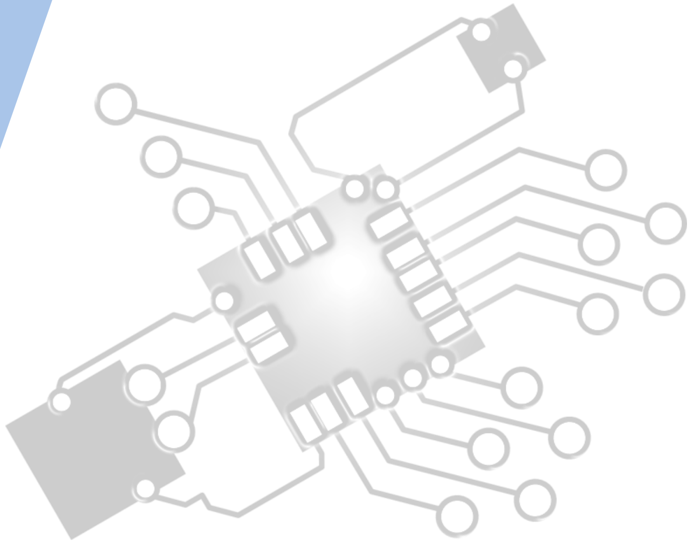
Computational thinking, problem-solving and programming:

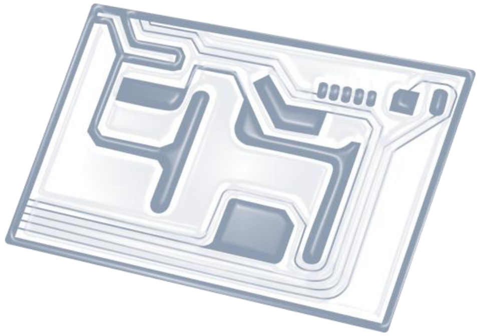
Connecting computational thinking and program design

IB Computer Science



Content developed by
Dartford Grammar School
Computer Science Department

HL Topics 1-7, D1-4



1: System design

2: Computer Organisation

3: Networks

4: Computational thinking

5: Abstract data structures

6: Resource management

7: Control

D: OOP

HL & SL 4.2 Overview

4.2.1 Describe the characteristics of standard algorithms on linear arrays

4.2.2 Outline the standard operations of collections

4.2.3 Discuss an algorithm to solve a specific problem

4.2.4 Analyse an algorithm presented as a flow chart

4.2.5 Analyse an algorithm presented as pseudocode

4.2.6 Construct pseudocode to represent an algorithm

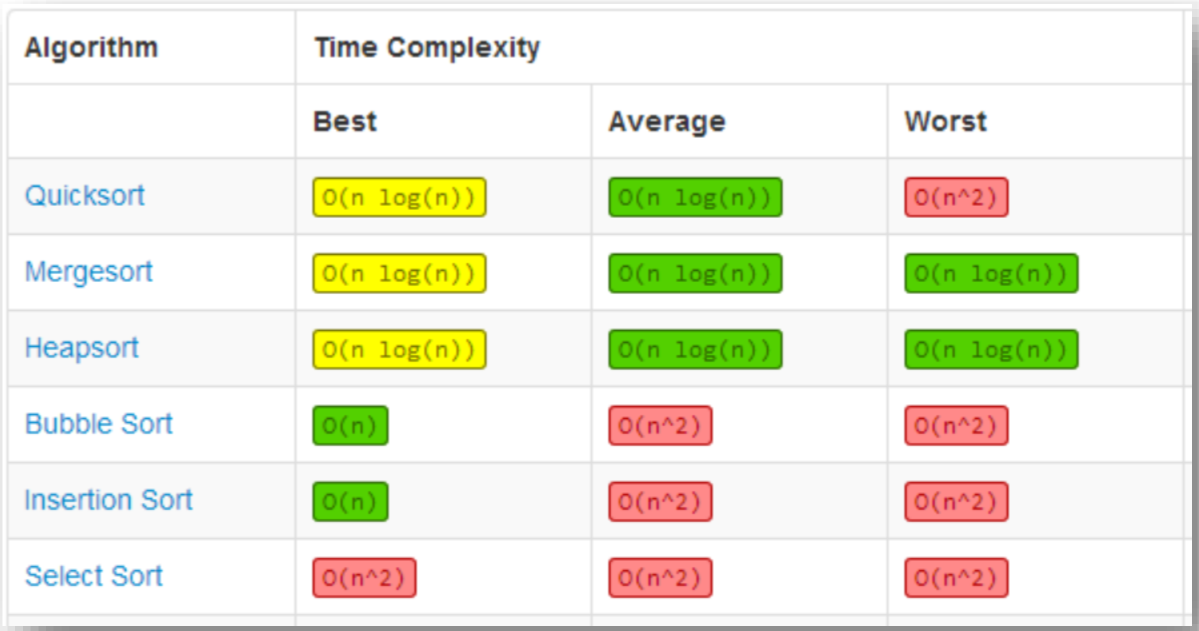
4.2.7 Suggest suitable algorithms to solve a specific problem

