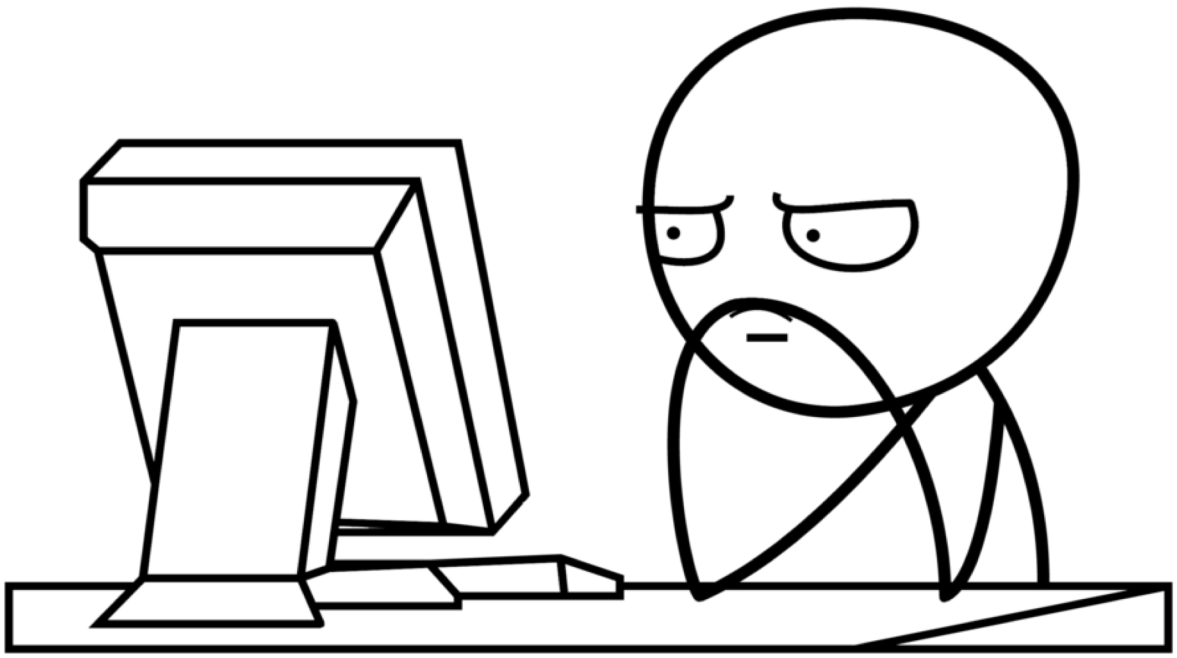
Computing @ turton

GCSE COMPUTER SCIENCE

KNOWLEDGE ORGANISER PACK

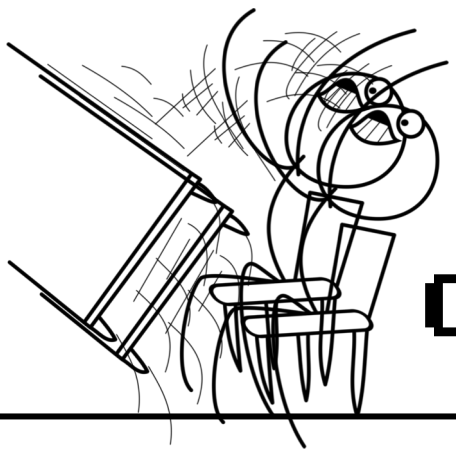
version 1.0



Computing @ turton

Knowledge organiser matrix

Knowledge Organiser	Knowledge Organiser Number	Paper 1 *Computational Thinking and Problem Solving*	Paper 2 *Computing Theory*
Keywords, concepts and flowcharts	1	✓	
Programming Theory 1	2	✓	
Programming Theory 2	3	✓	
Programming Theory 3	4	✓	
Programming Theory 4	5	✓	
Pseudocode	6	✓	
Pseudocode continued…	7	✓	
Language and translators	8	✓	
Searching	9	✓	
Sorting	10	✓	
Boolean logic	11	✓	✓
Data representation	12	✓	✓
Representing images and sound	13	✓	✓
Data compression	14	✓	✓
Hardware and software	15	✓	✓
Systems architecture	16	✓	✓
Storage	17	✓	✓
Networks	18		✓
Cyber security	19		✓
Ethical, legal and environmental issues	20		✓
Software development	21	✓	✓

