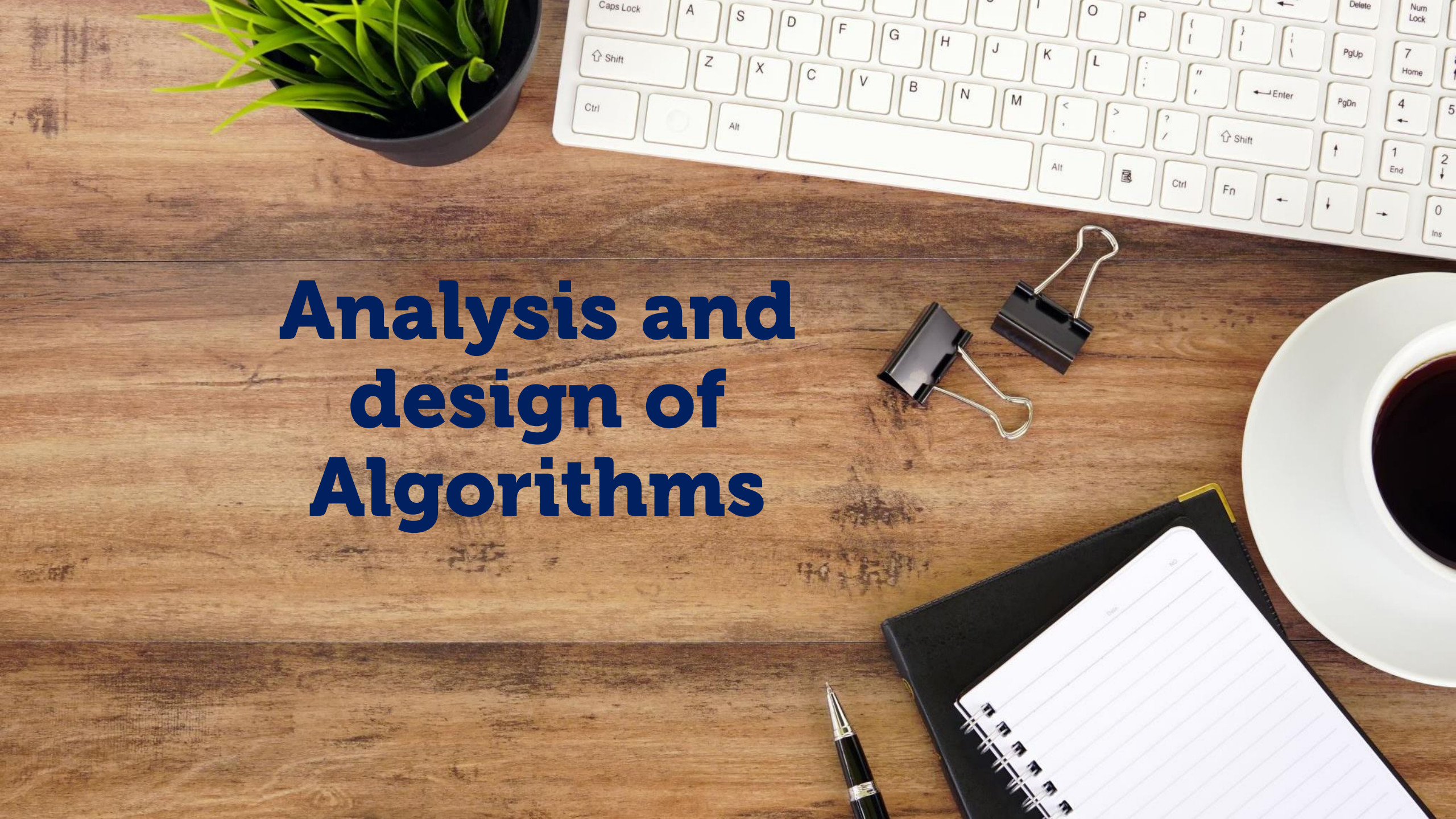

Analysis and design of Algorithms

# Objectives

- Analyse the suitability of different algorithms for a given task and data set
- Be familiar with measures and methods to determine the efficiency of different algorithms
- Define constant, linear, polynomial, exponential and logarithmic functions
- Use Big-O notation to compare the time complexity of algorithms
- Be able to derive the time complexity of an algorithm

# Computational algorithms

- What is an algorithm?
- Give some examples of computational algorithms

There are thousands of real-world problems to be solved using different algorithms; some of them still unsolved!

