OCR A-Level Computer Science Spec Notes

1.1 The characteristics of contemporary processors, input, output and storage devices - Summarized

1.1.1 Structure and function of the processor

(a) ALU; Control Unit; Registers

- **CPU** (Central Processing Unit): A general purpose-processor which completes instructions using the **FDE** cycle (Fetch-Decode-Execute).

CPU consists of:

- **ALU** (Arithmetic Logic Unit):
  - Carries out Logical/Arithmetic calculations in the CPU
  - Stored in the **ACC.**
  - Acts as a gateway to the processor for easy calculations.
- **CU** (Control Unit):
  - Controls **FDE** cycles
  - Decodes & Executes instructions + Coordinates Data around processor/computer
  - Synchronises actions using in-built **clock**
- **Registers**: Memory locations inside a computer that temporarily store data/information. They are faster to access than **RAM** especially during the **FDE** cycle
  - **GPR (**General Purpose Registers**):**
    - Temporarily store data than transferring using slower memory.
  - **PC** (Program Counter):
    - Stores the address of the next instruction to be processed
  - **ACC** (Accumulator):
    - Temporarily stores **ALU** calculations & deals with **I/O** data
  - **MAR** (Memory Address Register):
    - Temporarily stores address of the next instruction/data from main memory
  - **MDR** (Memory Data Register):
    - Contains the instructions of the memory location address specified in the MAR. Copies data/instructions to **CIR**
  - **CIR** (Current Instruction Register):
    - Hold the most recent instruction for decoding/execution by **CU**
- **Buses**: Parallel set of communication wires which carry instructions/data to/from registers to processors. There are 3 different buses in the CPU:
  - **Data Bus**: Carries data/instructions around the system (CPU <-> Register)
  - **Address Bus:** Carries information on the location of the data (MAR -> Main Memory)
  - **Control Bus**: Transmits **control signals** from **CPU** to sync rest of processor

(b) Fetch-Decode-Execute cycle

Fetch

- **PC** instruction fetched & stored in **Main memory** to **processor**
- **PC** passess address location to **MAR** through **address bus**
- **PC** is incremented in cycle & **Fetch signal** is sent to **control bus.**
- Contents of memory location is sent from memory to processor via **data bus** which is then stored on **MDR**
- Contents of **MDR/ACC** sent to **ALU** & calculation sent to **ACC**

Decode

- Load instruction from address in **MAR** & send to **MDR**
- Instruction copied from **MDR -> CIR**
- Instruction decoded into **opcode/operand** by **CU** in **CIR**

Execute

- The appropriate instruction **opcode** is carried out on the **operand** by the processor.

(c) CPU performance (clock speed, number of cores, cache)

CPU **performance** can be measured in different ways

- **Clock Speed**
  - Clock controls the process of executing instructions/fetching data
  - Can be **'overclocked' =** More **cycles per second**
  - **Heat Sink**: Fan to cool down overheating CPU
- **Number of Cores**
  - Multiple cores = Speed up **smaller problems**
  - **Multi Tasking** = **Different cores** run **different apps / All** work on **one app**
- **GPU (Graphics Processing Unit)**
  - Designed to handle **graphics/video faster** than a **CPU**
  - **CPU** directly sends **Graphics related tasks** to **GPU**
- **Cache**
  - Small memory which runs much faster than main memory **(RAM)**
  - By anticipating the data/instructions that are likely to be regularly accessed , the overall speed at which the **Processor** operates can be increased.
  - More space for **data/instructions** in **cache** memory
  - **RAM** needs to be accessed less frequently as accessing **cache** is quicker.
  - More **expensive** than RAM

(d) Pipelining

- Allow one instructions to be **decoded/executed** while the previous one is **fetched/decoded**
- **Jump instructions** can't be used with pipelining as the **wrong instruction** can be fetched/decoded which causes the pipeline to '**flush**'.

(e) Von Neumann, Harvard, contemporary architecture

Computers are built off from mainly **2 architectures:Von-Neumann/Harvard Architecture:**

Von Neumann Architecture:

- Single processor **CU** manages **program control**.
- Uses **FDE cycle** to execute one instruction at a time in a linear sequence.
- Program and data stored together in same memory format (**Problem** due to overwriting of data)
- Simple OS and easy to program
- **Von Neumann Bottleneck:** CPU has to wait for data transfer as it's much faster

Harvard Architecture:

- Data/instructions are stored in separate memory units with separate buses **(Complex)**
- So while data is being written to or read from the data memory, the next instruction can be read from the instruction memory (Von Neumann more cost effective)

Contemporary Processor Architecture:

- Modern high-performance CPU chips incorporate aspects of both architectures.

1.1.2 Types of processor

(a) CISC vs RISC processors

Reduced instruction Set Computer (**RISC**)	Complex instruction Set Computer (**CISC**)
- **Simple** processor design	- **Complicated** processor design
- **Simpler Instructions** used	- **Complex Instructions** used
- **One machine cycle** per instruction	- Each instruction (**Many cycles**)
- Allows pipelining	- No pipelining
- **Shorter** instruction set	- **Longer** instruction set
- Requires More **RAM**	- Requires Less **RAM**
- **Simple circuitry** is cheaper	- **Integrated circuitry** is more **expensive**
- Programs **run faster** due to simple instructions	- Programs **run more slowly** due to complicated circuit
- **Limited Instructions** available	- **Many Instructions** available
- An instruction performs a simple task so complex tasks can only be performed by **combining multiple instructions**	- An instruction can do complex tasks so no need to **combine many instructions**

(b) GPUs

- Specifically designed for **enhancing graphics**
- Have **inbuilt circuitry** & instruction set for graphics based calculations
- **Large number of cores** = run **highly parallelizable problem**s
- Perform **on-screen graphics transformations** quickly
- Tackles problems in: Science/Engineering, data mining, audio processing, password beaking, machine learning

Co-Processor: Extra processor to **supplement functions** of **primary processor** (CPU)

(c) Multicore and Parallel systems

Multicore processors

- More than **one processor** incorporated into **one chip**
- Focuses efforts of **multiple CPUs** into **1 task**
- Hard to program code to **decompose problems efficiently** for **multicore processing**

Parallel Systems

- A computer which does **multiple computations simultaneously** to solve a problem which takes **less time to do one job**
- Parallel processing isn't **suited to all to problems**. Most problems are only **partially parallelizable**.
- Allows **faster processing** and **speeds up arithmetic processes** as multiple instructions are processed at the same time and complex tasks are performed **efficiently**.
- **Complex OS & specific code** has to be written for **maximum efficiency** of parallel processing.

Different approaches to Parallel processing:

- **SIMD** (Single Instruction Multiple Data): The same instruction operates simultaneously on multiple data locations
- **MIMD** (Multiple Instructions Multiple Data): Different instructions operate concurrently on different data locations

1.1.3 Input, output and storage

(a) Applying different input, output, storage devices to a problem

Input Devices: Peripheral devices which pass data onto the computer and allow the user to communicate with the computer.

Output Devices: Peripheral devices used to report the results of processing from a computer to the user and allow the computer to communicate with the user.

Input Devices Examples: Keyboard, Mouse, Microphone, Scanner

Output Devices Examples: Printer, Speaker, Monitor, Actuators

Storage Devices

- A secondary storage device is the **physical hardware** that carries out the **storage action**.

When getting a storage device, the following needs to be considered:

- **Cost of media** (DVD disk vs an external hard disk)
- **Cost per GB** (Important for **backup of data**)
- **Speed** (**Read - Write** speed)
- **Capacity** (How much data it can store)
- **Potability** (How **heavy/light** the device is)
- **Durability** (How **long** can it last)

Archive: transfer (data) to a less frequently used storage medium such as magnetic tape.

Back-up: a copy of a file or other item of data made in case the original is lost or damaged.

(b) Magnetic, flash and optical storage devices

- **Peripheral devices** used to **permanently store data** when **Power OFF**
- 3 Main storage categories: **Magnetic/Flash/Optical**

Magnetic Storage	Flash Storage	Optical Storage
- Use of **magnetisable material** to read **magnetic patterns** of **platters** that run **mechanically** at **high speeds**	- Data is stored on **memory chips** - Can have contents **overwritten/erased** when **electrical charge** is applied	- Using a **laser** which reads the disc by looking at its **reflection**
- **High Capacity** at **Low Cost**	- No **moving parts** = **less power** - **High read/write** speeds - **Less Space** & **Run silently**	- **Cheap & resilient**
- **Noisy & Susceptible to damage** if moving **too quickly**	- **Expensive** form of storage	- Unreadable if there are **scratches**
- E.g HDD/Zip Drives/Magnetic Tape	- E.g SSD/Flash Drives(USB)/Flash memory Cards	- E.g CDs/DVDs/Blu-Ray discs

(c) RAM and ROM

RAM (Random Access Memory)	**ROM** (Read only Memory)
- User files/applications software/OS **temporarily stored**	- Small memory which can only be **READ into**
- Faster **read/write speed** than **secondary storage** media	- Stores **BIOS bootstrap program**. Stored here so it isn't deleted
- **Volatile**: Loses contents when **Power OFF**	- **Immediately** present when computer is turned on
- Data can be **written over** by allowing user to alter saved files in current use	- **Non Volatile**: Contents not lost when **Power OFF**
- **Large & reduces buffering**	- Memory contents can't be **altered/maliciously changed**

(d) Virtual storage

- Combination of **multiple storage devices** into **1 virtual storage device**
- Remote Storage/Software & Accessible anywhere
- If **one storage device fails**, can be replaced with **inexpensive storage device**
- Easy for **administrator** to monitor **one storage** device rather than **multiple**
- **Complicated system** so requirements to run are high

OCR A-Level Computer Science Spec Notes

1.2 Software and software development

1.2.1 Systems Software

(a) Function and purpose of operating systems

Operating System: Low-level software which controls a computer's basic functions such as:

- Controls communication to/from devices using **protocols**
- **Manage Software**: Loading/Uploading software to memory
- Provide **Security**: **Username/Password** control
- Handles **code translations** of: **compilers/interpreters/assemblers** to translate **HLL/LLL** into machine code.
- Provide a user interface **(UI) / HCI :** So user can interact with the computer e.g Command Line Interface (**CMD/CLI**)
- **Utility software** used to carry out **maintenance tasks** to maintain **hardware**
- Uses **job scheduling** to provide **fair access** to **processor** according to **set rules**.

(b) Memory management (paging, segmentation, virtual memory)

- Memory is limited so it needs to be **managed**.
- This is achieved by providing each **process** with a **segment** of the total memory
- This is so there is no **corruption of data** during memory transfer
- Ensures programs can't access each **other's memory** unless **legitimately required** to.
- Provides **security to OS**
- Allows programs **larger than main memory** to run
- Allows **separate processes** to run while **managing memory**

<table>
<tr><th>Paging</th><th>Segmentation</th><th>Virtual Memory</th></tr>
<tr><td>- Splits memory into fixed-size chunks made to fit the memory</td><td>- Splits memory into variable sized logical divisions which can hold whole programs</td><td>- When memory inefficient = allocated secondary storage memory used to allow programs to run</td></tr>
<tr><td colspan="2">- Are assigned to memory when needed to allow programs to run despite insufficient memory.</td><td>- Uses backing store as additional memory for temporary storage</td></tr>
<tr><td colspan="2">- Are stored on a backing store disk to swap parts of programs used for virtual memory.</td><td>- Swap pages to/from RAM (paging)</td></tr>
<tr><td colspan="2">- Allow programs to be stored in memory non-contiguously.</td><td>- Hold part of program not currently in use</td></tr>
<tr><td colspan="3">- May cause disk threshing when more time spent swapping pages from memory to disk than processing so computer may 'hang'.</td></tr>
</table>

(c) Interrupts (function of ISRs)

Interrupt: A signal from a device alerting the CPU for its immediate attention
- Obtain processor time via **generating a signal/message** to **processor** stating they need to be **serviced immediately**
- Breaks **current execution** which is occuring in the **processor**
- Interrupts have **different priorities**
- Start when **current FDE cycle** is **complete** to ensure **max efficiency of processor**
- Can only interrupt a **low-priority task** to avoid **delays/loss** of data

Interrupt Service Routine (**ISR**)
- Check **IR** to compare interrupt priority compared to task
- If **lower/equal priority** = current task continues
- If **higher priority** = CPU completes **FDE cycle**
- Contents of registers stored in **LIFO stack**
- Location of **ISR** is loaded by loading the **relevant value** into the PC
- When **ISR** is complete
  - Flags sent to **inactive state**
  - **Further interrupts checked & serviced** if necessary
  - Contents of stack **popped** and loaded back onto **registers** to **resume processing**

