OCR A-Level Computer Science Spec Notes

2.1 Elements of computational thinking

Computational Thinking: Take a complex problem, understand what the problem is and develop possible solutions.

2.1.1 Thinking abstractly

(a) The nature of abstraction.

- Abstraction is a **representation of reality**

(b) The need for abstraction.

- Needed to **encapsulate methods/data** so larger problems can be worked on without **too much detail**

(c) The difference between abstraction and reality

- Abstraction takes a **real life situation** and **removes unnecessary details** in order to reach a **solution quicker** by focusing on the **most important areas** of the problem.

(d) Devise an abstract model for a variety of situations.

- Examples of Abstractions: **Variables/Objects/Layers/Data Modules/Data Structures/Entity Diagrams**

2.1.2 Thinking ahead

(a) Identify the inputs and outputs for a given situation.

- Thinking ahead involves **planning potential inputs & outputs of a system**

(b) Determine the preconditions for devising a solution to a problem.

When **planning**, computer scientists will:

- **Determine outputs required** & **inputs** necessary to **achieve the outputs**
- Consider the **resources needed** & user expectations.

Strategies can be made to:

- Decide what is to be **achieved**
- Determine **prerequisites** & what's possible within **certain conditions**

(c) The nature, benefits and drawbacks of caching.

- **Illustration** of thinking ahead (**Caching**)
- **Caching**: Data stored in cache/RAM if needed again = Faster **future access**

(d) The need for reusable program components

Reusable program components

- Software is **modular** e.g **object/function**
- Modules transplanted into **new software / shared at run time** through the use of **libraries**
- Modules already **tested** = more **reliable** programs.
- Less **development time** as programs can be **shorter & modules shared**

2.1.3 Thinking procedurally

(a) Identify the components of a problem

- Thinking procedurally = **Decomposition**

(b) Identify the components of a solution to a problem

- **Large problems broken down** into **smaller problems** to **work** towards **solution**

(c) Determine the order of the steps needed to solve a problem

- **Order of execution** needs to be taken into account – may need data to be processed by **one module** before another can use it

(d) Identify the sub-procedures necessary to solve a problem

- **Large human projects benefit** from the same **approach.**

2.1.4 Thinking logically

(a) Identify the points in a solution where a decision has to be taken

- **Decisions** can be made **on the spot** or **before starting a task**
- It's important to know where decisions are taken as it affects program **inputs/outputs/functionality**

(b) Determine the logical conditions that affect the outcome of a decision

Consider:

- Are you planning the right thing?
- You need to think about the steps of a solution – will it yield the right results?
- What information do you have?
- Is it enough to form a certain (or acceptable) conclusion?
- What extra information do you need?
- What information do you have but don't need?

(c) Determine how decisions affect flow through a program

- Decisions made can either:
- **Speed up** the process
- **Decrease the speed** of the process
- Change the **inputs/outputs**
- Program **functionality** can **change**

2.1.5 Thinking concurrently

(a) Determine the parts of a problem that can be tackled at the same time

- Most modern computers can process a **number of instructions** at the **same time** (thanks to multi-core processors and pipelining).
- This means programs need to be **specially designed** to take **advantage** of this.
- **Modules processed** at the **same time** should be **independent**.
- **Well-designed programs** can save a lot of **processing time**.
- **Human activities** also **benefit from this**.
- **Project planning** attempts to **process stages simultaneously if possible**, so the project gets **completed more quickly**.

(b) Outline the benefits and trade offs that might result from concurrent processing

Concurrent (Parallel) Processing:

- **Carrying out more than one task at a time/a program has multiple threads**
- Multiple processors/Each thread **starts and ends at different times**
- Each processor **performs simultaneously/**Each thread **overlap**
- Each processor **performs tasks independently/**Each thread runs **independently**
- These **affect** the **algorithms** which are made

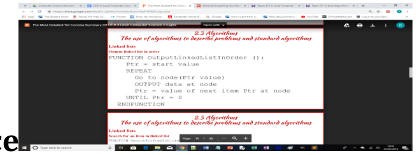
OCR A-Level Computer Science Spec Notes

2.2 Problem solving and programming

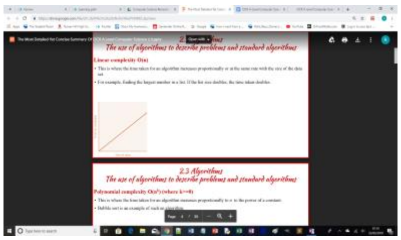
2.2.1 Programming techniques

(a) Programming constructs: Sequence, iteration, branching

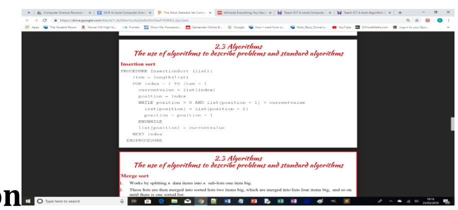
Programming Constructs (Methods of writing code)**:**



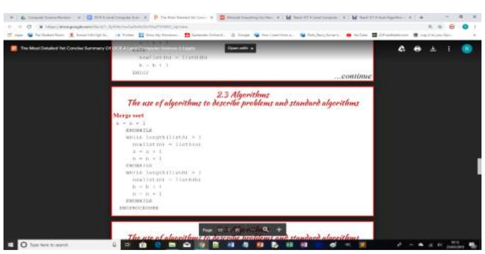
- **Sequence**
- **Series of statements** which are **executed one after another**
- Most common programming construct



