KNOWLEDGE ORGANISER GUIDANCE

It is advised that you print the relevant subject knowledge organisers and have them available to you when needed at all times.

An alternative recommendation would be to download the knowledge organisers for your subjects onto your electronic devices so you can access them when needed.

With the knowledge organiser you should make revision cards to help revise and build in time during independent study to test yourself weekly on the content.

While you have independent study, you should use your Knowledge Planner to study the relevant subject's Knowledge Organiser and learn the information provided.

Haggerston School

SIXTH FORM KNOWLEDGE ORGANISER

Computer Science

2023/2024

Aspiration Creativity Character

Computer Science

Paging

A method of manipulating memory which uses pages to stored code in fixed size blocks and allows programs to run despite insufficient memory. Uses virtual memory

Segmentation

A method of manipulating memory which uses segments to store code in different sized, logical sections. Uses virtual memory

Paging		Segmentation	
Advantages	Disadvantages	Advantages	Disadvantages
Allows programs to run despite insufficient memory using virtual memory	When virtual memory is used, if it takes too long for pages to be moved to the disk the computer will slow down (Disk thrashing)	Allows programs to run despite insufficient memory using virtual memory	When virtual memory is used, if it takes too long for segments to be moved to the disk the computer will slow down (Disk thrashing)
Pages are all of the same size		Segments have logical divisions which are more efficient	
Pages fit sections of memory		Segments are different sizes to match the sections of a program	
		Segments include complete sections of programs for easier reference	

Scheduling

A way of managing the amount of time programs have in the CPU

Memory management

A way of ensuring that programs in memory only access their own data or any authorised shared data with other programs

Virtual memory

A method of freeing available memory in the RAM by moving unused program sections to the hard drive. When the section of code is needed again it is returned to the RAM

Device drivers

Software which tell the OS how to communicate with a device, e.g. a printer driver

BIOS (Basic Input Output System)

Checks that the computer is functional and loads the OS's kernel into memory

- The bootstrap is responsible for loading the OS into memory
- Initial start up instructions are stored in ROM
- BIOS settings are stored in non-volatile flash memory

Intermediate Code

Code between source code and machine code which can be read by virtual machines

ISR (Interrupt Service Routine)

Determines what happens when an interrupt is raised

Interrupt

A signal which stops the fetch decode execute cycle from running normally in order to prioritise a different a device

Types of interrupt:

Hardware:
- Power pressed
- Memory parity error

Software:
- Illegal instruction
- Arithmetic overflow
- New log-on request

Input/output:
- Buffer almost empty
- Data transfer completion

Virtual machine

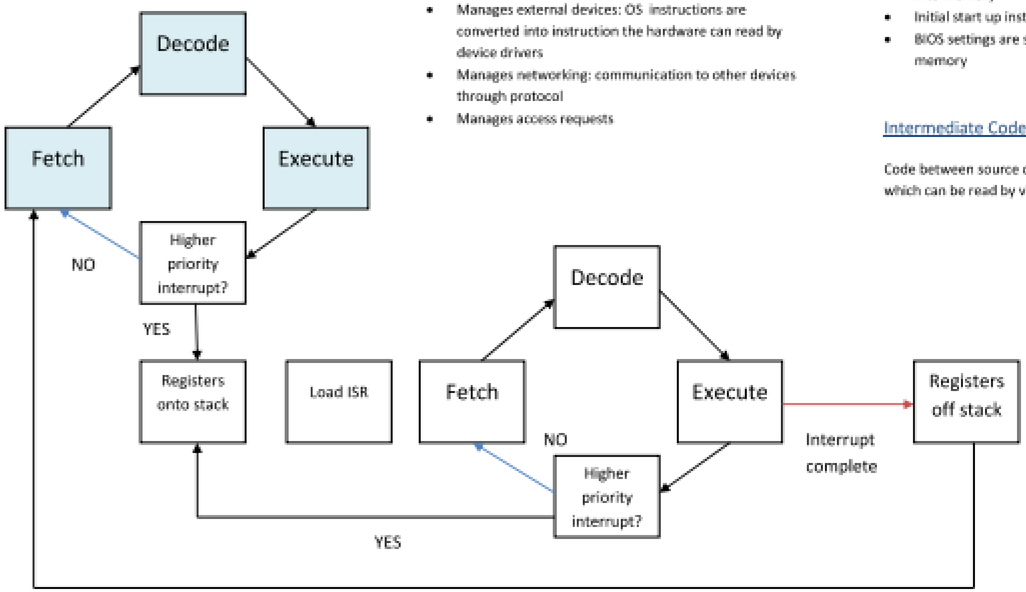
When software is used to take on the function of a physical machine

- Emulators provide the illusion that a program is running on native hardware

Operating System:

Software that manages the hardware, software, security and user interface of a computer

- Manages the interrupts in the processor
- Manages memory including paging, segmentation and virtual memory
- Manages external devices: OS instructions are converted into instruction the hardware can read by device drivers
- Manages networking: communication to other devices through protocol
- Manages access requests



Scheduling algorithms:

FCFS (First come first served):

Tasks are executed to completion and in order regardless of time

SJF (Shortest job first):

The shortest task is executed first to completion. The algorithm needs to know the time each job will take in advance

RR (Round robin):

Each task is given a certain amount of time. If it hasn't finished it rejoins the end of the queue

SRT (Shortest remaining time):

The shortest task is executed to completion or until a task with a shorter remaining time joins the queue

MLFQ (Multi-level feedback queues):

Multiple queues are used with different priorities and jobs are moved between the queues depending on their behaviour

Types of operating system:

Distributed OS:

Controls computers on a network and presents them to the user as one system

Embedded OS:

Specifically designed for a device and runs efficiently with little memory and low power CPUs e.g. in a washing machine

Multi-tasking OS:

Allows multiple application to be open at once by switching between running programs, e.g. Windows

Multi-user OS:

Allows multiple users to access a computer simultaneously with individual preferences, e.g. a supercomputer

Real-time OS:

Processes are always executed in a certain time frame to cater for unusually high demand, e.g. plane autopilot and hospital machine

SIXTH FORM KNOWLEDGE ORGANISER

Haggerston School

Aspiration Creativity Character

Computer Science

Computer Architecture

- **Von Neumann Architecture** has one control unit, ALU, registers and memory unit with a shared memory and data bus used for data and instructions.
- **Harvard Architecture** has separate memories for instructions and data. It is more commonly used in embedded processors
- Von Neumann Architecture is cheaper to develop as the control unit is simpler and allows programs to be optimised in size.
- Harvard Architecture allows data and instructions can be fetched in parallel and both memories can be different sizes.

Contemporary Processing

- Combines Harvard and Von Neumann architecture
- Von Neumann is used when working with data and instructions in main memory
- Harvard is used when working with cache.
- Has a separate instruction and data cache.

RAM and ROM

Random Access Memory (RAM)

- Volatile
- Holds data and programs which are currently in use
- High access speeds
- Very expensive per gigabyte

Read Only Memory (ROM)

- Non-volatile (Cannot be modified)
- Used to store fixed instructions such as the computer start up routine

Fetch Decode Execute Cycle and Registers

- The order operations take place to execute an instruction.
- Fetch Phase:
  - Address copied from the PC to the MAR.
  - Data bus copies the instruction from that location to the MDR
  - At the same time, the contents of the PC increase by 1
  - The value is them copied from the MDR to the CIR
- Decode Phase:
  - The contents of the CIR are split into operand and opcode
- Execute Phase:
  - The opcode is executed on the operand.

Busses and Assembly Language

- Assembly code uses mnemonics to represent instructions.
- Instructions are divided into operand and opcode
- Opcode is the type of instruction and the hardware to execute it.
- Operand is the address where the operation is performed.

Multi-core and Parallel Systems

- Multi-core CPUs have many cores which complete separate fetch-execute cycles independently.
- Parallel systems can carry out multiple instructions simultaneously using a single core using techniques like pipelining.

Unit 1.1 The Characteristics of Contemporary Processors, Input, Output and Storage Devices

CPU Components

- The **ALU** (Arithmetic and Logic Unit) carries out arithmetical and logical operations.
- The **CU** (Control Unit) directs operations inside the processor.
- **Registers** are small, fast memory cells used to temporarily store data.

Program Counter (PC)	Stores the address of the next instruction to be executed.
Accumulator (ACC)	Stores the results of calculations.
Memory Address Register (MAR)	Holds the address in memory that is to be written to or read from.
Memory Data Register (MDR)	Holds data which has been read or needs to be written.
Current Instruction Register (CIR)	Stores the current instruction, split into operand and opcode.

- Buses are parallel wires connecting two or more CPU components together.
- The number of parallel wires determines the bus width.
- The system bus contains the data bus, control bus, and address bus.

Data Bus	A bi-directional bus which transfers data and instructions between components.
Address Bus	Transmits the location in memory where data should be read or written.
Control Bus	A bi-directional bus which transmits control signals.

Flash Storage

- Fast and compact
- Logic gates store an electrical charge
- High represents a binary 1
- Low represents a binary 0
- Information is stored in blocks which are combined to form pages
- More expensive
- Limited lifespan

Solid State Drives

- Light and portable
- No moving parts
- More resistant to damage from movement than hard disk drives
- High data transfer rates
- Smaller capacity than hard disk drives

Magnetic Storage

- Two magnetic states represent binary
  - Polarised sectors represent 1
  - Unpolarised sectors represent 0
- Can be damaged by strong magnets

Hard Disk Drives

- High capacity
- Magnetic platters rotate at high speeds beneath a read/write head
- Multiple platters are stacked to maximise storage capacity
- Moving parts can become damaged

Magnetic Tape

- An older storage medium
- Tape is round onto reels within a cartridge.
- The tape drive spins the reels to move the tape across a reader

Floppy Disks

- A thin magnetic disk in a plastic case.
- Small and portable
- Typical storage capacity of 1MB

Virtual Storage

- A method of storing information remotely.
- Allows multiple computers to access data over a network or The Internet.
- Includes cloud storage and network accessible storage.
- Becoming more popular as network and Internet speeds increase.
- Relies on a network connection for access to data.
- Limited by network speed.

Input, Output and Storage Devices

- Input devices are used to send data to the computer, such as a keyboard, mouse or sensor.
- Output devices allow the computer to send information out, such as a speaker or screen.
- Storage devices allow data to be stored such as a hard drive.
- Some devices can be both an output and input device, such as a touch screen.
- Factors such as speed, accuracy, cost and relevance to the task should be considered when choosing devices.

Reduced Instruction Set Computers (RISC)	Complex Instruction Set Computers (CISC)
• Small instruction set • One instruction is one line of machine code • Used in personal computers	• Large instruction set • Instructions built into hardware • Used in microcontrollers and embedded systems • Compiler has less work to do • Less RAM is needed to store the code

Optical Storage

- Use lasers to read and write to a disk.
- Sectors of the disc are written in a spiral.
  - Pits scatters light representing 0
  - Lands reflects light representing 1
- Small and light so very portable
- Easily scratched
- Not very fast

Compact Disk (CD)

- Commonly used for audio but can store any data type
- Stores relatively little information

Digital Versatile Disc (DVD)

- Higher storage capacity than CDs
- Often used to store videos

Blu-Ray

- More than five times as much storage as a DVDs
- Used to store HD films

Pipelining

- Allows three instructions to be processed through the fetch, decode and execute cycle at the same time.
- Data is stored in a buffer close to the CPU until required.
- Whilst one instruction is being executed, another can be decoded and another fetched.
- Reduces the amount of CPU idle time.

Graphics Processing Unit (GPU)

- Had multiple processors working in parallel.
- Efficiently completes repetitive tasks.
- Used for image processing and machine learning.
- A co-processor (a secondary processor which supports the activities of the primary processor.

Factors Affecting CPU Performance

Clock Speed:

- Determined by the system clock
- All activities begin on a clock pulse
- Each operation starts when the clock changes from 0 to 1
- The clock speed is the number of clock cycles which can be completed in a second.
- Faster clock speed = better performance

Number of Cores:

- Each core is an independent processor which executes its own fetch-execute cycle
- CPUs with several cores can complete more than one fetch-execute cycle at the same time
- Some applications can only use one core.
- More cores = better performance

Amount and type of Cache Memory

- Cache memory is fast memory built into the CPU
- Instructions are held in cache allowing them to be accessed quickly if needed.
- As cache fills up, unused instructions are overwritten.
- More cache = better performance
- Cache can be Level 1, 2, or 3
- Level 1 is the fastest but smallest
- Level 3 is the slowest but largest

Computer Science

1: Systems Architecture

1.The Purpose of the CPU	
The Purpose of the CPU	To manage basic operations of the computer. To be the 'brains' of the computer
The main components of the CPU	Control Unit, Arithmetic Logic Unit, Cache
Von Neumann Architecture	The architecture that allows for the storage of instruction and data in the same location
The FDE Cycle	The cycle the CPU continuously carries out to process instructions
Binary	The number system used to store instructions and data in the computer
The role of a register in the CPU	it is a place to temporarily hold data and instructions as they are being processed by the CPU
The PC	The Programme Counter keeps the address of the next instruction to be processed
The MAR	The Memory Address Register is used to tell the CPU where to locate data in the Main Memory
The MDR	The Memory Data Register is used to store data that is fetched from the Main Memory
The ACC	The Accumulator stores results of logic operations a nd calculations used during processing

2.Common CPU Components and their Function	
The Control Unit has two functions	(1) Sending signals to control the flow of data and instructions, and (2) decoding instruction
Cache memory	A small section of extremely fast memory used to store commonly used instructions and data. Is it useful as the CPU can access the (fast) cache directly. L1 cache is closest to the CPU; L3 is the furthest
The ALU has the following functions	It carries out mathematical operations/logical operations/shifting operations on data; e.g. multiplication, division, logical comparisons
An Address	This is the location in the Main Memory (RAM) that stores data or instructions in the Van Neumann Architecture
Buses	Transfers information between the CPU and the Main Memory (and other places). E.g. the Address bus carries memory addresses between the CPU and RAM

3. The F-D-E (Fetch Decode Execute) Cycle	
The F-D-E Cycle repeatedly cycles	1.Fetch → 2. Decode → 3. Execute → 1.Fetch
The Fetch Stage	The address is generated by the Program Counter (PC) and is carried to the Memory Address Register (MAR) using the Address Bus. The PC then updates and stores the next memory address, ready for the next round of the cycle. The data or instruction that is in that memory location is placed on the data bus and carried to the processor and is stored in the Memory Data Register (MDR)
The Decode Stage	The data or instruction is then the Memory Data Register (MDR), decoded to find out if it is a piece of data or if it an instruction to do something such as ADD, STORE, SWITCH, REPEAT, etc...
The Execute Stage	The CPU performs the actions required by the instruction. If it is an instruction to control input or output devices, the Control Unit will execute the instruction. If it is a calculation then the Arithmetic and Logic Unit (ALU) will execute the instruction. The results of any calculations are recorded in the Accumulator

4. Performance of the CPU	
Cores	CPUs with multiple cores have more power to run multiple programs at the same time
Clock Speed	The clock speed describes how fast the CPU can run. This is measured in megahertz (MHz) or gigahertz (GHz) and shows how many fetch-execute cycles the CPU can deal with in a second
Cache Size	The more data that can be held in the cache, the shorter the trips the electric pulses need to make, so this speeds up the processing time of each of those billions of electrical signals, making the computer noticeably faster overall

5. Embedded Systems			
Definition	A computer system which forms part of an electronic device	Reasons	They are cheaper to make and smaller than a General Purpose Computer
Re-programmable	Not for different purposes but firmware can sometimes be upgraded	Examples	Washing machine. Smart Oven, Car Engine, Pacemaker

Computer Science

2: Primary and Secondary Storage

1.The purpose of RAM and ROM in a Computer System	
The purpose of RAM	RAM is the main memory (also called primary storage) for storing data and programs while they are in use
The purpose of ROM	ROM stores the boost sequence, which is a set of instructions that the computer executes every time it is switched on. ROM is essential since it loads the operating system
We use RAM rather than Secondary Storage	The RAM can be accessed at a much higher speed than the secondary storage. If the CPU was having to communicate directly with secondary storage for the F-D-E cycle, the computer would be incredibly slow
Volatility	ROM is non-volatile (it keeps its contents when the power is turned off). RAM is volatile (it loses its contents when the power is turned off)
Primary Storage Devices	Primary storage devices are internal to the system and are the fastest of the memory/storage device category. Typically, primary storage devices have an instance of all the data and applications currently in use or being processed. The computer fetches and keeps the data and files it in the primary storage device until the process is completed or data is no longer required. RAM, ROM, Graphics Card RAM, cache and registers are common examples of primary storage devices
Increasing RAM	This can speed the computer up since there is less need for virtual memory

2. The Need for Virtual Memory	
Definition of virtual memory	A temporary storage space taken up on a secondary storage device (e.g. hard disk) to allow more space for running programs and data than can fit in primary storage (RAM)
Use of virtual memory	Open applications/data that are not in current use are 'paged' out to the secondary storage. When they are needed, they are 'paged' back into primary memory
Advantage of virtual memory	Having virtual memory available allows a computer to run more programs at the same time, or to run larger programs; or to work with much larger amounts of data than could fit in the primary storage (main memory / RAM)
Disadvantage of virtual memory	It is relatively slow compared with RAM. The need to page data in and out of the secondary storage device slows down the computer. It can also lead to 'disk thrashing'

3. Secondary Storage	
Difference from primary storage	Primary storage (e.g. RAM, cache) is volatile. Secondary storage is non-volatile. It retains its data when the power is switched off
Cache memory	A small section of extremely fast memory used to store commonly used instructions and data. Is it useful as the CPU can access the (fast) cache directly. L1 cache is closest to the CPU; L3 is the furthest
ROM as secondary storage	Not really. ROM is read only. Secondary storage generally needs to be written to as well as read from

4. Common types of storage	
Optical	The surface of a CD is covered in microscopic dots. A laser would skim across the surface reading these. As the laser passes over, the pattern on the surface is picked up. If the laser hits a dot it is reflected differently to if there were no dot present. Eg. CD/CDR/CDRW/DVD/ BluRay
Magnetic	Magnetic hard drives uses silver coloured disks which are covered on both sides with a magnetic film divided into billions of tiny areas. Each one of those areas can be independently magnetised (to store a1) or demagnetised (to store a O). The read.write heads would flicker quickly over the surface as it reads and writes the data. Several platters would be installed in one hard drive to give greater storage capacity. E.g. Hard disk Drive/DAT/Tape Drive/Cassette
Solid State	Solid-state secondary storage does not have any moving parts. Solid state secondary storage stores data using circuit chips. they are sometimes called flash drives. E.g. USB drives/SD Cards/SSD Drives

5. Considerations for the Most suitable Storage Device	
Capacity	How much data needs to be stored?
Speed	How quickly can the data be stored? How quickly does it need to be read?
Portability	Does the device need to be transported? Are weight and size important?
Reliability	Is it mission critical? Will it be used over and over again?
Cost	How expensive is the media per byte of storage?

