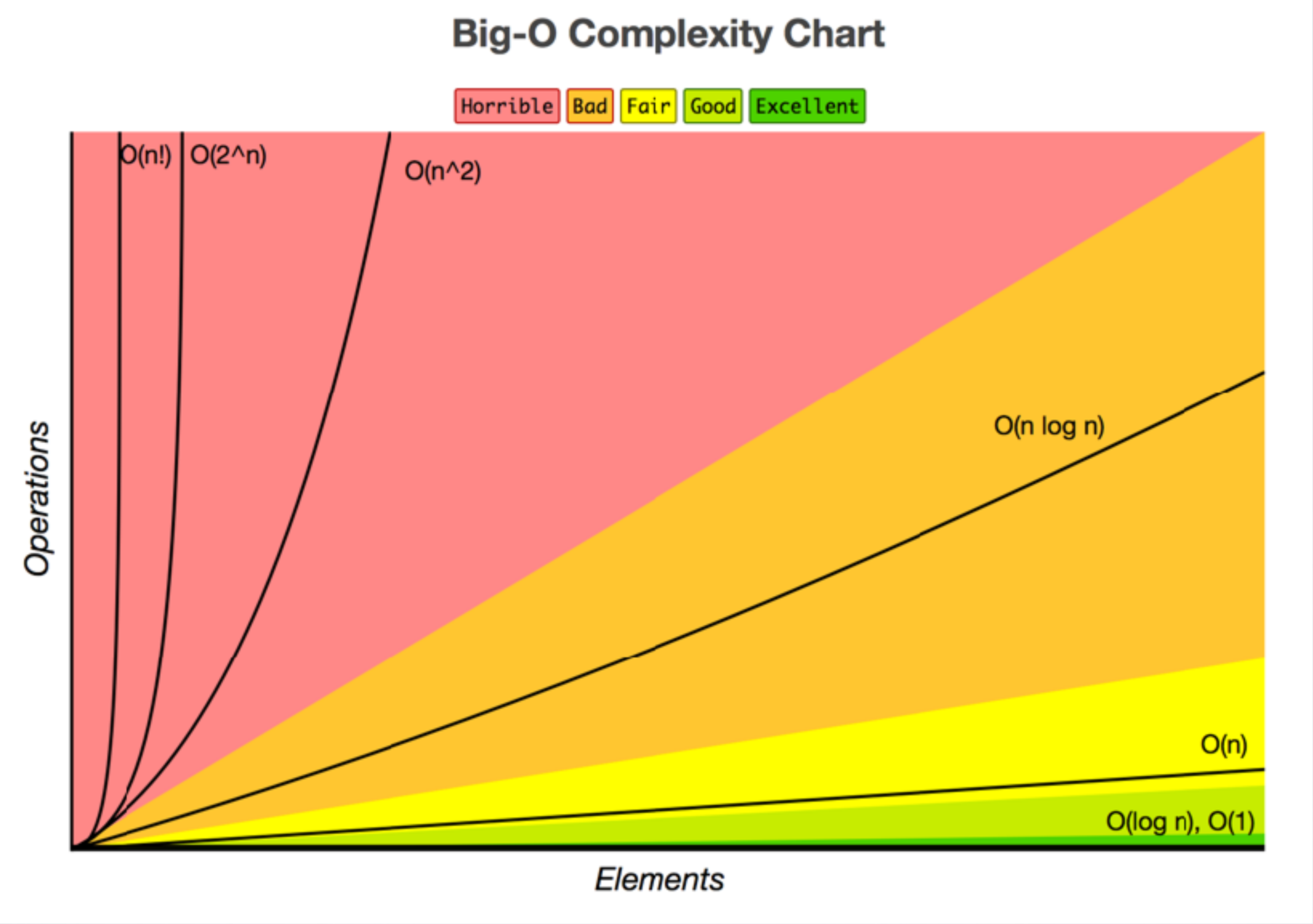
QUADRATIC TIME

- An algorithm runs in quadratic time if its execution time is proportional to the square of the input size
- Given a list, e.g. [1,2,3] get back all combinations:
  - [(1,1) (1,2), (1,3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]
- For every item, n, in the list we have to do n operations
- n * n == n^2, i.e. O(n^2)
- Example: Bubble, Selection, & Insertion sorts

```
def all_combinations(the_list):
    results = []
    for item in the_list:
        for inner_item in the_list:
            results.append((item, inner_item))
    return results
```