(d) Scheduling

Scheduler: Manages the amount of time allocated to different processes in the CPU. It has several purposes:
- **Maximise** # of jobs completed in set time
- **Maximise** # of users receiving fast response times with minimal delay
- Ensure all jobs are **processed fairly** so long jobs don't **monopolise** the processor
- Obtain the **most efficient** use of processor time and **utilise resources** dependent upon priorities
- Prevent **process starvation** from applications in **deadlock** failing to run

Scheduling Algorithms

Scheduling Algorithm	**Process of Scheduling**	**Advantages**	**Disadvantages**
Round Robin	- All jobs given equal amount of processor time. If not completed = sent to back of queue and next job is given time	- Simple to implement as jobs are relatively the same size	- The importance of the process is not taken into account - Some jobs require multiple processing tuns making round robin inefficient for longer jobs
First come first served	- Jobs completed in order of arrival. Other	-Simple algorithm which starts a job as soon as it reaches the front of the queue	- Once one job starts it prevents other jobs from being processed - Long jobs take longer

	processes wait in a queue		which decreases efficiency of processor - +Round Robin disadvantages
Shortest Job first	- Jobs ordered by how much time each job takes to complete	- Ensures max # of jobs completed - Minimises average time to process a task	- When a longer job is processed, it would be interrupted when a shorter job arrives in the queue (Could never complete job is short jobs keep coming) - +Round Robin disadvantages
Shortest remaining time	- Orders jobs by how much time they have remaining till completion	- Allows short processes to be handles very quickly - Ensures max # of jobs completed	- +Round Robin disadvantages
Multilevel Feedback queues	- This uses a number of queues. Each of these queues has a different priority.	- Enures higher priority processes run on time	- Complex to implement - Not efficient if jobs have similar priorities

(e) Distributed, embedded, multi-tasking, multi-user, real time OS

There are different types of operating systems:

- **Multi-tasking:** Allows more than one program to run simultaneously (Windows/Linux)
- **Multi-user:** Allows multiple users to operate one powerful computer using terminals
- **Embedded:** Handles a specific task on specific hardware (limited resources) (ATM)
- **Distributed:** Allows multiple computers (cluster) to work simultaneously on a problem as a single system. Shares data to reduce bottlenecks
- **Real-time:** The data is processed immediately and a response is given within a guaranteed time frame (Planes)
- **Batch**: The task of doing the same job over and over again (With different inputs/outputs)

(f) BIOS

Basic Input/Output system (**BIOS**) allows the computer to be 'booted up' when switched on

- When switched on, PC **points processor** to **BIOS memory to start u**p
- Check to see if computer is **functional/memory installed/processor functional**
- Stored in flash memory for **modification**

(g) Device drivers

- Normally **provided** with a **peripheral device** which contains instructions to enable the **peripheral** and **OS** to **communicate** and **configure hardware**.
- Enables multiple versions of OS to **communicate** with devices

(h) Virtual machines

- **Theoretical/Generalised computer** where a translator is available when programs are run
- Can **run OS** on a **software implementation**
- Uses an **interpreter** to run **intermediate code** (Slower than compiler)

Intermediate Code

- Partially translated/simplified code (high/machine code)
- Can be **produced by compiler** (if error free)
- Protects **source code** from being copied to keep intellectual property
- Platform **independent** = **improving portability**
- The program **runs more slowly** than **executable code** as it needs to be translated each time it is run by **additional software**.

1.2.2 Applications Generation

(a) Nature of applications

Software: Set of programs/instructions/code that runs on computers which makes **hardware work**. (**Applications/Utilities software**)

Applications Software:

- Allows user/hardware to carry out tasks
- E.g Word processor/spreadsheet packages/photo-editing suites/web browsers

(b) Utilities

Utilities Software:

- Small piece of systems software with **one purpose** usually linked with maintenance
- E.g Anti-Virus/Disk defragmentation/File managers

(c) Open source vs closed source

Open Source	**Closed Source**
- Free for others to examine/recompile	- Sold as license to use the software
- Users can create amended versions of program (Access to source code)	- Company/Developer holds copyright so users don't have access to source code
- No helpline since no commercial organisation	- Helpline/support available from company + regular updates + large user base
- E.g Linux/Firefox/Libre Office	- E.g MAC OS,iWork,Safari

(d) Translators: Interpreters, compilers, assemblers

Translators: Converts code from one language to another (Between HLL,LLL,source code,object,intermediate, executable,machine code). There are 3 types of translators:

- **Interpreters**
    - **Interprets & runs HLL code** by converting it to **machine code** & runs it before reading next line
    - Reports **one error at a time** (stops to show location of error)
    - Must be **present** each time the **program is run** so program runs **slower due to translation**
    - Source code (visible & changeable)
- **Compilers**
    - Converts **HHL source code** to **machine code**
    - Translates whole program as a **unit** + creates executable program when completed
    - Gives **list of errors** at end of compilation
    - **Not readable by humans** to protect intellectual property
    - **Machine dependent & architecture specific** (Different code needed)
    - Compiler is no longer needed when **executable code is used**
    - Produces **intermediate code** for **virtual machines**
- **Assemblers**
    - Uses **low-level source code** to translate assembly -> machine code
    - **Reserves storage** for instructions & data
    - One **assembly language instruction** is converted into one machine code instruction
    - **Many lines** needed for the simplest of tasks

(e) Stages of compilation

Compilation has several stages:

- **Lexical Analysis**
    - Comments/Whitespace removed from program
    - Remaining code turns into a **series of tokens** (sequences of characters)
    - **Symbol table** is created to keep track of variables/subroutines
- **Syntax Analysis**
    - **Abstract syntax tree** is built from **tokens** produced in lexical analysis
    - If any tokens **break rules of language** = Syntax errors generated
- **Code generation**
    - Abstract tree code is converted to **object code**
    - **Object code = machine code** before 'linker' is run
- **Code optimization**
    - **Tweaks code** to run as smoothly as possible

(f) Linkers, loaders, libraries

Linker: Combines compiled code with library code into a single executable file

Loader: Part of OS & responsible for loading a program into memory

Libraries: Pre-written bodies of code that can be used by programmers

- Save time/cover complex areas/different languages can be used together

1.2.3 Software Development

(a) Waterfall/Agile methodologies/Extreme programming/Spiral/RAD

Developing Software project:

Step	**Process**
Feasibility Study	- To carry out **enquiries** on whether the project is **possible** and **solvable**. Plans can be **revised** if there are **problems** - Analysts consider parameters such as: ● **Technical feasibility** – Is there **hardware/software** available to **implement the solution**? ● **Economic feasibility/cost benefit analysis** – Is the proposed solution possible to **run economically**? ● **Social feasibility** – Is the effect on the **humans involved** too **extreme** to be **socially acceptable/environmentally sound**? ● **Effect on company's practices and workforce** – Is there enough **operational skill** in the **workforce** to be **capable** of **running the new system**? ● **What is the expected effect on the customer?** - If customer **not impressed** then there **may not be a point**. ● **Legal/ethical feasibility** – Can the proposed system solve the **problem** within the **law**? ● **Time available** – Is the time scale **acceptable** for the **proposed system** to be **possible**?
Requirements Specification	- The **specification document** is developed between **client/software developers** creates an **understanding of a problem** and **solutions can be derived** - It states everything the new system is going to do including: ● **Input requirements** ● **Output requirements** ● **Processing requirements** ● **Clients agreement to requirements** ● **Hardware requirements** ● **Software requirements**
Testing	This process makes sure the project runs smoothly. There are 4 types of testing: - **Black-Box Testing:** Tests the **functionality** of the program without looking into the **internal structures/working**. **Only input/output** - **White-Box Testing:** Tests the **structure & workings of the application** as opposed to its functionality - **Alpha Testing:** Where **testers** in the **organisation test & identify** all possible **bugs/issues** before the product is released - **Beta Testing:** Test the program in a '**real environment**' with **limited end-users** so they provide **feedback** on the **functionality** of the program.
Documentation written throughout the process	- **Requirements specification:** Details **exactly what the system** will do - **Design:** Includes **algorithms/screen layouts/data storage** descriptions - **Technical Documentation:** Details how the **system works** for **future maintenance** E.g **Descriptions of code/modules & functionality** - **User Documentation:** Tell sythe user exactly how to **operate the system** E.g **Tutorials/Error messages descriptions/troubleshooting guide**

The waterfall lifecycle

- Series of **linear stages** presented in **order** (Can only go to next stage after previous is done)
- **Possible to back** if necessary
- List of stages: **Feasibility Study, Investigation/Requirements Elicitation, Analysis, Design, Implementation/Coding, Testing, Installation, Documentation, Evaluation, Maintenance**

Agile development methodologies

- A group of **methodologies** to cope with **changing requirements**
- Software produced in a **iterative manner** (Build on previous versions)

Extreme Programming (XP)

- Example of a **Agile Development Methodology** (Iterative in nature)
- **Customer is part of the team** to help decide '**users stories**' (Requirements/Tested)
- Each **iteration** creates a **version of the program** with code good enough to be **the final product**
- **Pair programming: One writes/one analyse = switch over**

The spiral model

- Designed to manage risk. 4 stages:
  - **Determine Objectives:** Determine objectives according to biggest risks
  - **Identify/Resolve Risks:** Risks identified & alternate solutions considered. Project stopped = Risk too high
  - **Development & Testing**: This is where the program is developed/tested
  - **Plan next iteration**: Determines what happens at next iteration

Rapid Application Development (RAD)

- Involves use of **prototypes**
- **Prototype** shown to **user** & **feedback given** to **amend prototype** until **user is happy**
- Constantly **developed** & **reviewed** by **user** until **user = satisfied**

(b) Merits and drawback of different methodologies

Waterfall Lifecycle	
Advantages	**Disadvantages**
- Suited to **large scale static projects**	- If changes occur, **hard to do = loss in time/money**
- Focuses on **early stage development**	- **Inflexible/limiting to change requirements**
- Focuses on **end user** (Can be involved in different parts of project)	- Dependent on '**clear requirements**' so there is little '**splash-back**'
- Progress of **development easily measurable**	- Produces **excessive documentation = time consuming**
- Generally **more progress forward** than backward	- **Missing system components** tend to be found during **design/development**
- **Orderly sequence guarantees quality**	- Performance **can't be tested until fully**

written documents	**completed**

Agile development methodologies (Extreme Programming)	
Advantages	**Disadvantages**
- New **requirements** adapted **throughout**	- **Client** has to be part of team which might be **inconvenient for them** - **Lack of documents** due to emphasis on coding = **not suitable for larger projects**
- **End-User** is integral throughput	
- **Pair programming allows code** to be **efficient/robust/well written**	
- Code is **created quickly** and **modules** available for user as they are done	

The spiral model	
Advantages	**Disadvantages**
- **Large amount of risk analysis** significantly **reduces risk** as **risks are fixed in early development stages**	- **High skilled team** needed for **risk analysis**
- **Software prototype** created early and updated in every **iteration**	- **Development costs high** due to number of **prototypes** created & **increased customer collaboration**

Rapid Application Development (RAD)	
Advantages	**Disadvantages**
- **End user** can see a **working prototype** early in project	- Emphasis on **speed & development** affects **overall system quality**
- **End user** more involved & can **change requirements** so **clear direction** on where the **program is heading**	- **Potential for inconsistent designs** & **lack of detail in documentation**
- Overall **development time** is **quicker reducing costs**	- Not suitable for **safety critical systems**
- **Concentration** on **essential elements** for **fast completion**	

1.2.4 Types of Programming Language

(a) Need for variety of programming paradigms

Paradigms = Methods

Many types of programming languages which are high/low level languages:

- **High-level languages**
- Uses language more similar to human language (English + Mathematical Expressions)
- Can be converted to **machine code**

- **Low-level languages**
- Directly linked to **architecture** of computer
- Machine/Assembly code are **low level**

(b) Procedural languages

- High level, 3rd gen, imperative languages
- Uses **sequences/selection/iteration**
- Program gives a **series of instructions** line by line on **what/how to** so an operation
- Statements are called **functions/procedures**
- **Breaks down** the solution into **subroutine blocks** which are rebuilt and combined to form the program
- Tasks completed in a **specific** way
- **Logic of program = series of procedure calls**
- E.g VB.NET/Python/C

(c) Assembly language (LMC)

Assembly code:

- Machine oriented language
- Closely related to **computer architecture**
- Uses **mnemonics** for instructions
- Translated by a **assembler**
- Easier to write than **machine code**, but **more difficult** than **HLL**.
- **Descriptive names** for **data stores**
- **Each instruction** is translated into **1 machine code instruction**.

LMC: fictional processor designed to illustrate the principles of how processors and assembly code work.