6. Typical uses	
Optical	Read only distribution on a large scale (CD/DVD). Relatively small capacity
Magnetic	High data capacity. Reasonably fast. Low cost. Cloud storage on server farms
Solid State	Low power. Small. Rugged. Silent. Very fast. Medium data capacity

Unit 1.2 Software and Software Development (Page 1)

Operating Systems (OS)
- Provide an interface between the user and computer
- Features include Memory management, Resource management, File management, Input Output Management, Interrupt management, Utility software, Security, User interface

Algorithms
- A set of instructions used to solve a set problem.
- Inputs must be clearly defined.
- Must always produce a valid output.
- Must be able to handle invalid inputs.
- Must always reach a stopping condition.
- Must be well-documented for reference.
- Must be well-commented.

Memory Management
- Computers often need more memory than is available and so must efficiently manage the available memory and share it between programs.

Paging
- Memory is broken down into equal sized parts called pages.
- Pages are swapped between main and virtual memory.

Segmentation
- Memory is split up into segments.
- Segments can vary in size.
- These segments represent the logical flow and structure of a program.

Virtual Memory
- Part of the hard drive can be used as RAM.
- Access is slower than RAM.
- Paging is used to move sections which are not in active use into virtual memory.

Interrupts
- A signal generated by hardware or software to tell the processor it needs attention.
- Have different priorities.
- Stored with a priority queue in an interrupt register.

Interrupt Service Routine (ISR)
- At the end of the fetch, decode, execute cycle the interrupt register is checked.
- If there is an interrupt with a higher priority than the current task:
  - The contents of the registers are transferred into a stack .
  - The appropriate (ISR) is loaded into RAM.
  - A flag is set, noting that the ISR has begun.
  - The flag is reset when the ISR has finished.
  - This process repeats until no more interrupts exist.

Virtual Machines
- A software implementation of a virtual computer
- Intermediate code is halfway between machine code and object code.
- It is independent of process architecture allowing it to run across different systems.
- It takes longer to execute
- Virtual machines can be used to help protect from malware, test software, or run software with different versions or OS requirements.

Applications Software
- Used by an end user to perform a specific task.
- e.g. word processor or web browser

Systems software
- Manages computer resources to maintain performance
- e.g. operating system or device driver.

Utility Software
- Has a specific function to maintain OS performance
- e.g. backup or compression software

Scheduling
- The operating system schedule processor time between running programs.
- These are known as jobs and held in a queue.
- Pre-emptive scheduling routines actively start and stop jobs
- Non pre-emptive routines start jobs then leave them to complete

Round Robin Routine
- Each job is given a time slice of processor time to run in.
- When a job has used up it's time slice it is returns to the start of the queue and receives another.
- This repeats until the job is complete.

First come first served routine
- Jobs are processed in the order they entered the queue

Multilevel feedback queue routine
- Uses multiple queues, each with a different priority

Shortest job first routine
- The queue is ordered by the amount of processor time needed.
- The shortest jobs are completed first.

Shortest time remaining routine
- The queue is ordered based on the time left to completion.
- Jobs with the least time needed to complete are finished first

	Advantages	Disadvantages
Round Robin	All jobs are eventually attended to.	Longer jobs take much longer. Takes no account of priority.
First Come First Served	Easy to implement.	Takes no account of priority.
Multilevel Feedback	Considers job priority.	Tricky to implement
Shortest Job First	Works well for batch systems	Requires additional processor time to order the queue Takes no account of priority.
Shortest Time remaining	Increased throughput	Requires additional processor time to order the queue Takes no account of priority.

Types of Operating System
Distributed
- Runs across several devices
- Spreads task load across multiple computers

Embedded
- Built to perform a specific small task
- Built for a specific device and hardware
- Limited functionality
- Less resource intensive

Multi Tasking
- Allows multiple tasks to be completed simultaneously
- Uses time slicing to switch between applications

Multi User
- Several users can use a single computer
- A scheduling algorithm allocates processor time between jobs

Real Time
- Performs tasks within a guaranteed time frame
- Used in time critical systems.

BIOS
- Basic Input Output System.
- Runs when a computer first turns on.
- Runs tests then loads the main OS into memory.
- Power On Self Test (POST) makes sure all hardware is connected and functional
- Tests the CPU, Memory and external devices.

	Open Source	Closed Source
	Provided along with the source code. No license required to use.	Needs a license to use. Source code is not available. Protected by Copyright
Advantages	Online, free, community support. Many individuals will work on the code meaning it is of high quality. Free.	The company provides support and documentation. Professionally developed. More secure. Regular updates
Disadvantages	Not always well supported or documented. Variable quality code. Less secure.	Code cannot be customised to meet user needs. License may restrict use. More expensive.

Translators
- Covert source code into object code.

Compiler
- Translates code all in one go.
- Compilation process is longer.
- Produces platform specific code.
- Complied code can be run without a translator.

Interpreter
- Translates and executes code line by line.
- Will error if a line contains an error.
- Slower to run than compiled code.
- Code is platform independent.
- Useful for testing.

Assembler
- Assembly code is platform specific, low level code.
- Translates assembly code to machine code.
- 1 line of assembly code = 1 line of machine code.

Stages of Compilation
Lexical Analysis
- Comments and whitespace removed
- Identifiers and keywords replaced with tokens
- Token info stored in a symbol table

Syntax Analysis
- Tokens checked against language rules
- Flags syntax errors
- Abstract Syntax Tree Produced

Code Abstraction
- Machine code produced using Abstract Syntax Tree

Optimisation
- Reduces execution time
- Most time consuming part
- Removes redundant code.

Device Drivers
- Code which allows the OS to interact with hardware
- Specific to the OS and architecture type

Linkers
- Link external modules and libraries used in the code.
- Static linkers copy the library code directly into the file, increasing its size.
- Dynamic linkers just add the addresses of the module or library.

Loaders
- Provided by the OS to fetch the library or module from the given location in memory

Libraries
- Libraries include pre compiled, error free, code which can be used within other programs
- Common functions can quickly and easily be reused across multiple programs
- Saves the time and effort associated with developing and testing code to perform the same task over and over again.

Ways to Address Memory
- Machine code comprises an operand and opcode.
- Operand is the value relating to the data on which the instruction should be performed.
- Opcode holds the instruction and the addressing mode.
- The addressing mode is how the operand should be interpreted.

Addressing Modes
- Immediate Addressing – The operand is the value itself and the instruction is performed on it.
- Direct Addressing – The operand provides the address of the value the instruction should be performed on.
- Indirect Addressing – The operand holds the address of a register. The register holds the address of the data.
- Indexed Addressing – An index register stores a certain value. The address of the operand is found by adding the index register and the operand.

Computer Science

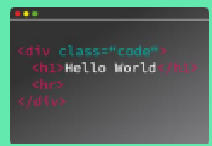
Unit 1.2 Software and Software Development (Page 2)

	Merits	Drawbacks	Uses
Waterfall	• Straightforward to manage • Clearly documented	• Lack of flexibility • No risk analysis • Limited user involvement	Static, low-risk projects with little user input.
Agile	• High quality code • Flexible to changing requirements • Regular user input	• Poor documentation	Small to medium projects with unclear initial requirements.
Extreme Programming	• High quality code • Constant user involvement means high usability	• High cost as two people are needed • Teamwork is essential • User needs to be present	Small to medium projects with unclear initial requirements requiring excellent usability.
Spiral	• Thorough risk-analysis • Caters to changing user needs • Prototypes produced throughout	• Expensive to hire risk assessors • Lack of focus on code efficiency • High costs due to constant prototyping	Large, risk-intensive projects with a high budget.
Rapid Application Development	• Caters to changing requirements • Highly usable finished product • Focus on core features, reducing development time	• Poorer quality documentation • Fast pace and late changes may reduce code quality	Small to medium, low-budget projects with short time-frames.

Assembly Language

- One level up from machine code.
- Low level language.
- Uses abbreviations for machine code called mnemonics.
- Processor specific.
- One line in assembly language equals one line in machine code.

Mnemonic	Instruction	Function
ADD	Add	Add the value at the memory address to the value in the Accumulator
SUB	Subtract	Subtract the value at the memory address from the value in the Accumulator
STA	Store	Store the value in the Accumulator at the memory address
LDA	Load	Load the value at the memory address to the Accumulator
INP	Input	Allows the user to input a value to be held in the Accumulator
OUT	Output	Prints the value in the Accumulator
HLT	Halt	Stops the program at that line
DAT	Data	Creates a flag with a label at which data is stored.
BRZ	Branch if zero	Branches to an address if the value in the Accumulator is zero. A conditional branch.
BRP	Branch if positive	Branches to a given address if the value in the Accumulator is positive. A conditional branch.
BRA	Branch always	Branches to a given address no matter the value in the Accumulator. An unconditional branch.



Development Methodologies

Extreme Programming

- An agile model.
- Development team includes developers and user representatives.
- The system requirements are based on "user stories".
- Produces highly usable software and high quality code.
- Programmers work no longer than 40 hours per week.
- Hard to produce high quality documentation.

Rapid Application Development

- An iterative methodology.
- Uses partially functioning prototypes.
- Users trial a prototype.
- Focus groups gather user requirements.
- This informs the next prototype.
- This cycle repeats.
- Used where user requirements are unclear.
- Code may be inefficient.

Agile Methodologies

- A collection of mythologies.
- Aimed to improve flexibility.
- Adapt quickly to changing user requirements.
- Sections of the program are developed in parallel.
- Different stages of development can be carried out simultaneously.
- A prototype is provided early and improved in an iterative manner.
- Low focus on documentation.
- High focus on user satisfaction.

Spiral Programming

- Used for high risk projects.
- Has four stages:
- Analyse requirements.
- Locate and mitigate risks.
- Develop, test and implement.
- Evaluate to inform the next iteration.
- The project may be terminated if it is deemed too risky.
- Specialist risk assessors are needed.

Waterfall

- The stages are completed in order.
- The clear structure makes this model easy to follow.
- Changes mean that all stages must be revisited.
- User involvement is low.

Procedural Programming

- Simple to implement.
- Applicable to many problems.
- Is not suited to every problem.
- Uses traditional data types and structures.

Structured Programming

- A subsection of procedural programming
- The flow is given four structures: sequence, selection, iteration and recursion.

Object Orientated Programming

Advantages

- Reusable
- Code is more reliable
- Code is easy to maintain and update
- Classes can be reused, saving time and effort

Disadvantages

- Requires an alternative style of thinking
- Not suited to every problem
- Not best suited for small problems

Attributes, Methods, Classes and Objects.

- **Class** – a template for an object. Defines the behaviour and state of the object.
- **State** – defined by attributes giving the object's properties.
- **Behaviour** – defined by the methods. Describes the action an object can perform.
- **Instantiation** – using a class to create an object.
- **Object** - an instance of a class. Classes can create multiple objects.
- **Setter** – a method which sets the value of an attribute.
- **Getter** – a method which retrieves the value of an attribute.
- **Constructor method** – Allows a new object to be created from a class. Every class must have one.
- **Inheritance** - process where a subclass will inherit all methods and attributes of a superclass.
- **Polymorphism** – allows objects to behave differently depending on their class.
- **Overloading** – avoiding a method by passing different parameters to a method.
- **Overriding** – redefining a method to allow it to produce a different output or function differently.

Programming Paradgims

Imperative

Code specifies clearly the actions to be performed

Procedural

Widely used

Can be applied to a wide range of problems

Written as a set of instructions which are carried out one by one

Easy to write

Object Orientated

Useful for problems which can be broken down into distinct components

Based around objects which have attributes and methods

Focus on reusability, maintainability and ease of updating

Declarative

States the desired result with the language deciding how to accieve it. Users are unaware how the result is obtained

Functional

Based around functions

Function calls can be combined

Linked to mathematics

Logic

Rules and facts of the problem are defined

Queries are used to find answers to these problems

1.2 Software and software development

Hierarchy of software

Application Software

General purpose Software – Software that is designed to be widely used in many ways for both business and personal use (eg applications such as word processing, presentation software, spreadsheet, and web browser).

Specialist Software – Software that is developed for a specific use or for a specific business, scientific, or educational area. For instance, air traffic control systems and stock control systems would fall under this category.

Bespoke Software – The is tailor made software that is developed for a specific organisation or client. Bespoke software is expensive but meets the specific needs of an organisation.

System software

System software is concerned with the running of the computer. Its purpose is the control the computer hardware and manage the application software.

Program translators allow programs to be translated into machine code so that code can be run on a computer. Translators include interpreter, compiler and assembler.
Libraries are collections of prewritten code that can be used in software projects. Thee libraries significantly speed up the development process. Libraries can be reused across multiple applications.
Utility programs are applications that help with the running of the machine.

Common utility programs include:
Auto backup and restore: Incremental backup is useful because only files that have changed or been added since the last full backup needed to be backed up.
Anti-virus: Scans the computer to identify malicious code
Firewall Scans input and output packets and prevents malicious packets accessing the computer.
Disk defragmentation: Organises files on a disk to be located contiguously. Often after defragmentation performance is improved because a file can be accessed from one location on a disk. Files can become fragmented when the original file increases in size and no longer fits into a contiguous location and has to be split over multiple locations.

A computer system has both hardware and software.

Hardware is the physical components that make up a device or computer system. These include both the internal components (eg motherboard, CPU, RAM) and also peripheral and networking devices such as printers and routers.

Software is the computer code, programs and algorithms that give instructions to the hardware to make it perform the desired task. Without the software the hardware will not get any instructions and it will not do anything.

The role of the Operating System

- The most important piece of system software is the operating system.
- The operating system is system software with the role of managing the hardware and software resources.
- The OS handles management of the processor, memory, input/output devices, applications and security.
- The OS hides the complexity of the hardware from the user and provides a user interface.

Application management – Application software does not need to concern itself with interaction and complexities of managing the hardware because this is dealt with by the operating system. Application software needs to run on top of operating system which takes care of interaction with the hardware resources.

Processor resources – Allows multiple applications to be run simultaneously by manages the processing time between applications and cores and switching processing between applications very quickly. Multiple applications will access the processor resources via a schedule that alternates processing between applications. High priority applications will have more CPU time, but it means that lower priority applications will take longer to run.

Memory management – The OS distributes memory resources between programs and manages transfer of data and instruction code in and out of memory. Ensures that each application does not use excessive memory.

Input / Output devices – The OS controls interaction with input (eg keyboard) outputs (eg. Monitor) and storage (eg hard disk) using hardware drivers. Allows users to save files to the hard disk for instance.

Computer Science

Normalisation

- The process of designing a relational database.
- Aims to produce the best and most effective design.

Normalisation Considerations

- Remove redundant or duplicated components.
- Ensure data in linked tables is consistent.
- Allow complex queries to be carried out.
- Ensure records can be added or removed without problems.

First Normal Form

- Attributes may contain a single value only.

Second Normal Form

- In First Normal Form.
- Partial dependencies are not allowed.

Third Normal Form.

- In Second Normal Form.
- Non key dependencies are not allowed.

Run Length Encoding

- A lossless compression method.
- Repeated values are replaced with a single instance of the value and the number of times the value occurs.
- It relies on all consecutive pieces of data being the same.
- It offers poor reduction in file size if there is little repetition.

Indexing

- Stores the position of each record when records are ordered by a certain attribute.
- The primary key is automatically indexed.
- Allows data to be found and accessed quickly

Capturing Data

- There are many ways to capture the data needed for a database.
- The most appropriate way will depend on the type and quantity of data needed and available resources.
- Data may be manually entered by a human or scanned in using optical character recognition, sensors or barcodes.

Selecting, Managing and Exchanging Data

- Data may be selected based around set criteria
- Only data matching the criteria is input to the data
- SQL can be used to sort, structure and filter the data
- Data may need to be transferred between systems or organisations
- This is know as data exchange
- This can be accomplished using EDI (Electronic Data Exchange)

Entity Relationship Modelling

- One to One – Each entity can be associated with one other entity only.
- One to Many – A single table many entities in another table.
- Many to Many – Many entities in one table are linked to many in another table.

Unit 1.3 Exchanging Data Page 1

Referential Integrity

- Ensures consistency.
- Ensures that information is not removed if it is needed elsewhere in the database.

Transaction Processing

- A single operation executed on data.
- Must be processed in line with ACID

ACID

- Atomicity, Consistency, Isolation, Durability.
- Atomicity - the whole transaction must be processed.
- Consistency - transactions must maintain the referential integrity rules between linked tables.
- Isolation - executing transactions at the same time must produce the same result as if they were executed one after the other.
- Durability - when a transaction has been executed it will not be undone.

Record Locking

- Prevents records being accessed by more than one transaction at the same time.
- Prevents inconsistencies and data loss.
- Can result in deadlock

Redundancy

- Multiple copies of the data are kept in different physical locations.
- If data in one copy is lost or damaged it can be retrieved from another copy.

SQL Commands

- SELECT - returns fields from a table.
- FROM - specifies the table or tables.
- WHERE - specifies the search criteria.
- LIKE – used to specify wildcard criteria in conjunction with the % character.
- AND, OR – match more than one criteria.
- JOIN - allows rows from multiple tables to be returned and defines how the tables are linked
- INSERT INTO - inserts a new record in an existing table.
- DELETE - delete a record from a table.
- DROP – delete an entire table.

SQL Examples

```
SELECT CustomerName,Address FROM Customers
WHERE CustomerName LIKE '%Smith%'

SELECT CustomerName,Address FROM Customers
WHERE CustomerName LIKE '%Smith%' AND
CustomerAddress LIKE '%Road%'

SELECT Orders.OrderID,
Customers.CustomerName, Orders.OrderDate
FROM Orders
JOIN Customers ON
Orders.CustomerID=Customers.CustomerID

INSERT INTO Customers (CustomerName,
ContactName, Address, City, PostalCode,
Country)
VALUES ('Cardinal', 'Tom B. Erichsen',
'Skagen 21', 'Stavanger', '4006', 'Norway')

DELETE FROM Customers WHERE
CustomerName='Alfreds Futterkiste'

DROP TABLE Shippers
```

Databases

- An entity is item about which information is stored such as books, or customers.
- Attributes are the categories in which data is collected such as height or name.

Flat File Database

- Consists of a single file.
- Usually based around a single entity.
- Only one table.

Relational Database

- Uses many tables to store data about different entities.
- These tables are linked together.

Primary Key

- A unique identifier, different for each object in the database.
- Usually and ID number or other unique ID.

Foreign Key

- Used to link two tables together.
- The primary key from a different table.

Secondary Key

- Used to enable searching or sorting.
- Usually a common field like name.

Hashing

- Turns an input into a value of a fixed size.
- The input is known as a key.
- The output is known as a hash.
- The hash cannot be turned into the key.
- A hash table stores keys and their matching values.
- They can be used to lookup data in an array.
- They are used in situations where lots of data needs to be looked up in a constant time.
- Algorithms which perform this task are called hash functions.
- The output of a hash function should be smaller than the input.
- If two inputs produce the same hash this is known as a hash collision.
- Using a second hash function and storing items together with the hash helps to overcome collisions.
- Good hash functions are quick to run and have a low rate of collision.