# Problems solved by algorithms

- **Routing** problems:
  - Routing packets of data around the Internet by the shortest route
  - Finding the shortest route for a salesman to cover his territory
- **Timetabling** commercial aircraft crews so that they do not exceed their permitted flight hours
- **Searching** information on the Internet or from a database
- **Encrypting** communications so that they cannot be hacked
- **Sorting** large amounts of data
- **Writing a compiler** program to translate a high level language to machine code

# What are the properties of a good algorithm?

- Has clear and precisely stated steps that produce the correct output for any set of valid inputs
- Should allow for invalid inputs
- Must always terminate at some point
- Should execute efficiently, in as few steps as possible
- Should be designed in such a way that other people will be able to understand it and modify it if necessary

# How efficient is an algorithm?

- We can sum the numbers from 1 to 1000 using the following algorithm:

```
sum = 0
for count = 1 to 1000
      sum = sum + count
next count
```

- Or, we can use a single statement to calculate the sum:

```
sum = 1000 * (1000 + 1) / 2
```

- Which is more efficient? Why?

# A measure of efficiency

- The number of assignment statements to be executed in the first algorithm is 1002

```
n = 1000
sum = 0
for count = 1 to n
      sum = sum + count
next count
```

- In the second algorithm, only two statements are executed:

```
n = 1000
sum = n * (n + 1) / 2
```

- The number of assignment statements is a good basic measure of efficiency

# Execution time vs problem size

- For most problems, the larger the size of the problem or amount of data, the longer the execution time
- How many steps will it take to sum n items?

```
sum = 0
for count = 1 to n
      sum = sum + count
next count
```

- How many statements in the second algorithm?

```
sum = sum + n * (n + 1) / 2
```

# Mathematics revision

- Recall from mathematics
  - A function is defined by an equation such as $y = 3x + 2$
  - A generalised form would be $f(n) = 3n + 2$

# Linear functions

- A linear function takes the form f(n) = an + b where a and b are constants
  - f(n) = 3n, f(n) = n + 5, f(n) = 6n + 1 are all examples of linear functions, and increase in a straight line
  - f(1) is a constant function – no matter how large n gets, f(1) stays the same size. f(2), f(3), f(100) are all constant functions
  - The order of magnitude for a linear function is written O(n)
    (note: O stands for “Order”)

# Quadratic functions

- A quadratic function takes the form

  $f(n) = an^2 + bn + c$ where a, b and c are constants

- As n becomes large, the $n^2$ term increases very much faster than either of the other terms
- The order of magnitude of a quadratic function is written $O(n^2)$
- The dominant term is the one used when comparing algorithms, so the $bn$ and $c$ terms are not considered

# Logarithmic functions

- A logarithmic function takes the form $f(n) = a\log_2 n$

  "The logarithm of a number is the power to which the base must be raised to make it equal to the number"

- $8=2^3$ ➔ $\log_2 8=3$ and $1024=2^{10}$ ➔ $\log_2 1024=10$
- f(n) increases very slowly in relation to n
- The order of magnitude of a logarithmic function is written O(log n)
- We're only concerned with base 2, so it is not necessary to specify it in the notation O(log n)

# Big-O notation

- Big-O notation is a measure of the time complexity of an algorithm
  - It is a useful approximation of the time taken to execute an algorithm for a given number of items in a data set
- An algorithm of time complexity O(n) increases linearly
  - 10,000 items will take approximately twice as long as 5,000 items to process
- There is no such thing as, for example, O(2n + 1)
- Only the dominant term counts, so it is O(n)