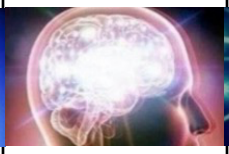


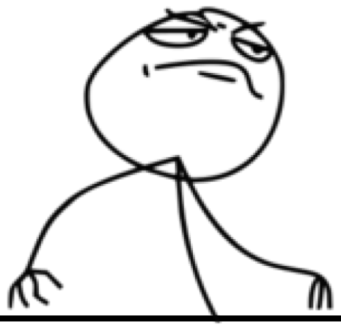
GCSE Computer Science

Computing @ turton

My Knowledge Checklist

Tick (✓) as you go along throughout the year…

Knowledge Organiser Reference	*"Know a **little**"* 	*"Know a **lot**"* 	*"Know it **all**"* 
0. Keywords, Concepts and Flowcharts			
1. Programming Theory 1			
2. Programming Theory 2			
3. Programming Theory 3			
4. Pseudocode 1			
5. Pseudocode 2			
6. Data Structures and String Handling			
7. Types of language			
8. Searching			
9. Sorting			
10. Boolean logic			
11. Data representation			
12. Representing images and sound			
13. Data compression			
14. Hardware and software			
15. Systems architecture			
16. Storage			
17. Networks			
18. Cyber security			
19. Ethical, legal and environmental issues			
20. Software development			
	*Know **everything!***		



GCSE Computer Science

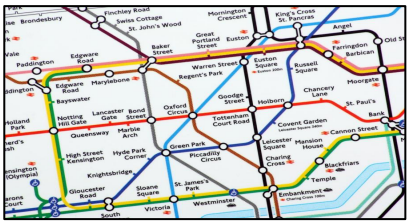
Keywords and concepts

Computing @ Turton

Algorithm - An algorithm is a sequence of steps that can be followed to complete a task. *Be aware that a computer program is an implementation of an algorithm and that an algorithm is not a computer program.*

Decomposition - Decomposition means breaking a problem into a number of sub-problems, so that each sub- problem accomplishes an identifiable task, which might itself be further subdivided.

Abstraction - The process of removing unnecessary detail from a problem. E.g. The London tube map is a form of abstraction. The map tells you what line each station is on and which other lines are connected. Very useful for a person travelling. Not useful to an engineer who is planning where to dig tunnels for a new line.



3 Programming Constructs

1 Sequence

In a sequence structure, an action or event leads to the next in a predetermined order.

```
qty = input()
total = qty * price
print(total)
```

2 Selection

A question is asked, depending on the answer the program takes one, two or more courses of action.

```
x = input()
if x > 5 then
    print("too big")
else
    print("just right!")
endif
```

3 Iteration

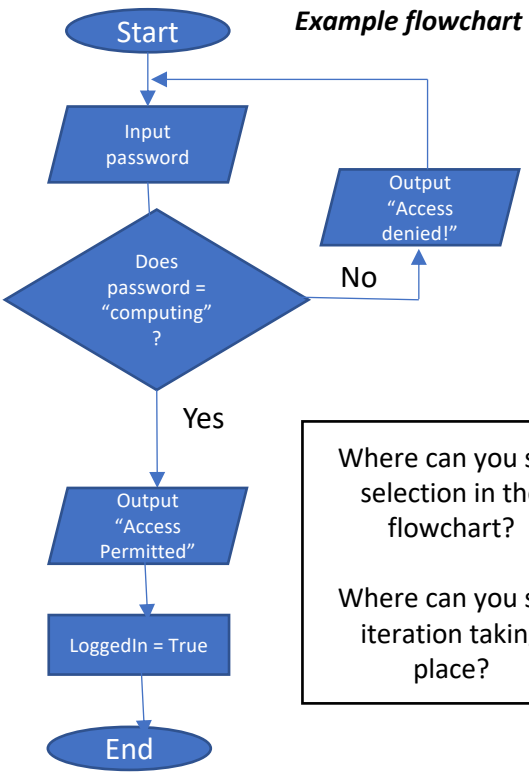
A process wherein a set of instructions or structures are repeated in a sequence a set number of times or until a condition is met.

```
for count = 1 to 10
    print("ROVERS!")
next count
```

Flowcharts

Symbol	Name
	Start/end
	Arrows
	Input/Output
	Process
	Decision

Example flowchart



Where can you see selection in the flowchart?