Encryption

- Used to keep data secure.
- Used when transmitting or storing data in ways where others may have access to it.
- Scrambles the data to prevent it being easily read.
- Encryption keys are used to encrypt and decrypt data.

Symmetric Encryption

- The same private key is used by the sender and receiver.
- The same key is used to encrypt and decrypt data.
- A key exchange process is used to share the key.
- Data can be read should the key be intercepted.

Asymmetric Encryption

- Uses two different keys.
- The public key is used to decrypt data and can be shared anywhere.
- The private key is used to encrypt data and must be kept securely.
- The two keys are known as a key pair and are related to each other.
- Encrypting a message using the public key verifies that it was sent and encrypted by the owner of the key.

Dictionary Encoding

- A lossy compression method.
- Commonly used data is replaced with an index.
- The compressed data is stored with a dictionary.
- The dictionary can restore original data.
- The dictionary links the commonly used data to the index.

Search Engines

- Search a database of web addresses to find resources based on criteria set by the user.
- Rely on an index of pages through which they search.
- Web Crawlers build the index by traversing The Internet exploring all links on the page.
- Crawlers collect keywords, phrases and metadata from pages.

Unit 1.3 Exchanging Data Page 2

Network Topologies

Bus Network

All devices are connected to a single cable (called the bus)

A terminator is at each end of the cable.

Advantages:
- Easy to install extra devices.
- Cheap to install as it doesn't require much cable.

Disadvantages
- If the cable fails or is damaged the whole network will fail.
- Performance becomes slower ad additional devices are connected due to data collisions.
- Each device receives all data, a security risk

Star Network

All nodes are connected to one or more central switches.

Often used with wireless networks.

Advantages:
- Every device has its own connection so failure of one node will not affect others.
- New devices can be added by simply connecting them to the switch.
- Usually have higher performance as a message is passed only to its intended recipient.

Disadvantages:
- If the switch fails it takes out the whole network.
- Requires a lot of cable so can be expensive.

Mesh Networks

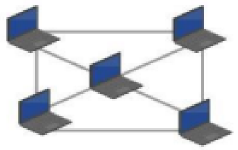
No central connection point, with each device connecting directly to others.

Full mesh networks have every device connected to every other device.

Partial mesh networks have each device connected to several others but not necessarily every other device.

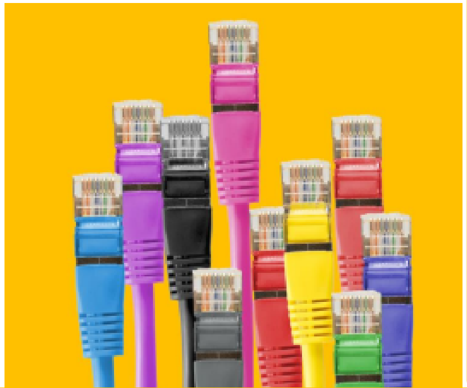
Advantages
- Messages can be received more quickly.
- Messages have many possible routes they can take.
- Multiple connections mean that no device should be isolated
- Each device can talk to more than one node at the same time.
- Devices can be added without interruption.

Disadvantages
- Can be impractical and expensive to setup.
- Require a lot of maintenance



Computer Networks

- A network is two or more computers connected together for the purposes of transmitting data.
- The physical topology defines the physical layout of the network
- The logical topology defines the way data flows through the network
- A protocol is a set of rules for communication between devices.
- They allow devices from different vendors to communicate
- A LAN (local area network) covers a small physical area.
- A WAN (wide area network) covers a large physical area.



Internet Protocols

TCP/IP Stack
- Transfer Control Protocol / Internet Protocol.
- A group (stack) of protocols which work together.
- Controls the flow of data packets through the network.

DNS
- Domain Name System
- Allows websites and other network devices to be identified by a human readable name.
- DNS Server converts the name to an IP Address.
- A hierarchy.
- Each domain name is separated by a dot.
- The names to the right are highest in the hierarchy.

Application Layer
- Top of the stack.
- Specifies the required protocol needed by the application the user is using.

Transport Layer
- Uses TCP to establish a connection through the network between the source and recipient devices.
- Splits data into packets labelled with a packet number.
- Requests retransmission of any packets lost during transit.

Network Layer
- Adds a source and destination IP Address to packets.
- Routers use this address to forward packets through the network to their destination.

Link Layer
- The physical connection between devices.
- Uses a MAC Address to communicate.

LANs and WANs
- LAN – Local Area Network – covers a small area.
- WAN – Wide Area Network – covers a large area.

Client Server Network

- Clients connect to a central server.
- The server is a powerful computer central to the network.
- It holds all the data.
- More secure setup.
- Clients do not need to be backed up.
- Data and resources can be shared easily.
- Expensive to setup.
- More secure.

Peer to Peer Network

- Computers are connected directly to each other.
- Computers share data with one another.
- Quick, cheap and easy to setup.
- Less secure.
- Easier to maintain.

Proxy Server

- Sits between a user and the resource they are accessing.
- Protects users' privacy.
- Caches frequently accessed websites to increase performance.
- Reduces web traffic.
- Uses rules to block access to sensitive information.

PageRank Algorithm

- Ranks each web page
- Higher ranked pages appear first when results are shown.
- Rank based on the number of incoming links on the page and the rank of these pages.
- This is stored in a directed graph.
- The sites are nodes and the links between the pages are the arcs.

PageRank(x) = (1-d) + d[(PageRank(T1) ÷ Count(T1)) + … + (PageRank(Tn) ÷ Count(Tn)]

Compression

- Reduces the space needed to store or transmit a file.
- Important when sharing files over a network or The Internet and when dealing with limited storage space.
- Increased the number of files which can be sent or received.
- Lossy compression removes some information whilst compressing the file. Original cannot be retrieved.
- Lossless compression reduces the size of the file without losing any information. Original can be retrieved.

Network Hardware

NIC
- Network Interface Card
- May be wired or wireless.
- Allows a device to connect to a network.
- Has a unique MAC (Media Access Control) address assigned to it.

Switches
- Controls the flow of data through the network.
- Used in star topologies.

Wireless Access Points (WAPs)
- Allows devices to connect wirelessly to a network.
- Used in mesh networks.
- Often used with a router to allow devices Internet access.

Routers
- Used to connect two or more networks together.
- Often used between a home/office network and an ISP to allow Internet access.

Gateway
- Used to connect networks using different protocols.
- Translates protocols to allow devices to communicate.
- Changes the packet headers.

Packet and Circuit Switching

- Packet Switching
- Data is split into packets.
- Packets are sent across the network.
- Packets may take different routes through the network.
- Circuit Switching
- A direct link is created between devices.
- The link is maintained for the entire conversation.
- Both devices must transfer data at the same rate.

Firewalls

- Prevent unauthorised access to the network.
- Has two NICs.
- Data enters one NIC and is compared to a set of rules.
- Traffic which matches the rules is passed out the other NIC.

Server Side Processing

- Client sends all data to the server for processing. Examples include SQL and PHP.
- It requires no plugins on the client.
- Servers can usually perform large or complex calculations more quickly.
- It is not browser dependent.
- It is more secure

Client Side Processing

- Client processes the data locally.
- Examples include JavaScript.
- Web pages can immediately respond to actions.
- Code executes more quickly.
- It gives more control over the behaviour and look of websites

SIXTH FORM KNOWLEDGE ORGANISER

Haggerston School

Aspiration Creativity Character

Computer Science

Unit 1.4 Data Types, Data Structures and Algorithms

Bitwise Manipulation

Shifts

- Shifts on binary numbers are called logical shifts.
- May be a logical shift left or logical shift right.
- Move all the bits of the number a specific number of places left or right.
- Involves adding a number of zeros at the beginning or end.
- This gives a multiplication for left shifts and division for right shifts by two to the power of the number of places shifted.
- Moving one place will double or halve the number.

Masks

- Combines binary numbers with a logic gate such as AND or XOR.
- May multiply or otherwise change the involved numbers.

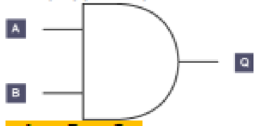
Karnaugh Maps

- Used to simplify Boolean expressions
- Can be used for truth tables with between two and four variables
- Values in columns and rows must be written using grey code
- Columns and rows only differ by one bit

1) Write the truth table as a Karnaugh Map
2) Highlight all the 1s
3) Only groups of 1s with edged equal to a power of 2 may be highlighted
4) Remove variables which change within the highlighting
5) Keep variables which do not change

Boolean Operators

AND - two conditions must be met for the statement to be true
Written as AND or .

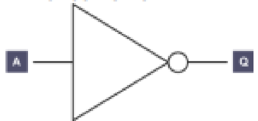


A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

OR - at least one condition must be met for the statement to be true
Written as OR or +

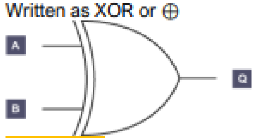
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

NOT – inverts the result, e.g. NOT(A AND B) will only be false when both A and B are true
Written as NOT or ¯



A	Q
1	0
0	1

XOR – Also know as Exclusive OR. Works the same as an OR gate, but will output 1 only if one or the other and not both inputs are 1.
Written as XOR or ⊕



A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

Data Structures

Records

- A row in a file or table
- Widely used in databases
- Made up of fields

Lists

- A number of items
- Items can occur more than once
- Data can be of more than one data type

Tuples

- An ordered set of values
- Cannot be changed once initialised
- Initialised with regular rather than square brackets

Arrays

- An ordered set of elements, each of the same type.
- A 1D array is like a list.
- A 2D array is like a table.
- A 3D array is like a multi page spreadsheet.
- 2D arrays are searched first by the rows and then the columns.

Linked Lists

- Dynamic data structure.
- Stores an ordered list.
- Contents need not be in contiguous data locations.
- Items are called nodes.
- Each node contains a data field and a link or pointer field.
- The data field contains the data itself.
- The pointer field contains the address of the next item.

Graphs

- Notes connected by edges or arcs.
- Directed graphs allow edges to be traversed in one direction only.
- Undirected graphs allow edges to be traversed in both directions.
- Weighted graphs attach a cost to each arc.
- Implemented using an adjacency list or adjacency matrix.
- Adjacency matrix - easy to add nodes and to work with.
- Adjacency list - space efficient.

Trees

- Connected graphs with root and child nodes.
- A note is an item in the tree.
- An edge connects two nodes together.
- A roof is a node with no incoming nodes.
- A child is a node with incoming edges.
- A parent is a node with outgoing edges.
- A subtree is a section of a tree consisting of a parent node with child nodes.
- A leaf is a node with no child nodes.
- A binary tree is a tree where each node has two or fewer children.
- Binary trees store information in a way which is easy to search.
- They often store each node with a left and right pointer.

Positive Integers in Binary

- Each binary digit is called a bit
- Eight bits form a byte
- Four bits (half a byte) is called a nybble
- The most significant bit is furthest left
- The least significant bit is furthest right

Data Types

Integer

- A whole number
- May be positive, negative or 0
- Cannot have a fraction or decimal point
- Often used for counting objects
- e.g. 5, -1, 0, 10

Real

- Positive or negative number
- May have a decimal point
- Often used for measurements
- e.g. 5, -10, 100.556, 15.2

Character

- A single symbol
- May be a letter, number or character
- Uppercase and lowercase letters are different characters
- e.g. A, a, 5, M, ^, @

String

- A collection of characters
- Can store one or many strings
- Often used to contain text
- Leading 0s are not trimmed so useful for storing phone numbers

Boolean

- True or False only

Binary Subtraction

- Use Two's Complement.
- Use the same rules as adding a negative number.
- Use binary addition with a negative two's complement number.

Trace Tables

- A method of recording the values used within an algorithm at each stage of processing to help in troubleshooting
- Tests algorithms for logic errors which occur when the algorithm is executed.
- Simulates the steps of algorithm.
- Each stage is executed individually allowing inputs, outputs, variables, and processes to be checked for the correct value at each stage.
- A great way to spot errors

```
X = 3
Y = 1
while X > 0
    Y = Y + 1
    X = X - 1
print(Y)
```

Stage	X	Y	Output
1	3	1	
2		2	
3	2		
4		3	
5	1		
6		4	
7	0		
8			4

Normalisation

- Maximises the precision in any number of bits.
- Adjust the mantissa so that it begins with 01 for positive numbers and 10 for negative numbers.

Binary Addition

- 0 + 0 = 1
- 0 + 1 = 1
- 1 + 1 = 10
- 1 + 1 + 1 = 11

Combining and Manipulating Boolean Operations

- Boolean operators can be combined to form Boolean equations
- This follows the same way as combining standard maths operators
- The equation can be represented by a truth table
- Sometimes a long expression can share a truth table with a shorter expression
- It is better to use the shorter version.

Computer Science

1.4 Data types, data structures and algorithms

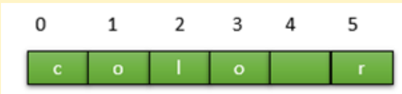
Abstract Data Types: Abstract data types allow us deal with the operations and behaviours of a data type and not to be concerned with their operation which is abstracted away.

Data Structures

Static data structure

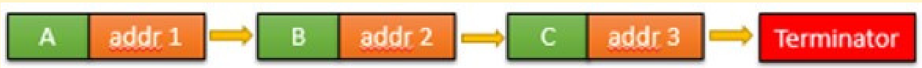
This is a fixed block of memory that is reserved at the start of the program. This is a contiguous space on disk. The next memory location is the next address and its position can be implied, so there is no need to explicitly point to it.

Suppose we want to remove the 'u'. This is not easy for static memory location because we must move all the succeeding elements up one place.

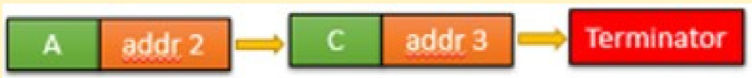


Dynamic Data Structure

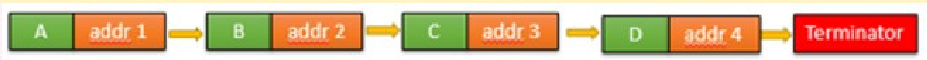
Dynamic memory allocation is where memory is allocated and deallocated during the running of the program. The memory is allocated on the heap. The heap allows random allocation and access of memory. dynamic memory allocation uses linked lists where each element points to the address of the succeeding element.



To remove an element just requires pointing to a different address



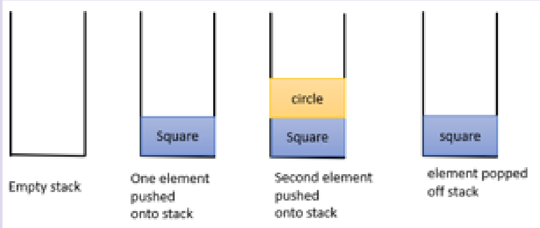
Conversely to add an element just requires pointing to that address



	Advantages	Disadvantages
Static data structures	Memory locations are fixed and can be accessed easily and quickly and are in a contiguous position in memory	Memory is allocated even when not is not being used
Dynamic data structures	More flexible and more efficient than static data structures because we only use memory that is needed. Uses linked lists and makes it much easier to remove and add element.	Data structure may be fragmented so can be slow to access.

Stacks

Stacks are a ***last in first out*** file system just like a stack of plates. That is the last item added to the stack isd the first to be retrieved.



Stack operations:

push: add element to the stack

pop: remove element from the stack

peek/top: view the top element on a stack without removing

isEmpty: test to see if stack is empty

isFull: test to see if stack is full

Uses of stacks:

- Can reverse a sequence of numbers by popping a value from o ne stack and pushing to another
- Used in Reverse Polish Notation
- Stack frames used in subroutine calls

Queues

A queue is a ***first in first out*** data structure. Typically queues are used in buffering where a sequence of instructions are sent to a printer for instance, and the printer prints of the document in order in which the instructions arrived. Lists can be used to represent queues.

Queue operations:

Add: add element tot he end of a queue

remove: remove element from front of queue

isEmpty: test to see if queue is empty

isFull: test to see if queue is full

Computer Science

1.4 Data types, data structures and algorithms

Linear Queue

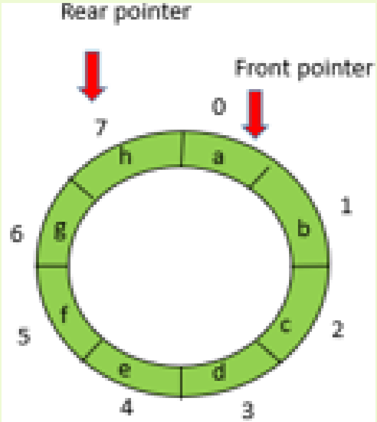
As an item is removed from the queue all the other items move up one space. For a long queue this can take a lot of processing.

Linear queue using pointers

As an item is removed from the queue the pointer representing the start of the queue also moves up. We need to know the length of the queue and how many elements have been removed. The problem with this method is that we end up with a lot of empty cells in memory that are now unused at the front of the list.

Circular queue:

In a linear array when items are removed removed from the memory location those memory locations are allocated but are no longer used. Circular queues get around this problem by 'recycling' these memory locations at the back of the queue.



Priority queue:

Each element is assigned a priority. Highest priority items are removed first. If elements have the same priority then the item nearest the front of the queue is removed first. So in this case 0 would be removed.

Alternatively, the queue could store items in priority order and the item removed from the front of the queue as with a linear queue.

Graphs

A graph is a way of representing the relation between data. A graph is made up of vertices/nodes that are connected by edges or arcs. This could represent a rail or road network

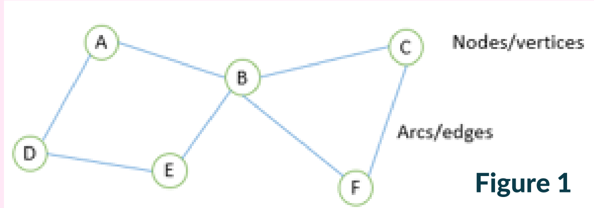
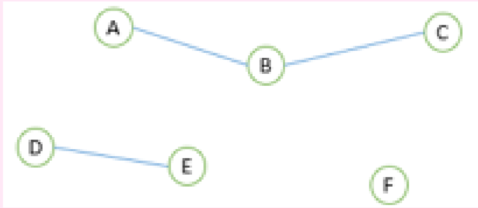


Figure 1



Graphs do not need to be connected. this is a valid graph.

Weighted graph

Weighted graphs add a value to an arc. This might represent the distance between places or the time taken between train stations.