4.2.8 Deduce the efficiency of an algorithm in the context of its use

4.2.9 Determine the number of times a step in an algorithm will be performed for given input data

Topic 4.2.8

Deduce the **efficiency** of an algorithm in the context of its use



Algorithm	Time Complexity		
	Best	Average	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))
Bubble Sort	O(n)	O(n^2)	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)
Select Sort	O(n^2)	O(n^2)	O(n^2)

Teacher's notes:

- *Students should **understand** and **explain** the **difference in efficiency** between a **single loop**, **nested loops**, a **loop that ends when a condition is met** or questions of similar complexity.*
- *Students should also be able to **suggest changes** in an algorithm that **would improve efficiency**, for example, using **a flag** to stop a search immediately when an item is found, rather than continuing the search through the entire list.*

What effects run time of an algorithm?

(a) computer used, the hardware platform

(b) representation of abstract data types (ADT's)

(c) efficiency of compiler

(d) competence of programmer (programming skills)

(e) **complexity of underlying algorithm**

(f) **size of the input**

Generally (e) *and* (f) *are the most important*

Definition: Complexity

- Complexity of an algorithm is a measure of the **amount of time** and/or **space** required by an algorithm for an **input** of a **given size (n)**.
- **Time** for an algorithm to run **t(n)** is characterised by the size of the input.
- We usually try and estimate the **WORST CASE**, and sometimes the **BEST CASE**, and very rarely the **AVERAGE CASE**.

What do we measure?

- In analysing an algorithm, rather than a piece of code, we will try and predict the number of times **the principle activity** of that algorithm is performed.
- For example, if we are analysing a **sorting algorithm** we might count the **number of comparisons** performed.

Best vs Worst vs Average case

- **Worst Case**
  - is the maximum run time, over all inputs of size n, ignoring effects (a) through (d) above. That is, we only consider the "number of times the principle activity of that algorithm is performed".
- **Best Case**
  - In this case we look at specific instances of input of size n. For example, we might get best behaviour from a sorting algorithm if the input to it is already sorted.
- **Average Case**
  - Arguably, average case is the most useful measure. It might be the case that worst case behaviour is pathological and extremely rare, and that we are more concerned about how the algorithm runs in the general case. Unfortunately this is typically a very difficult thing to measure.