Where can you see iteration taking place?

Remember, more than one algorithm can be used to solve a problem!

Programming Theory 1

Variables and constants

Computing @Turton

Variable – Sometimes we need computers to remember the information we give it. A variable can be thought of as a box (memory location) that the computer can use to store a value. The value held in the box may change or vary. A program can use as many variables as it needs.

A variable is made up of three parts:

- A name (identifier)
- A type **(data type – see below)**
- A value (what you are storing)

name = "Mr rifai"
*The variable is called **name**, its data type is a **string,** and its value is **Mr Rifai***

Assignment - In order to change the data value stored in a variable, you use an operation called assignment. Different values may be assigned to a variable at different times during the execution of a program.

```
x = 5    #here we are assigning 5 to the variable x
name = input()   #here whatever the user types
                 in will be assigned to the
                 variable name.
```

Data types	
String	Combination of characters that appear on the keyboard (alphanumeric)
Integer	A whole number
Real	A decimal/fractional number
Boolean	True/False or Yes/No
Character/Char	Used for single letters

Example:

String	Float or Real	Integer	Boolean
Title	Rating	TimesViewed	Favourite
Zombie Attack	9.5	83	True
True Love	8.0	5	True
Mission: Pluto	2.5	1	False

Scope – The scope of a variable can be **local** or **global**.

- ***local*** variables only work in the procedure or loop they are created in.
- ***global*** variables can be accessed from any point in a program.

Declaration – Declaring a name for a variable is saying what the data type will be and where it will be stored in memory.
E.g. Dim name as String

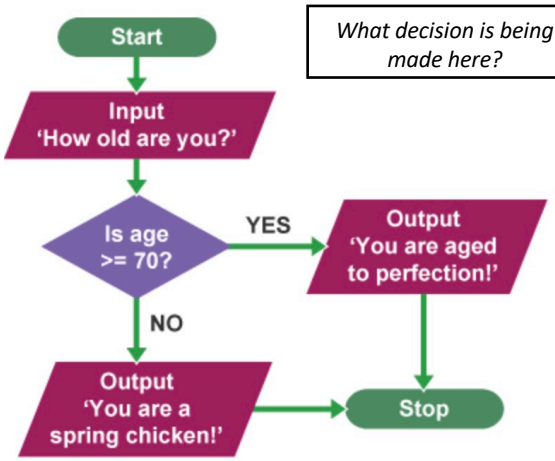
Constant – Similar to a variable, it is still a named memory location in the program **BUT** the value cannot be changed while the program is running E.g. If we wanted to store the VAT for a shop program we would set it as a constant at the start: ***VAT = 0.2*** *or in VB* ***Const VAT As Real = 0.2***

Selection

At some point, a program will have to ask a question because it has reached a step where one or more options are available. Depending on the answer given, the program will follow a certain step and ignore the others. E.g. If you have a queue jumping ticket go to the front else queue up!

These decisions lead to different paths through the program. Without selection it would not be possible to include different paths in programs. ***Think of the decisions involved in any game you have played….***

Selection is implemented using **IF** statements.



What decision is being made here?

Remember if it's in speech marks, it's a STRING!

Programming theory 2
Selection continued...

For **selection** in programming you can use ***if …else***

```
age = int(input("How old are you?"))
if age >= 70:
  print("You are aged to perfection!")
else:
  print("You are a spring chicken!")
End if
```

You can use ***else if*** to provide more choices.

```
age = int(input("How old are you?"))
if age >= 70 then:
  print("You are aged to perfection!")
elseif age == 50 then:
  print("Wow, you are half a century old!")
else:
  print("You are a spring chicken!")
```

Iteration

The third programming construct is ***iteration***. Means repetition, so ***iterative*** statements always involve performing a loop in the program to ***repeat*** a number of statements.

There are 2 types of iteration:

1. **Indefinite** – iteration continues until some specified condition is met.
   e.g. WHILE…END WHILE and REPEAT…UNTIL
2. **Definite** – Iteration is carried out a set number of times and is decided in advance.
   e.g. FOR….NEXT loops in programming.