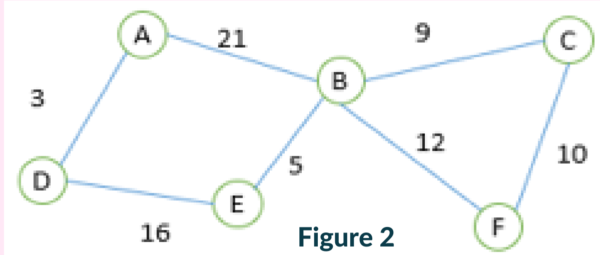


Figure 2

Adjacency Matrix with no weighting

- Graphs can be represented as adjacency matrices
- Graphs with no weights are given a value of 1 for connected nodes

	A	B	C	D	E	F
A	-	1	-	1	-	-
B	1	-	1	-	1	1
C	-	1	-	-	-	1
D	1	-	-	-	1	-
E	-	1	-	1	-	-
F	-	1	1	-	-	-

Adjacency Matrix with weighting

	A	B	C	D	E	F
A	-	21	-	3	-	-
B	21	-	9	-	5	12
C	-	9	-	-	-	10
D	3	-	-	-	16	-
E	-	5	-	16	-	-
F	-	12	10	-	-	-

Adjacency List with no weighting

A	[D, B]
B	[A, E, C,F]
C	[B, F]
D	[A, E]
E	[D, B]
F	[B, C]

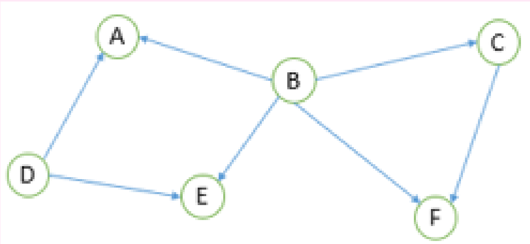
Graphs can also be represented as adjacency lists.
Adjacency list for figure 1

1.4 Data types, data structures and algorithms

Adjacency List with weighting

Graphs can also be represented as adjacency lists. Adjacency list for figure 2

A	{D:3, B:21}
B	{A:21, E:5, C:9, F:12}
C	{B:9, F:10}
D	{A:3, E:16}
E	{D:16, B:5}
F	{B:12, C:10}



Directed graphs

Undirected graphs have connections in both directions. Directed graphs only apply in one direction and are represented with edges with arrow heads on one end.

Directed graph as adjacency list

A	
B	[A, E, C, F]
C	[F]
D	[A, E]
E	
F	

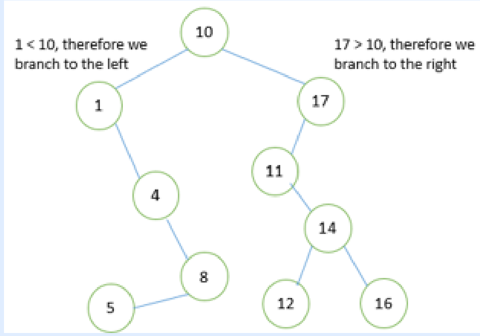
Directed graph as adjacency matrix

		To					
		A	B	C	D	E	F
From	A						
	B	1		1		1	1
	C						1
	D	1				1	
	E						
	F						

Binary Tree

- In a binary tree a node can only have a maximum of two child nodes
- A binary tree can be used for sorting a sequence of numbers
- The first number is the root node
- If the number is smaller than the node then we branch left, if it is bigger, we branch right

A binary tree for a sequence of numbers: 10,1,17,4,8,11,14,16,5,12

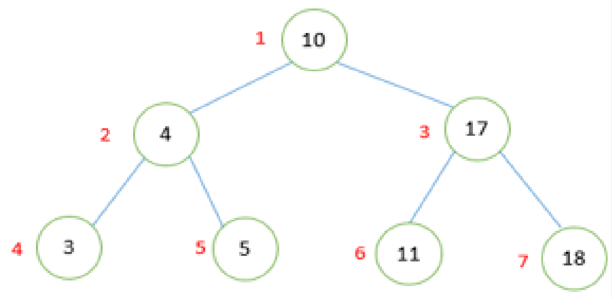


Tree data structure

- We can represent a tree data structure with three lists/arrays
- An array contains the value at the nodes
- An array that points to the location of left child of the node in the values array
- An array that points to the location of right child of the node in the values array
- If a node does not have child node then this is indicated with a -1 or null

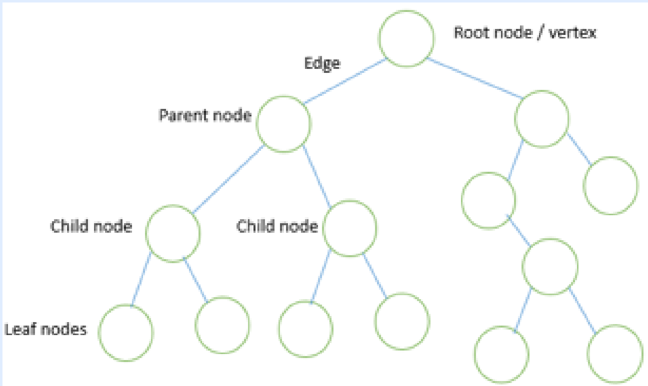
Node index	Left Tree	Right Tree	Values
1	2	3	10
2	4	5	4
3	6	7	17
4	-1	-1	3
5	-1	-1	5
6	-1	-1	11
7	-1	-1	18

Null pointers



Trees

- A tree is a connected, undirected graph with no cycles
- *Connected*: Every node is connected either indirectly to directly to every other node
- *No Cycles*: There is only one path between nodes
- *Undirected*: can traverse in both directions along the edges
- A *rooted tree* has a root node that has no parent and all other nodes are descended from the root. All other nodes can be a parent and/or a child node.
- A leaf has no children



Computer Science

1.4 Data types, data structures and algorithms

Hash Table

- Hashing allows stored data to be accessed very quickly without the need to search though every record. This is achieved by relating the data itself to its index position using a key. There are several hashing algorithms that can achieve this.
- If the calculated number is bigger than the length of the list then you will need to apply the modulo
- Collisions occur when a bin is already occupied. In such a situation the data are placed in the next available bin
- You can rehash with a higher modulus and number of elements when the number of collisions become high
- The load factor is the number of occupied bins delivered ny the number of total bins
- The hash table should contain more bins than there are elements that you would like to store by a load factor of 0.75
- If the load factor is exceeded, we can rehash using a larger hash table with a greater number of bins.

Worked Example

Put the numbers 81, 93, 76, 51, 17, 61 into a hash table with 10 elements. Because the values are bigger than the length of the list, we apply the modulo which is the length of the table.

81 MOD 10 - 1 (81 goes into index position 1)
93 MOD 10 = 3
76 MOD 10 = 6
51 MOD 10 = 1 (a collision has occurred, place in next available position)
17 MOD 10 - 7
61 MOD 10 - 1

Other hashing algorithms

If the data you want to convert has letter and not numbers, you can convert the data to corresponding ASCII values.

Worked Example

Let us consider the following names: Bart, Homer, Lise, Milhouse, Ralk. We have a has table with 10 elements.

Homer	72 + 111 + 109 + 101 + 114	507 MOD 10	7
Bart	66 + 97 + 114 + 99	393 MOD 10	3
Lisa	76 + 105 + 115 + 97	393 MOD 10	3 (collision)
Milhouse		898 MOD 10	8
Ralf		389 MOD 10	9

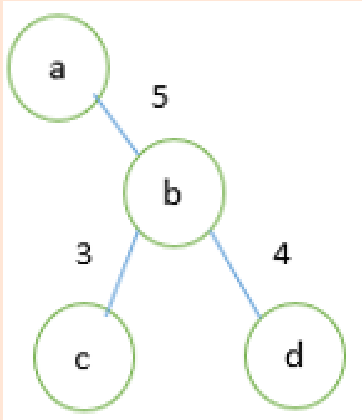
0	1	2	3	4	5	6	7	8	9
			Bart	Lisa			Homer	Milhouse	Ralf

Dictionaries

A dictionary is an abstract data type. It contains a list of pairs of values with a key that is associated with a value. We use key to access a value.

```
dict = {key1: value1, key2: value2, ..., keyN: valueN}
```

Create empty dictionary	id={}
Create a dictionary	id= {23:"James",25:"Thomas",18:"Gordon",32:"Percy"}
Return a value associated with a key	id[23]->James
Add a value	id[33]="Trevor"
List values	id
Remove a value	del id[32]



Using a dictionary to represent a graph

g={"a":{"b":5}, "b":{"a":5,"c":3,"d":4}, "c":{"b":3}, "d":{"b":4}}

Vectors

Vector Notation

Function Representation

A vector can be represented as a Function (*f*: S → R) where S is the set that maps to R. For instance S=[0,1,2,3,4] and R=[4.0,5.5,6.7,9.1,-2.3]

0→4.0
1→5.5
2→6.7
3→9.1
4→-2.3

List/1-D array representation

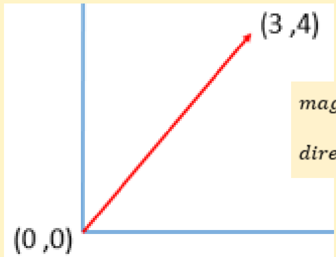
e.g. A 5 vector over R would be: [4.0,5.5,6.7,9.1,-2.3]

Dictionary representation

A 5 vector could be represented as a dictionary with both sets and mapping e.g. R={4.0,1:5.5,2: 6.7,3: 9.1,4: -2.3}

Visualisation of a vector

We can represent a vector as geometric point in space. A 2-vector e.g. [3,4] can be represented by an arrow with its tall at [0,0] and its head at [3,4]. Vectors have both magnitude and direction.



$$magnitude = \sqrt{3^2 + 4^2}$$

$$direction = \tan^{-1}(4/3)$$

Vector addition

Each element in the vector is added to the corresponding value at that element in the other vector.

Worked example:

Find a+b where a =[2, 3, 6, 8] and b=[3, 1, 4, 5]

a = [2, 3, 6, 8]
+ + + +
c = [3, 1, 4, 5]
a+b = [2+3, 3+1, 6+4, 8+5]
a+b = [5, 4, 10, 13]

Scalar vector multiplication

Vectors can be multiplied by scalars (single numbers). Each element is multiplied by the scalar

Worked Example

Find 2a where a= [2, 3, 6, 8]
2a = [(2x2) , (3x2) , (6x2) , (8x2)]
2a = [4, 6, 12, 16]

Dot product

The dot product of two vectors is calculated by multiplying the corresponding element in both vector and adding together all the elements. Given vector a and b such that
a = [a1, a2, ..., an] and b = [b1, b2, ..., an]
Then a.b = (a1 x b1) + (a2, x b2) + ... , + (an x bn)

Worked Example

Find a.b where a= [2, 3, 6, 8] and b= [3, 1, 4, 5]
a =[2, 3, 6, 8]
x x x x
b =[3, 1, 4, 5]
a.b =[6 + 3 + 24 + 40]
a.b = 73

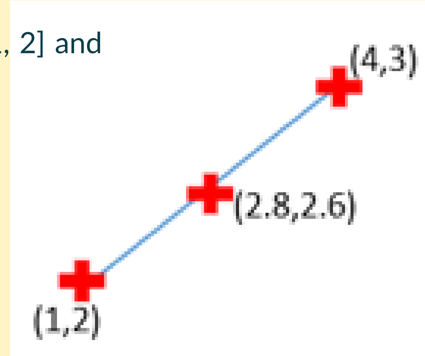
Convex combination of 2 vectors

Every convex combination of 2 points lines on a line between the two points 2 points. This has the form *au + bv* where a + b = 1 and *a, b* >= 0

Worked Example

Find the convex combination *au + bv* of vectors u=[1, 2] and v=[4, 3], where a=o.4 and b=0.6

au = [1*0.4, 2*0.4]
au = [0.4, 0.8]
bv = [4*0.6, 3*0.6]
bv = [2.4, 1.8]
au+bv = [2.4+0.4, 0.8+1.8]
au+bv = [2.8, 2.6]



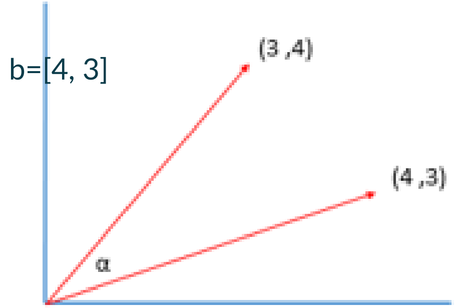
Angle between 2 vectors

The angle between two vectors is calculated as:
cos(a) = a.b / |a|.|b|

Worked Example

Calculate the angle between two vectors a=[3, 4], b=[4, 3]

a.b = (3 x 4) + (4 x 3) = 24
$|a|=\sqrt{3^2+4^2}=5$
$|b|=\sqrt{4^2+3^2}=5$
24 / 5 . 5 = 24/25 = 0.96 = 16.3°



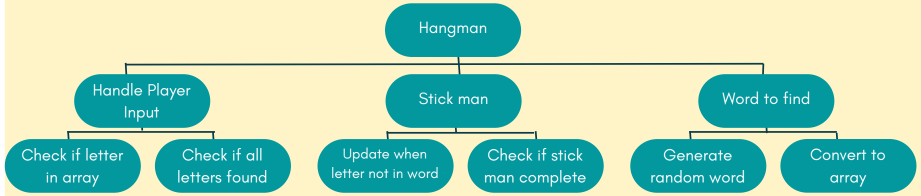
Computer Science

2.1 Elements of computational thinking

Thinking Procedurally

Problem decomposition

Decomposition is the breaking down of a complex problem into smaller more manageable problems that are easier to solve. Each component of the program completes a specific task. This allows algorithms to be more modular.



Each 'end of branch' is a module/subroutine to be programmed. This is known as top-down design. The diagram above is called a hierarchy chart.

Advantages of Decomposition

- Large programs are broken down into subtasks/subroutines that are easier to program and manage
- Each subroutine (i.e. module) can be individually tested
- Modules can be re-used several times in a program or elsewhere
- Frequently used modules can be saved in a library and used by other programs. For example, in C# rnd, sqrt. Having components that have already been written, debugged and tested will save the programmer time.
- Several programmers can simultaneously work on different modules. shortening development time
- Programs are more reliable and have fewer errors
- Programs take less time to test and debug
- A well-organised modular program is easier to follow
- New features can be added by adding new modules

Thinking Concurrently

Parallel Processing

- Requires a processor/CPU with multiple cores
- Each core processes different instructions at exactly the same time
- Impossible on a single core processor
- CPUs can contain up to 64 cores (and counting)

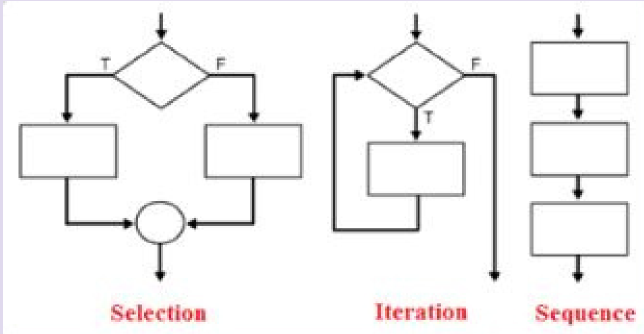
Thinking Logically

Tools for Designing Algorithms

- **Hierarchy charts**: Useful for identifying the major task and breaking these down into subtasks
- **Flowcharts:** Useful for getting down initial ideas for individual subroutines
- **Pseudocode:** will translate easily into program code

Programming Structures

- **Sequence:** one line is executed after another
- **Selection:** if, elif, else; switch, case, endswitch
- **Iteration:** while, endwhile; do, until; for, next loops



Flowchart Symbols

Symbol	Name	Function
	Start/end	An oval represents a start or end point
	Arrows	A line is a connector that shows relationships between the representative shapes
	Input/Output	A parallelogram represents input or output
	Process	A rectangle represents a process
	Decision	A diamond indicates a decision

Programming Errors

- When you first start programming, the most common errors you make will be syntax errors
- Logic errors are another type of error. They occur not because of an error in the syntax, but instead because you get unexpected results
- Logic errors normally occur at points where selection occur (if...else) or at points of iteration

Concurrent Processing

- Happens on a processor with a single core
- The core appears to process different instructions at the same time, but it is an illusion
- Each process is given slices of processor time, giving the appearance that several tasks are being performed simultaneously

Threads

- A process can be broken down into multiple threads – instructions to be completed one after the other in sequence
- A single core can cope with two threads simultaneously
- A four-core CPU would have (be able to handle) eight threads (simultaneously)
- Thread can start and end at different times
- Thread can overlap in their execution (fetch-decode-execute)

2.1 Elements of computational thinking

Computer Science

Thread locking

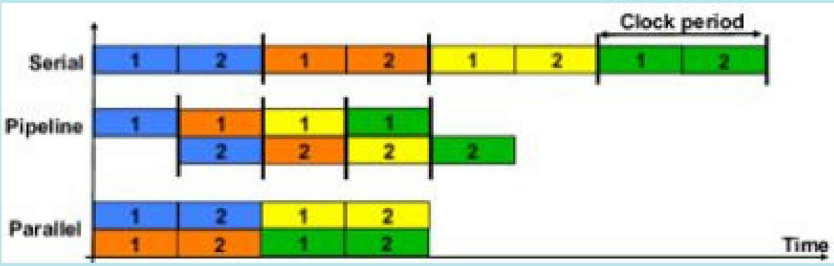
- Sometimes you might have a situation where you don't want threading to occur
- This would be when you don't want two operations to be happening simultaneously, because it will create a bug or similar problem
- For example, two threads are incrementing a counter, both by one. The result should be 117, but since they are happening simultaneously, the outcome is not expected:

o Counter value is 115
o First thread reads the value of the counter from the memory (115)
o First thread increases the local counter value (116)
o Second thread reads the value of the counter from the memory (115)
o Second thread increases the local counter value (116)
o Second thread saves the local counter value to the memory (116)
o First thread saves the local counter value to the memory (116)
o Value of the counter is 116

- In such situations, as part of the code, you can lock threads for certain operations, preventing this from happening (first operation completes before the second is implemented)

Pipelining

- Involves splitting larger tasks, and overlapping the processing of them
- With regards the CPU, to speed up processing time, while one instruction is fetched, another can be decoded and a third executed
- Can also relate to, in an algorithm, the output from one procedure being used for the input for another



Problem Solving

Enumeration

- An exhaustive search for all possible solutions until one works
- Also known as brute force - testing every combination of possible routes until you find the shortest one

Simulation

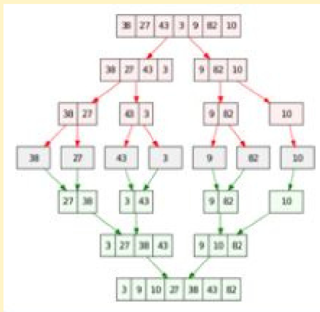
- This is where the situation is simulated to help find the best solution to the problem
- Might require an entirely computer-based simulation, e.g. to solve queuing problems
- Might required a physical model too. E.g. to investigate air resistance on a model of a new F1 car design

Pattern Recognition

- This involves utilising a database of previously experienced patterns in order to find a match
- May take heuristic approach to find a best fit

Divide and Conquer

- This involves reducing the size of a problem with every iteration
- The best-known example is the binary search, which is a method of searching a sorted list for a particular item
- Another is a merge sort



Backtracking

- Backtracking is an approach to a problem where partial solutions are built up to produce a full solution
- If a pathway fails, some of the partial solutions up to that point are discarded and you start again from the last potentially successful point
- Same as trial and error or trial and improvement

Data Mining and Big Data

- Data mining is the process of digging through large sets of data in order to (one or more of); find hidden links and relationships, recognise patterns and trends and predict future trends
- Big data was a term coined in the early 2000s to describe vast amounts of information now available to the computing world

Heuristic Methods

- There are often other options for solving problems apart from brute force methods
- One method is to find a solution which is likely to be correct, or which is nearly but not quite, perfect but sufficient, in a reasonable time frame. This is called a heuristic approach

Performance Modelling

- It is often important to know how a system will perform in real life before implementing it
- To save money, time and in the interest of safety, models (simulations) are built (physical and/or computational) to predict what will happen in real life
- It can also be used to stress-test a program with large volumes of test data before going live.