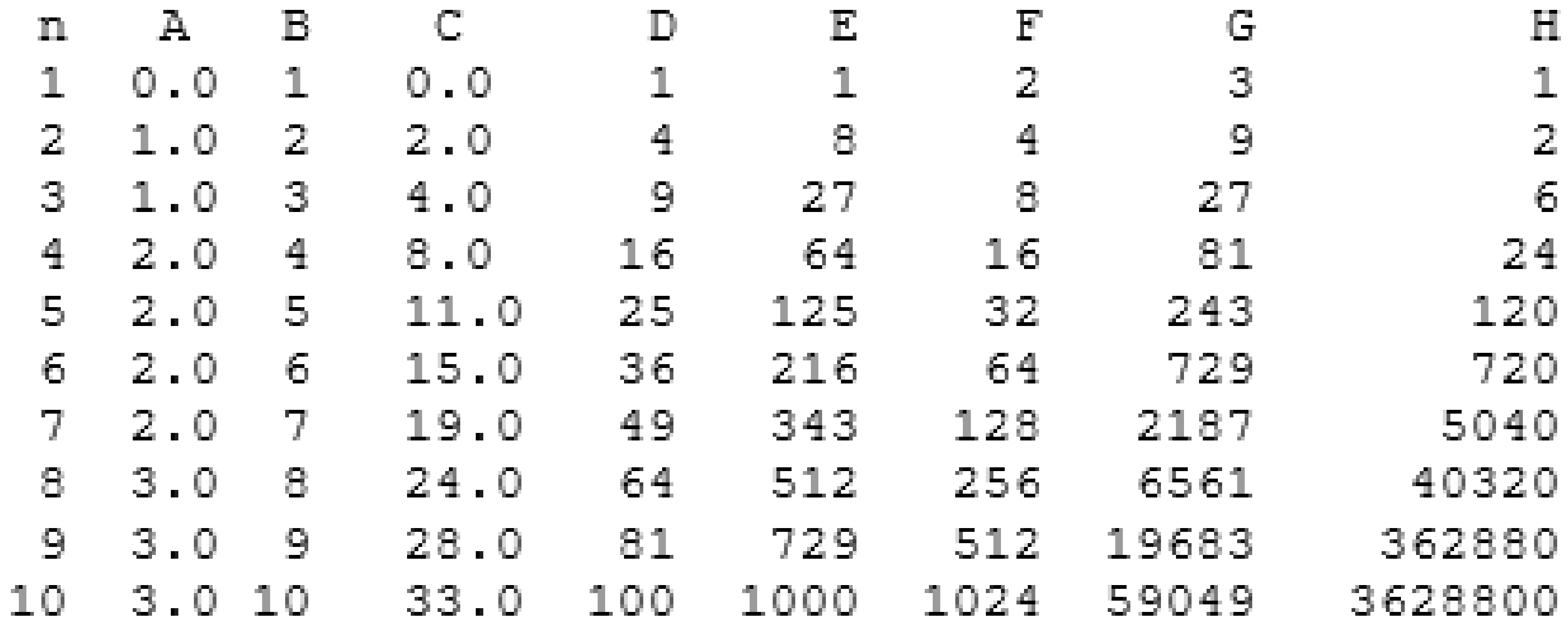
The *growth rate* of t(n)

- Suppose the worst case time for algorithm **`A`** is:
  `t(n) = 60*n*n + 5*n + 1` for input of size **`n`**.
- We ignore the coefficient that is applied to the most significant (dominating) term in t(n).
- Consequently this only affects the "units" in which we measure. It does not affect how the worst case time grows with **`n`** (input size) but only the units in which we measure worst case time
- Under these assumptions we can say:
  `t(n)` grows like **`n*n`** as **`n`** increases or **`t(n) = O(n*n)`**
- which reads "*t(n) is of the order n squared*" or as "*t(n) is **big-oh** n squared*"

Example (the tyranny of growth)

- In order (from least to most complex)…
  - **A = (log$_2$ n)** *{log to base 2 of n}*
  - **B = n** *{linear in n}*
  - **C = (n * (log$_2$ n))** *{n log n}*
  - **D = (n^2)** *{quadratic in n}*
  - **E = (n^3)** *{cubic in n}*
  - **F = (2^n)** *{exponential in n}*
  - **G = (3^n)** *{exponential in n}*
  - **H = (n!)** *{factorial in n}*