# Analysing an algorithm

- What is the number of steps in the following computation?

```
total = 0
for student = 1 to n
  for mark = 1 to 3
    total = total + results[student][mark]
  next mark
next student
for student = 1 to n
  average[student] = (mark[1]+mark[2]+[mark[3])/3
next student
```

# Deriving Big-O

- There are 1 + 3n + n assignment statements
- The algorithm has time complexity O(n)

```
total = 0
for student = 1 to n
  for mark = 1 to 3
    total = total + results[student][mark]
  next mark
next student
for student = 1 to n
  average[student] = (mark[1]+mark[2]+[mark[3])/3
next student
```

# Logarithmic functions

- “Divide and conquer” algorithms work by halving the size of the problem at each pass
  - A binary tree search algorithm is a good example of this
  - For a balanced tree of 1024 (i.e. $2^{10}$) items, only 10 items need to be examined
  - For a tree of 2048 (i.e. $2^{11}$) items, only 11 items need to be examined
- Time complexity increases very slowly as the problem size increases
- It is logarithmic, that is, O(log n)

# Task 1

| n | $n^2$ | $5n^2$ | 10n | 2 | $f(n)= 5n^2+ 10n + 2$ |
|---|---|---|---|---|---|
| 10 | | | | 2 | |
| 100 | | | | 2 | |
| 1000 | | | | 2 | |
| 10,000 | | | | 2 | |

# Permutations

- A permutation of n items is the number of ways the n items can be arranged
- There are two types of permutation:
  - Repetition allowed; for example, a combination lock with 4 digits 0 to 9
  - No repetition allowed; for example, you have 4 differently coloured balls in a bag, and you draw them out one at a time

# Permutations with repetition

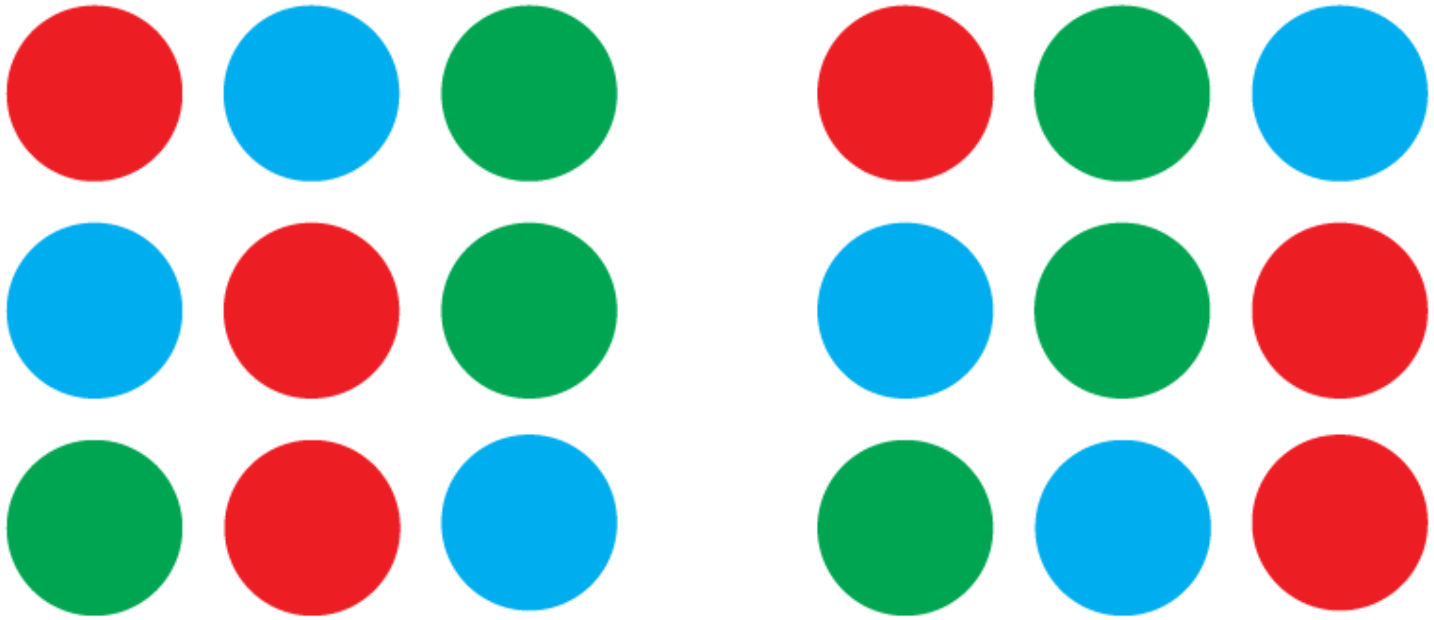
- Suppose you have a combination lock with 2 digits
  - There are 10 possibilities for the first digit
- For each of these numbers, the second number can be any digit between 0 and 9
  - 00, 01, 02, 03…..09
  - 10, 11, 12, 13…. 19 …. …. 90, 91, 92, 93…..99
- There are 100 ways of choosing just 2 digits!
  - How many ways of choosing 3 digits? 4 digits?
- This problem is $O(10^n)$.  Can you explain why?