Computer Science

Unit 2.2 Problem Solving and Programming

Variables

- Variables can be either global or local scope.
- Scope refers to the section of code where the variable can be accessed.
- A local variable in a subroutine has precedence over a global variable with the same name.

Local Variables

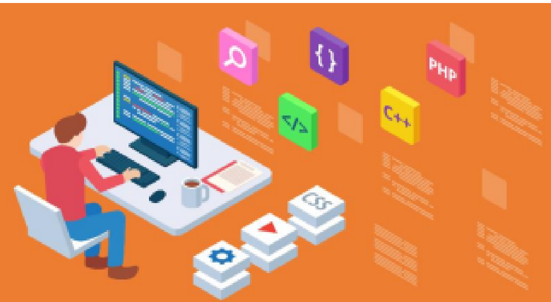
- Can only be accessed within the subroutine where they were defined.
- Multiple variables with the same name can exist in different subroutines.
- Are deleted when the subroutine ends.
- Ensures subroutines are self contained.

Global Variables

- Can be accessed through the whole program.
- Used for values needed throughout the program.
- Risk the variable is unintentionally edited.
- Uses memory for longer.

Problem Recognition

- Stakeholders say what they need from the solution.
- This information is used to produce a clear list of system requirements and a definition of the problem.
- We may consider the strengths and weaknesses of a current system.
- We may consider the required inputs, outputs and the volume of stored data.



Modularity

- Large or complex programs can be split into smaller self contained modules.
- This makes it easier to divide tasks between a team and manage the project.
- It simplifies maintenance since each component can be handled individually.
- It improves the reusability of code.
- Top Down (Stepwise) Refinement
- A technique used to modularise programs.
- The problem is broken into sub problems until each sub problem is a single task.
- Modules form blocks of code called subroutines.

Programming Constructs

- Sequence – Code is executed line by line from the top down.
- Breaching – A block of code is run only if a condition is met using IF and ELSE statements
- Count Controlled Iteration – A block of code is run a certain number of times. Uses FOR, WHILE or REPEAT UNTIL statements.
- Condition Controlled Iteration – A block of code is run while or until a condition is met. Uses FOR, WHILE or REPEAT UNTIL statements.

Integrated Development Environment

- Programs used to write code.
- Contains a set of tools which make it easier for programmers to write, develop and debug code.
- May include stepping, variable watching, breakpoints, source code editor and debugging tools.

Recursion

- When a subroutine calls itself during execution.
- Continues until a stopping condition is met.

Advantages

- Requires fewer lines of code.
- Easier to express functions such as factorials recursively.

Disadvantages

- Risk of stack overflow if memory runs out.
- Often challenging to trace and locate errors.

Object Orientated Techniques

- Object oriented programming languages use classes.
- A class is a template for an object.
- An object is an instance of a class.
- It defines the behaviour and state of objects.
- Object state uses attributes.
- Object behaviour uses methods.
- Encapsulation is used to edit attributes.
- Top down design applies encapsulation to modules.
- Modules are built to be self contained and reusable.

Can a Problem be Solved by Computational Methods?

- Not all problems can be solved in this way.
- Some may need too many resources or time.
- Problems which can be solved using algorithms lend themselves well to being solved via computational methods.
- We must identify whether the problem can be solved using computational methods before we attempt to solve it.

Problem Solving Strategies

Backtracking

- Uses algorithms, often recursively.
- Builds a solution methodically.
- Based on paths which have been visited and found to be correct.
- The algorithm backtracks to the previous stage if an invalid path is found.

Data Mining

- Identifies patterns or outliers in large data sets, often collected from multiple sources.
- These data sets are known as big data.
- It spots correlations between data and other trends which might not be easy to see.
- Can be used to make predictions about the future.
- A useful tool to assist in business and marketing.

Heuristics

- A non optimal or rule of thumb approach.
- Used to find an approximate solution to a problem.
- Used where the standard solution takes too long.
- Does not produce a 100% accurate or complete solution.
- Provides an estimate for intractable problems.
- Performance Modelling
- Mathematical method to test loads on systems.
- A cheaper and less time consuming method of testing applications.
- Used for safety critical systems where a trial run can't be carried out.

Pipelining

- Modules are divided into individual tasks.
- Tasks are developed in parallel.
- Allows faster completion.
- The output of one process is often the input of another.
- Often used in RISC processors, which perform different parts of the Fetch-Decode-Execute cycle at the same time.

Visualisation

- Presenting data using charts or graphs.
- Makes it easier for humans to understand.
- Allows trends or patterns to be more easily identified.

Functions and Procedures

- Named code blocks which perform a particular task.
- Functions must always return a single value.
- Procedures do not have to return a value.
- Parameters can be passed to them by either reference or value.

Passing by Reference

- The address of the parameter only is given to the subroutine.
- The subroutine works on the value at the given address.

Passing by Value

- A copy of the value is passed to the subroutine.
- The original value is unchanged.
- The copy is deleted at the end of the subroutine.
- Exam questions will use this technique unless told otherwise.
- Exam questions will use the format function function(x:value, y:value)

Divide and Conquer

- A problem solving technique with three parts.
- Divide - halve the size of the problem with each iteration.
- Conquer - solve the subproblems.
- Merge - combine the solutions.
- It is applied in binary search, quick sort and merge sort.
- It is a quick way to simplify complex problems.

Problem Decomposition

- The problem is broken down into smaller subproblems.
- This is repeated until each subproblem can be represented using a single subroutine.
- This reduces the complexity of the problem and makes it easier to solve.
- It enables programmers to see which areas can be solved using pre-existing libraries or modules.
- It makes the project easier to manage.
- Subproblems can be assigned to different specialist teams or individuals.
- Modules can be designed and tested individually before being combined.
- It makes it possible to develop modules in parallel and therefore finish more quickly.
- It is easier to debug the code and locate errors.

Abstraction

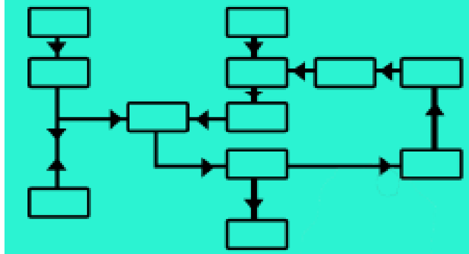
- Represents real world entities using computational elements.
- Excessive details are removed to simplify the problem.
- This may then match a problem which has previously been solved.
- Existing modules, functions or libraries can then be used to solve the problem.
- Levels of abstraction divide a complex problem into smaller parts.
- Different levels can be assigned to teams whilst hiding details of other layers.
- This makes the project easier to manage.
- Abstraction by generalisation groups together sections with similar functionality.
- This allows segments to be coded together, saving time.

Computer Science

Unit 2.3 Algorithms

Algorithms
- A set of instructions used to solve a set problem.
- Inputs must be clearly defined.
- Must always produce a valid output.
- Must be able to handle invalid inputs.
- Must always reach a stopping condition.
- Must be well-documented for reference.
- Must be well-commented.

Designing Algorithms
- The priority for an algorithm is to achieve the given task.
- The second priority is to reduce time and space complexity.
- There may be a conflict between space and time complexity and the requirements and situation for an algorithm will dictate which is more important.
- To reduce space complexity, make as many changes on the original data as possible. Do not create copies.
- To reduce time complexity, reduce the number of loops.



Queues
- FIFO (First in first out)
- Often an array.
- The front pointer marks the position of the first element.
- The back pointer marks the position of the next available space.

Queue Functions
- Check size `size()`
- Check if empty `isEmpty()`
- Return top element (but don't remove) `peek()`
- Add to the queue `enqueue(element)`
- Remove element at the front of the queue and return it `dequeue()`

Big-O Notation
- 0(1) - Consistent time complexity - The amount of time is not affected by the number of inputs.
- 0(n) - Linear time complexity - The amount of time is directly proportional to the number of inputs.
- 0(nn) - Polynomial time complexity - The amount of time is directly proportional to the number of inputs to the power of n.
- 0(2n) - Exponential time complexity - The amount of time will double with every additional input.
- 0(log n) - Logarithmic time complexity - The amount of time will increase at a smaller rate as the number of inputs increases.

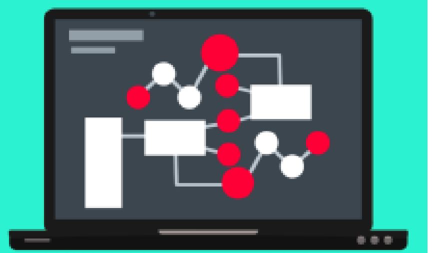
Sorting Algorithms
- Places elements into a logical order.
- Usually numerical or alphabetical.
- Usually in ascending order.
- Can be set to work in descending order.

Bubble Sort
- Compares elements and swaps as needed.
- Compares element 1 to element 2.
- If they are in the wrong order, they are swapped.
- This process is repeated with 2 and 3, 3 and 4, and so on until the end of the list is reached.
- This process must be repeated as many times as there are elements in the array.
- Each repeat is referred to as a "pass".
- Can be modified to improve efficiency by using a flag to indicate if a swap has occurred during the pass.
- If no swaps are made during a pass the list must be in the correct order and so the algorithm stops.
- A slow algorithm.
- Time complexity of 0(n2)

Insertion Sort
- Places elements into a sorted list.
- Starts at element 2 and compares it with the element directly to its left.
- When compared, elements are inserted into the correct position in the list.
- This repeats until the last element is inserted into the correct position.
- In the 1st iteration 1 element is sorted, in the 2nd iteration 2 are sorted etc.
- Time complexity of 0(n2)

Merge Sort
- A divide and conquer algorithm.
- Formed of a Merge and MergeSort function.
- MergeSort divides the input into two parts.
- It then recursively calls MergeSort on each part until their length is 1.
- Merge is called.
- Merge puts the groups of elements back together in a sorted order.
- You will not be asked about the detailed implementation of this algorithm but do need to know how it works.
- It is more efficient than bubble and merge sort.
- It has a worst case time of O(n log n)

Quick Sort
- Selects an element and divides the input around it.
- Often selects the central element, which is known as the pivot.
- Elements smaller than the pivot are listed to its left.
- Larger elements are listed to its right.
- The process is repeated recursively.
- Slow.
- Time complexity of O(n2)



Searching Algorithms
- Used to locate an element within a data structure.
- Many different forms exist.
- Each is suited to different purposes and data structures.

Linear Search
- Most basic search algorithm.
- Works through the elements one at a time until the requested element is found.
- Does not need data to be sorted.
- Easy to implement.
- Not very efficient.
- Time Complexity is 0(n)

Binary Search
- Only works with sorted data.
- Finds the middle element, then decides on which side of the data the requested element is.
- The unneeded half is discarded and the process repeats until either the requested element is found or it is determined that the requested element does not exist.
- A very efficient algorithm.
- Time Complexity is 0(log n)

Stacks
- FILO (First In Last Out)
- Often an array.
- Uses a single pointer (the top pointer) to track the top of the stack.
- The top pointer is initialised at -1, with the first element being 0, the second 1 and so on.

Stack Functions
- Check size `size()`
- Check if empty `isEmpty()`
- Return top element (but don't remove) `peek()`
- Add to the stack `push(element)`
- Remove top element from the stack and return it `pop()`

Space Complexity
- The amount of storage space the algorithm takes up.
- Does not have a defined notation.
- Copying data increases the storage used.
- Storage space is expensive so this should be avoided.

Linked Lists
- Contains several nodes.
- Each node has a pointer to the next item in the list.
- For node N, N.next will access the next item.
- The first node is the head.
- The last node is the tail.
- Searched using a linear search.

Time Complexity
- How much time an algorithm needs to solve a problem.
- Measured using big-o notation.
- Shows the amount of time taken relative to the number of inputs.
- Allows the required time to be predicted.

Logarithms
- The inverse of an exponential.
- An operation which determines how many times a certain number is multiplied by itself to reach another number.
- x y = log(x)
- 1 (20) 0
- 8 (23) 3
- 1024 (210) 10

Path Finding Algorithms

Dijkstra's Algorithm
- Finds the shortest path between two points.
- The problem is depicted as a weighted graph.
- Nodes represent the items in the scenario such as places.
- Edges connect the nodes together.
- Each edge has a cost.
- The algorithm will calculate the best way, known as the least cost path, between two nodes.

A* Algorithm
- Provides a faster solution than Dijkstra's Algorithm to find the shortest path between two nodes.
- Uses a heuristic element to decide which node to consider when choosing a path.
- Unlike Dijkstra's Algorithm, A* only looks for the shortest path between two nodes, instead of the shortest path from the start node to all other nodes.

Trees
- Consists of nodes and edges.
- Cannot contain cycles.
- Edges are not directed.
- Can be traversed using depth first or breadth first.
- Both methods can be implemented recursively.

Depth First (Post Order) Traversal
- Moves as far as possible through the tree before backtracking.
- Uses a stack.
- Moves to the left child node wherever possible.
- Will use the right child node if no left child node exists.
- If there are no child nodes, the current node is used.
- the algorithm then backtracks to the next node moving right.

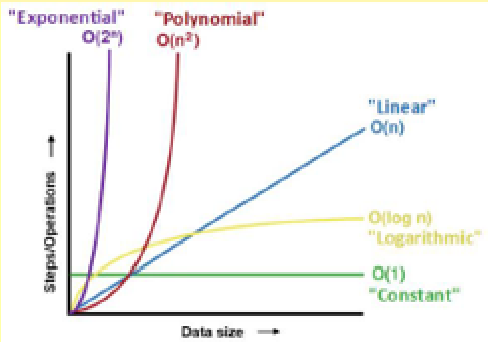
Breadth First
- Starts from the left.
- Visits all children of the starting node.
- Then visits all nodes directly connected to each of these nodes in turn.
- Continues until all nodes have been visited.

Computer Science

2.3 Algorithms

Big-O Notation

The time complexity of an algorithm is the worst-case number of operations required for an algorithm to complete given a data size of n



- Time complexity = measure of the time required by a computer to run the algorithm, given input values of size n
- Space complexity = amount of computer memory required to run the algorithm, given input values of size n
- Big-O value shows how time/memory increases input data size increases
- The default Big-O value normally considered is the worst-case, though the best case and average case should be considered
- The best time complexity is O(1), then O(log n), then O(n), etc...

Search Algorithms

Linear Search

If you have to search for items in a file (or in an array), and the list/array are not in any particular order (i.e. sorted), you will have to search through the items one by one.

```
function binarySearch (alist, itemSought)
    LB = 0
    UB = length(alist) - 1
    while LB <= UB
        mid = (LB + UB) DIV 2
        if alist[mid] = itemSought then
            return mid
        else
            if alist[mid] < itemSought then
                LB = mid + 1
            else
                UB = mid - 1
            endif
        endif
    endwhile
    return -1
endfunction
```

As the size of the data set doubles, the maximum number of possible checks also doubles. This means the time complexity is O(n).

Binary Search

- Can only be performed on an ordered list
- Examine the middle value. Use (LB + UB)/2 and round down if there's an even number of items (i.e. DIV)
- Check if item you are looking for is more than or less then this item
- Whichever half it must be in, discard the other half including the middle item you had
- Repeat until found

As the size of the data set doubles, the maximum number of possible checks only increases by one. This means the time complexity is O(log n).

```
function linearSearch (alist, itemSought)
    index = -1
    i = 0
    found = False
    while i < length(alist) and found = False
        if alist[i] = itemSought then
            index = i
            found = True
        endif
        i = i +1
    endwhile
    return index
endfunction
```

Binary Search - Recursive Version

```
function binarySearch (alist, itemSought, LB, UB)
    if UB < LB then
        return -1
    else
        mid = (LB + UB) DIV 2
        if alist[mid] > itemSought then
            return binarySearch (alist, itemSought, LB, mid-1)
        else
            if alist[mid] < itemSought then
                return binarySearch (alist, itemSought, mid+1, UB)
            else
                return mid
            endif
        endif
    endif
endfunction
```

In the best case, both searches have equal complexity.

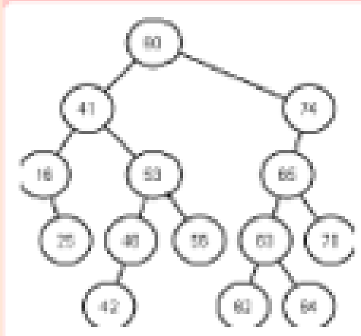
However, in average and worst case, binary search is more efficient (O(log n) is better the O(n)).