- **Branching/Selection**
- Decisions made on the state of a **Boolean expression**
- Program **diverges** to another part on program based on whether a **condition** is **true or false**
- **IF statement** is a common example of Selection

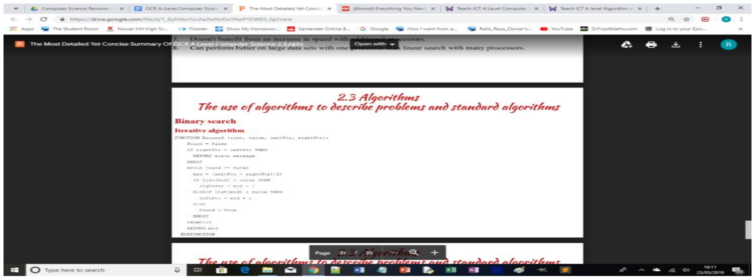


- **Iteration**
- = **repetition**. A section of code **repeated** for a **set amount of time / until condition is met**
- Loop: When a section of code is repeated
- Example of **For Loop ->**
- Example of **While Loop ↓**



(b) Recursion, how it can be used and compares to an iterative approach

- **Subroutine/Subprogram/Procedure/Function** that **calls itself**
- Another way to produce **iteration**



(c) Global and local variables

Variables: Named locations that store data in which contents can be changed during program execution

- Assigned to a **data type**
- **Declared/Explicit statement**

Global Variables

- **Defined/declared** outside **subprograms** (Functions/Procedures etc)
- Can be **'seen' throughout** a program
- Hard to **integrate** between **modules**
- **Complexity** of program **increases**
- Causes **conflicts** between names of other **variables**
- **Good programming practice** to not use global variables (**Can be altered**)

Local Variables

- Declared in a **subroutine** and **only accessible within** that subroutine
- Makes **functions/procedures reusable**
- Can be used as a **parameter**
- **destroyed/deleted** when **subroutine exits**
- **same variable names** within two different modules will not **interfere** with **one another**
- Local variables **override global variables** if they have the **same name**

(d) Modularity, functions, procedures, parameters

Modularity: Named locations that store data in which contents can be changed during program execution

- Program **divided** into **separate tasks**
- Modules **divided** into **smaller modules**
- Easy to **maintain, update and replace** a part of the system
- Modules can be **attributed** to different **programmers strength**

- Less code produced

Functions

- Subroutine/subprogram/module/named sub-section of program/block which most of the time **returns a value**
- Performs **specific calculations & returns a value of a single data type**
- Uses **local variables** & is used commonly
- Value returned **replaces function call** so it can be used as a **variable** in the **main body of a program**

Procedures

- Performs **specific operations** but **don't return a value**
- Uses **local variables**
- **Receives** & usually accepts **parameter values**
- Can be called my **main program**/another **procedure**
- Is used as any **other program instruction** or **statement** in the main program

Parameters

- **Description/Information** about **data supplied** into a **subroutine** when called
- May be given **identifier/name** when called
- Substituted by **actual value/address** when called
- May **pass values** between **functions & parameters** via **reference/ by value**
- Uses local variables

Passed by Value:

- A copy is made of the actual value of the **variable** and is passed into the procedure.
- Does not change the **original variable value**.
- If changes are made, then only the **local copy** of the **data** is **amended** then **discarded**.
- No **unforeseen effects** will occur in other modules.
- Creates new **memory space**

Passed by Reference:

- The **address/pointer/location** of the value is passed into the **procedure**.
- The **actual value** is not **sent/received**
- If changed, the **original value** of the **data** is also **changed** when the **subroutine ends**

- This means an **existing memory space** is used.

(e) Use of an IDE to develop/debug a program

IDE (Integrated Development Environment) contains the tools needed to **write/develop/debug a program**. Typical IDE has the following tools:

- **Debugging tools**
- **Inspection** of variable names
- **Run-time detection** of errors
- Shows **state of variables** at where **error occurs**
- **Translator diagnostics:**
- Reports **syntax errors**
- Suggests **solutions** & informs programmer to **correct error**
- Error message can be **incorrect/misinterpreted**
- **Breakpoint:**
- Tests program at **specified points/lines of code**
- Check **values** of **variables** at that **point**
- Set **predetermined point** for program to **stop & inspect code/variables**
- **Variable watch:**
- Monitors **variables/objects**
- Halt **program** if condition is **not met**
- **Stepping:**
- Set program to **step through one line at a time**
- Execution **slows down** to observe path of **execution** + **changes to variable names**
- Programmer can **observe the effect** of each line of code
- Can be used with **breakpoints** + **variable watch**

(f) Use of object orientated techniques

- Many programs written using objects (**Building blocks**)
- **Self contained**
- Made from **methods & attributes**
- Based on **classes**
- Many objects can be based in the **same class**
- **Most programs** made using object-oriented techniques

2.2.2 Computational methods

(a) Features that make a problem solvable by computational methods

Computability: Something which is not affected by the speed/power of a machine

Computational methods can help to break down problems into sections for example:

- **Models of situations/hypothetical solutions** can be **modelled**
- **Simulations** can be run by **computers**
- **Variables** used to **represent data items**
- Algorithms used to **test possible situations** under **different circumstances**

Features that make a problem solvable by computational methods:

- Involves **calculations** as some issues can be **quantified** - these are easier to process **computationally**
- Has **inputs, processes and outputs**
- Involves **logical reasoning**.