LMC instruction set:

Mnemonic	Function	Example Instruction	Explanation
`ADD`	Add	`ADD n`	Add the contents of n to the ACC
`SUB`	Subtract	`SUB n`	Subtract the contents of n from the ACC

`STA`	Store	`STA n`	Store the number n
`LDA`	Load	`LDA n`	Load the contents of n into the ACC
`BRA`	Branch always	`BRA number`	Unconditional jump to number label
`BRZ`	Branch if zero	`BRZ number`	Jump to number label if ACC contents is zero
`BRP`	Branch if positive	`BRP number`	Jump to number label if ACC contents is positive
`INP`	Input	`INP`	Prompt for a number to be input
`OUT`	Output	`OUT`	Outputs the contents of the ACC
`HLT`	End Program	`HLT`	Stops program execution
`DAT`	Data Location	`n DAT 10`	Creates data location n and stores the number 10 in it

(d) Modes of memory addressing

Different ways of accessing memory in low level languages:

- **Direct addressing**
  - **Simplest & most common** type of addressing
  - **Address** in the **memory** where the value actually is that should be used
  - “Instruction ADD 10 means go find data value in data location ‘10’ and add that value to the accumulator’
  - Used in **assembly language**

- **Indirect addressing**
  - The **operand** is the address of the data to be used by the **operator**
  - Useful for **larger memories**
  - E.g. in ADD 23, if address 23 stores 45, address 45 holds the number to be used.

- **Indexed addressing**
  - Modifies the address given by adding the number from the **Index Register** to the address in the instruction.
  - Allows **efficient access** to a range of memory locations by incrementing the value in the IR e.g. used to access an array
  - E.g Adding data value 5 to data location 20, 6 is at 21 etc
  - **Final address = base address + index**

- **Immediate addressing**
    - Used in **assembly language**
    - Memory remains as **constant** as it **doesn't change** (address field = constant)
    - Data in the **operand** is the **value** to be used by the **operator** e.g. ADD 45 adds the data value stored in data location '45' to the value in the ACC.

(e) Object-oriented languages (**OOP**)

- **Programming paradigm** which enables programs to **solve problems** by implementing components such as **objects** to work together to **create a solution**
- Most programs have OOP in them (Java/C++/C#)
- Components of OOP include:
    - **Classes**: Template used to define an object. Specifies what methods/attributes the object should have.
    - **Object**: Self-contained instance of a class based off real world entities made from attributes and methods.
    - **Methods:** Subroutines which forms the actions an object can carry out
    - **Attributes:** Value stored in variable associated with an object.
    - **Constructor:** Method describes how an object is created.

Features of OOP

- **Encapsulation**
    - Process of **hiding data within objects** to keep attributes **private**
    - Prevents objects being amended in **unintended ways**
    - **Private attributes** can only be amended by **public methods = maintains data integrity**
- **Inheritance**
    - When a class **inherits** it's **parents attributes & methods**
    - This class might have it's **own methods/attributes** which could **override** methods of the parent class (unless **superclass** is used)
    - The class can be used as a base for **different objects** to save time
- **Polymorphism**
    - Meaning **"Many Forms"**
    - Applies same method to **different objects = treated in same way**
    - Code written is able to **handle different objects** in the same way to reduce the **volume produced**

OCR A-Level Computer Science Spec Notes

1.3 Exchanging Data

1.3.1 Compression, Encryption and Hashing

(a) Lossy vs Lossless compression

Compression- The reduction of file sizes to:

- **Reduce** download times
- Make best use of **bandwidth**
- **Reduce file storage** requirements

There are 2 types of **compression:**

- **Lossy**
    - Some **data stripped out** to **reduce file size**
    - Information not r**ecoverable** hence **deleted** since it has **least importance**
    - Typically used for I**mages/Videos/Music files**. Data removed is not **noticeable** by **humans**
    - Common lossy formats: **JPEG/MP3/MPEG**
- **Lossless**
    - **Retains all data** by **encoding it efficiently**
    - The **original file** can be **regenerated**
    - Common lossless formats : **ZIP/GIF/PNG**

(b) Run length encoding and dictionary coding

There are 2 types of **encoding:**

- **Run Length Encoding**
    - **Stores redundant data** (pixels/words/bits) into groupings of bits
    - **Indexed and stored** on a dictionary/table + # of occurrences
    - Used in **TIFF/BMP files**
- **Dictionary Encoding**
    - **Compression algorithm** which uses a **known dictionary/own dictionary** to **encode data.**
    - File consists of **dictionary + sequence of occurrences**
    - **Substitutes entries** for **unique code** e.g (function = F_N)
    - Used for **ZIP/GIF/PNG files**

(c) Symmetric and asymmetric encryption

Encryption:

- The process of **scrambling data** that the only way to read it is to **decrypt it**
- Uses **encryption keys** (long random numbers) to **encrypt/decrypt messages**
- **Public key** = available to all / **Private key** = Available to owner only
- Long process to **encrypt & decrypt**

There are 2 types of **encryption:**

- **Symmetric Encryption**
    - **Same key** used to **encrypt/decrypt**
    - Requires **both parties** to have **copy of key**
    - **Can't be transferred over internet** = Easy to decrypt

    - **Stronger** than asymmetric (Same length)
- **Asymmetric Encryption**
    - **Different keys** to **encrypt/decrypt** = **More secure**
    - Public key = encrypt / Private key = decrypt
    - Example: TLS (**Transport Layer Security**) uses symmetric & asymmetric

(d) Different uses of hashing

Hashing:

- **Used to produce/check passwords**
- Stores data in **abbreviated form** e.g 123456 -> 456
- Difficult to **regenerate hash value -> original value**
- Vulnerable to **brute-force attacks**
- **Low chance of collision** (Different inputs = same output) = ↓ risk of files being the same
- **Easy to check** – the login attempt is hashed again

1.3.2 Databases

(a) Flat file and relational databases

Databases: Structured & Persistent stores of data for ease of processing

- Allow data to be: **Retrieved quickly/updated easily/filtered for different views**

Flat file Databases

- **Simple data structures** which are easy to **maintain** (limited data storage)
- Limited use due to **redundant/inconsistent data**
- No **specialist knowledge to operate**
- Harder to **update & data format** is **difficult** to **change**

Relational Databases

- Based on **linked tables** (relations)
- Based on **entities** (Rows & Columns)
- Each row (**tuple**) in a table is **equivalent** to a record and is **constructed** in the **same** way.
- Each **column** (attribute) is equivalent to a **field** and must have just **one data type**.
- Improves **data consistency & integrity**
- Easier to change data **format & update records**
- Improves **levels of security** so easier to access data
- **Reduces data redundancy** to avoid wasting storage

Primary Key (PK)

- Is a **unique identifier** in a table used to **define each record**.

Foreign Key (FK)

- **PK** in one table is used as an **attribute or FK** in another to **provide links** or relationships between tables.
- Represents a (**one to many**) **relationship** where the FK is at the "**many**" end of the relationship to avoid **data duplication**.
- This allows **relevant data** to be **extracted** from different tables.

Secondary Key (SK)

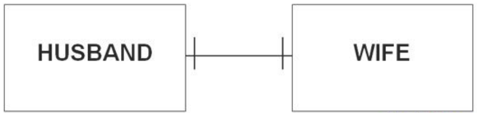
- An **attribute** that allows a **group of records** in a table to be **sorted** and **searched differently** from the PK and data to be accessed in a **different order**.

Entity Relationships

- Used to plan **RDB**
- Diagrams to show **relation**
- Helpful in **reducing redundancy**

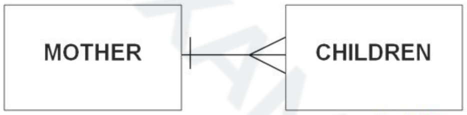
One-One Relationship

- Not suitable for relationship tables



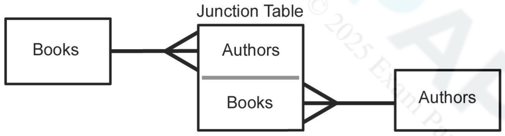
One-Many Relationship

- Used in well designed RBS



Many-Many Relationship

- Leads to data redundancy



Indexing

- The **PK** is **normally indexed** for **quick access**.
- The **SK** is an **alternative index** allowing for **faster searches** based on **different attributes**.
- The **index** takes up **extra space** in the **database**.
- When a **data table** is **changed**, the **indexes** have to be **rebuilt**.

Serial files

- Are **relatively short** and **simple** files.
- **Data records** are **stored chronologically** i.e. in the order in which they are entered.
- New data is always **appended** to the **existing records** at the end of the **file**.
- To **access a record**, you search from the **first item** and read each preceding item.
- **Easy** to **implement**.
- Adding **new records** is easy.
- **Searching** is easy but **slow**.

Sequential files

- Are **serial files** where the data in the file is **ordered logically** according to a **key field** in the record.

Indexed sequential files

- Records are **sorted according** to a **PK**
- A **separate index** is kept that allows **groups or blocks** of records to be accessed **directly** and **quickly**
- **New records** need to be inserted in the **correct position** and the index has to be **maintained** and **updated** to be kept in **sync** with the **data**
- Is more **difficult** the **manage** but accessing **individual files** is much **faster**
- More **space efficient**
- More **suited** to **large files**

Database Management System (DBMS)

- Is **software** that **creates**, **maintains** and **handles** the **complexities** of **managing a database**.
- May provide **UI**.
- May use **SQL** to **communicate** with other programs.
- Provides **different views** of the **data** for **different users**.
- Provides **security features**.
- **Finds**, **adds** and **updates** data.
- **Maintains indexes**.
- Enforces **referential integrity** and **data integrity rules**.
- Manages **access rights**.
- Provides the **means** to **create the database structures**: queries, views, tables, interfaces and outputs.

Queries

- **Isolate** and **display** a **subset of data**.
- **QBE**: query by example.

(b) Methods of capturing, selecting, managing, exchanging data

There are multiple ways to **capture/select/manage/exchange data** based on the scenario and what needs to be obtained. For example, a hotel would want the guests information so they can process payments.

(c) Normalisation to 3NF

Normalisation: There are 3 stages to normalisation:

- **1NF**
    - Separates **multiple items/ sets of data** in each row to remove **duplicate values**
- **2NF**
    - Removes data that occurs on **multiple rows** & puts data into **new table**
    - **Creates relationship links** between **tables** as necessary by **repeated fields**
- **3NF**
    - Removes fields not **directly related** to the **primary key** to their own **linked table** so every value left **depends on the key**

(d) SQL: **Structured Query Language**

SQL Command	Explanation & Example
CREATE TABLE	Creates an Empty Table: *Create Table_Name (* *column1 datatype,* *column2 datatype,* *column3 datatype,*)
DROP	Remove database components (ALTER TABLE can be used to delete column): ALTER TABLE *green* DROP COLUMN *name*;
INSERT	Adds values into records in tables: INSERT INTO *example(name, dob)* VALUES
DELETE	Deletes data from table_name: DELETE FROM "example" WHERE
SELECT	Lists the field name to be displayed: SELECT "Name"
WHERE	Lists the search criteria for the field value: WHERE "*Name*" = '*Fred*'
AND	Works when both expressions are true: "*Name*" = '*Cox*' AND "*Order*" < 3
FROM	Lists the table the data comes from:: FROM "*tblCustomer*"

(e) Referential integrity

- **Transactions** should maintain **referential integrity**.
- This means keeping a database in a **consistent state** so changes to data in one table must take into **account data** in **linked tables**
- Enforced by **DBMS**.

(f) Transaction processing (ACID), record locking and redundancy

ACID rules protect integrity of database:

- **Atomicity:** A change is either performed or not. Half finished changes not saved.
- **Consistency:** Any change must retain the overall state of database
- **Isolation:** A transaction must not be interrupted by another
- **Durability:** Changes must be written to storage in order to preserve them

Record locking
- **Preventing simultaneous access** to objects in databases to **prevent losses** in updates or data inconsistencies
- A record is **locked** when a user retrieves it from editing/updating
- Anyone else trying to access record is **denied access** until record is completed/cancelled

Data Redundancy
- Is **unnecessary repetition of data** that leads to inconsistencies
- Data should have **redundancy** so if part of a database is **lost** it should be **recoverable from elsewhere**
- Redundancy can be provided by RAID setup or mirroring servers.