Linear Search vs Binary Search Time Complexity

Time Complexity	Best case	Average Case	Worst Case
Linear Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)

Binary Tree Search

Similar to binary search algorithm. except instead of using midpoints, half od the tree/subtree is eliminated with each pass after examining its root



Binary Tree Search - Time Complexity

```
function binarySearch (itemSought, currentNode)
    if currentNode = None then
        return False
    else
        if itemSought = item at currentNode then
            return True
        else
            if itemSought < item at currentNode then
                if left child exists then
                    return binarySearch (itemSought, left child)
                else
                    return False
                endif
                if right child exists then
                    return binarySearch (itemSought, right child)
                else
                    return False
                endif
            endif
        endif
    endif
endfunction
```

- The number of items to search is halved with each pass
- Conversely, the (maximum) number of passes increases by one as the tree is doubled in size
- This gives the same time complexity as the binary search, O(log n)

Computer Science

Linear Search Algorithm

- The purpose of the linear search algorithm is to find a target item within a list
- Compares each list item one-by-one against the target until the match has been found and returns the position of th eitem in the list
- If all items have been checked and the search item is not in the list, then the program will run through to the end of th elist and return a suitable message indicating that the item is not in the list
- The algorithm runs in linear time. If n is the length of the list, then at worst the algorithm will make n comparisons. At best, it will make 1 comparison and on average it will make (n+1)/2 comparisons
- The performance of the algorithm will be improved iof the target item is near the start of the list
- The time complexity of the linear search algorithm is 0(n)

Example

Find the position of letter "Z" within the following list. Assume we do not have visibility of the list:

Index position	0	1	2	3	4	5	6	7
Value	V	A	S	Z	X	R	T	G

We compare it with the value in index position 0. We find that the value is "v" so we need to move on to the next index position. At index position 1 and 2, we still have not found z. However, we get to index position 3 and we compare the target with the value and we find they match, so the algorithm returns the index position and stops.

```
Pseudocode
i ← 0
x ←  len (listOfItems)
pos ←  -1
found ←  False
WHILE i < x AND NOT found
 IF listOfItems[1] == itemSearch THEN
  found ←  True
  pos ←  i + 1
 ENDIF
 i=i+1
ENDWHILE
OUTPUT pos
```

Worked example: given the following values for *listOfItems* and *itemSearch*, we have the following trace table

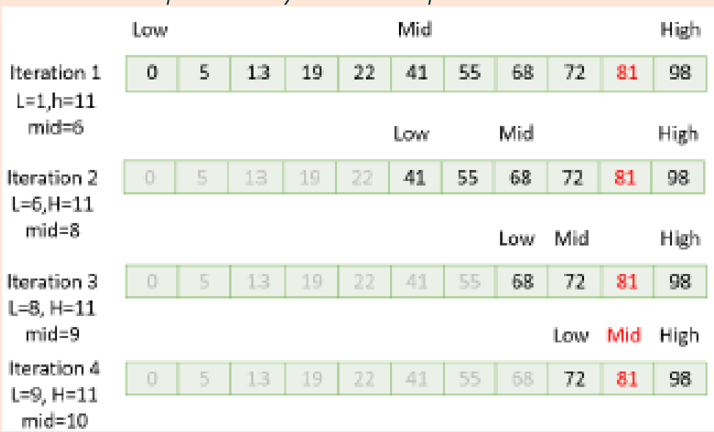
```
listOfItems ←  [6,3,9,1,2]
itemSearch ←  1
```

i	x	pos	found	itemSearch	listOfItems [1]	Output
0	5	-1	False	1	6	
1					3	
2					9	
3					1	
4		4	True			4

Binary Search Algorithm

- The binary search algorithm works on a sorted list by identifying the middle value in the list and comparing it with the search item
- If the search item is smaller, the mid element becomes the new high value for the search area
- If the search item is larger, the mid element becomes the low value for the search area
- This keeps repeating until the search item is found
- When the search item is found, the index position of the item is returned
- At each iteration, the search are halved in size. Consequently, this is an efficient algorithm
- The time complexity if the binary search algorithm is 0(log n)

Examples: Binary search in operation to find 81



Pseudocode

```
low ← 1
high ← LENGTH(arr)
mid ← (low + high) DIV 2
WHILE val # A[mid]
 IF A[mid] < val THEN
  low ← mid
 ELIF A[mid] > val THEN
  high ← mid
 ENDIF
  mid ←  (low + high) DIV 2
  ENDWHILE
OUTPUT mid
```

*Worked example: given the following values for **arr** and **val**, we have the following trace table:*

mid	high	low	A[mid]	A[high]	A[low]
6	11	1	41	98	0
8	11	6	68	98	41
9	11	8	72	98	68
10	11	9	81	98	72

Linear search versus binary search

	Advantages	Disadvantages
Linear Search	• Very simple algorithm and easy to implement • No sorting required • Good for short lists	• Slow because it searches through the whole list • Very inefficient for long lists
Binary Search	• Much quicker than linear search because it halves the search zone at each step	• The list needs to be ordered

Computer Science

2.3 Algorithms — Sorting Algorithms

Python implementation using lists

```
def binaryTreeSearch(node,searchItem)
 path.append(values[node])
 if values[node] == searchItem:
  return "Value in Tree. Path: "+str(path)
 elif values[node] < searchItem:
  if treeRight[node] == -1:
   return "Value not in Tree"
 return binaryTreeSearch(treeRight[node] ,searchItem)
 elif values[node] > searchItem:
  if treeLeft[node] == -1:
   return "Value not in Tree"
  return binaryTreeSearch(treeLeft[node] ,searchItem)

path = []
# node[0,1,2,3,4,5,6,7,8,9]
values = [10,1,17,4,11,8,14,5,12,16]
treeLeft = [1,-1,4,-1,-1,7,8,-1,-1,-1]
treeRight=[2,3,-1,5,6,-1,9,-1,-1,-1]
print (binaryTreeSearch(0, 5))
```

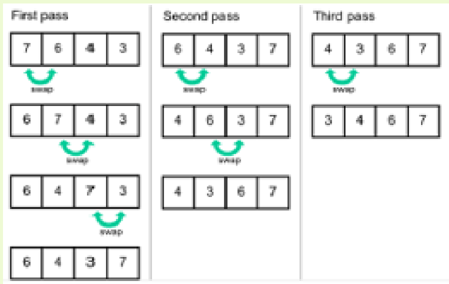
Tracing

Call num	*Call*	*Output*	*Return*
1	*BinarySearchTree(10,5)*	*10*	
2	*BinarySearchTree(1,5)*	*1*	
3	*BinarySearchTree(4,5)*	*4*	
4	*BinarySearchTree(8,5)*	*8*	
5	*BinarySearchTree(5,5)*	*5*	*5*

Sorting Algorithms

Bubble Sort

- Go through the array, comparing each item to the one next to it
- Of it is greater then the next one, swap them over
- The last element will be the largest one after the first pass
- There will be a total of n-1 passes. The number of comparisons reduce by one with each pass.




```
numbers = [9, 5, 4, 15, 3, 8, 11]
numItems = length(numbers)
i = 0
swapMade = True
while i < (numItems - 1) and (swapMade = True)
    swapMade = False
    for j = 0 to (numItems - i - 2)
        if numbers[j] > numbers[j+1]
            #swap the numbers
            temp = numbers[j]
            numbers[j] = numbers[j+1]
            numbers[j+1] = temp
            swapMade = True
        endif
    next j
    i = i + 1
endwhile
print (numbers)
```

Insertion Sort

Much like you would sort a hand of playing cards. From the left, move each card into the correct position relative to those its left.

```
function insertionSort(aList)
    n = length(aList)
    for index = 1 to n - 1
        itemInHand = aList[index]
        position = index
        while position > 0 and aList[position-1] > itemInHand
            aList[position] = aList[position-1]
            position = position - 1
        endwhile
        aList[position] = itemInHand
    next index
endfunction
```

Bubble Sort vs Insertion Sort Time Complexity

	Time Complexity			Space Complexity
Algorithm	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$

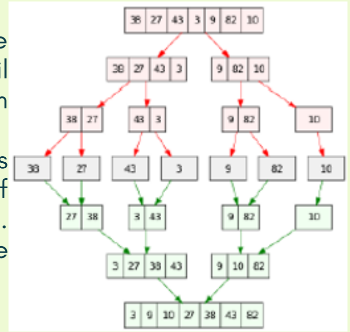
- Both have the same best, average and worst case time complexity
- However, in real-world terms, the insertion sort is considered slightly more efficient – in most average situations, there will tend to be slightly fewer iterations required to take place than for a bubble sort
- Also, a bubble sort requires items to be swapped, while an insertion sort requires items to be simply moved (which is a less complex process)

Bubble Sort vs Insertion Sort Time Complexity

- For a list of size n, both algorithms will require n memory locations
- No matter how big the data set gets, the amount of space required (extra to the data itself) remains the same
- Both algorithms are 'inplace' – the sorting takes place within the data set itself, not outside of it
- Thus, the space complexity of both algorithms is 0(1) (i.e constant no matter how large the data set is)

Merge Sort

- Successively split the lists into sublists until there is only one item in each sublist
- Merge pairs of sublists into sequenced lists of 2, then 4, ther 8 etc.. items until all items are in one merged list
- This is the sorted list




```
function mergesort(array a)
    if length(a) == 1 return a
    array L - [a[0] ... a[n/2]]
    array R - [a[n/2+1] ... a[n]]
    L - mergesort(L)
    R - mergesort(R)
    return merge(L, R)
endfunction

function merge(L, R)
    sortedArray as array
    while (L and R are BOTH not empty)
        if (L[0] > R[0])
            add R[0] to the end of sortedArray
            remove R[0] from R
        else
            add L[0] to the end of sortedArray
            remove L[0] from L

    while (L is not empty)
        add L[0] to the end of newArray
        remove L[0] from L
    while (R is not empty)
        add R[0] to the end of newArray
        remove R[0] from R
    return newArray
endfunction
```

- This is a recursive function
- This first function continually subdivides the list until we get individual 'lists' of one element each
- Due to the nature of recursion, the 'merge' function occurs as part of the unwinding, gradually merging the lists together, two at a time

2.3 Algorithms Sort Algorithms

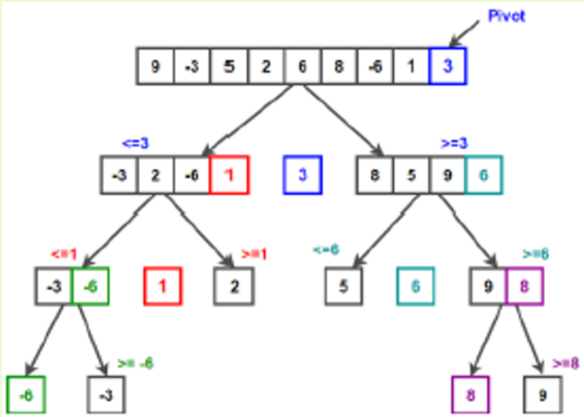
Merge Sort Time Complexity

- Since this uses a divide and conquer approach, as seen for a binary search (doubling the number of items only adds one more iteration), the time complexity is 0(log n)
- However, for each 'set' of n items to sort, there will be n sublists that need to be combined
- This means the time complexity has to be multiplied by a factor of n
- So, overall time complexity is 0(n log n)
- This is the same in the best case, average case and worst case

Merge Sort Space Complexity

- The merge sort requires additional memory for storing the left and right halves of the list as they are combined (worst case, this will be n items in both halves combined)
- This gives a space complexity of 0(n)

Quick Sort



- Select a pivot value, e.g. first item in the list, but could be any
- Divide the remainder of the list in two portions:
  1. all elements less than the pivot value must be in the first partition
  2. all elements greater than the pivot value must be in the second partition
- Recursively repeat the process until each partition holds only once item. Recombining the elements from the bottom will mean the list is now sorted

```
function partition (alist, start, end)
   pivot = alist[start]
   leftmark = start + 1
   rightmark = end
   done = False
   while done = False
      while leftmark <= rightmark and alist[leftmark] <= pivot
         leftmark = leftmark + 1
      endwhile
      while alist[rightmark] >= pivot and rightmark >= leftmark
         rightmark = rightmark - 1
      endwhile
      if rightmark < leftmark
         done = True
      else
         // swap the list items
         temp = alist[leftmark]
         alist[leftmark] = alist[rightmark]
         alist[rightmark] = temp
      endif

   // swap the pivot with alist[rightmark]
   temp = alist[start]
   alist[start] = alist[rightmark]
   alist[rightmark] = temp
   return rightmark
endfunction

function quicksort(alist, start, end)
   if start < end
      // partition the list
      split = partition(alist, start, end)
      // sort both halves
      quicksort(alist, start, split-1)
      quicksort(alist, split+1, end)
   endif
   return alist
endfunction
```

Quick Sort Time Complexity and Space Complexity

- Again uses a divide and conquer approach, as seen for a binary search (doubling the number of items only adds one more iteration), the time complexity is 0(log n)
- However, each of the n items has to be compared against the current pivot value, meaning the time complexity has to be multiplied by a factor of n, so overall average case time complexity is 0(n log n)
- In the worst case, every data item would need to be involved in a swap or change of position for each iteration. The worst case time complexity is $0(n^2)$
- The space complexity is 0(log n)

Quick Sort vs Merge Sort Time Complexity

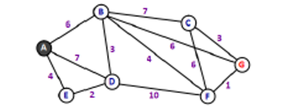
	Time Complexity			Space Complexity
Algorithm	Best Case	Average Case	Worst Case	Worst Case
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)

- The majority of sorts will be average case, so no real difference in time complexity
- Only in the worst case does a merge sort outperform a quick sort in terms of time complexity
- The merge sort has a much worse space complexity
- For very large data sets, this problem with space complexity that the merge sort has compared to the quick sort is a real problem
- Can result in more use of virtual memory, impacting time and performance as this secondary storage is required to be accessed more regularly
- For these reasons the quick sort is generally regarded to be the 'best'

Dijkstra's Shortest Path Algorithm

Dijkstra's Inefficiency

- Dijkstra's algorithm will potentially visit every node in order to find the shortest distance between two nodes
- Dijkstra's algorithm takes no account of the best general direction to head in. The only thing considered is the distance between nodes (no matter whether you are heading towards your destination, or away from it)

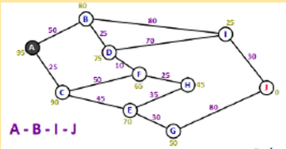


Node	Shortest Distance From A	Previous Node
✓✓A	—	—
✓✓B	6	A
✓✓C	13	B
✓✓D	~~7~~ 6	~~A~~ E
✓✓E	4	A
✓✓F	10	B
G	~~12~~ 11	~~B~~ F

A - B - F - G

A* Algorithm

- Similar to Dijkstra's algorithm, but uses two costs
- Dijkstra's algorithm has one cost for each path, the real cost (e.g. distance) from one node to another
- The A* algorithm uses this cost too, but also an approximate cost from each node to the goal. You could also think of it as a 'crow flies' value – the rough direct distance from each node to the destination



A - B - I - J

Node	Path (p)	Heuristic (h)	p + h	Previous Node
✓✓A	–	–	–	—
✓✓B	50	80	130	A
✓✓C	25	90	115	A
✓✓D	75	75	150	B
✓✓E	70	70	140	C
✓✓F	75	65	140	C
✓✓G	100	50	150	E
✓✓H	~~105~~ 100	45	~~150~~ 145	~~E~~ F
✓✓I	130	25	155	B
J	~~180~~ 160	0	~~180~~ 160	~~G~~ I

- Although it might sometimes be a good idea to travel away from your destination for a short distance (e.g. to get on the motorway), in general it is best to travel toward the destination
- The A* algorithm is likely to outperform Dijkstra's algorithm because it is likely to visit less nodes, find a more direct, optimum path more quickly, and consequently be more efficient

2.3 Algorithms — Sorting Algorithms

Bubble Sort

- The purpose of sorting algorithms is to order an unordered list. Item can be ordered alphabetically or by number
- Bubble sort steps through a list and compares pairs of adjacent numbers. The numbers are swapped if they are in the wrong order. for an ascending list, if the left number is bigger than the right number, the items are swapped, otherwise the numbers are not swapped
- The algorithm repeatedly passes through the list until no more swaps are needed
- The time complexity of the algorithm is 0(n^2)

Example: Sort the following sequence in ascending order using bubble sort: 5,3,4,1,2

Pass 1	5	3	4	1	2	
	3	5	4	1	2	Compare 5 and 3 – swap
	3	4	5	1	2	Compare 5 and 4 – swap
	3	4	1	5	2	Compare 5 and 1 – swap
	3	4	1	2	5	Compare 5 and 2 – swap; end of pass 1
Pass 2	3	4	1	2	5	Compare 3 and 4 – no swap
	3	1	4	2	5	Compare 4 and 1 – swap
	3	1	2	4	5	Compare 4 and 2 – swap
	3	1	2	4	5	Compare 4 and 5 – no swap; end of pass 2
Pass 3	1	3	2	4	5	Compare 3 and 1 – swap
	1	2	3	4	5	Compare 3 and 2 – swap
	1	2	3	4	5	Compare 3 and 4 – no swap
	1	2	3	4	5	Compare 4 and 5 – no swap; end of pass 3
	1	2	3	4	5	

Bubble sort pseudocode

```
A ← [5,3,4,1,2]
sorted ←  False
WHILE not sorted
 sorted ←  True
 FOR i TO LEN (A)-1:
  IF A[i] > A[i+1]:
   temp ← A[i]
   A[i] ← A[i+1]
   A[i+1] ← temp
   sorted ← False
  ENDIF
  ENDFOR
ENDWHILE
OUTPUT A
```

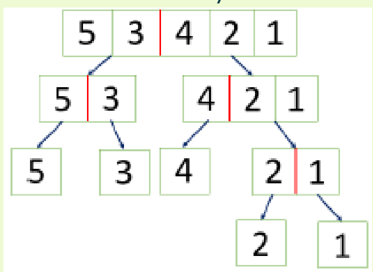
Merge Sort

Merge sort pseudocode

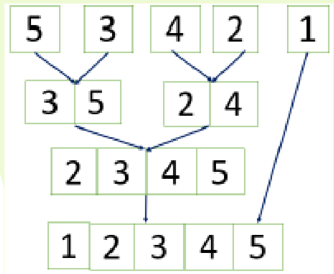
```
SUBROUTINE MergeSort(List, Start, End)
 IF Start < End THEN
  Mid ←  (Start + End) DIV 2
  List1 ← MergeSort (List, Start, Mid)
  List2 ←  MergeSort (List, Mid + 1, End)
  List3 ←  []
  WHILE LEN(List1_ > 0 AND LEN(List2) > 0
   IF List1[1] > List2[1] THEN
    APPEND List2[1] TO List3
    POP List2[1] FROM List2
   ELSE
    APPEND List1 [1] TO List3
    POP List1[1] FROM List1
   ENDIF
  ENDWHILE
  WHILE LEN(List1) > 0
   APPEND List1[1] TO List3
   POP List1[1] FROM List1
  ENDWHILE
  WHILE LEN(List2) > 0
   APPEND List2[1] TO List3
   POP List2[1] FROM List2
  ENDWHILE
  RETURN List3
 ELSE
  List4 ← []
  APPEND List[Start] To List4
  RETURN List4
ENDSUBROUTINE
```

- Merge sort is a type of divide and conquer algorithm
- There are two steps: divide and combine
- Merge sort works by dividing the unsorted list sublists. It keeps on doing this until there is 1 item in each list
- Pairs of sublists are combined into an ordered list containing all items in the two sublists. The algorithm keeps going until there is only 1 ordered list remaining
- Merge sort is a recursive function that calls itself
- The time complexity of merge sort is 0(n log n)

Step 1: Divide – Keep dividing until there is only 1 in each list



Step 2: Combine



1. the first items in the the two sublists are compared and the smallest value is copied to the parent list
2. The copied item is then removed from the sublist
3. When there are no items left in one of the sublists, the remaining items in the other sublists are then copied, in order to the parent list

```
Tracking the code
L=[5,3,4,1,2]
MergeSort(L,1,5)
```

Call	Start	End	Mid	List Returned
1	1	5	3	
2	1	3	2	
3	1	2	1	
4	1	1		[5]
3	1	2	1	
5	2	2		[3]
3	1	2	1	[3,5]
2	1	3	2	
6	3	3		[4]
2	1	3	2	[3,4,5]
1	1	5	3	
7	4	5	4	
8	4	4		[1]
7	4	5	4	
9	5	5		[2]
7	4	5	4	[1,2]
1	1	5	3	[1,2,3,4,5]

Merge Sort vs Bubble Sort

	Advantages	Disadvantages
Bubble Sort	• Very simple and robust algortihm	• Can be slow particularly for long lists. As the number of items increases, the time taken for the algorithm to run increases dramatically
Merge Sort	• Much faster then bubble sort, especially when the number of elements is large	• More complex to understand • Step 1: Divide • Step 2: Combine

Comparing Algorithms

- The time efficiency of algorithms refers how long an algorithm takes to run as a function of the size of the input
- More than one algorithm can be used to solve the same problem
- For instance, to calculate the sum of a sequence of numbers, we can use the following algorithm:

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

where *n* is the number we wish to sum the values up to. Using this calculation the time remains constant regardless the value of n. In other words, regardless of how many numbers we wish to add up, the time taken will always be the same.