(b) Problem recognition

- A problem should be **recognised/identified** after looking at a **situation** and **possible solutions** should be divided on how to **tackle the problems** using **computational methods**

(c) Problem decomposition

Problem Decomposition

- **Splits problem** into **subproblems** until each problem can be **solved**.
- Allows the use of **divide and conquer**
- Increase **speed of production**.
- Assign areas to specialities.
- Allows use of **pre-existing modules & re-use of new modules**.
- Need to ensure **subprograms** can **interact correctly**.
- Can **introduce errors**.
- Reduces **processing/memory requirements**.
- Increases **response speeds of programs**.

(d) Use of divide and conquer

Divide and Conquer: When a task is split into **smaller tasks** which can be tackled more easily

(e) Use of abstraction

Abstraction: Process of separating ideas from particular instances/reality

- **Representation of reality** using various methods to display real life features
- **Removes unnecessary details** from the main purpose of the program
- E.g Remove parks/roads on an Underground Tube Map

Examples of Abstraction: Variables/data structure/network/layers/symbols (maps)/Tube Map

(f) Applying computational methods

Other computational Methods:

- **Backtracking**
- **Strategy** to **moving systematically** towards a **solution**
- **Trial & Error** (Trying out series of actions)
- If the pathway **fails** at some point = **go to last successful stage**
- Can be used **extensively**
- **Heuristics**
- **Not always worth trying to find the 'perfect solution'**
- Use '**rule of thumb'** /educated guess **approach** to arrive at a solution when it is unfeasible to analyse all possible solutions
- Used to **speed up finding solutions** for **A* algorithm**
- Useful for too many **ill-defined variables**
- **Data mining**
- Examines **large data sets** and looks for **patterns/relationships**
- **Brute force** with **powerful computers**
- Incorporates: **Cluster analysis, Pattern matching, Anomaly detection, Regression Analysis**
- Attempts to show relationships between **facts/components/events** that may not be obvious which can be used to **predict future solutions**
- **Visualisation**
- A computer process presents data in an **easy-to-grasp way** for humans to **understand** (visual model)
- Trends and patterns can often be better comprehended in a **visual display**.
- Graphs are a **traditional form** of visualisation.
- **Computing techniques** allow **mental models** of what a **program** will do to be produced.
- **Pipelining**
- Output of **one process fed into another**

- **Complex jobs** placed in **different pipelines** so **parallel processing** can occur
- Allow **simultaneous processing of instructions** where the processor has **multi-cores**
- Similar to factory production in real life
- **Performance modelling**
- Example of **abstraction**
- **Real life objects/systems** (computers/software) can be **modelled** to see how they perform & behave when in use
- **Big-O notation** used to measure **algorithm behaviour** with increasing input
- **Simulations predict performance** before real systems created

OCR A-Level Computer Science Spec Notes

2.3 Algorithms

2.3.1 Algorithms

(a) Analysis and design of algorithms for a given situation

Algorithms: Set of instructions that complete a task when execute

- Algorithms run by computers are called **'programs'**
- Scale algorithms by:
- The **time** it takes for the algorithm to complete
- The **memory/resources** the algorithm needs. '**space**'.
- **Complexity (Big O notation)**

(b) The suitability of different algorithms for a given task and data set, in terms of execution time and space

There are **different suitable algorithm**s for **each task**

- **Space efficiency:**
- The measure of how much memory (**space**) the algorithm takes as its input (**N**) is scaled up
- Space **increases linearly** with N
- Code space is **constant/data space** is also **constant**
- **Time efficiency**
- Measure of how much **time** it takes to **complete an algorithm** as its input (**N**) increases
- Time increases **linearly** with N

- **Sum of numbers = n(n+1)/2**
- **Big O notation**
- Refer to ((c) Measures and methods to determine the efficiency of algorithms (Big O) notation (constant, linear, polynomial, exponential and logarithmic complexity))

(c) Measures and methods to determine the efficiency of algorithms (Big O) notation (constant, linear, polynomial, exponential and logarithmic complexity)

(Big O) notation

- Shows **highest order component** with any constants removed to evaluate the **complexity** and **worst-case scenario** of an **algorithm**.
- Shows how **time increases** as **data size increases** to show **limiting behaviour**.

Big O Notation

- **O(1) – Constant complexity** e.g. printing first letter of string.
- **O(n) – Linear complexity** e.g. finding largest number in list.
- **O(kn) – Polynomial complexity** e.g. bubble sort.
- **O(k^n) – Exponential complexity** e.g. travelling salesman problem.
- **O(logn) – Logarithmic complexity** e.g. binary search

(d) Comparison of the complexity of algorithms

Complexity

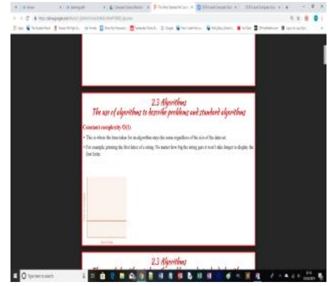
- Complexity is a measure of how much time**, memory space** or **resources** needed for an algorithm **increases** as the **data size** it works on **increases**.
- Represents the **average complexity** in **Big-O notation**.
- Big-O notation just shows the **highest order component** with any **constants removed**.
- Shows the **limiting behaviour** of an algorithm to classify its complexity.
- Evaluates the **worst case scenario** for the **algorithm**.