Indefinite = Condition-controlled loop
Definite = Counter-controlled loop

WHILE …END WHILE loop
The condition is tested **before each iteration.**
And the statements in the loop will be **executed** if the **condition** is **true.**
The statements in the loop may not be executed (if the condition is initially false)

```
num = input()
WHILE num > 0
   total = total + num
   num = input()
END WHILE
print total
```

WHILE Loops are used when the number of repetitions is **NOT** known in advance
WHILE Loops are known as **condition-controlled**, as the loop ends when a **condition** is met.

```
x = 1
WHILE x < 6
    print x
    x = x + 1
END WHILE
```

REPEAT …UNTIL loop
Similar to the **WHILE** loop. Difference being that the Boolean expression is tested at the **end** of the loop!
This means the loop is **always performed** at least **once**!

```
num = input()
REPEAT
  total = total + num
  num = input()
UNTIL num = 0
print total
```

In the above code, when num = 0, the loop will stop.
The condition is tested **AT THE END** of the loop – hence the instructions within the loop **GET EXECUTED AT LEAST ONCE**
Also **condition-controlled** and used when repetitions **NOT known** in advance.

```
x = 1
REPEAT
    print x
    x = x + 1
UNTIL x > 5
```

FOR …NEXT loop
Useful when you **know in advance** the number of iterations you wish to perform.
Uses a counter variable.

```
FOR i = 1 to 5
   print ("ROVERS")
NEXT
```

The above code will iterate 5 times and print *ROVERS* five times. The counter variable i starts at 1 and ends at 5 and **jumps** out of the loop.
Counter-controlled as the Counter variable is used to stop the Loop.
Used when the number of repetitions are **known in advance.**
(Finite number of Loops)

```
FOR x = 1 TO 5
    print x
NEXT
```

The code for each of the programs above outputs the same thing, 1,2,3,4,5.

Programming Theory 3
Nested selection/iteration

3

Computing @ Turton

Nested selection is IF statements within IF statements. ***Indentation*** is important!

```
x = input("Enter your age: ")
if x > 21 then:
    if x > 100 then:
        print("You are too old, go away!")
    else:
        print("Welcome, you are of the right age!")
    end if
else:
    print("You are too young, go away!")
end if
```

Nested iteration is a loop programmed within a loop. See the example below of a times table program.

```
for times_table = 1 to 12:
    for count = 1 to 12:
        product = times_table * count
        print times_table, "x", count, "=", product
    next
next
```

Nesting is made clear by indenting the code. Indenting makes the start and end of the if OR loop more clearer! Use TAB to indent!

SUBROUTINES

A **subroutine** is a named block of code which performs a specific task in a program. It can be called using its name (identifier) in the main program. The two types of subroutine you need to know are ***procedures*** and ***functions.***

Procedures don't need to ***return*** values back to the ***main*** program.

```
PROC displaymenu()
    print("Option 1: Display rules")
    print("Option 2: Start new game")
    print("Option 3: Quit")
    print("Enter 1, 2, or 3: ")
END PROC

#main program
displaymenu()
```

When executed, main program runs first (Sub Main) in VB

A ***function*** MUST return a ***value*** back to the ***main*** program.

```
FUNCTION getchoice()
    print("Option 1: Display rules")
    print("Option 2: Start new game")
    print("Option 3: Quit")
    print("Enter 1, 2, or 3: ")
    choice = input()
    return choice
END FUNCTION
#main program
option = getchoice()
print("You have chosen ", option)
```

This program runs the *getchoice()* function and stores the **return** value in the variable **option** in the main program.

Parameters – Frequently, you need to **pass** values or variables to a subroutine from the main program.

```
1   SUB cylinderVolume(r,len)
2      pi ← 3.142
3      vol ← pi*r*r*len
4      RETURN vol
5   ENDSUB
6   #main program
7   OUTPUT "Enter the radius of the cylinder:"
8   radius ← USERINPUT
9   OUTPUT "Enter the length of the cylinder:"
10  length ← USERINPUT
11  volume ← cylinderVolume(radius,length)
12  OUTPUT "The volume of the cylinder is ", volume
```

- Main program runs first (line 7)
- User enters value for **radius** and **length** of cylinder (Lines 8 & 10)
- The values of the parameters **radius** and **length** are passed to the subroutine where they are referred to using the identifiers **r** and **len** (Line 1)
- Order of passing parameters is important e.g. **radius** gets passed to **r** & **length** gets passed to **len**.
- Names do not need to be the same e.g. length != len

Remember, Functions differ from procedures in that functions return values, unlike procedures which do not. However parameters can be passed to both procedures and functions.