We could use alternative algorithm to calculate the sum of a sequence of numbers.

```
sum ← 0
FOR i ← 1 to n
        sum ← sum + i
ENDFOR
OUTPUT sum
```

Using this algorithm , the number of operations increases in linear time with the size of the input. Therefore, the time taken for the algorithm to run will grow in linear time as in size of the input increases. Clearly this is more inefficient than the first algorithm even though it solves the same problem.

Another area where algorithms differ in their efficiency is in regard to the memory requirements of algorithms. For instance, programs that read in huge data files into memory can end up taking up large space in memory.

When developing algorithms, it is important to consider the hardware constraints of the system you are developing, e.g. mobile phone which has limited processing and space capability. If you have large memory, then your algorithm can afford to be less space efficient. Likewise, if you have access to tremendous processing power algorithm (e.g. supercomputer), you may not need to be time efficient, although it is still desirable to make algorithms as efficient as possible.

Maths for Bog O Notation

A function allows us to map a set of input values to a set of output values *y = f(x)*

where x is a value from the domain and y a value from the codomain

domain -> codomain

A linear function takes the form $y = mx + c$, where m is the gradient and c the intercept on the y axis.

A polynomial function takes the form $y = ax^2 + bx + c$

An exponential function takes the form $y = a^x$

A logarithm function takes the form $y = a\log_n x$

Permutations illustrate how the number of operations grows factorally when we add additional dimensions to some problems.

How many different combinations can sequence of digits have?

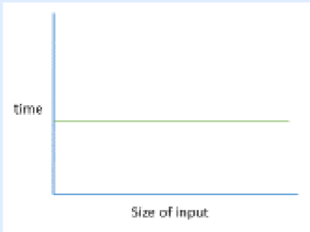
No. of digits	No of combinations
2	2
3	6
4	24
5	120

Big-O notation gives us an idea of how long a program will run if we increase the size of the input. We need to consider how many operations will need to be carried out for a given size of input. This gives is the time complexity of the algorithm.

Constant Time O(1)

The time remains constant even when the number of input increases. E.g. calculating the sum of a sequence of numbers.

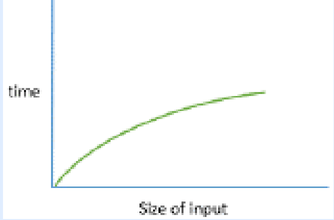
*sum = (n+1) * n/2*



Regardless of how many numbers we wish to add up, the time taken will always be the same.

Logarithmic Time O(log n)

The time taken for the algorithm toi sun will grow slowly as in size of the input increase

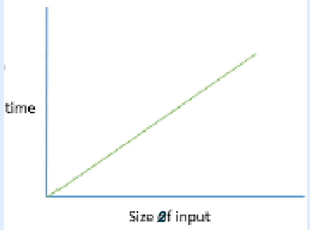


Linear Time O(n)

The time taken for the algorithm to run will grow linear time as in size of the input increases.

E.g. inefficient algorithm to calculate the sum of a sequence of numbers

```
sum = 0
for i=0 to n
      sum = sum + 1
output (sum)
```

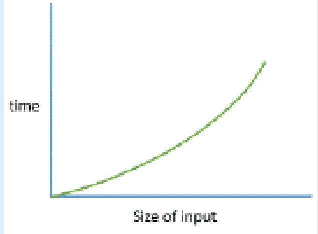


Polynomial Time O(n)

The time taken for the algorithm to run will grow proportionally to the square of the size of the data set.

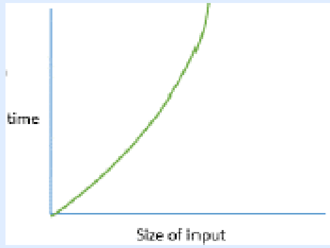
Normally when you have nested for loop, this will have a polynomial time complexity.

```
for i=0 to n
 for j=0 to n
   Do something
```



Exponential Time O(2)

The time taken for the algorithm will grow as the power of the number of inputs, so the time taken for the algorithm to run will grow very quickly as more input data are added.



The time taken for an algorithm to run will depend on the hardware (e.g. processor clock speed, RAM size), even though the number of operations will be constant for a fixed output

Tractable problems are problems that have a polynomial or less time solution e.g. O(1), O(n), O(log n), O(n^2)

Intractable problem are problems that can be theoretically solved but take longer than polynomial time e.g. O(n!), O(2^n)

Heuristic algorithms are used to provide approximate but not exact solutions to intractable problems.

2.3 Algorithms

The Travelling Salesman Problem

The idea is to find the shortest route to visit all cities. This is a permutation of the number of cities, so has a factorial time complexity, so quickly becomes an intractable problem with an unfeasibly huge number of permutations.

To solve this we use an heuristic algorithm. This provides and acceptable solution to the problem but it may not be the optimal or best solution. So for the travelling salesman problem, we may find a short route but not necessarily the shortest route. Heuristic algorithms for the travelling salesman problem include the following:

- Greedy algorithm: take the shortest route to the next city
- Visit the cities in a circle
- Brute force algorithm: apply to small but different subsets of cities. Apply the brute force algorithm to fewer, manageable problems rather than a single, intractable problem

Time complexity of common algorithms

Linear Search	0(n)
Binary Search	0(log n)
Binary Tree Search	0(log n)
Bubble Sort	$0(n^2)$
Merge Sort	0(n log n)
Travelling Salesman Problem	0(n!)
Brute force password cracker where n is the legnth of the password	$0(A^n)$

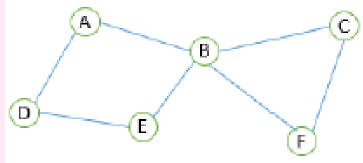
Unsolvable problems. Some problems cannot be solved by a computer. The Halting problem is one such problem and shows that some problems cannot be solved algorithmically.

The Halting problem states that there is no computer program that exists that can determine whether another computer program will halt or will continue to run forever, given some specific input.

Traversing Graphs

We can use depth first traversal or breadth first traversal to traverse a graph: *Graph used in example to follow:*

A	[B, B]
B	[A, E, C, F]
C	[B, F]
D	A, E]
E	[D, B]
F	[B, C]



Breadth First Traversal

Breadth first traversal starts at a node and explores all the neighbour nodes before moving into the next ;evel of nodes. A breadth first traversal uses an iterative approach. A typical application of a breadth first traversal is for determining the shortest path of an unweighted graph.

```
breadth_first_traversal (node)
 queue = []
 visited = []
 queue.append (node)
 visited.append (node)


 while queue is not empty
  node = queue.pop (0)
  print (node, end = " ")
  for i in graph [node] :
   if i not in visited
    queue.append(i)
    visited.append(i)


graph={'A':['D','B'],\
'B':['A','E','C','F'], 'C': ['B','F'],\
'D': ['A','E'],'E':['D','B'], 'F':['B', 'C']}

breadth first traversal ("A")
```

Node	i	output	visited	queue
A			[A]	[A]
		A		[]
	D		[A,D]	[D]
	B		[A,D,B]	[D,B]
D		D		[B]
	A			
	E		[A,D,B,E]	[B,E]
B		B		[E]
	A			
	C			
	C		[A,D,B,E]	[E,C]
	F		[A,D,B,E,C,F]	[E,C,F]
		E		[C,F]
		C		[F]
		F		[]

Depth First Traversal

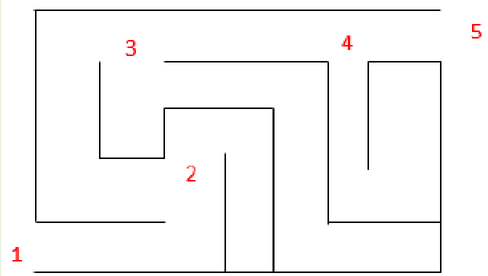
Depth first traversal starts at a node and traverses along each path as far as it goes before backtracking to the next branch. Depth first traversal uses recursion. An application of a depth first traversal is for navigating a maze.

```
# Uses recursive calls
depth_first_traversal (node)
 visited.append (node)
 for i in graph [node]:
  if i not in visited
   depth_first_traversal (i)

# Graph represented as an adjacency
list
graph={"A":["D","B"], "B":["A","E","C",
"F"],\
"C": ["B","F"], "D": ["A","E"],\
"E":["D","B"],"F":["B","C"]}
```

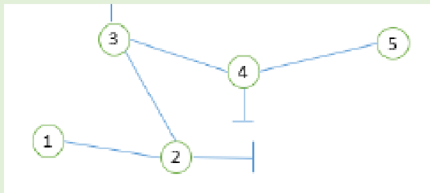
Call	Node	i	visited
			[]
1	A		[A]
2	D	D	[A,D]
		A	
3	E	E	[A,D,E]
		D	
4	B	B	[A,D,E,B]
		A	
		E	
5	C	C	[A,D,E,B,C]
		B	
6	F	F	[A,D,E,B,C,F]

Navigating a maze with depth first traversal

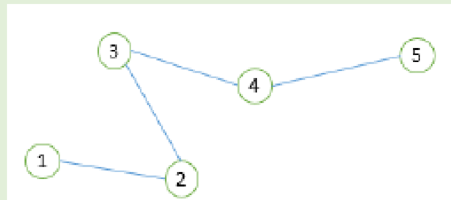
Nodes are placed at the start and end points as well as at locations where there are alternative paths



Graph representation of maze with dead ends



Graph representation of maze without dead ends



Computer Science

2.3 Algorithms

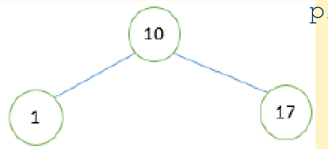
Tree Traversal

There are three ways of traversing a binary tree:

- Pre-order tree traversal
- Post-order tree traversal
- In-order tree traversal

When traversing a tree we start at the root node. We can then visit the node (that is, obtain the value of the node), traverse left or traverse right.

The order in which we visit, traverse left or traverse right depends on the traversal method that we use.

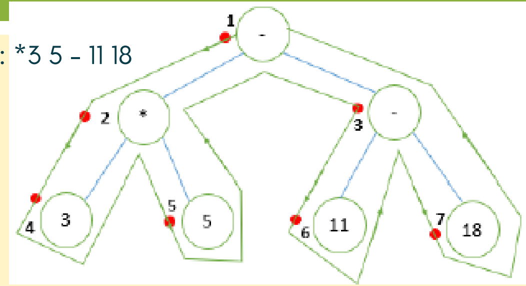


	Pre-order traversal	Post-order traversal	In-order traversal
Order	1. visit node 2. left traversal 3. right traversal	1. left traversal 2. right traversal 3. visit node	1. left traversal 2. visit node 3. right traversal
Example	10, 1, 17	1, 17, 10	1, 10, 17
Example application	Prefix Notation, Copying a tree	Reverse Polish Notation	Ordering a sequence of numbers, binary tree search

Pre-order traversal

```
pre_order_traversal (node):
    print(values[node])
    if tree_left[node] != -1:
      pre_order_traversal(tree_left[node])
    if tree_right[node] != -1:
     pre_order_traversal(tree_right[node])
values=["+","-","*",2,4,6,7]
tree_left=[2,4,6,-1,-1,-1,-1]
tree_right=[3,5,7,-1,-1,-1,-1]
pre_order_traversal(1)
```

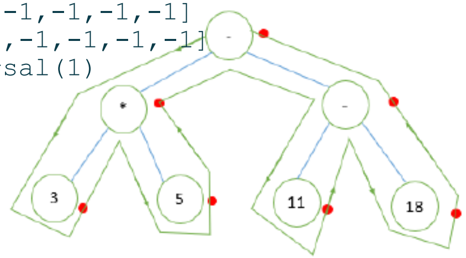
Sequence output: *3 5 – 11 18



Node	Value [node]	Tree_right [node]	Tree_left [node]	Output
1	+	3	2	+
2	-	5	4	-
4	2	-1	-1	2
2	-	5	4	
5	4	-1	-1	4
1	+	3	2	
3	*	7	6	*
6	6	-1	-1	6
7	7	-1	-1	7

Post-order traversal

```
post_order_traversal (node):
    if tree_left[node] != -1:
      post_order_traversal(tree_left[node])
    if tree_right[node] != -1:
     post_order_traversal(tree_right[node])
    print(values[node])
values=["+","-","*",2,4,6,7]
tree_left=[2,4,6,-1,-1,-1,-1]
tree_right=[3,5,7,-1,-1,-1,-1]
post_order_traversal(1)
```

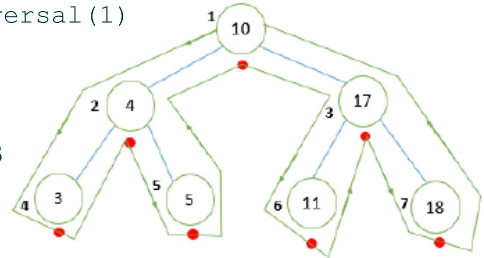


Call	Node	Value [node]	Tree_right [node]	Tree_left [node]	Output
1	1	+	3	2	
2	2	-	5	4	
3	4	2	-1	-1	2
2	2	-	5	4	
4	5	4	-1	-1	4
2	2	+	5	4	-
5	3	*	7	6	
6	6	6	-1	-1	6
5	3	*	7	6	
7	7	7	-1	-1	7
5	3	*	7	6	*
1	1	+	3	2	+

In-order traversal

```
in_order_traversal (node):
    if tree_left[node] != -1:
     in_order_traversal(tree_left[node])
    print(values[node])
    if tree_right[node] != -1:
     in_order_traversal(tree_right[node])
# node_index[1,2,3,4,5,6,7]
values=[10,4,17,3,5,11,18]
tree_left=[2,4,6,-1,-1,-1,-1]
tree_right=[3,5,7,-1,-1,-1,-1]
in_order_traversal(1)
```

Sequence output:
3,4,5,10,11,17,18



Node	Value [node]	Tree_left [node]	Tree_right [node]	Output
1	10	2	3	
2	4	4	5	
4	3	-1	-1	3
2	4			4
5	5	-1	-1	5
1	10			10
3	17	6	7	
6	11	-1	-1	11
3	17			17
7	18	-1	-1	18

Computer Science

2.3 Algorithms   Reverse Polish Notation

Infix Notation

We are all familiar with infix notation where the operators appear between the operands (i.e. the numbers) that you want to apply the operator to.

Reverse Polish Notation (Postfix)

RPN uses postfix notation where the operators follow the operand. Using infix notation to add two numbers we get:

`<operand> <operator> <operand> 3 + 4`

In RPN (postfix notation) this becomes:

`<operand> <operand> <operator> 3 4 +.`

Many interpreters and compliers automatically convert between infix notation to postfix notation, so there is no requirement to write code using the less familiar postfix notation.

Advantages of Postfix

- Simpler for computer to evaluate
- Do not need brackets
- Operators appear in correct order of precedence of operators, so there are fewer operations

RPN Algorithm

1. Go through each character in the postfix expression from left to right
2. If character is a number, then push number onto the stack
3. Otherwise, if the character is an operator (+,-,/,X), then pop the top 2 numbers from the stack
4. Evaluate the 2 numbers using the operator
5. Push result back onto the stack

Worked example: Solve the following expression: 53 1 + – 6 x

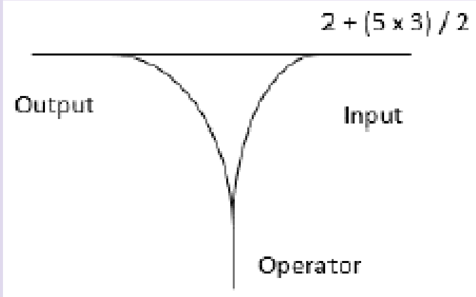
Stack at each step: *Answer is 6. Infix expression (5-(1+3))x6*

1	2	3	4	5	8	9
5	3	1	4	1	6	6
	5	3	5		1	
		5				
~~5~~31+–6x	531+–6x	~~531~~+–6x	531+–6x	531+–6x	~~531+–6~~x	~~531+–6x~~
Push 5 onto stack	Push 4 onto stack	Push 1 onto stack	Pop 1,3 Evaluate 1+3=4 Push result on stack	Pop 4,5 Evaluate 5-4=1 Push result on stack	Push 6 onto stack	Pop 6,1 Evaluate 6x1=6 Push result on stack

Convert from infix to Postfix notation

Step 1	Add brackets	(3 + ((5 x 3) / (7 – 4)))
Step 2	Write out the operands with spaces	3 5 3 7 4
Step 3	Starting with the inner most brackets, move the operator to after the operands from between the operands	3 5 3x7 4– 3+(15/3) 3 5 3x7 4/ 3+5 3 5 3x7 4/+ 8

Alternative Shunting Yard Algorithm to convert from infix to postfix notation



Worked example: Convert the following expression to RPN: 2 + (5x3)/2

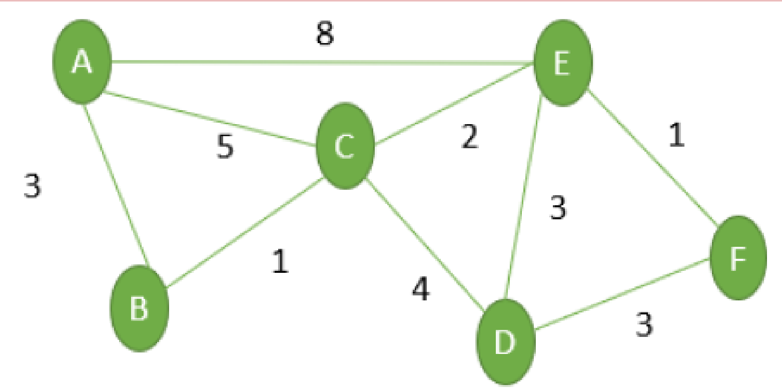
Symbol	Action	Output queue	Operator stack
2	Push operand onto output queue	2	
+	Push operator onto operator stack	2	+
5	Push operand onto output queue	2 5	+
x	Push operand onto operator stack, x has higher precedence than +	2 5	x+
3	Push operand onto output queue	2 5 3	x+
/	Pop stack to output, x has same precedence as /. Push on operator stack, / has higher precedence than +	2 5 3 x 2 5 3 x	+ /+
2	Pop operand onto output queue	2 5 3 x 2	/+
	Pop whole stack onto output queue	2 5 3 x 2/+	

Computer Science

Dijkstra's shortest path algorithm

- The purpose of Dijkstra's algorithm finds the shortest path between nodes / verticies in a weighted graph
- Selects the unvisited node with the shortest path
- Calculates the distance to each unvisited neighbour
- Updates the distance of each unvisited neighbour if smaller
- Once all neighbours have been visited, mark nodes as visited

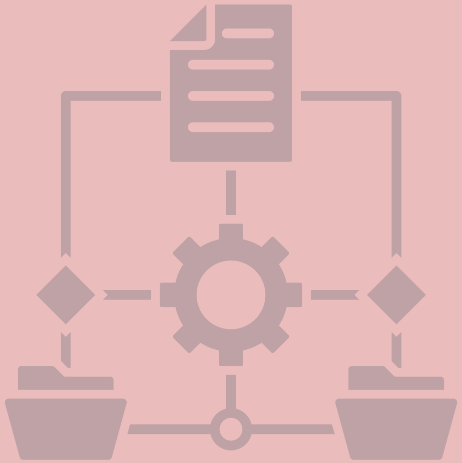
Example Graph



Start at node A because it is the unvisited node with the shortest distance to node A. The distance to each unvisited neighbour is 3 and 5 for B and C respectively. B has the shortest distance to node A so this is the next unvisited node we select. At B, there is only 1 neighbour (C). The distance is updated because the route A-B-C (4) has less cost than the route A-C(5). E is the next unvisited node with the shortest distance and is has neighbours D and F. F has the less cost out of the two and is then selected as the next unvisited node. The shortest route is A-C-E-F.