Types of Complexity

Complexity	Description	Graph

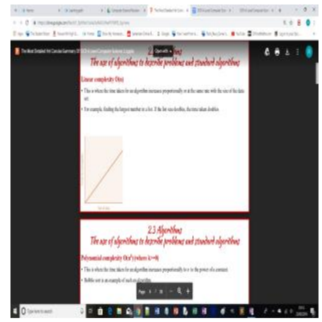
Constant complexity O(1)

- **Time taken** for an algorithm stays the **same** regardless of the **size** of the data set
- **Example:** Printing the first letter of a string. No matter how big the string gets it won't take longer to display the first letter.



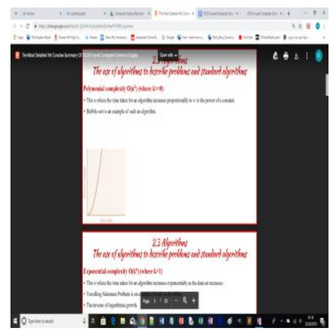
Linear complexity O(n)

- This is where the **time taken** for an algorithm **increases proportionally** or at the **same rate** with the **size of the data set**.
- Example: Finding the largest number in a list. If the list size doubles, the time taken doubles.



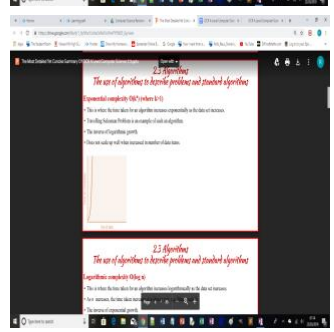
Polynomial complexity O(kn) (where k>=0)

- This is where the time taken for an **algorithm increases proportionally to n** to the **power** of a **constant**.
- Bubble sort is an example of such an algorithm.



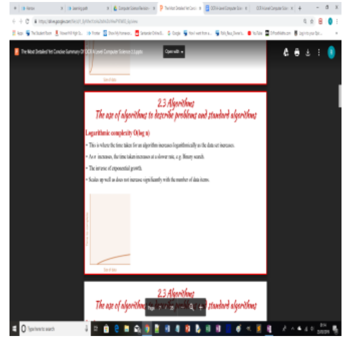
Exponential complexity O(k^n) (where k>1)

- This is where the time taken for an **algorithm increases exponentially** as the data set **increases**.
- **Travelling Salesman Problem** = example algorithm.
- The inverse of **logarithmic growth**.
- Does not scale up well when **increased** in **number** of **data items**.



Logarithmic complexity O(log n)

- This is where the time taken for an algorithm **increases logarithmically** as the **data set increases**.
- As **n increases**, the **time taken increases** at a **slower rate**, e.g. Binary search.
- The **inverse of exponential growth**.
- **Scales up well** as does not **increase significantly** with the **number of data items**.

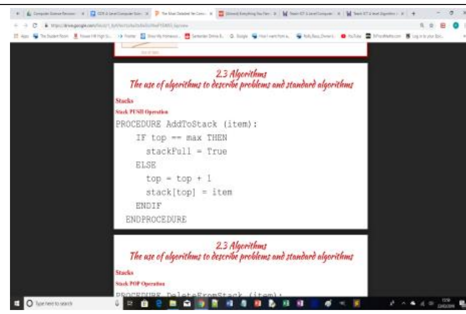


(e) Algorithms for the main data structures (stacks, queues, trees, linked lists, depth-first (post-order) and breadth-first traversal of trees)