1.3.3 Networks

(a) Characteristics of a networks, importance of protocols/standards

Network: interconnected set of devices

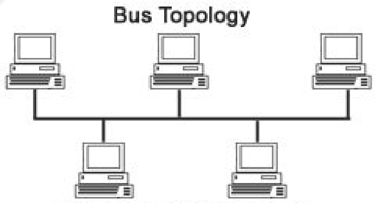
Frame: A unit of data sent on a network

Private Networks

Advantages	Disadvantages
- Security (Control of access)	- Specialist staff/security/backups needed
- Confidence of availability	

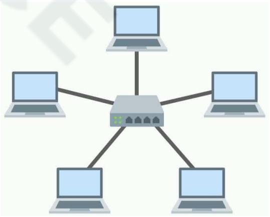
Network Topologies: the layout of a network

Different types of Network Topologies



- **Bus**
  - Nodes attached to single **backbone** = vulnerable to changes
  - Prone to **data collisions**
  - Uncommon now

- **Ring**
  - Nodes attached to exactly **2 other nodes**
  - Data sent in 1 direction to **avoid collisions**
  - Easily **disrupted**



- **Star**
  - **Most** networks are star layouts
  - **Resilient**
  - **Speratrate** link from each node to **switch/hub**

Standards/protocols- Set of rules relating to the **communication of devices & data transmitted between them:**
- Examples: TCP/IP stack

Open Systems Interconnection (OSI) model

- An openly available (non-proprietary) network model.

7 layers in the **OSI model**:

- 7 – **Application**: collecting and delivering data in the real world.
- 6 – **Presentation**: data conversions.
- 5 – **Session**: manages connections.
- 4 – **Transport**: packetizing and checking.
- 3 – **Network**: transmission of packets, routing.
- 2 – **Data Link**: access control, error detection and correction.
- 1 – **Physical**: network devices and media.

(b) The internet structure

- **The TCP/IP Stack:**
    - Suite of protocols cover **data formatting, addressing, routing** and **receiving**. Equivalent to layers **7,4,3,2** of **OSI model**

4 layers of abstraction

Layer	Purpose
Application (7)	Capturing/delivering data & packaging
Transport (4)	Establishment/termination of connections via routers
Network (3)	Provides transmission between different networks. Concerned with IP addressing and direction of datagrams.
Link (2)	Passes data onto physical network (Copper wire/optical fibre/wireless)

- (Domain Name System) **DNS:**
    - **Hierarchical system** for **naming resources** on a network
    - Human readable equivalent to IP address (e.g www.google.co.uk instead of 64.256.201.765)
    - Domain names translates **URLs** to **IP addresses**
    - If server can't resolve it passes request **recursively** to another server which sends **IP address** to **browser** so it can retrieve **website hosted** from server.
- **Protocol layering**
    - Form of **abstraction**
    - Divides **complex system** into **component parts** of functionality
    - Gradually allows work to be completed & allows **efficient problem solving**
    - Each layer **communicates** only with **adjacent layers**

Layers of abstraction

Layer	Purpose
Application (7)	The hardware that provides the connections.
Network (3)	Concerned with routes from sender to recipient.
Physical (1)	Hardware that provides the connections

- **Network Types (WAN/LAN etc)**
    - **Local Area Network (LAN)**
        - Confined to one location (school/business)
        - Infrastructure maintained by organisations that owns it
    - **Wide Area Network (WAN)**
        - Covers a large geographical area
        - Makes use of communication providers (BT,Virgin)
        - Internet is a WAN but special case (multiplicity of users)

- **Packet and circuit switching**

Packet Switching	Circuit Switching
- Connectionless node	3 Stages: - connection establishment - data transmission - connection termination.
- Divides message into data units called **packets**	- Exclusive **dedicated channel** which physically connects devices together
- Sent across the **most efficient** route (Not predetermined)	- Suitable for **intensive data transfer**
- At each node = destination read = most convenient route taken	- Packets **remain in order** but **reassembled** at **destination**
- Packets arrive **out of order** (reordered at destination)	- All packets go on **same route** in order
- Only as **fast** as **slowest packet**	- Sets up route between **2 computers** for duration of message
- Errors **resubmitted** if any occur	- Ties up **large areas of network** so no other data can use any part of the circuit until the **transmission is complete**.
- **Error checking** promotes **successful transmission.**	

(c) Network security

Authentication

- Protects users using a **username & password**
- As networks are more easily **hacked** into, new security systems implemented by using:
    - **Multiple credentials /smart cards/ biometric information (fingerprints/iris scans)**

Firewalls

- Various **combinations** of **hardware/software** that isolate a network from the outside world
- Configurable to **deny access** to **certain addresses/data**

Proxies

- Computers **interposed** between **networks & remote resource**
- Control **input/output** from a **network**

Encryption

- Most traffic is made **unintelligible** to **unauthorised individuals**
- Key is needed for **sender to encrypt** and **receiver to decrypt**
- **Bigger the key = more encryption**
- **Asymmetric key encryption** (Public/Private key)

(d) Network hardware

Network Interface Card/Controller

- Generates/Receives electrical signals
- Works at the physical/data link layers

Router

- **Device to connect networks**
- **Receives/Forwards data packets**
- Directs packets to next device (Uses table/algorithm to decide route)

MAC address

- **48 bit identifier**
- Permanently added to device by manufacturer
- Human readable **group of 6 bytes**

Switches

- Devices to **connect** to other devices on networks
- **Packet switching** to send data to specific destinations (Using hardware addresses)
- Operates at **Lvl ⅔** of **OSI model**

Hubs

- Connects **nodes together** by broadcasting a signal to all possible destinations
- Correct destination accepts signal

Wireless Access Points

- Usually **connected to a router**
- **Data link layer**
- Used to **connect devices to Wifi**

(e) Client-server and peer to peer

Client server

- **High end computers** act as **servers**
- Client **computer requests services** from server
- Services provided: File storage/access, printing, internet access, security features (login)
- **Less complex = more accessible**
- Computers don't have to be **powerful/expensive**
- Servers upgraded to fix **security issues/provide** more features

Peer-Peer Server

- All computers = **equal status**
- Computers can act as **client &/ server**

- Useful on internet so traffic can **avoid servers**
- Cheaper as its **private** so no expensive hardware/bandwidth needed
- More likely to be **fault tolerant**

1.3.4 Web Technologies

(a) HTML, CSS, JavaScript

World Wide Web (WWW)

- Collection of **billions** of **web pages**
- Written in **HTML** (have hyperlinks)
- Tags to indicate how text is to be handled.
- Assets: **Images/Videos/Forms/Applets**

Browsers

- Software that **renders HTML pages**
- Find **web resources** by **accepting URLs** and following links
- Find resources on **private networks**
- Browser examples: **Chrome/Safari/Opera/IE/Firefox**

Standards: Set of guidelines used universally so all computers can access the same resources.

Examples of standards

- **HTML (Hyper Text Markup Language)**
    - Create web pages & elements
    - Has **tags**: Mark out elements on page to show browser how to process element
    - **Links:** redirects user from current page to page referred by link
- **CSS (Cascading Style Sheets)**
    - Determines how **tags affect objects**
    - Used to **standardise** an appearance of a webpage
    - Changes made can **affect whole site** instead of one page
    - **Content** and **formatting** are kept **separate**
    - Simpler **HTML** used as **CSS** can be used in **multiple files**
    - Adjustable for **different devices**
- **JavaScript**
    - Programing language which runs on **browsers & controls elements**
    - Embedded into **HTML** with <script> tags to add functionality such as:
        - Validation/animation/Newer content
    - Used on **client side** = less strain on server & server side as it can be amended
    - Can run on any **browser** (normally interpreted)

(b) Search engine indexing

Search Engines: Web based software utilities that enable users to find resources on the web

- Builds **indexes**
- Uses **algorithms** to **complete searches** & **web-crawling bots** to collect indexes
- Supports many human languages

Search Engine Indexing (SEI)

- Is the process of **collecting** and **storing data** from **websites** so that a search engine can quickly **match** the content against **search terms**.

(c) PageRank algorithm
- Developed by **Google**
- Attempt to **rank pages by usefulness/importance**
- Takes into account: # of **inward/outward links** & # of sites that **link** to current site
- The PageRank of the linking sites – the algorithm iteratively calculates the **importance** of each site so that links from sites with a **high importance** are given a **higher ranking** than those linked from sites of low importance.

$$PageRank\ of\ site = \sum \frac{PageRank\ of\ inbound\ link}{Number\ of\ links\ on\ that\ page}$$

OR

$$PR(u) = (1 - d) + d \times \sum \frac{PR(v)}{N(v)}$$

(d) Server and client side processing

Server Side Processing: Processing that takes place on the web server

Pros	Cons
- **High security**: data sent to server for processing then sent back	- **Extra load** on the server makes running the server **more expensive**
- **Hides code from user** to protect copyright & being amended	
- No need to rely on browser having correct interpreter	

Is best used where processing is **integral** e.g. generating content and accessing data including secure data so any data passed must be checked carefully.

Client Side Processing: Processing that takes place on the web browser

Pros	Cons
- More processing = Reduced load on server = Reduced data traffic	- Code is visible so can be **copied** - Browser may not run the code as it doesn't have the **capability/ user intentionally** disabled client code
- **Quick feedback** to user	
- More **responsive** code	
- Data doesn't need to be sent to server and back	

Is best used when it’s not **critical code** that runs. If it is critical then it should be carried out on the **server**. Is also best where **quick feedback** to the user is needed – an example being games.

OCR A-Level Computer Science Spec Notes

1.4 Data types, data structures and algorithms

1.4.1 Data Types

(a) Primitive data types

Data types (All stored in the computer in **Binary):**

- **Integer**: Single **whole number** e.g (5,37,-102)
- **String**: A **sequence** of **alphanumeric characters** e.g (**3A*s**)
- **Real**: Numbers with **decimal/fractional** components e.g (3.14, 0.6)
- **Character**: Single **digit/letter/symbol** e.g (s,G,9,&)
- **Boolean:** Used to represent **Binary logic** (True/False, 0/1)

(b) Represent positive numbers in binary

- **Binary** is a **base 2 number system** whereas **denary** has a **base 10**
- To convert from **Binary -> Denary** (How I personally do it) **I will convert 200 to denary**:
  - Create this **nifty table** (It looks btec but still) (apparently called the **tabular method** according to Teach ICT) :

128	64	32	16	8	4	2	1
1	1	0	0	1	0	0	0

  - Firstly we know **128** goes into **denary** so put a **1** under **128** in the table
  - Then do **200-128 = 72**. Now **64** goes into **72** once so put a **1** in that
  - Now do **72-64 = 8**. 32 & 16 don't go into 8 but **8** does do put a **1** in there
  - **Fill** the **rest** of the boxes with **0**
  - Finito (**Answer: 11001000**)
  - **To summarize, keep subtracting and seeing whether the numbers in the top row go into the subtracted value.** It's hard to explain just practice lmao.

(c) Sign and magnitude & two's complement for negative numbers

Sign & Magnitude:

- In denary, store a sign bit, a '+' or '–' as **part of the number**
- Simply use the most left-handed bit, to store these as a binary value, **0 for + and 1 for –**

<u>**Corresponding Steps**</u> (example 127 & -127)

Sign Bit	64	32	16	8	4	2	1
0 (+)	1	1	1	1	1	1	1

= 127

Sign Bit	64	32	16	8	4	2	1
1 (-)	1	1	1	1	1	1	1

=-127

Two's Complement: An easy method for subtraction (**Overpowered** if use correctly**)**:

1. Convert subtraction number into **binary**
2. Start from most **right** and keep all values the same until you reach the first **'1'.** Then after that **switch '1's with '0's and '0's for '1's'**
3. Add the **binary numbers** and **discard** the **overflow**

<u>**Corresponding Steps**</u> (example Convert 75-35)

1. 35 in Binary = 00100011
2. 11011101 (-35)
3. Add 75 therefore (75+(-35)) = 01001011 + 11011101 = 0101000

(d) Addition and subtraction of binary numbers

Binary Addition (Check answer by doing in denary then converting**)**

- **0+0=0** / **1+0=10** (0 but **carry 1** to next calc) / **1+1 = 11** (1 but **carry 1** to next calc)

Binary Subtraction **(**Check answer by doing in denary then converting**)**

- **1-0=1** / **1-1=0** / **10- 1 = 1**

(e) Represent positive numbers in hexadecimal

Hexadecimal uses a **Base 16 Number System (4 bit system** as **2^4=16**)

- Same as denary upto 9 then letters are used where:
    - **10=A/11=B/12=C/13=D/14=E/15=F**

(f) Convert positive integers between binary, denary and hex

Denary-> Binary (E.g **Convert 81 to Binary**)

- Refer to **(b) Represent positive numbers in binary**

Binary -> Denary (E.g Convert 0101 1010 to Denary)

- Plug in Binary numbers into **Nifty Table**

128	64	32	16	8	4	2	1
0	1	0	1	1	0	1	0

- Just add the numbers which have a 1 below them (64+16+8+2 = 90)

Binary -> Hexadecimal (E.g Convert 0101 1010 to Hexadecimal)

- Split byte into **2 nibbles (0101 1010**)
- Convert each nibble separately into Hexadecimal (0101 = 5 / 1010 = 10 = A)
- Combine the result together: **5D**

Denary -> Hexadecimal

- Convert Denary into Binary
- Follow instructions for: **Binary -> Hexadecimal**

Hexadecimal -> Binary

- Split each Hex letter/number up
- Convert each letter/number into **binary equivalent**
- Join binary up again

Hexadecimal -> Denary

- Follow instructions for: **Hexadecimal-> Binary**
- Convert Binary into Denary

Alternate Way

- multiply each corresponding Hex digit with increasing powers of 16
  3B = $3\times\mathbf{16}^{1}+11\times\mathbf{16}^{0}$ = 48+11 = 5910

$$\text{decimal} = d_{n-1}\times16^{n-1} + \ldots + d_3\times16^3 + d_2\times16^2 + d_1\times16^1+d_0\times16^0$$

(g) Representation and normalisation of floating point numbers (**Mantissa is normally 5 bits & exponent is 3. Question will tell you if it changes**)

Floating Point Numbers: A way of storing decimals in Binary

1. If a number is **positive/negative**, look at the first binary digit: 0 =positive, 1 = negative
2. Split into **mantissa** and **exponent**.
3. Use the **exponent** to float the binary point back into place (put decimal point after first number)
4. Convert to denary.