Dijkstra Pseudocode

```
Q ← []
distance ← []
previous node ← []
FOR i← 1 TO NUMBER_OF_VERTICIES
 Append i to Q
 Append 100 to distance
 Append -1 to previous_node
ENDFOR
distance[1] ← 0
WHILE LEN(Q) != 0
 u ← Q[1]
 Pop u from Q
 FOR v  in Q
  IF matrix[u][v] > 0:
   a=distance[u] + matrix[u][v]
   IF a<distance[v]
    distance[0]=a
    previous_node[v]=u
   ENDIF
  ENDIF
 ENDFOR
ENDWHILE
```



Trace table given then following matrix

u/v	1	2	3	4
1	0	2	5	3
2	0	0	1	0
3	0	0	0	0
4	0	0	0	0

q	u	v	a	Distance				Previous_node			
1,2,3,4				100	100	100	100	-1	-1	-1	-1
				0							
2,3,4	1	2	2		2				1		
		3	5			5				1	
		4	3				3				1
3,4	2	3	3			3				2	
4	3										
-	4										

Computer Science

Beginner's Python Cheat Sheet

Variables and Strings

Variables are used to store values. A string is a series of characters, surrounded by single or double quotes.

Hello world

```
print("Hello world!")
```

Hello world with a variable

```
msg = "Hello world!"
print(msg)
```

Concatenation (combining strings)

```
first_name = 'albert'
last_name = 'einstein'
full_name = first_name + ' ' + last_name
print(full_name)
```

Lists

A list stores a series of items in a particular order. You access items using an index, or within a loop.

Make a list

```
bikes = ['trek', 'redline', 'giant']
```

Get the first item in a list

```
first_bike = bikes[0]
```

Get the last item in a list

```
last_bike = bikes[-1]
```

Looping through a list

```
for bike in bikes:
    print(bike)
```

Adding items to a list

```
bikes = []
bikes.append('trek')
bikes.append('redline')
bikes.append('giant')
```

Making numerical lists

```
squares = []
for x in range(1, 11):
    squares.append(x**2)
```

Lists (cont.)

List comprehensions

```
squares = [x**2 for x in range(1, 11)]
```

Slicing a list

```
finishers = ['sam', 'bob', 'ada', 'bea']
first_two = finishers[:2]
```

Copying a list

```
copy_of_bikes = bikes[:]
```

Tuples

Tuples are similar to lists, but the items in a tuple can't be modified.

Making a tuple

```
dimensions = (1920, 1080)
```

If statements

If statements are used to test for particular conditions and respond appropriately.

Conditional tests

```
equals                x == 42
not equal             x != 42
greater than          x > 42
  or equal to         x >= 42
less than             x < 42
  or equal to         x <= 42
```

Conditional test with lists

```
'trek' in bikes
'surly' not in bikes
```

Assigning boolean values

```
game_active = True
can_edit = False
```

A simple if test

```
if age >= 18:
    print("You can vote!")
```

If-elif-else statements

```
if age < 4:
    ticket_price = 0
elif age < 18:
    ticket_price = 10
else:
    ticket_price = 15
```

Dictionaries

Dictionaries store connections between pieces of information. Each item in a dictionary is a key-value pair.

A simple dictionary

```
alien = {'color': 'green', 'points': 5}
```

Accessing a value

```
print("The alien's color is " + alien['color'])
```

Adding a new key-value pair

```
alien['x_position'] = 0
```

Looping through all key-value pairs

```
fav_numbers = {'eric': 17, 'ever': 4}
for name, number in fav_numbers.items():
    print(name + ' loves ' + str(number))
```

Looping through all keys

```
fav_numbers = {'eric': 17, 'ever': 4}
for name in fav_numbers.keys():
    print(name + ' loves a number')
```

Looping through all the values

```
fav_numbers = {'eric': 17, 'ever': 4}
for number in fav_numbers.values():
    print(str(number) + ' is a favorite')
```

User input

Your programs can prompt the user for input. All input is stored as a string.

Prompting for a value

```
name = input("What's your name? ")
print("Hello, " + name + "!")
```

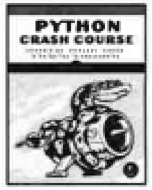
Prompting for numerical input

```
age = input("How old are you? ")
age = int(age)

pi = input("What's the value of pi? ")
pi = float(pi)
```

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



Haggerston School

SIXTH FORM KNOWLEDGE ORGANISER

Aspiration Creativity Character

Computer Science

Beginner's Python Cheat Sheet - Lists

What are lists?

A list stores a series of items in a particular order. Lists allow you to store sets of information in one place, whether you have just a few items or millions of items. Lists are one of Python's most powerful features readily accessible to new programmers, and they tie together many important concepts in programming.

Defining a list

Use square brackets to define a list, and use commas to separate individual items in the list. Use plural names for lists, to make your code easier to read.

Making a list

```
users = ['val', 'bob', 'mia', 'ron', 'ned']
```

Accessing elements

Individual elements in a list are accessed according to their position, called the index. The index of the first element is 0, the index of the second element is 1, and so forth. Negative indices refer to items at the end of the list. To get a particular element, write the name of the list and then the index of the element in square brackets.

Getting the first element

```
first_user = users[0]
```

Getting the second element

```
second_user = users[1]
```

Getting the last element

```
newest_user = users[-1]
```

Modifying individual items

Once you've defined a list, you can change individual elements in the list. You do this by referring to the index of the item you want to modify.

Changing an element

```
users[0] = 'valerie'
users[-2] = 'ronald'
```

Adding elements

You can add elements to the end of a list, or you can insert them wherever you like in a list.

Adding an element to the end of the list

```
users.append('amy')
```

Starting with an empty list

```
users = []
users.append('val')
users.append('bob')
users.append('mia')
```

Inserting elements at a particular position

```
users.insert(0, 'joe')
users.insert(3, 'bea')
```

Removing elements

You can remove elements by their position in a list, or by the value of the item. If you remove an item by its value, Python removes only the first item that has that value.

Deleting an element by its position

```
del users[-1]
```

Removing an item by its value

```
users.remove('mia')
```

Popping elements

If you want to work with an element that you're removing from the list, you can "pop" the element. If you think of the list as a stack of items, pop() takes an item off the top of the stack. By default pop() returns the last element in the list, but you can also pop elements from any position in the list.

Pop the last item from a list

```
most_recent_user = users.pop()
print(most_recent_user)
```

Pop the first item in a list

```
first_user = users.pop(0)
print(first_user)
```

List length

The len() function returns the number of items in a list.

Find the length of a list

```
num_users = len(users)
print("We have " + str(num_users) + " users.")
```

Sorting a list

The sort() method changes the order of a list permanently. The sorted() function returns a copy of the list, leaving the original list unchanged. You can sort the items in a list in alphabetical order, or reverse alphabetical order. You can also reverse the original order of the list. Keep in mind that lowercase and uppercase letters may affect the sort order.

Sorting a list permanently

```
users.sort()
```

Sorting a list permanently in reverse alphabetical order

```
users.sort(reverse=True)
```

Sorting a list temporarily

```
print(sorted(users))
print(sorted(users, reverse=True))
```

Reversing the order of a list

```
users.reverse()
```

Looping through a list

Lists can contain millions of items, so Python provides an efficient way to loop through all the items in a list. When you set up a loop, Python pulls each item from the list one at a time and stores it in a temporary variable, which you provide a name for. This name should be the singular version of the list name.

The indented block of code makes up the body of the loop, where you can work with each individual item. Any lines that are not indented run after the loop is completed.

Printing all items in a list

```
for user in users:
    print(user)
```

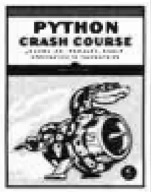
Printing a message for each item, and a separate message afterwards

```
for user in users:
    print("Welcome, " + user + "!")

print("Welcome, we're glad to see you all!")
```

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



Beginner's Python Cheat Sheet — If Statements and While Loops

What are if statements? What are while loops?

If statements allow you to examine the current state of a program and respond appropriately to that state. You can write a simple if statement that checks one condition, or you can create a complex series of if statements that idenitfy the exact conditions you're looking for.

While loops run as long as certain conditions remain true. You can use while loops to let your programs run as long as your users want them to.

Conditional Tests

A conditional test is an expression that can be evaluated as True or False. Python uses the values True and False to decide whether the code in an if statement should be executed.

Checking for equality

A single equal sign assigns a value to a variable. A double equal sign (==) checks whether two values are equal.

```
>>> car = 'bmw'
>>> car == 'bmw'
True
>>> car = 'audi'
>>> car == 'bmw'
False
```

Ignoring case when making a comparison

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
```

Checking for inequality

```
>>> topping = 'mushrooms'
>>> topping != 'anchovies'
True
```

Numerical comparisons

Testing numerical values is similar to testing string values.

Testing equality and inequality

```
>>> age = 18
>>> age == 18
True
>>> age != 18
False
```

Comparison operators

```
>>> age = 19
>>> age < 21
True
>>> age <= 21
True
>>> age > 21
False
>>> age >= 21
False
```

Checking multiple conditions

You can check multiple conditions at the same time. The and operator returns True if all the conditions listed are True. The or operator returns True if any condition is True.

Using and to check multiple conditions

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 and age_1 >= 21
False
>>> age_1 = 23
>>> age_0 >= 21 and age_1 >= 21
True
```

Using or to check multiple conditions

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 or age_1 >= 21
True
>>> age_0 = 18
>>> age_0 >= 21 or age_1 >= 21
False
```

Boolean values

A boolean value is either True or False. Variables with boolean values are often used to keep track of certain conditions within a program.

Simple boolean values

```
game_active = True
can_edit = False
```

If statements

Several kinds of if statements exist. Your choice of which to use depends on the number of conditions you need to test. You can have as many elif blocks as you need, and the else block is always optional.

Simple if statement

```
age = 19

if age >= 18:
    print("You're old enough to vote!")
```

If-else statements

```
age = 17

if age >= 18:
    print("You're old enough to vote!")
else:
    print("You can't vote yet.")
```

The if-elif-else chain

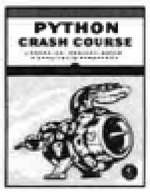
```
age = 12

if age < 4:
   price = 0
elif age < 18:
    price = 5
else:
    price = 10

print("Your cost is $" + str(price) + ".")
```

Conditional tests with lists

You can easily test whether a certain value is in a list. You can also test whether a list is empty before trying to loop through the list.

Testing if a value is in a list

```
>>> players = ['al', 'bea', 'cyn', 'dale']
>>> 'al' in players
True
>>> 'eric' in players
False
```

Computer Science

Beginner's Python Cheat Sheet — Functions

What are functions?

Functions are named blocks of code designed to do one specific job. Functions allow you to write code once that can then be run whenever you need to accomplish the same task. Functions can take in the information they need, and return the information they generate. Using functions effectively makes your programs easier to write, read, test, and fix.

Defining a function

The first line of a function is its definition, marked by the keyword def. The name of the function is followed by a set of parentheses and a colon. A docstring, in triple quotes, describes what the function does. The body of a function is indented one level.

To call a function, give the name of the function followed by a set of parentheses.

Making a function

```
def greet_user():
    """Display a simple greeting."""
    print("Hello!")

greet_user()
```

Passing information to a function

Information that's passed to a function is called an argument; information that's received by a function is called a parameter. Arguments are included in parentheses after the function's name, and parameters are listed in parentheses in the function's definition.

Passing a single argument

```
def greet_user(username):
    """Display a simple greeting."""
    print("Hello, " + username + "!")

greet_user('jesse')
greet_user('diana')
greet_user('brandon')
```

Positional and keyword arguments

The two main kinds of arguments are positional and keyword arguments. When you use positional arguments Python matches the first argument in the function call with the first parameter in the function definition, and so forth.

With keyword arguments, you specify which parameter each argument should be assigned to in the function call. When you use keyword arguments, the order of the arguments doesn't matter.

Using positional arguments

```
def describe_pet(animal, name):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet('hamster', 'harry')
describe_pet('dog', 'willie')
```

Using keyword arguments

```
def describe_pet(animal, name):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet(animal='hamster', name='harry')
describe_pet(name='willie', animal='dog')
```

Default values

You can provide a default value for a parameter. When function calls omit this argument the default value will be used. Parameters with default values must be listed after parameters without default values in the function's definition so positional arguments can still work correctly.

Using a default value

```
def describe_pet(name, animal='dog'):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet('harry', 'hamster')
describe_pet('willie')
```

Using None to make an argument optional

```
def describe_pet(animal, name=None):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    if name:
        print("Its name is " + name + ".")

describe_pet('hamster', 'harry')
describe_pet('snake')
```

Return values

A function can return a value or a set of values. When a function returns a value, the calling line must provide a variable in which to store the return value. A function stops running when it reaches a return statement.

Returning a single value

```
def get_full_name(first, last):
    """Return a neatly formatted full name."""
    full_name = first + ' ' + last
    return full_name.title()

musician = get_full_name('jimi', 'hendrix')
print(musician)
```

Returning a dictionary

```
def build_person(first, last):
    """Return a dictionary of information
    about a person.
    """
    person = {'first': first, 'last': last}
    return person

musician = build_person('jimi', 'hendrix')
print(musician)
```

Returning a dictionary with optional values

```
def build_person(first, last, age=None):
    """Return a dictionary of information
    about a person.
    """
    person = {'first': first, 'last': last}
    if age:
        person['age'] = age
    return person

musician = build_person('jimi', 'hendrix', 27)
print(musician)

musician = build_person('janis', 'joplin')
print(musician)
```

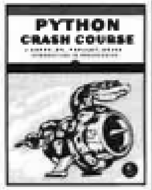
Visualizing functions

Try running some of these examples on pythontutor.com.

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



SPaG

Grammar: Write in Sentences

A sentence is a group of words that make sense. Sentences start with a capital letter and end with a full stop, question mark or exclamation mark. All sentences contain clauses. You should try to use a range of sentences when writing. There are three main types of sentences.

Simple sentence: A sentence containing one main clause with a **subject** and a **verb**.

He reads.

Literacy is important.

Compound sentence: Two simple sentences joined with a conjunction. Both of these simple sentences would make sense on their own. Varying conjunctions makes your writing more interesting.

He read his book because **it was written** by his favourite author.

Literacy is important so **students had** an assembly about reading.

Complex sentence: A longer sentence containing a main clause and one or more subordinate clause(s) used to add more detail. The main clause makes sense on its own. However, a subordinate clause would not make sense on its own, it needs the main clause to make sense. The subordinate clause is separated by a comma (s) and/or conjunction. The clause can go at the beginning, middle or end of the sentence.

He read his book even though it was late.

Even though it was late, he read his book.

He read his book, even though it was late, because it was written by his favourite author.

How can you develop your sentences?

1. Start sentences in different ways. For example, you can start sentences with adjectives, adverbs or verbs.

Adjective: Funny books are my favourite!

Adverb: Regularly reading helps me develop a reading habit.

Verb: Looking at the front cover is a good way to choose a reading book.

2. Use a range of **punctuation**.

3. **Nominalisation**

Nominalisation is the noun form of verbs; verbs become concepts rather than actions. Nominalisation is often used in academic writing. For example:

It is important to read because it helps you in lots of ways.

Becomes: Reading is beneficial in many ways.

Germany invaded Poland in 1939. This was the immediate cause of the Second World War breaking out. Becomes: Germany's invasion of Poland in 1939 was the immediate cause of the outbreak of the Second World War.

Connectives and Conjunctions	
Cause And Effect	Because So Consequently Therefore Thus
Addition	And Also In addition Further (more)
Comparing	Whereas However Similarly Yet As with/ equally/Likewise
Sequencing	Firstly Initially Then Subsequently Finally After
Emphasis	Importantly Significantly In particular Indeed
Subordinate	Who, despite, until, if, while, as, although, even though, that, which

SPaG: Spelling and Punctuation

Punctuation

Use a range of punctuation accurately when you are writing.

. Full stop Marks the end of a sentence.

, Comma Separates the items on a list or the clauses in a sentence.

' Apostrophe Shows possession (belonging) or omission (letters tak en away).

" " Quotation marks Indicate a quotation or speech.

' ' Inverted commas Indicate a title.

? Question mark Used at the end of a sentence that asks a question.

! Exclamation mark Used at the end of a sentence to show surprise or shock.

: Colon Used to introduce a list or an explanation/ elaboration/ answer to what preceded. A capital letter is only needed after a colon if you are writing a proper noun (name of person or place) or two or more sentences.

; Semi-colon Joins two closely related clauses that could stand alone as sentences. Also used to separate items on a complicated list. A capital letter is not needed after a semi-colon unless you are writing a proper noun (name of person or place).

Brackets Used to add extra information which is not essential in the sentence.

Spelling

Use the following strategies to help you spell tricky words.

1. Break it into sounds (d-i-a-r-y)
2. Break it into syllables (re-mem-ber)
3. Break it into affixes (dis + satisfy)
4. Use a mnemonic (necessary – one collar, two sleeves)
5. Refer to word in the same family (muscle – muscular)
6. Say it as it sounds – spell speak (Wed-nes day)
7. Words within words (Parliament – I AM parliament)
8. Refer to etymology (bi + cycle = two + wheels)
9. Use analogy (bright, light, night, etc)
10. Use a key word to remember a spelling rule (horrible/drinkable for -ible & -able / advice/advise for -ice & -ise)
11. Apply spelling rules (writing, written)
12. Learn by sight (look-cover-say-write check)