Data Structures | **Description** | **Algorithm**

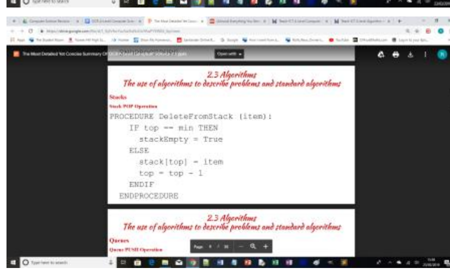
Stack PUSH

- When a data item is **added** to the **top** of a stack



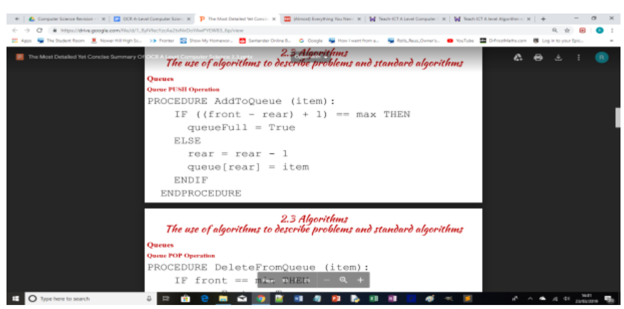
Stack POP

- When a data item is **removed** from the **top** of a stack



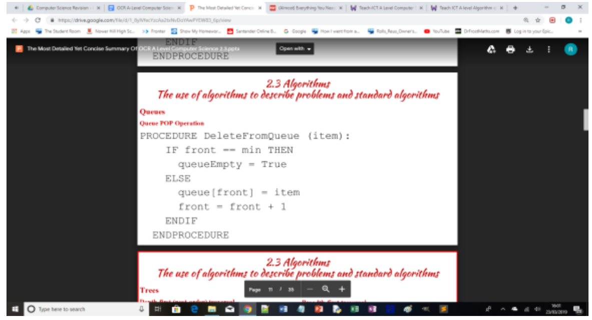
Queue PUSH

- When a data item is **added** to the **back** of a queue



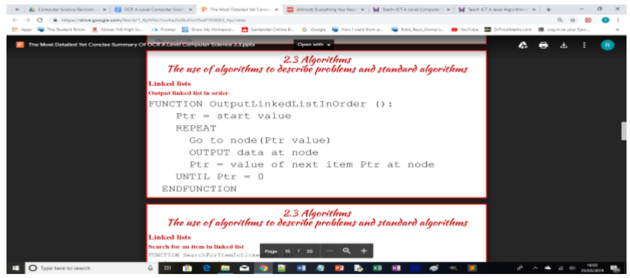
Queue POP

- When a data item is **removed** from the **front** of a queue



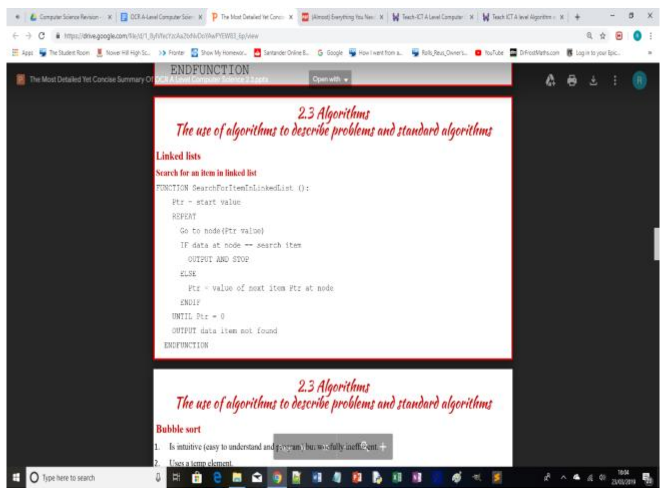
Linked List (Output in Order)

- When the contents of a linked list are **displayed in order**



Linked List (Add item to list)

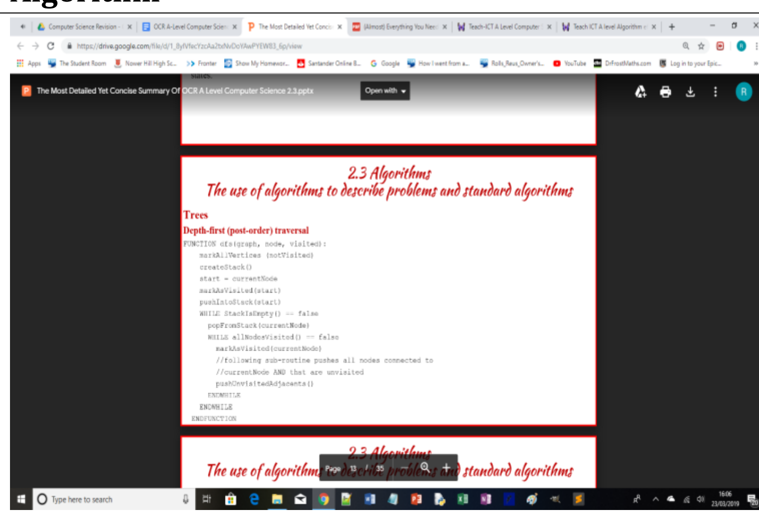
- When a data item is added **anywhere on a linked list**



Tree Traversal	Description	Algorithm

Depth first (post-order)

- Visit **all nodes** to the **left of the root node**
- Visit **right**
- Visit **root node**
- Repeat **three points for each node**

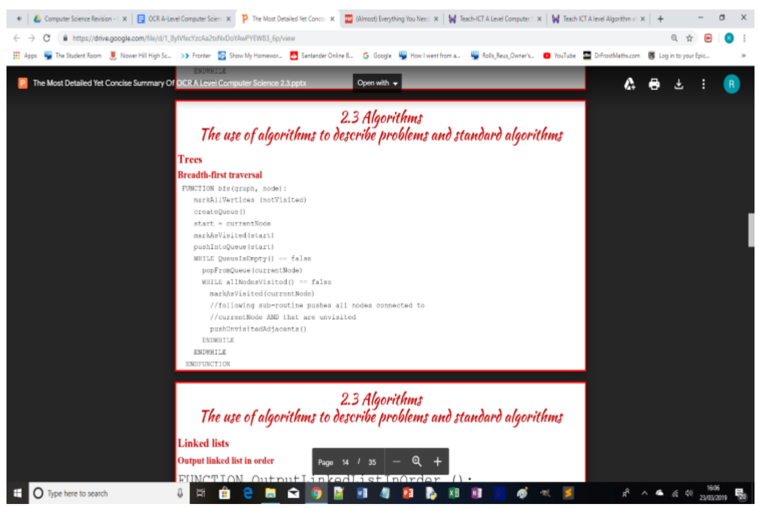


visited

- Depth first isn't **guaranteed** to find the **quickest solution** and possibly **may never find the solution** if no precautions to revisit **previously visited states.**