Edinburgh Napier
UNIVERSITY



FROM: HTTP://BIGOCHEATSHEET.COM/

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

FROM: HTTP://BIGOCHEATSHEET.COM/

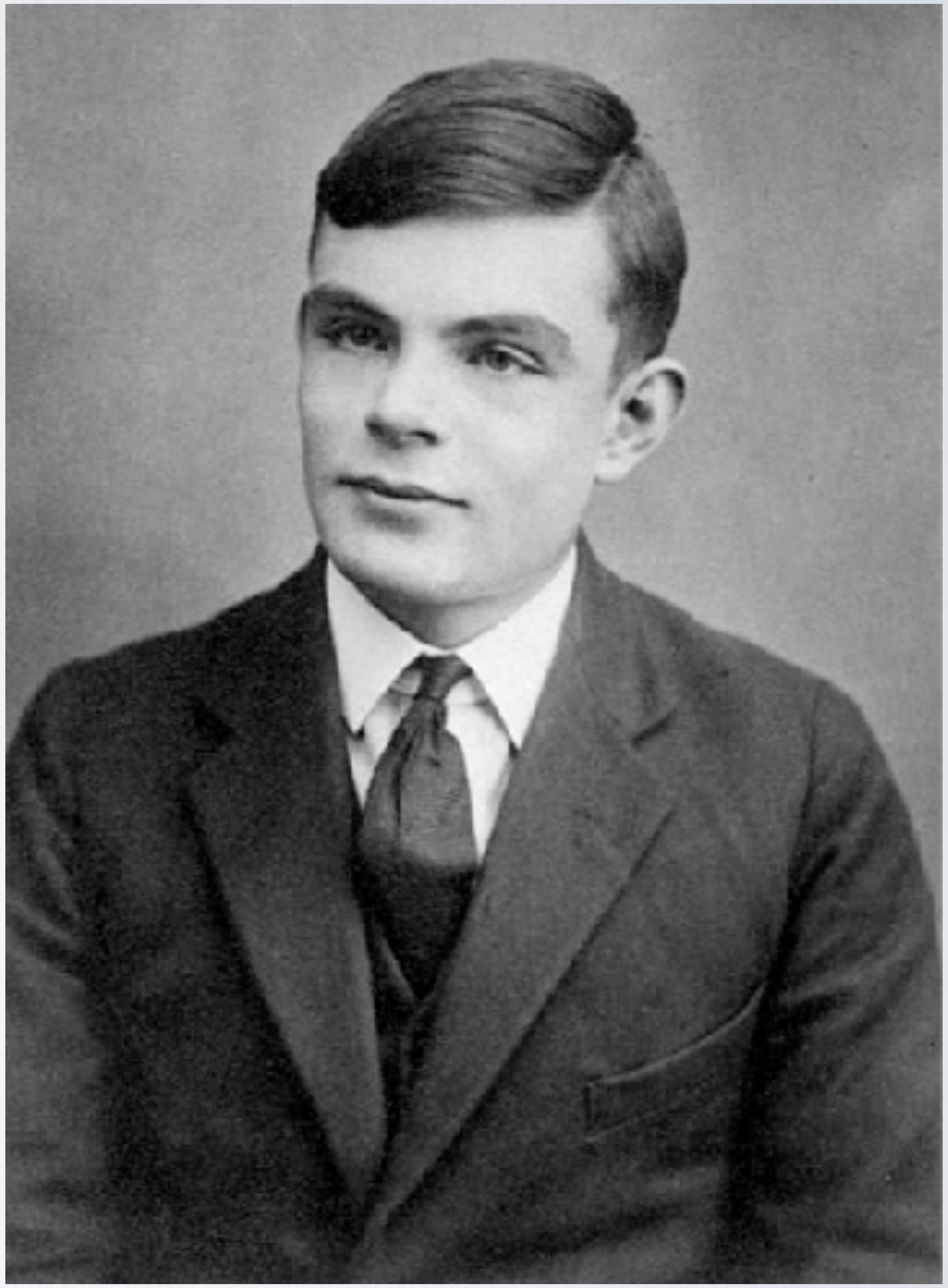
PRACTICAL SKILLS

- Many professional programmers will tell you that they don't use Big O notation and it hasn't been important to the careers.
- Perhaps in the most literal sense they are correct - they haven't specifically said that something has a big O or big Omega or big Theta value, but…
  - … they do, through experience, develop a sense for how long certain tasks take, how much memory is needed, whether a given problem is tractable on the resources available. &.c
- However, ask any programmer how they evaluate & optimise their code they will talk about things like
  - Input size & looping as indicators of where a program will spend time computing
  - Profiling (using tools to determine where your program spends time)