Programming theory 4
Data structures

4

Computing @Turton

A data structure is simply a way of **representing the dat**a held in a computer's memory. There are many data structures available to programmers e.g. ***arrays***, ***records***, lists and more.

An **array** is one method of storing data in an organised structure. If we were making a game and we wanted to store player names and their scores we can store these inside two arrays.

First we must **declare** the arrays so the program knows what size array to create. **The index starts at 0:**

playerNames[4] #declares an array with 5 spaces
gameScores[4]

You can then tell the program exactly what names and scores by writing the following:

gameScores = (124, 99, 121, 105, 132)
playerNames = ("Katie","Patrick","Tom","Rosie","Michael")

Arrays **do not** store mixed **data types**. The first array *gameScores[]* can only store integers. The array *playerNames[]* can only hold strings.

Two-dimensional array – a o*ne-dimensional array can be seen as data elements organised in a row. A t**wo-dimensional array** is similar to a one-dimensional array, but it can be visualised as a grid (or table) with rows and columns.*

To declare a 10x10 grid (10 rows and 10 columns) we could say:

gameGrid[9][9]

Each element in the array can be accessed using its **index** value. Think of them as co-ordinates. An **index** is used to point at a data element in an array. *gameGrid[0][0]* would be pointing to row number 1 and column number 1.

gameScores[] mentioned previously has five elements:
gameScores = (124, 99, 121, 105, 132)
gameScores[0] would return element **124**
gameScores[0] = 95 would change what's being held at position 0 to **95**.

String handling

The ability to **manipulate** alphanumeric data there are multiple functions we use in order to do this.

Function	Meaning
All example code will be using the string "lovelace". *Index numbers will vary depending on language!*	
SUBSTRING(start, end, *string*)	Extract a portion of a string from another. *SUBSTRING(0,3,"lovelace")* would return "love"
POSITION(*string, char*)	Returns index position of a character in a string. POSITION("lovelace","v") would return 3
LENGTH(string)	Return the length of the string LENGTH("lovelace") would return 8.
ASCII(character)	Return the ASCII value of the character. ASCII("A") would return 65.
CHAR(ASCII Value)	Return the Character corresponding to the Numeric ASCII Value. CHAR(65) would return "A"

String concatenation – concatenate means chain strings together to create new ones.
E.g. *print("love" + "lace")* would print *"lovelace".* Here we used the + to concatenate the two strings.

Type conversion – Integers can be converted to strings and vice versa e.g.
int("1") would convert the character "1" to the integer 1.

str(123) converts the integer 123 into a string "123"

Remember an array holds <u>multiple</u> values, whereas an ordinary variable holds a <u>single</u> value!

PSEUDOCODE

5

Computing @ Turton

Variables – Variables are assigned using the = operator.

x = 3
name = "Bob"

A variable is declared the first time a value is assigned. Variables declared inside a function or procedure are local to that subroutine. Variables in the main program can be made global by the keyword global. A global variable is accessible to any subroutine.

Casting – Variables can be typecast using the int, str and float functions.

str(3) returns "3"
int("3") returns 3
Float("3.14") returns 3.14

Outputting to Screen
print("hello") or output("hello")

Selection e.g. IF statements or Select Case

```
if choice = "a" then
    print("you selected a")
elseif choice = "b" then
    print("you selected b")
else
    print("that wasn't a choice!")

Select Case
   Case "a"
      print("you selected a")
   Case "b"
      print("you selected b")
   Case else
      print("that wasn't a choice!")
```

String Handling

To get the length of a string.
stringname.length or len(stringname)
To get a substring (string within a string):
Stringname.substring(startposition, numberof characters)

E.g.

```
sometext = "Computer Science"
print(sometext.length)
Print(sometext.substring(3,3))
```

Will display:
16
put

Comparison Operators

==	Equal to
!=	Not equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

Aritmetic Operators

+	Addition eg x=6+5 gives 11
-	Subtraction eg x=6-5 gives 1
*	Multiplication eg x=12*2 gives 24
/	Division eg x=12/2 gives 6
MOD	Modulus eg 12MOD5 gives 2
DIV	Quotient eg 17DIV5 gives 3
^	Exponentiation eg 3^4 gives 81

Iteration – Counter controlled – Definite

```
for i = 0 to 7
   print("Hello")
next i
```

Will print hello 8 times (0-7 inclusive)

Iteration – Condition controlled – Indefinite

```
while answer != "computer"
   answer = input("What is the password?")
end while

do
   answer = input("What is the password?")
until answer == "computer"
```

Remember MOD gives you the remainder and DIV gives you the integer rounded down!