Breadth first

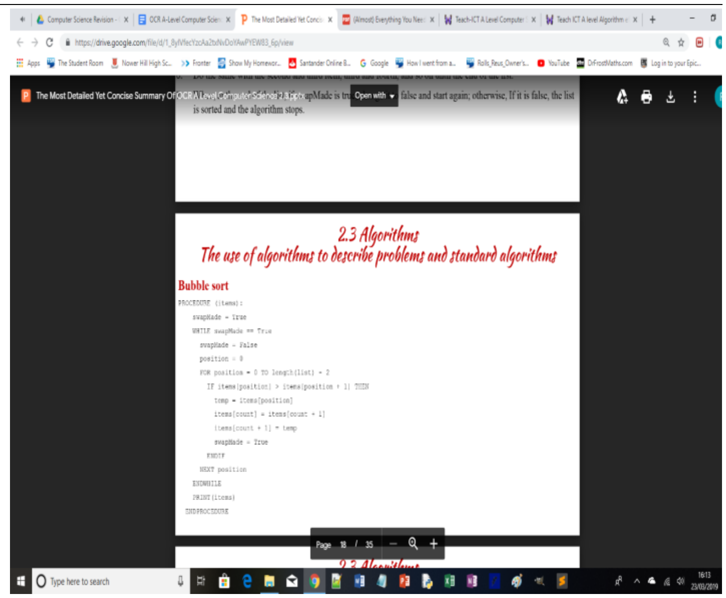
- Visit **root node**
- Visit all **direct subnodes (children)**
- Visit all **subnodes of first subnode**
- Repeat **three points** for each **subnode visited**
- Breadth first requires **more memory**



than Depth first search.

- It is **slower** if you are looking at **deep parts** of the tree.

(f) Standard algorithms (bubble sort, insertion sort, merge sort, quick sort, Dijkstra's shortest path algorithm,A* algorithm, binary search and linear search)

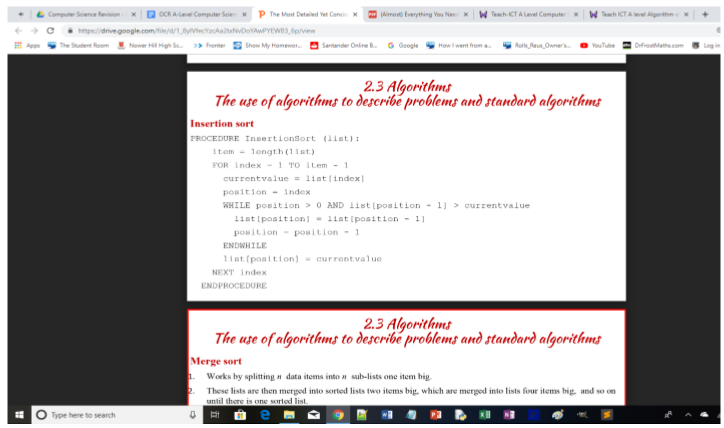
Sort	Description	Algorithm
Bubble Sort	• Is **intuitive** (easy to understand and program) but **inefficient**. • Uses a **temp element**. • Moves through the data in the **list repeatedly** in a **linear way** • Start at the **beginning** and **compare** the **first item** with the **second**. • If they are out of order, **swap them** and set a **variable swapMade true**. • Do the same with the **second and third item**, **third and fourth**, and so	

on until the **end of the list.**

- When, at the end of the list, **if swapMade is true**, change it to **false** and **start again**; otherwise, If it is **false**, the **list is sorted** and the **algorithm stops.**

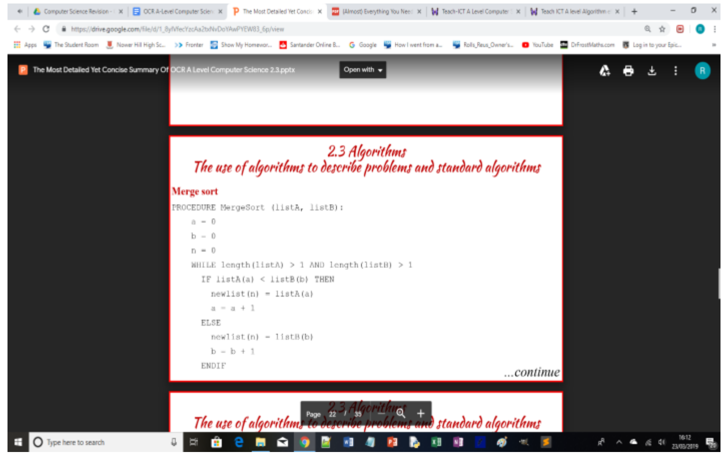
Insertion Sort

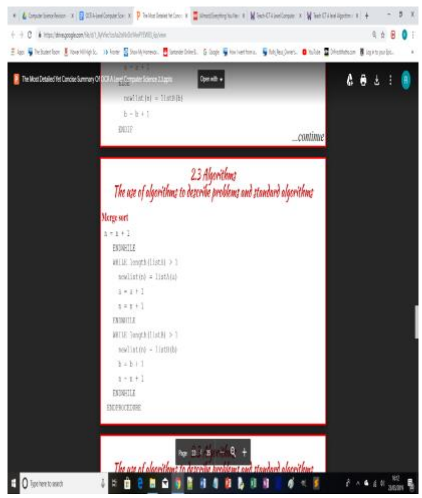
- Works by **dividing** a list into **two parts**: **sorted and unsorted**
- Elements are inserted **one by one** into their **correct position** in the **sorted section** by **shuffling them left** until they are **larger** than the item to the **left** of them until all items in the list are **checked.**
- **Simplest sort algorithm**
- **Inefficient** & takes longer for **large sets of data**