COMPUTABILITY

ALAN TURING (1912-54)

- English Computer Scientist (before computers really existed)
- Father of both theoretical Computer Scientist & Artificial Intelligence (also worked in Mathematics, Philosophy & Theoretical Biology)
- Worked for GCHQ during WW2 performing cryptanalysis
- Post WW2 worked on the National Physics Laboratory ACE & the Manchester computers (SSEM, Baby, &c.)
- Replaced Gödel's formal language describing results on the limits of proof & computation with a simple hypothetical device: A formal description of a computational device that became known as a Turing Machine.



TURING'S WORK

- Started work on the **Halting Problem** (we'll get to that) in 1936
- Proved that you cannot create a program that solves (gives an answer) to the Halting Problem for all possible inputs
- Elements of the proof of this specified a mathematical definition of a computer and program - this became the known as the **Turing Machine**
- Significant because one of the first problems proven to be unsolvable.

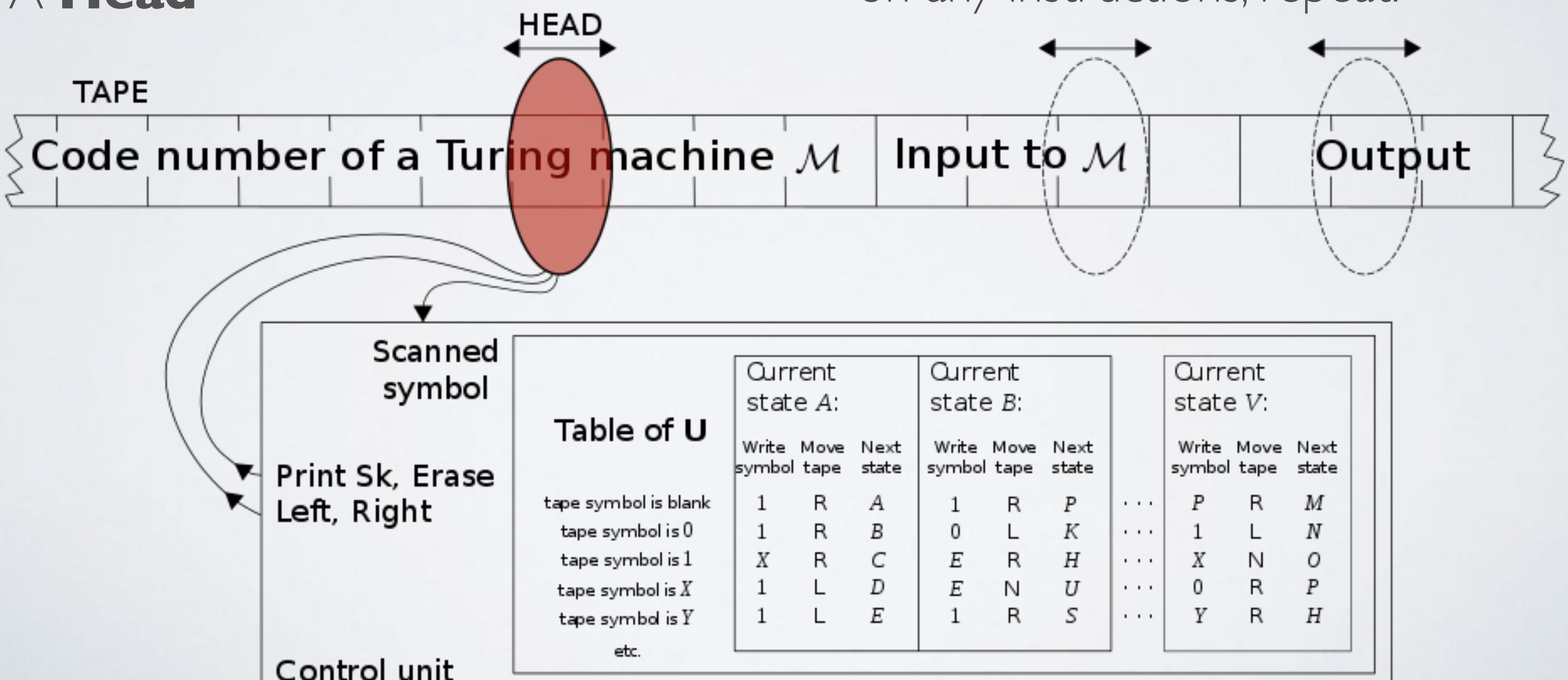
TURING'S MODEL OF COMPUTATION

- Theoretical mathematical model of computation:
    - **An abstract machine that manipulates symbols on a strip of tape according to a set of rules**
- *Recall we saw another model earlier in the von Neumann architecture*
- Simpler but paradoxically also more powerful
    - e.g. removes practical issues such as bus (& bottleneck)
    - Can simulate any computer algorithm
    - Strictly a **Universal** Turing Machine - A Turing Machine that can take as input a description of another Turing Machine
- von Neumann more than sufficient for most reasoning about data structures & algorithms (close to reality) but Turing necessary to study limits of computation NB. von Neumann very likely influenced by Turing in design or earliest physical computers
- Strictly impossible to implement (we cannot build anything of infinite size)

Edinburgh Napier UNIVERSITY

TURING MACHINE

- A **Tape** of infinite length (*think of tape is being like memory with each cell laid out next to the other in a long line*)
- A **Head**
- A **State Register**
- A finite **Table of instructions**
- Read tape, decode information, act on any instructions, repeat.



HALTING PROBLEM

- Determining from a description of an arbitrary program & an input whether the program will finish running or continue forever
- Phrased in terms of Turing machines:
  - Given a description of a turing machine & initial input, asks whether the program, when executed on the input, will halt (complete) or continue forever.
  - Been shown that not possible to construct a Turing machine that can answer this question
    - e.g. have a function halts() into which we pass a program. Function then returns true if halts & false otherwise