Negative Values:

1. Split into **mantissa** and **exponent**.
2. Evaluate the **exponent**
3. Move the binary point **one place** to the left (If exponent is -1 for example)

The number of bits chosen for the mantissa & exponent affects the **range** and the **accuracy** of the values that can be stored:

- If **more bits** are used for the mantissa = more **accurate** values.
- But the **range** is **limited** by the **small exponent**.
- If more bits are used for the **exponent**, = **range** of **values** stored is **greater**.
- But the accuracy is **limited** by the **smaller mantissa**.

(h) Floating point arithmetic, +ve, -ve, addition, subtraction

Addition of Floating Point Numbers (E.g **01011 001 + 01100 010**)

1. Figure out what the **exponents** of the **2 bytes** (001 = 1 & 010 =2)
2. Shift **Mantissas** according to **exponents** (010110 -> 01.011 & 01100 = 011.00)
3. Add **digits** together (01.011 + 11.00 = 100.011)

Subtraction of Floating Point Numbers

1. Figure out what the **exponents** of the **2 bytes** (001 = 1 & 010 =2)
2. **Shift Mantissas** according to **exponents** (010110 -> 01.011 & 01100 = 011.00)
3. Add **digits** together (01.011 + 11.00 = 100.011)

(I) Bitwise manipulation and masks

Bitwise manipulation: The CPU is able to shift and mask binary to complete a range of operations.

- Binary can be logically shifted left/right
- Shifting Left = *2 & Shifting Right = /2
- E.g Shifting **0001** (**1**) to the left = **0010** (2) & **1*2=2**

Masking: Data used for bitwise operations. Using a mask (Byte/Nibble/bit etc) can be altered by a bitwise manipulation.

- **NOT** performs a bitwise swap of values in a binary number **(0 -> 1 & 1 -> 0)**
- **AND** excludes bits by placing a 0 in the appropriate bit in the mask
    - **(0 AND 0 =0 / 0 AND 1 = 0 / 1 AND 1 = 1)**
- **OR** resets bits by placing a 1 in the appropriate bit in the mask.
    - **(0 OR 0 =0 / 0 OR 1 = 1 / 1 OR 1 = 1)**
- **XOR** checks if corresponding bits are the same.
    - **(0 XOR 0 =0 / 0 XOR 1 = 1 / 1 XOR 1 = 0)**

(j) Character representation (ASCII and UNICODE)

Character Set

- Normally equates to the **symbols** on the **keyboard** that are represented by the computer by unique **binary numbers** and may include **control codes**
- Number of bits used for one character is **1 byte**
- number of characters tend to be a **power of 2** and uses more bits for an **extended set**.

ASCII (American Standard Code for Information Interchange)

- **ASCII** is a **256 character set** which is based on a **8-digit binary pattern** (7 bits + parity bit)
- The **limited character set** makes it impossible to display other **characters** & symbols outside the **English alphabet**

UNICODE

- UNICODE was originally a **16-bit coding system** but now has over **65000**
- Updated to remove the **16-bit restriction** by using a **series of code pages** with each page representing the **chosen language symbols**.
- Original **ASCII representations** are included with the **same numeric values**

1.4.2 Data Structures

(a) Arrays (3D), records, lists, tuples

Arrays

- **Data structure** which contains a **set of data items** of the **same data type** grouped together under a **single identifier**
- **Static data structure** (Size can't change)
- Each element can be **accessed & addressed quickly** by accessing the **index/subscript**
- Stored **contiguously** in memory
- **Multi dimensional** (1D (**Spreadsheet**), 2D (**Table**), 3D (**Multiple Tables**))

Records

- Data stores organised by **attributes** (fields) containing **one item** of data

Lists

- **Abstract data type** where the same item can occur **twice**
- Data stores **organised** by an **index**

Tuples

- Ordered set of values which are **immutable** (can't be modified)
- **Multiple data types** stored as it's similar to a list

(b) Linked lists, graphs, stack, queue, tree, binary search tree, hash table

Linked Lists

- **Dynamic data structure**
- Uses **index values/pointers** to sort lists in specific ways
- Can be organised into more than **one category**
- Needs to be **traversed** until **desired element** is found
- To add data: data added to the next **available space** & **pointers adjusted**
- To remove data: Pointer from **previous item set** to **item that will be removed** which **bypasses** the **removed item**
- The contents may not be **stored contiguously** in memory.

Graphs

- Set of **vertices/nodes** connected by **edges/arcs**
- Can be represented by an adjacency **matrix**
- Edges can be:
  - **directional** or **bi-directional**
  - **directed** or **undirected**
  - **weighted** or **unweighted**
- Searched by **breadth/depth first traversal**

Stack

- **LIFO** (Last In First Out)
- **2 pointers (Top/bottom)** Top Pointer = **Stack Pointer**
- Data is **added** (**PUSH**) and **removed** (**POP**) from the **top of the stack**
- **Stack overflow**: When data is trying to go into **stack** but **stack is full**

Queue

- **FIFO** (First In First Out)
- **2 pointers (Start/End)** Start Pointer = **Queue Pointer**
- Data is **added (enqueue)** from **end** & **data removed (dequeue)** from the **top of queue**

Tree

- Are **dynamic branching** data structures.
- They consist of **nodes** that have **sub nodes** (**children**).
- The **first node** at the start of the tree (**root node**)
- The lines that join the nodes are called (**branches**)

Binary search tree

- Each node has a maximum **of 2 children** from a **left branch** and a **right branch**.
- To add data to the tree, it is placed at the end of the list in the **first available space** and added to the tree following the rules:
  - If a child node is **less than a parent node**, it goes to the **left** of the parent.
  - If a child node is **greater than a parent node**, it goes to the **right** of the parent.

Hash table

- Enable **access to data** that is not **stored in a structured manner**.
- **Hash functions** generate an **address** in a table for the data that can be **recalculated** to **locate** that data.

1.4.3 Boolean Algebra

(a) Defining a problem using Boolean logic

Boolean Logic

- **NOT** (**Negation**) Symbol: ¬ (e.g if A=0 -> ¬A =1 / A=1 -> ¬A = 0)
- **AND** (**Conjunction**) Symbol: ^ (e.g if A=1 & B=1 -> A^B =1 Otherwise A^B = 0)
- **OR** (**Disjunction**) Symbol: v (e.g if A=1 / B=1 -> AvB =1 Otherwise AvB = 0)
- **XOR** (**Exclusive Disjunction**) Symbol: v̲ (e.g if A=1 / B=1 (Not other)-> AvB =1 Otherwise Av̲B = 0)
- **NAND** (**Conjunction**) Symbol: ¬(A ∧ B) (e.g if A=1/0 & B=1/0 -> AvB = 0/1 Otherwise AvB = 1)
- **NOR** (**Disjunction**) Symbol: ¬(A V B) (e.g if A=1/0 & B=1/0 -> AvB = 1/0 Otherwise AvB = 0)

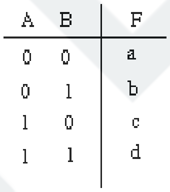
(b) Manipulating Boolean expressions (Karnaugh maps)

Karnaugh maps

- Are a **visual method** for simplifying **logical expressions**.
- They **show** all the outputs on a grid of all **possible outcomes** (Truth Table)
- The method is to **create blocks** of **1s** as **large** as possible so that the 1s are covered by as **few blocks as possible** and **no 0s are included**.
- The blocks can **wrap** around the diagram if necessary, in **both directions**, from **side to side** or from **top to bottom**.

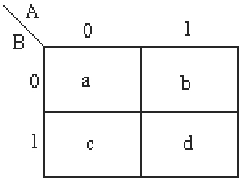
The rules for using Karnaugh maps:

- **No 0s (zeros)** allowed & diagonal blocks
- Larger groups the better
- Every 1 must be **within a block**
- Overlapping blocks allowed
- Wrap around blocks allowed
- Aim for **smallest possible groups**



A	B	F
0	0	a
0	1	b
1	0	c
1	1	d

Truth Table.



B \ A	0	1
0	a	b
1	c	d

F.

Karnaugh Map

(c) Simplifying statements in Boolean algebra using rules

Equivalence / Iff (if and only if)

Symbol (AND):

- ≡ e.g. (A ∧ B) ≡ ¬(¬A V ¬B) ------> (A AND B ≡ NOT (NOT A OR NOT B))

Alternative notations (XOR):

- e.g. (A ⊻ B) ≡ (A ∧ ¬B) V (¬A ∧ B) ------> (A XOR B ≡ (A AND NOT B) OR (NOT A AND B))

Boolean algebra

There are rules, similar to arithmetic (**Statistics** if you take **A-level Maths**), for manipulating

Boolean expressions:

Boolean Rule	Boolean Expression	Description
De Morgan's laws	• ¬(A V B) ≡ ¬A ∧ ¬B • ¬(A ∧ B) ≡ ¬A V ¬B	• A NOR B ≡ NOT A AND NOT B • A NAND B ≡ NOT A OR NOT B
Distribution	• A ∧ (B V C) ≡ (A ∧ B) V (A ∧ C) • A V (B ∧ C) ≡ (A V B) ∧ (A V C)	• A AND (B OR C) ≡ (A AND B) OR (A AND C) • A OR (B AND C) ≡ (A OR B) AND (A OR C)
Association	• (A ∧ B) ∧ C ≡ A ∧ (B ∧ C) • (A V B) V C ≡ A V (B V C)	• (A AND B) AND C ≡ A AND (B AND C) • (A OR B) OR C ≡ A OR(B OR C)
Commutation	• A ∧ B ≡ B ∧ A • A V B ≡ B V A	• A AND B ≡ B AND A • A OR B ≡ B OR A
Double Negation	• ¬(¬ A) ≡ A	• NOT(NOT A) ≡ A
Simplification Expressions (1 = True 0 = False)	**AND** • A ∧ A ≡ A • A ∧ 0 ≡ 0 • A ∧ 1 ≡ A • A ∧ ¬A ≡ 0 **OR** • A V A ≡ A • A V 0 ≡ A • A V 1 ≡ 1 • A V ¬A ≡ 1	**AND** • A AND A ≡ A • A AND 0 ≡ 0 • A AND 1 ≡ A • A AND NOT A ≡ 0 **OR** • A OR A ≡ A • A OR 0 ≡ A • A OR 1 ≡ 1 • A AND NOT A ≡ 1
Absorption	• A V (A ∧ B) ≡ A • A ∧ (A V B) ≡ A	• A OR (A AND B) ≡ A • A AND (A OR B) ≡ A

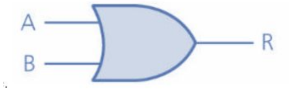
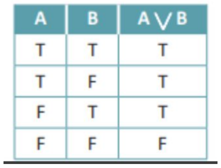
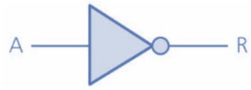
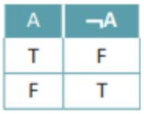
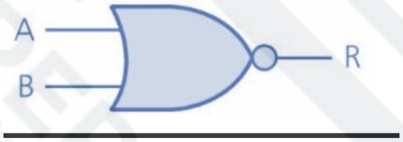
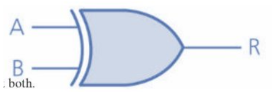
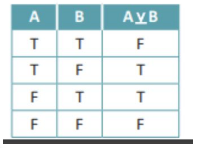
(d) Logic gate diagrams and truth tables

Logic Gates: Building block of a digital circuit used to implement Boolean functions

Truth Table: Mathematical table used with logic gates to list out all possible scenarios of the corresponding logic gate(s)