Merge Sort

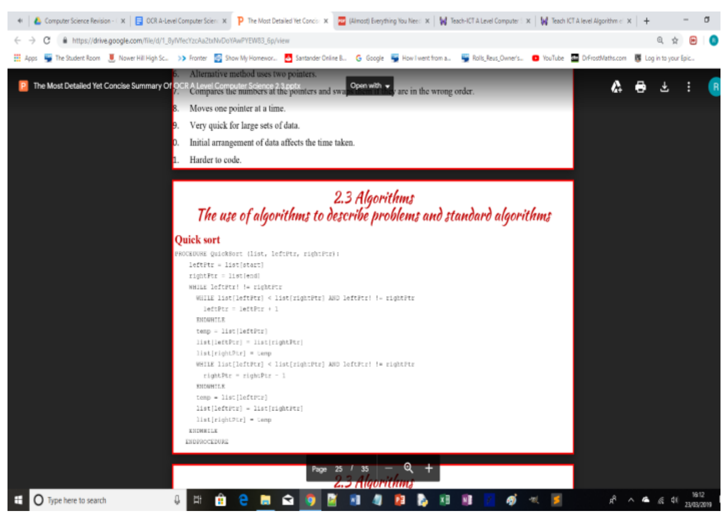
- Works by splitting **n data items** into **n sublists one item big.**
- These lists are then **merged** into **sorted lists two items big**, which are **merged into lists four items big**, and so on until there is **one sorted list.**
- Is a **recursive algorithm** = require **more memory space**
- Is **fast** & **more efficient** with **larger volumes** of data to sort.

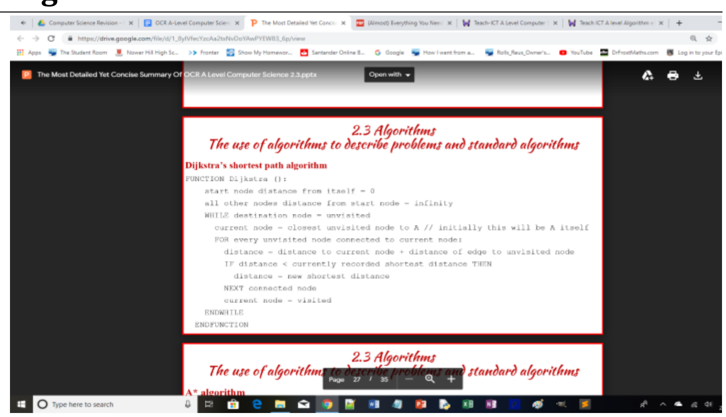




Quick Sort

- Uses **divide and conquer**
- Picks an item as a **'pivot'**.
- It then creates two **sub-lists**: those **bigger** than the pivot and those **smaller.**
- The same process is then applied **recursively/iteratively** to the **sub-lists** until all items are **pivots**, which

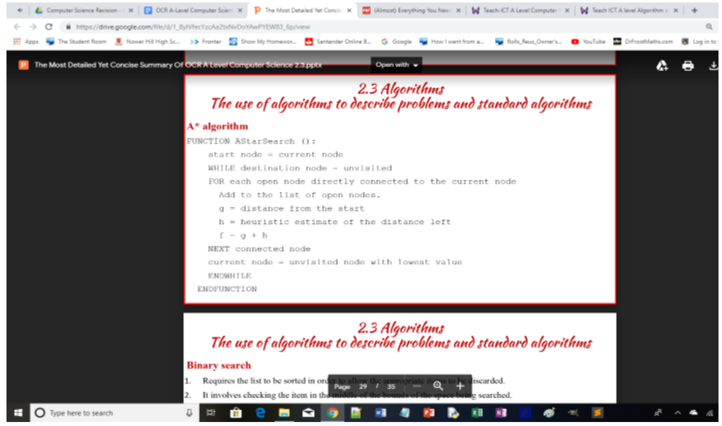


	• will be in the **correct order**. • Alternative **method uses two pointers**. • **Compares** the numbers at the **pointers** and swaps them if they are in the **wrong order**. • Moves **one pointer at a time**. • **Very quick** for **large sets of data**. • Initial arrangement of **data affects the time taken**. • **Harder to code.**	
Path Algorithms	**Description**	**Algorithm**
Dijkstra's shortest path algorithm	• Finds the **shortest path** between **two nodes** on a graph. • It works by keeping **track of the shortest distance** to each **node** from the **starting node**. • It **continues** this until it has **found the destination**	

node.