Pseudocode continued...

Logical Operators in programming e.g. while x <=5 AND flag = false

AND (conjunction)		
INPUT		OUTPUT
A	B	$A \wedge B$
T	T	T
T	F	F
F	T	F
F	F	F

OR (disjunction)		
INPUT		OUTPUT
A	B	$A \vee B$
T	T	T
T	F	T
F	T	T
F	F	F

NOT (negation)	
of ¬ A	
A	$\neg A$
T	F
F	T

Subroutines

Example 1

```
function triple(number)
   return number*3
end function

sub main #main program
y = triple(7)       #calling the triple function
                    passing 7 into the number
                    parameter.
```

Example 2

```
procedure greeting(name)
   print("hello" + name)
end procedure

sub main #main program
greeting("Mr Rifai")
```

Arrays

Arrays will be 0 based (index starts at 0)

```
array names[4].    #declares array with 5 spaces
names[0]="Janine"
names[1]="Emily"
names[2]="Alison"
names[3]="Ahmed"
names[4]="Elijah"

print (names[3])
```

Would print "Ahmed"

Example of 2D Array

```
array board[7,7]    #declares array with 8 rows
and 8 columns
board[0,0]="Pawn"
```

Reading to a file

To open a file to read ***openRead*** is used and ***readLine*** *to return* a line of text from the file. The following program makes x the first line of sample.txt

```
myFile = openRead("sample.txt")
x = myFile.readLine()
myFile.close()
```

Writing from a file

To open a file to write to, ***openWrite*** is used and ***writeLine*** to add a line of text to the file. In the program below hello world is made the contents of sample.txt (any previous contents is overwritten).

```
myFile = openWrite("sample.txt")
myFile.writeline("Hello World")
myFile.close()
```

Comments

Used so a programmer can annotate code so others can easily understand. Makes maintenance easier and to are used help find bugs. Denoted by a **#** or **//**

```
while x < 5:        //we will enter the while loop if the condition is true. The condition is if x > 5
  print("Hello)
```

Remember, functions must always return a value! Procedures don't need to return anything.

LANGUAGE AND TRANSLATORS

7

Computing @ Turton

1st Generation (1GL)	2nd Generation (2GL)	3rd Generation (3GL)
Machine code	**Assembly language**	**High level code**
LOW-LEVEL LANGUAGE (LLL)	**LOW-LEVEL LANGUAGE (LLL)**	**HIGH-LEVEL LANGUAGE (HLL)**
At the very lowest level of operation a computer follows **binary** instructions. Uses 1s and 0s. Processors (CPUs) use machine code and each processor has its own machine code instruction set. In machine code the instructions are made up of two parts e.g. *01101011* **0110** **1011** Opcode | Operand ✓ Closer to architecture (CPU) ✓ No need to **translate** ✗ Difficult for humans to understand ✗ Opcodes have to be memorised	**Assembly code** is created by the developers of processors. Assembly languages are **architecture dependent**. Uses **mnemonics** such as *ADD* or *MOV* to shorten instructions. Often used to develop software for embedded systems and for controlling specific hardware components. ✓ Memory efficient ✓ Greater control of hardware features ✗ Needs to be **translated** by. an **assembler** so a CPU can understand. ✗ Not fully portable due to architecture dependence	Most computer programs are written in a high-level language e.g. Python, Visual Basic & Java. This is code that humans use to program. Uses statements in English and mathematical symbols. E.g. if, function, MOD. **3 types of HLL:** 1. Structure language 2. Procedural language 3. Object oriented language ✓ Easy to understand and write ✓ Easy to maintain and learn ✓ Portable ✗ Needs to be **translated** so a CPU can understand. ✗ Less memory-efficient ✗ Slower than LLL programs. ✗ Cannot communicate directly with the hardware (CPU)

Converting from high level/assembly to machine code - Before a CPU can execute code, it must be converted to machine code. The application used to provide this conversion is called a **translator.**

3 types of translator

1 Assembler –

- *An assembler translates assembly language into machine code.*
- ***Assembly language** has a 1:1 correspondence with machine code.*

2 Compiler –

- Used when source code has been fully developed.
- Translates the <u>whole code</u> in **one** go.
- Reports errors at **end**
- Once translated, it is stored as an <u>executable file</u>. (.exe)
- Because this is a standalone program, it can then be run on other compatible computers (with needing software).