Logic Gates Examples:

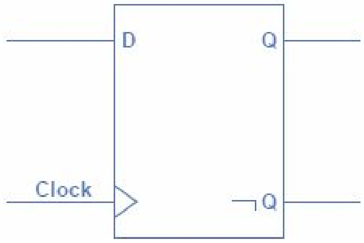
Type of GATE	Boolean Expression	Diagram	Truth Table

<table>
<tr><td>AND</td><td>A ∧ B</td><td>
</td><td>
<table><tr><th>A</th><th>B</th><th>A∧B</th></tr><tr><td>T</td><td>T</td><td>T</td></tr><tr><td>T</td><td>F</td><td>F</td></tr><tr><td>F</td><td>T</td><td>F</td></tr><tr><td>F</td><td>F</td><td>F</td></tr></table></td></tr>
<tr><td>OR</td><td>A V B</td><td>
</td><td>
<table><tr><th>A</th><th>B</th><th>A∨B</th></tr><tr><td>T</td><td>T</td><td>T</td></tr><tr><td>T</td><td>F</td><td>T</td></tr><tr><td>F</td><td>T</td><td>T</td></tr><tr><td>F</td><td>F</td><td>F</td></tr></table></td></tr>
<tr><td>NOT</td><td>¬A</td><td>
</td><td>
<table><tr><th>A</th><th>¬A</th></tr><tr><td>T</td><td>F</td></tr><tr><td>F</td><td>T</td></tr></table></td></tr>
<tr><td>NAND</td><td>¬(A ∧ B)</td><td>
</td><td><table><tr><th>A</th><th>B</th><th>¬(A ∧ B)</th></tr><tr><td>F</td><td>F</td><td>T</td></tr><tr><td>F</td><td>T</td><td>T</td></tr><tr><td>T</td><td>F</td><td>T</td></tr><tr><td>T</td><td>T</td><td>F</td></tr></table></td></tr>
<tr><td>NOR</td><td>¬(A V B)</td><td>
</td><td><table><tr><th>A</th><th>B</th><th>¬(A V B)</th></tr><tr><td>F</td><td>F</td><td>T</td></tr><tr><td>F</td><td>T</td><td>F</td></tr><tr><td>T</td><td>F</td><td>F</td></tr><tr><td>T</td><td>T</td><td>F</td></tr></table></td></tr>
<tr><td>XOR</td><td>A ⊻ B</td><td>
</td><td>
<table><tr><th>A</th><th>B</th><th>A⊻B</th></tr><tr><td>T</td><td>T</td><td>F</td></tr><tr><td>T</td><td>F</td><td>T</td></tr><tr><td>F</td><td>T</td><td>T</td></tr><tr><td>F</td><td>F</td><td>F</td></tr></table></td></tr>
</table>

(e) D type flip flops, half and full adders

D type flip flops

- Store the state of a data bit in RAM
- **D = Delay**
- 2 Inputs: data(D) & clock
- 2 Outputs: the delayed data (**Q**) and the inverse of the delayed data (**¬Q**)



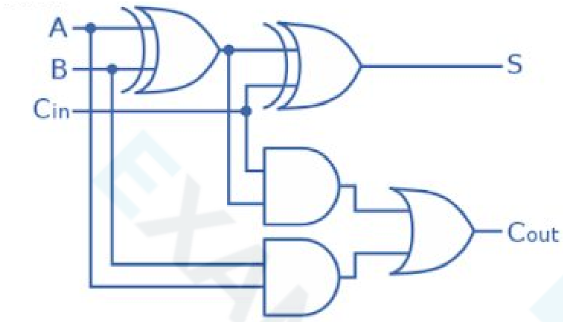
Half Adders

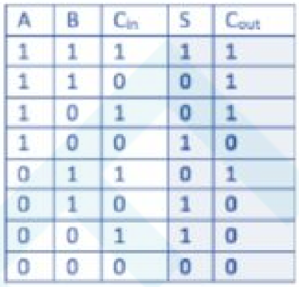
- Half adders logic circuit with 2 inputs & outputs
- The sum (S) is an XOR gate (A XOR B)
- The carry (C) is an AND gate (A AND B)

Full adders

- Combination of half adders to make a full adder
- The sum (S) is an XOR gate
- The carry (C) is an AND gate
- A B and Cin are added together = The result is given in the sum (S) N And a carry bit in Cout

  Series of full adders combined together allows computers to add binary numbers.





A	B	Cin	S	Cout
1	1	1	1	1
1	1	0	0	1
1	0	1	0	1
1	0	0	1	0
0	1	1	0	1
0	1	0	1	0
0	0	1	1	0
0	0	0	0	0

OCR A-Level Computer Science Spec Notes

1.5 Legal, moral, cultural and ethical issues

1.5.1 Computing related legislation

(a) **Data Protection Act 1998**

- Is designed to protect **personal data** and focuses on **controlling the storage of data** about the **data subject**.
- All data users must **register** with the **Data Commissioner**

There are eight provisions:

- Data must be **processed fairly & lawfully**
- Data must be **adequate, relevant & not excessive**
- Data must be **accurate & up to date**
- Data must not be **retained for longer than necessary**
- Data can only be **used for the purpose for which it was collected**
- Data must be kept **secure**
- Data must be **handled in accordance with people's rights**
- Data must not be **transferred outside the EU without adequate protection**

(b) **Computer Misuse Act 1990**

Law aimed at **illegal hackers** who hack to **exploit systems**

- Offence to gain **unauthorised access** to **computer material**
  - WIth intent to **commit/facilitate commision** of **further crimes**
  - With intent to **change the operation** of a **computer (Disturbing Viruses)**

(c) **Copyright, Design and Patents Act 1988**

- Any **individual/organisation** who **produces media/software/intellectual property** has the **ownership protected** by the **act**
- Other parties not allowed to **copy/reproduce/redistribute** without **permission** from **copyright owner**

(d) **Regulation of Investigatory Powers Act 2000**

This act is about the use of the internet by **criminals/terrorists.** Regulates how authorities **monitor our actions**. Certain organisations can:

- Demand **ISPs** to provides **access** to a **customer's communications.**
- Allow **mass surveillance** of **communications**.
- Demand **ISPs** fit **equipment** to **facilitate surveillance**
- Demand **access** be **granted** to **protected information**
- Allow **monitoring** of an **individual's internet activities**.
- Prevent the **existence** of such **interception activities** being **revealed in court**.

1.5.2 Moral and ethical issues

The individual moral, social, ethical and cultural opportunities and the risks of digital technology:

- **Computers in the workforce**
  Skill Sets for people have changed as technology advances:
  - **Robot manufacturing**: Less direct manufacturing roles & more technical/maintenance roles
  - **Online shopping**: Less in-store jobs/more distribution (logistics) jobs
  - **Online banking**: Closure of high street bank branches

- **Automated decision making**
  Decisions which can be made by computers/systems. Depends of quality/accuracy of data & precision of algorithm
  - **Electrical Power Distribution**: Rapid responses to changing circumstances
  - **Plant Automation**
  - **Airborne collision avoidance systems**
  - **Credit assessments**: Banks use system to create automatic assessments
  - **Stock Market Dealing**: Automated Could have caused 'flash crash' (2010)

- **Artificial intelligence**
  Perceived to either be beneficial or disadvantageous. AI is used daily for example:
  - **Credit-card checking**: Looks for unusual credit card use to identify fraudulent activity
  - **Speech recognition:** Identify keywords/patterns to interpret meaning of speech
  - **Medical diagnosis systems**: Self-diagnose illnesses & help medics in making diagnoses
  - **Control systems:** Monitor/interpret/predict events
- **Environmental effects**
  - **Computers are composed of**: airborne dioxins, polychlorinated biphenyls (PCBs), cadmium, chromium, radioactive isotopes, mercury
  - **Handled** with **great care** during **disposal**
  - **Shipped off** to countries with **lower environmental standards**
  - Workers/children extract **scrap metal** from **discarded parts** which are **recycled/sold**

- **Censorship and the Internet**
  - **Suppression** on what can be **accessed/published**
  - Material which is acceptable **depends on the person**
  - Some countries apply **censorship for political reasons**
  - Organisations e.g **schools** apply censorship that is beyond **national censorship** to **protect the individuals** from material regarded as **unsuitable** by the **organisation**

- **Monitor behaviour**
  - **CCTV** used to **monitor behaviour**
  - Organisations **track** an **individual's work** to see if they are on **target**

- Organisations might **track social media** to ensure **behaviour outside social media** is **acceptable**

- **Analyse personal information**
  - Analysing data about an individual's behavior used to:
    - **Predict market trends**
    - **Identify criminal activity**
    - **Patterns to produce effective treatments for medical conditions**

- **Piracy and offensive communications**
  - Communications Act (CA) 2003
    - This Act makes it illegal to **'steal' Wi-Fi access** or send o**ffensive messages or posts.**
    - Under this Act, in 2012, a young man was jailed for 12 weeks for posting offensive messages and comments about the **April Jones murder** and the **disappearance of Madeleine McCann**

- **Layout, colour paradigms and character sets**
- Equality Act (2010)
    - This Act makes it illegal to **discriminate against individuals** by not providing a **means of access** to a service for a **section of the public**.
    - This means web service providers have to make services more accessible e.g:
      - **Make it screen reader friendly**
      - **Larger fonts/ Screen magnifier option**
      - **Image tagging**
      - **Alternate text for images**
      - **Colour changes to factor colour blind people**
      - **Transcripts of sound tracks/subtitles**

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

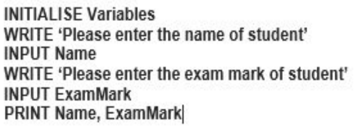
2.2 Problem solving and programming

2.2.1 Programming techniques

(a) Programming constructs: Sequence, iteration, branching

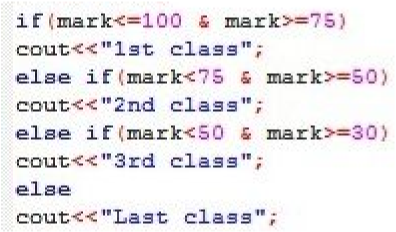
Programming Constructs (Methods of writing code):

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



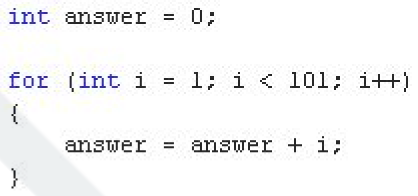
```
INITIALISE Variables
WRITE ‘Please enter the name of student’
INPUT Name
WRITE ‘Please enter the exam mark of student’
INPUT ExamMark
PRINT Name, ExamMark
```

- **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

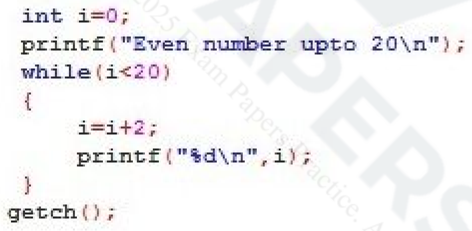


```
if(mark<=100 & mark>=75)
cout<<"1st class";
else if(mark<75 & mark>=50)
cout<<"2nd class";
else if(mark<50 & mark>=30)
cout<<"3rd class";
else
cout<<"Last class";
```

- **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** ↓



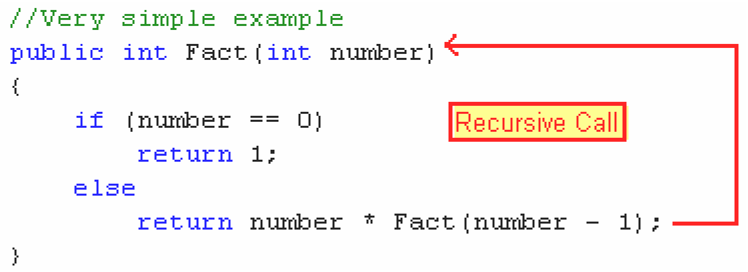
```
int answer = 0;

for (int i = 1; i < 101; i++)
{
    answer = answer + i;
}
```



```
 int i=0;
 printf("Even number upto 20\n");
 while(i<20)
 {
     i=i+2;
     printf("%d\n",i);
 }
getch();
```

(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**



```
//Very simple example
public int Fact(int number)
{
    if (number == 0)
        return 1;
    else
        return number * Fact(number - 1);
}
```

(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

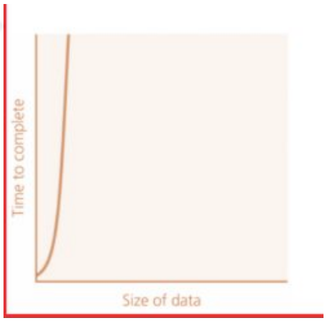
- **O(1) – Constant complexity** e.g. printing first letter of string.
- **O(n) – Linear complexity** e.g. finding largest number in list.
- **O(kn) – Polynomial complexity** e.g. bubble sort.
- **O(k^n) – Exponential complexity** e.g. travelling salesman problem.
- **O(logn) – Logarithmic complexity**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