A* algorithm

- **Improvement** on **Dijkstra's algorithm**.
- **Heuristic approach** to estimate the **distance** to the **final node**, = **shortest path** in **less time**
- Uses the **distance** from the **start node plus** the **heuristic estimate** to the **end node.**
- Chooses which **node** to **take next** using the **shortest distance + heuristic**.
- All **adjoining nodes** from this **new node are taken.**
- Other **nodes** are **compared again in future checks.**
- Assumed that this **node** is a **shorter distance**.
- **Adjoining nodes** may **not** be **shortest path** so may need to **backtrack to**

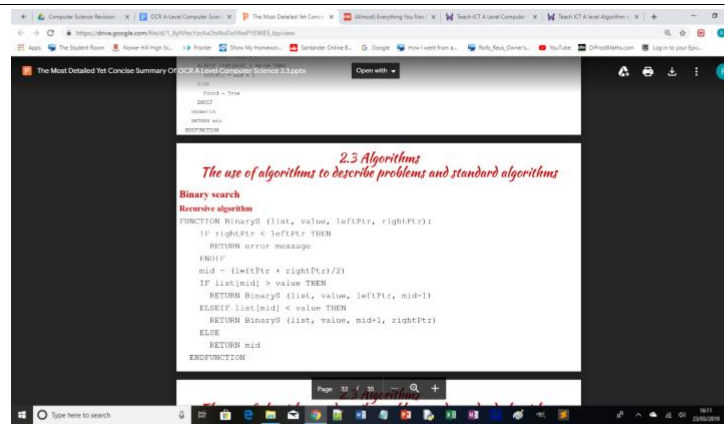


previous nodes.

<table>
<tr><th>Search Type</th><th>Description</th><th>Algorithm</th></tr>
<tr><td>Binary Search

Recursive</td><td>

Requires the list to be sorted in order to allow the appropriate items to be discarded.
It involves checking the item in the middle of the bounds of the space being searched.
It the middle item is bigger than the item we are looking for, it becomes the upper bound.
If it is smaller than the item we are looking for, it becomes the lower bound.
Repeatedly discards and halves the list at each step until the item is found.
Is usually faster in a large set of data than linear search because fewer items are

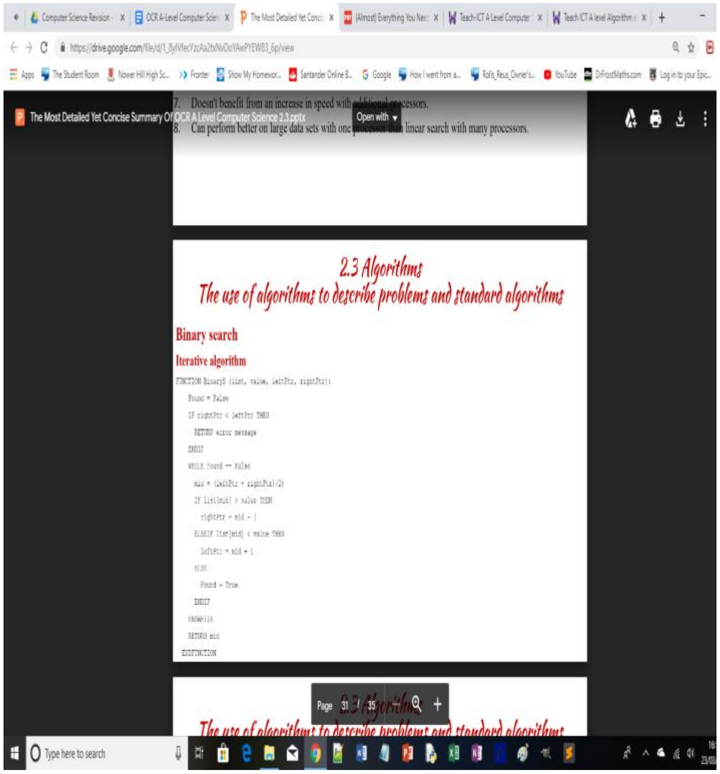
</td><td>
</td></tr>
</table>

checked so is more **efficient for large files**.

- Doesn't benefit from **increase in speed** with **additional processors**.
- Can perform better on **large data sets** with **one processor** than **linear search with many processors**.

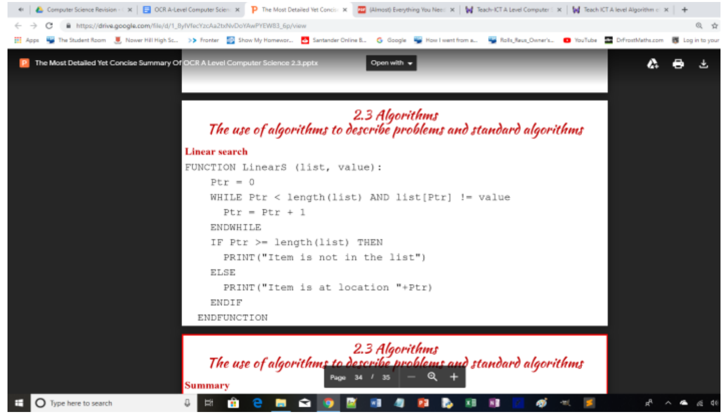
Binary Search

Iterative



Linear Search

- Start at the **first location** and check each **subsequent location** until the **desired item is found** or the **end of the list is reached**.
- Does not need an **ordered list** and **searches through all items** from the **beginning one by one.**
- Generally performs much better than binary search if the **list is small** or if the item being searched for is **very close to the start of the list**
- Can have **multiple processors** searching **different areas** at the same time.
- Linear search **scales very** with **additional processors.**



Summary

	Worst Case	Best Case
Bubble Sort	n^2	n

Insertion Sort	**n^2**	**n**
Merge Sort	**n log n**	**n log n**
Quick Sort	**n^2**	**n log n**
Binary Search	**log_2 (n)**	**1**
Linear Search	**n**	**1**