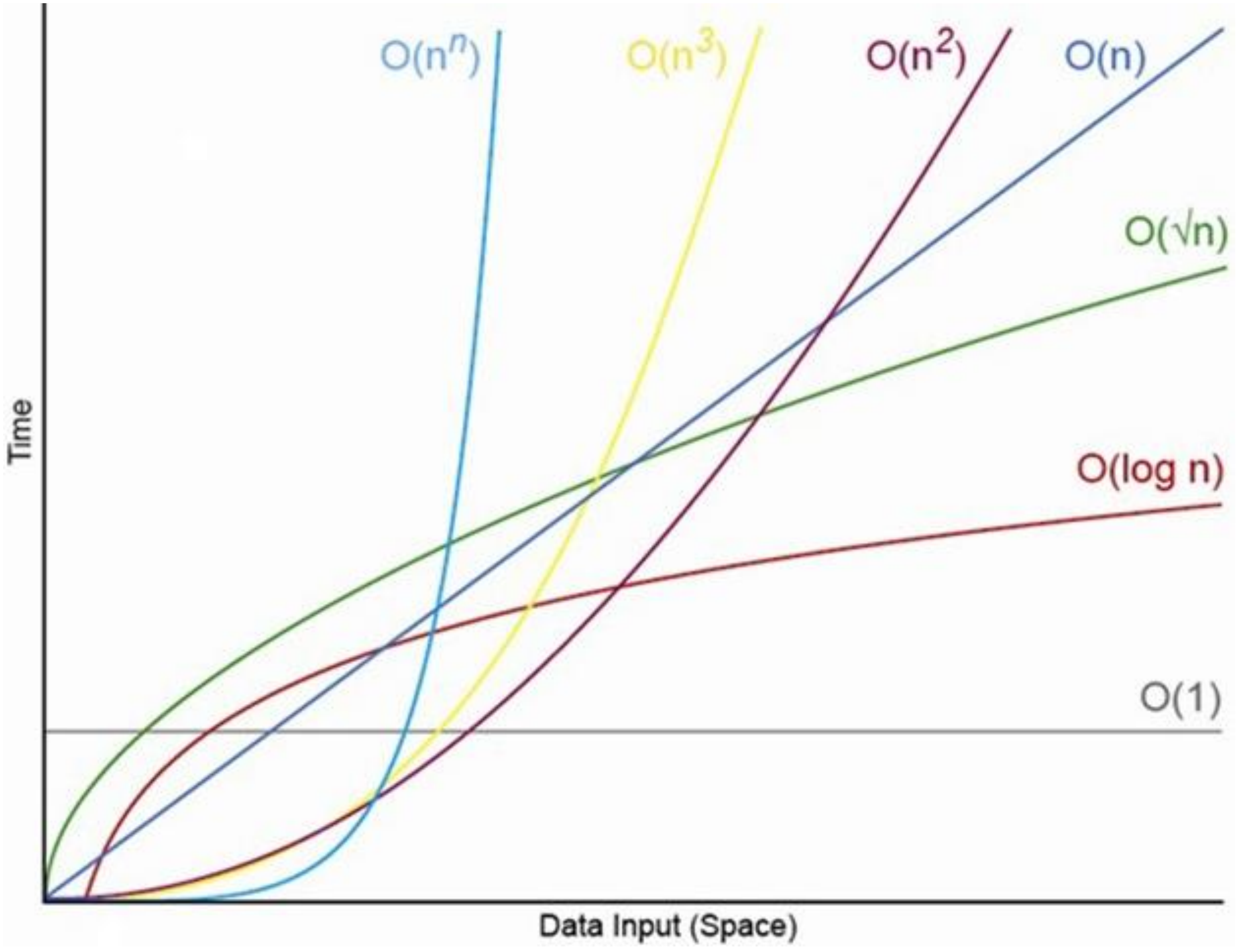
Tabulated below, are a number of functions against *n* (from 1 to 10)



n	A	B	C	D	E	F	G	H
1	0.0	1	0.0	1	1	2	3	1
2	1.0	2	2.0	4	8	4	9	2
3	1.0	3	4.0	9	27	8	27	6
4	2.0	4	8.0	16	64	16	81	24
5	2.0	5	11.0	25	125	32	243	120
6	2.0	6	15.0	36	216	64	729	720
7	2.0	7	19.0	49	343	128	2187	5040
8	3.0	8	24.0	64	512	256	6561	40320
9	3.0	9	28.0	81	729	512	19683	362880
10	3.0	10	33.0	100	1000	1024	59049	3628800

Big O notation summary

Notation	Type	Examples	Description
O(1)	Constant	Hash table access	Remains constant regardless of the size of the data set
O(log n)	Logarithmic	Binary search of a sorted table	Increases by a constant. If n doubles, the time to perform increases by a constant, smaller than n amount
O(< n)	Sublinear	Search using parallel processing	Performs at less than linear and more than logarithmic levels
O(n)	Linear	Finding an item in an unsorted list	Increases in proportion to n. If n doubles, the time to perform doubles
O(n log(n))	n log(n)	Quicksort, Merge Sort	Increases at a multiple of a constant
O(n^2)	Quadratic	Bubble sort	Increases in proportion to the product of n*n
O(c^n)	Exponential	Travelling salesman problem solved using dynamic programming	Increases based on the exponent n of a constant c
O(n!)	Factorial	Travelling salesman problem solved using brute force	Increases in proportion to the product of all numbers included (e.g., 1*2*3*4...)


O(n^n)
O(n^3)
O(n^2)
O(n)
O(√n)
O(log n)
O(1)
Time
Data Input (Space)

On a more basic note:

- A **single loop** that repeats **n** times takes **n** time to run
- A **nested loop** that repeats **n** times takes **n x n** times to run (potentially MUCH longer)
- A loop that checks a condition/flag (usually a WHILE loop) only loops while it has to – no unnecessary looping!
- *So, want a loop to run faster?* Try using a flag-based (WHILE) loop that will stop once the item you’re searching for is found. The alternative (FOR loop) would check EVERYTHING every time it runs.