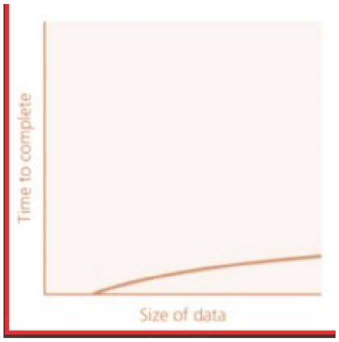
(d) Comparison of the complexity of algorithms

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	

```
PROCEDURE AddToStack (item):
    IF top == max THEN
      stackFull = True
    ELSE
      top = top + 1
      stack[top] = item
    ENDIF
  ENDPROCEDURE
```

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

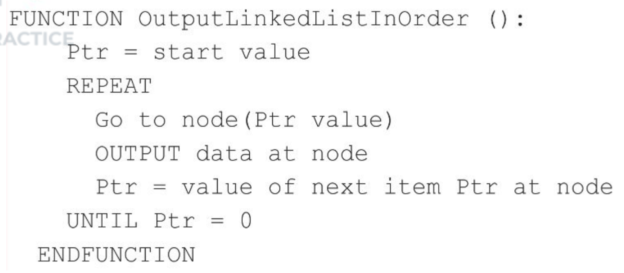
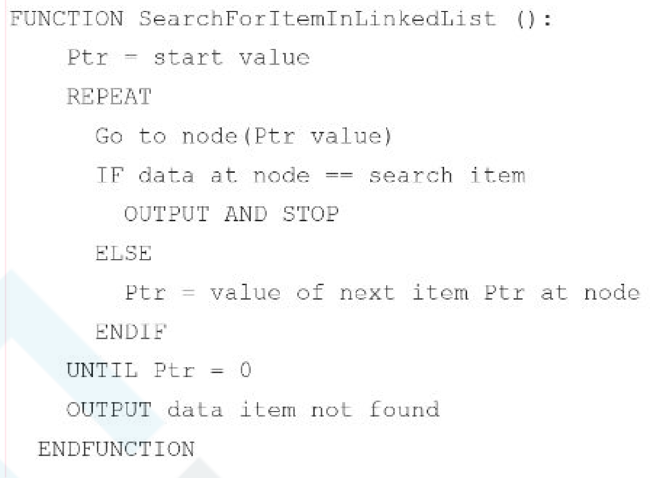
```
PROCEDURE DeleteFromStack (item):
    IF top == min THEN
      stackEmpty = True
    ELSE
      stack[top] = item
      top = top - 1
    ENDIF
  ENDPROCEDURE
```

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

```
PROCEDURE AddToQueue (item):
    IF ((front - rear) + 1) == max THEN
      queueFull = True
    ELSE
      rear = rear - 1
      queue[rear] = item
    ENDIF
  ENDPROCEDURE
```

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

```
PROCEDURE DeleteFromQueue (item):
    IF front == min THEN
      queueEmpty = True
    ELSE
      queue[front] = item
      front = front + 1
    ENDIF
  ENDPROCEDURE
```

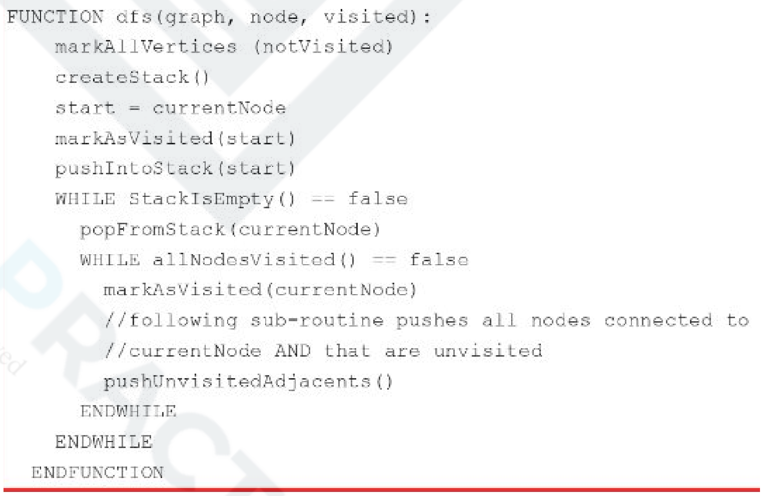
Linked List (Output in Order)	- When the contents of a linked list are **displayed in order**	
Linked List (Add item to list)	- When a data item is added **anywhere** on a **linked list**	

Linked List (Output in Order)

```
FUNCTION OutputLinkedListInOrder ():
    Ptr = start value
    REPEAT
      Go to node(Ptr value)
      OUTPUT data at node
      Ptr = value of next item Ptr at node
    UNTIL Ptr = 0
  ENDFUNCTION
```

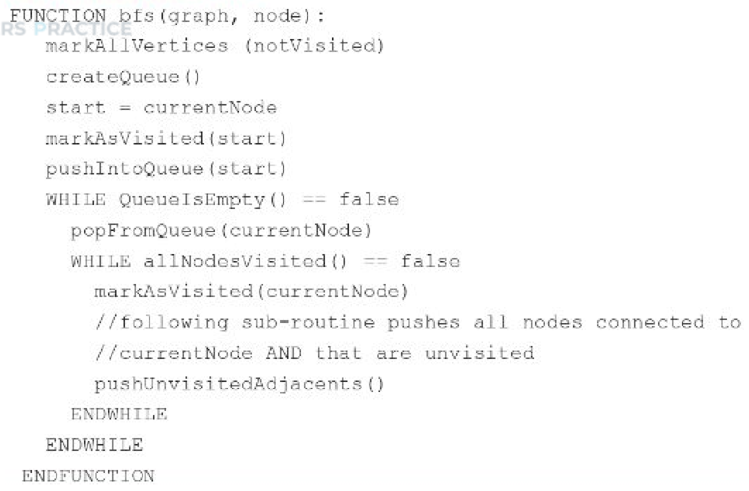
Linked List (Add item to list)

```
FUNCTION SearchForItemInLinkedList ():
    Ptr = start value
    REPEAT
      Go to node(Ptr value)
      IF data at node == search item
        OUTPUT AND STOP
      ELSE
        Ptr = value of next item Ptr at node
      ENDIF
    UNTIL Ptr = 0
    OUTPUT data item not found
  ENDFUNCTION
```

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.**	

Depth first (post-order)

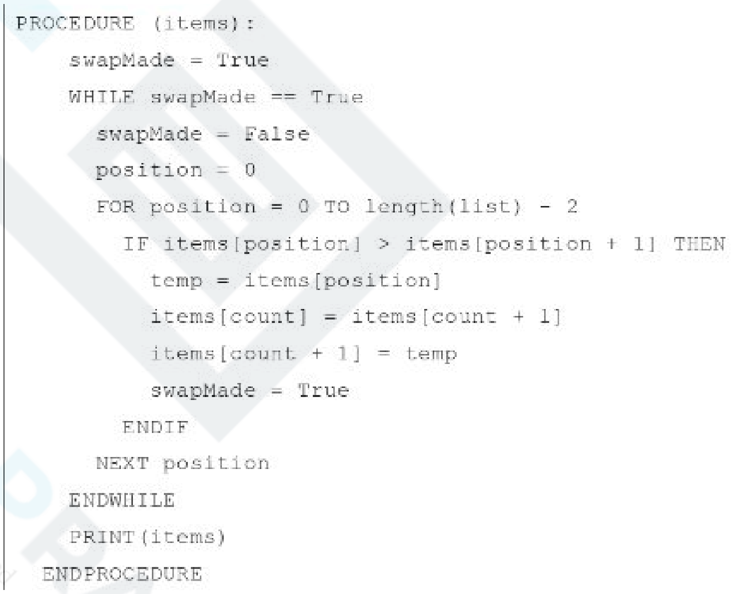
```
FUNCTION dfs(graph, node, visited):
    markAllVertices (notVisited)
    createStack()
    start = currentNode
    markAsVisited(start)
    pushIntoStack(start)
    WHILE StackIsEmpty() == false
      popFromStack(currentNode)
      WHILE allNodesVisited() == false
        markAsVisited(currentNode)
        //following sub-routine pushes all nodes connected to
        //currentNode AND that are unvisited
        pushUnvisitedAdjacents()
      ENDWHILE
    ENDWHILE
  ENDFUNCTION
```

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.	

Breadth first

```
FUNCTION bfs(graph, node):
   markAllVertices (notVisited)
   createQueue()
   start = currentNode
   markAsVisited(start)
   pushIntoQueue(start)
   WHILE QueueIsEmpty() == false
     popFromQueue(currentNode)
     WHILE allNodesVisited() == false
       markAsVisited(currentNode)
       //following sub-routine pushes all nodes connected to
       //currentNode AND that are unvisited
       pushUnvisitedAdjacents()
     ENDWHILE
   ENDWHILE
 ENDFUNCTION
```

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

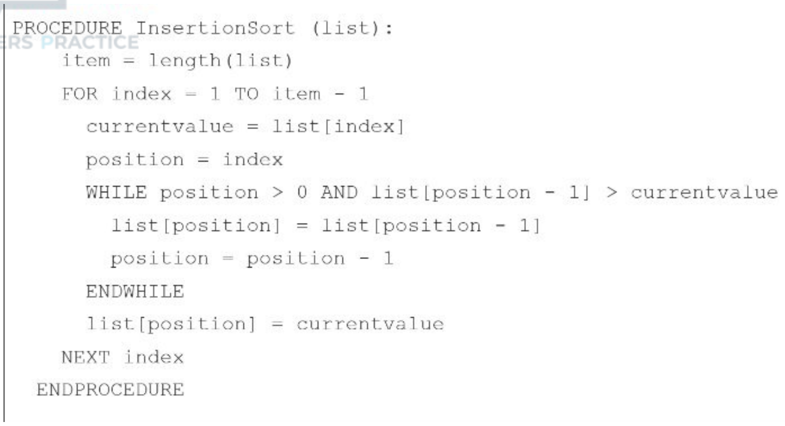
Sort	Description	Algorithm
Bubble Sort	- Is **intuitive** (easy to understand and program) but **inefficient**. - Uses a **temp element**. - Moves through the data in the **list repeatedly** in a **linear way** - Start at the **beginning** and **compare** the **first item** with the **second**. - If 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**.	

Bubble Sort

```
PROCEDURE (items):
    swapMade = True
    WHILE swapMade == True
      swapMade = False
      position = 0
      FOR position = 0 TO length(list) - 2
        IF items[position] > items[position + 1] THEN
          temp = items[position]
          items[count] = items[count + 1]
          items[count + 1] = temp
          swapMade = True
        ENDIF
      NEXT position
    ENDWHILE
    PRINT(items)
  ENDPROCEDURE
```

Insertion Sort

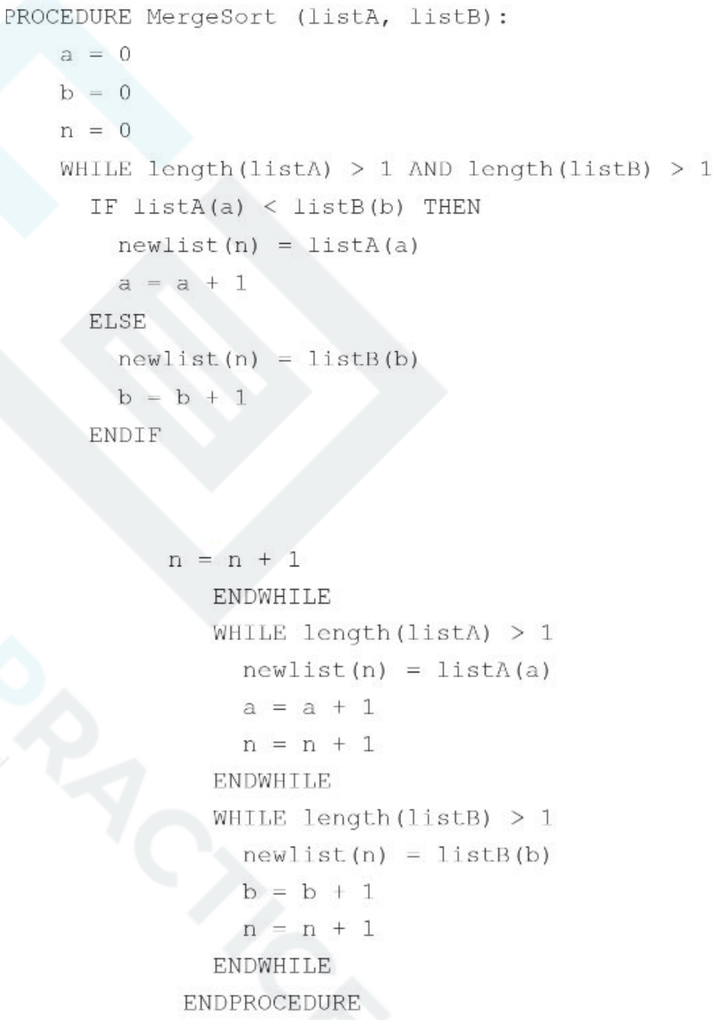
- Works by **dividing** a list into **two parts**: **sorted and unsorted**
- Elements are inserted **one by one** into their **correct position** in the **sorted section** by **shuffling them left** until they are **larger** than the item to the **left** of them until all items in the list are **checked**.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