  - Only way to know for certain is the run the machine & see what happens. If it halts then you know it halts,, otherwise…?
  - Example of an **undecidable** or **non-computable** problem
- An instance of a class of problems called **decision problems**

IMPORTANCE OF THE HALTING PROBLEM

- Many computing science (& mathematical) problems are instances of the halting problem in disguise (i.e. they can be reconfigured or *generalised* into a version of the halting problem)
- Halting problem is equivalent to asking:
  - "Does this computer program ever stop?"
  - "Does this computer program have any security vulnerabilities?"
- If had halts() then could prove/disprove nearly every open math problem
  - Does an odd perfect number exist?
  - NB. Riemann hypothesis, Goldbach conjecture, Poincare conjecture

THOUGHT EXPERIMENT

- You have various apps on your mobile device
- Think of each app as a Turing machine
- Sometimes an app crashes your phone because they get caught in a loop and never halt
- Dev team releases an app that checks for this (checker app)
- Checker app takes another app as input, If apps stops then checker app accepts it but If app loops then checker rejects it
- App dev create an app called paradox. It loads the checker app then loads itself into the checker app, e.g. Paradox (Checker (Paradox)))
- This reverses the output of the checker app. If checker accepts paradox then paradox will loop and crash otherwise it will halt (so the rejection is undeserved),e.g.
  - Paradox(Checker(Paradox)) = Paradox(Checker(loop)) = Paradox(reject) = Halt
  - Paradox(Checker(Paradox)) = Paradox(Checker(halt)) = Paradox(accept) = Loop
  - Contradiction

COMPUTABILITY & COMPUTATION

Edinburgh Napier UNIVERSITY

- Ability to solve a problem in an affective manner
- A problems computability is closely related to the existence of an algorithm to solve it.
- Have talked about use of Turing machines
- These are powerful computational models but there are less powerful (but still interesting models of computation), e.g. (Non-)Deterministic Finite Automaton, Pushdown Automaton
- Different models can do different tasks, e.g. semantic clarity, easier to implement

SUMMARY

- Algorithms
- An introduction to complexity (time & space)
- Some practical methods for determining complexity
- A tiny digression into computability

Edinburgh Napier
UNIVERSITY

QUESTIONS ???

IS THIS THE LAST QUESTION?
(THINK ABOUT IT….)

Edinburgh Napier
UNIVERSITY

WHAT DID WE LEARN?

- *We can now…*

- Inspect code & roughly determine the order of complexity of its computations
- Describe the features & function of the Turing Machine
- Understand some of the limits of computation