# Permutations with no repetition

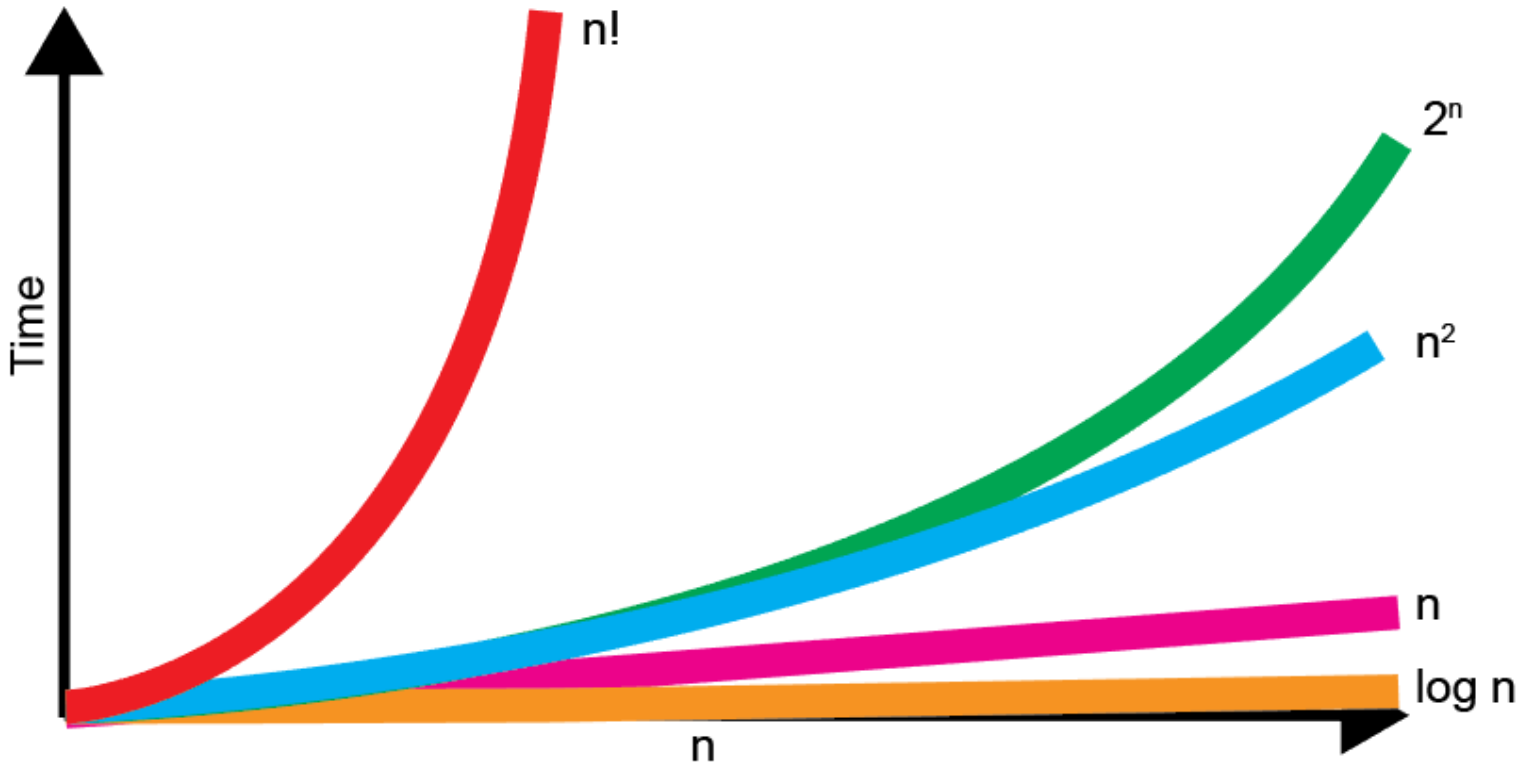
- Suppose you have a bag with 3 coloured balls, red, blue, green, and you pick out one at a time
- There are 3 possibilities for the first ball, two possibilities for the second ball, one for the third
  - There are 3 x 2 x 1 ways of picking out the three balls
- What if there were 5 differently coloured balls?
  Or 7 differently coloured balls?
  - Can you generalise the formula to show the time complexity of an algorithm that prints out all the ways of picking n different coloured balls in different orders?