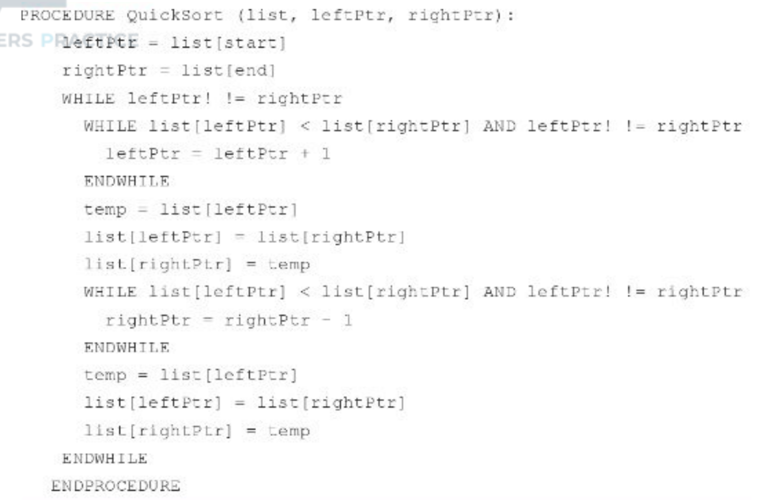
```
PROCEDURE InsertionSort (list):
   item = length(list)
   FOR index = 1 TO item - 1
     currentvalue = list[index]
     position = index
     WHILE position > 0 AND list[position - 1] > currentvalue
       list[position] = list[position - 1]
       position = position - 1
     ENDWHILE
     list[position] = currentvalue
   NEXT index
 ENDPROCEDURE
```

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.

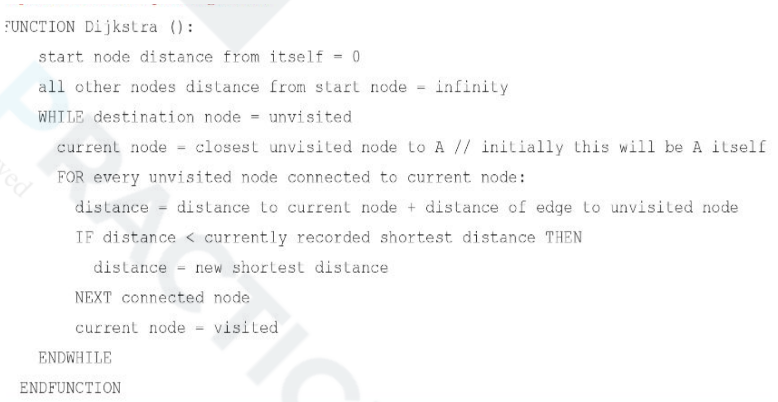


```
PROCEDURE MergeSort (listA, listB):
    a = 0
    b = 0
    n = 0
    WHILE length(listA) > 1 AND length(listB) > 1
      IF listA(a) < listB(b) THEN
        newlist(n) = listA(a)
        a = a + 1
      ELSE
        newlist(n) = listB(b)
        b = b + 1
      ENDIF


           n = n + 1
              ENDWHILE
              WHILE length(listA) > 1
                newlist(n) = listA(a)
                a = a + 1
                n = n + 1
              ENDWHILE
              WHILE length(listB) > 1
                newlist(n) = listB(b)
                b = b + 1
                n = n + 1
              ENDWHILE
            ENDPROCEDURE
```

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**.	

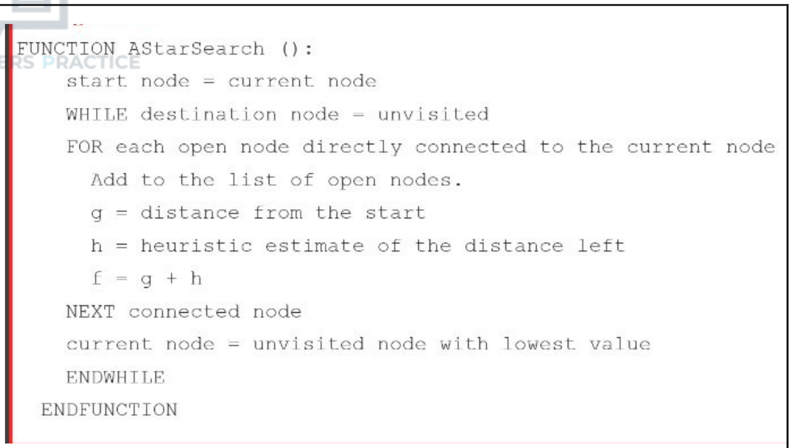
Quick Sort

```
PROCEDURE QuickSort (list, leftPtr, rightPtr):
    leftPtr = list[start]
    rightPtr = list[end]
    WHILE leftPtr! != rightPtr
      WHILE list[leftPtr] < list[rightPtr] AND leftPtr! != rightPtr
        leftPtr = leftPtr + 1
      ENDWHILE
      temp = list[leftPtr]
      list[leftPtr] = list[rightPtr]
      list[rightPtr] = temp
      WHILE list[leftPtr] < list[rightPtr] AND leftPtr! != rightPtr
        rightPtr = rightPtr - 1
      ENDWHILE
      temp = list[leftPtr]
      list[leftPtr] = list[rightPtr]
      list[rightPtr] = temp
    ENDWHILE
  ENDPROCEDURE
```

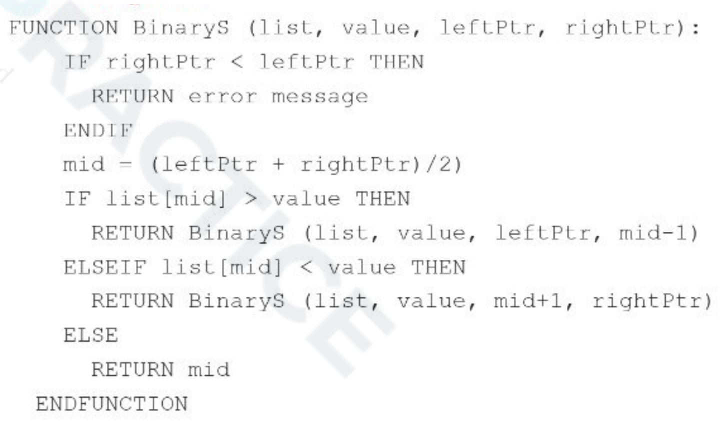
Path Algorithms	Description	Algorithm
Dijkstra's shortest path algorithm	- Finds the **shortest path** between **two nodes** on a graph. - It works by keeping **track of the shortest distance** to each **node** from the **starting node**. - It **continues** this until it has **found the destination node**.	

Dijkstra's shortest path algorithm

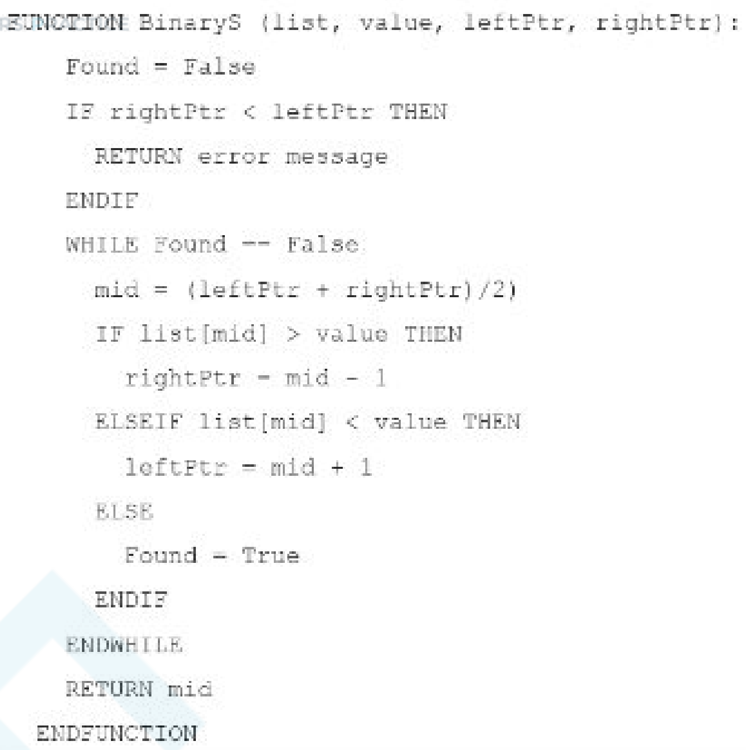
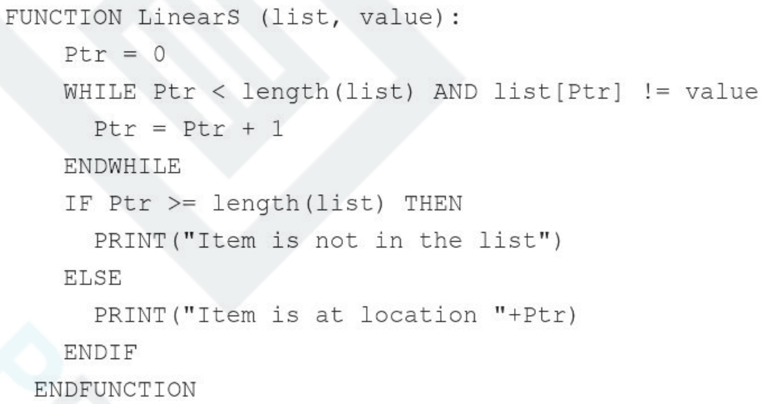
```
FUNCTION Dijkstra ():
    start node distance from itself = 0
    all other nodes distance from start node = infinity
    WHILE destination node = unvisited
      current node = closest unvisited node to A // initially this will be A itself
      FOR every unvisited node connected to current node:
        distance = distance to current node + distance of edge to unvisited node
        IF distance < currently recorded shortest distance THEN
          distance = new shortest distance
        NEXT connected node
        current node = visited
    ENDWHILE
  ENDFUNCTION
```

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**.	

```
FUNCTION AStarSearch ():
   start node = current node
   WHILE destination node = unvisited
   FOR each open node directly connected to the current node
     Add to the list of open nodes.
     g = distance from the start
     h = heuristic estimate of the distance left
     f = g + h
   NEXT connected node
   current node = unvisited node with lowest value
   ENDWHILE
 ENDFUNCTION
```

Search Type	Description	Algorithm
Binary Search Recursive	- 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	

```
FUNCTION BinaryS (list, value, leftPtr, rightPtr):
   IF rightPtr < leftPtr THEN
     RETURN error message
   ENDIF
   mid = (leftPtr + rightPtr)/2)
   IF list[mid] > value THEN
     RETURN BinaryS (list, value, leftPtr, mid-1)
   ELSEIF list[mid] < value THEN
     RETURN BinaryS (list, value, mid+1, rightPtr)
   ELSE
     RETURN mid
 ENDFUNCTION
```

Binary Search Iterative	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 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**.	
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.**	

Binary Search Iterative

```
FUNCTION BinaryS (list, value, leftPtr, rightPtr):
    Found = False
    IF rightPtr < leftPtr THEN
      RETURN error message
    ENDIF
    WHILE Found == False
      mid = (leftPtr + rightPtr)/2)
      IF list[mid] > value THEN
        rightPtr = mid - 1
      ELSEIF list[mid] < value THEN
        leftPtr = mid + 1
      ELSE
        Found = True
      ENDIF
    ENDWHILE
    RETURN mid
  ENDFUNCTION
```

Linear Search

```
FUNCTION LinearS (list, value):
    Ptr = 0
    WHILE Ptr < length(list) AND list[Ptr] != value
      Ptr = Ptr + 1
    ENDWHILE
    IF Ptr >= length(list) THEN
      PRINT("Item is not in the list")
    ELSE
      PRINT("Item is at location "+Ptr)
    ENDIF
  ENDFUNCTION
```

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