# Permutations with no repetition

- There are 3! ways of picking or arranging 3 balls
  - The time complexity of an algorithm to print out the different ways is O(n!)

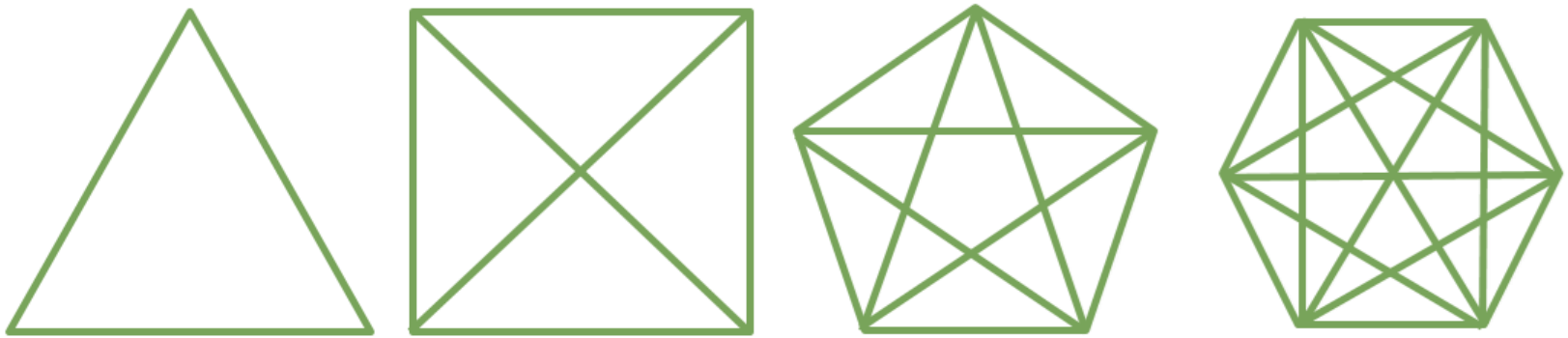


# Comparison of time complexities



# Task 2

- Try the questions in **Task 2** of the worksheet



# Summary

- Big-O notation is used to describe the time complexity of different algorithms
  - It describes how fast the runtime increases as the number of items to be processed increases
- Algorithms with time complexity O(n!) and O($2^n$) are hopelessly inefficient for large values of n
- Algorithms of O(log n) are extremely efficient for large values of n, with computation time increasing only slightly as the number of items doubles

# Summary

- To calculate the order of an algorithm in Big-O notation, count the number of assignment statements in the algorithm
- Three assignment statements in two nested FOR loops each of length $n$, for example, is $3n^2$ statements
- Only the dominant term is significant, and any constant coefficient is ignored, so if there are, say, $3n^2 + 5n + 2$ statements, the time complexity is $O(n^2)$