3 Interpreter

- Translates code one line at a time.
- Debugging is easier. Each line of code is analysed and checked before being executed.
- Errors reported **during** translation
- Uses less memory, source code only has to be present one line at a time in memory

IDE – Integrated Development Environment – An application used to create software for example Python's IDLE. Assists the programmer during development. An IDE may support many languages such as Visual Studio (VB, C#)

Features of an IDE – 1) Debugger – Used to identify, find errors. **2**) Syntax highlighting, colour co-ordination. **3)** Provides an Interpreter/Compiler. **4)** Source code editor – Allows you to edit code. **5)** Auto-complete. **6)** Provides access to libraries e.g. import

Remember, all code written in high-level or assembly language MUST be TRANSLATED into machine code!.

SEARCHING

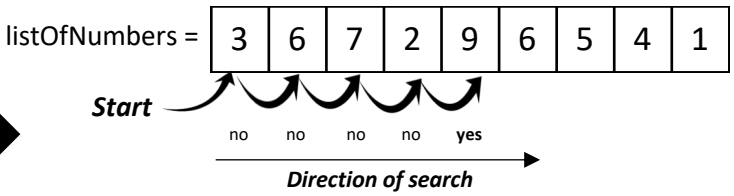
8

Computing @ Turton

Linear Search
Searching for a **keyword** or value is the foundation of many computer programs. The most basic kind of search is a **linear search**.
Method:

- Starts with the first item in the data set
- Compares item to search criteria
- If no match found, the next item is compared
- This continues until a match is found or until you reach the end of the data set
- Also known as a **sequential** search as it moves along each item *sequentially*.

If we were looking for the value **9** in *listOfNumbers[]* below:



Pseudocode

```
Function linearsearch(listOfNum, item)
    index = -1
    i = 0
    found = False
    while i < length(listOfNum) AND NOT found
        if listOfNum[i] = item then
            index = I
            found = True
        end if
        i = i + 1
    end while
    return index
End Function
```

✓ Good for small searches
✓ List does NOT NEED to be **sorted**
✗ Very inefficient and slow for large lists

If the item you are looking for is at the end it will take a LONG time!

Binary Search – A much more efficient method of searching a list for an item. This list though, has to be **IN ORDER**!
Method:

- Split the list in half and compare the midpoint to the item being searched
- If item is at the midpoint it has been found!
- If it isn't is the item being search for higher or lower than the midpoint.
- If **higher** discard the first half of the list until the midpoint.
- If **lower** discard the second half of the list (midpoint to the end)
- **Repeat** the process again of finding the midpoint, examining the item, if higher or lower than item sought and discarding items.
- The item will either be found or will not be in the list

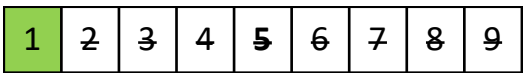
✓ Very efficient, faster than linear search as it halves the data set at each step.
✓ Fewer comparisons needed

✗ Can't be used on an **unsorted** list!
✗ More complicated to implement and is inefficient on very small lists

When looking for the number **1** in *listOfNumbers[]*

Find midpoint which is 5. 1 is less than 5 so we discard the second half of the list.

Find the midpoint of the new list which is 2 (using DIV which rounds down). 1 is less than 2 so discard second half.



Find midpoint of new list which is 1. Match against item we are searching for which is 1. ITEM FOUND!

Pseudocode

```
Function binarysearch(listOfNum, item)
    index = -1    first = 0    found = False
    last = len(listOfNum) – 1
    while first <= last AND found = False
        midpoint = ((first + last) DIV 2)
        if listOfNum[midpoint] = item then
            found = True,   index = midpoint
        else
            if listOfNum[midpoint] < item then
                first = midpoint + 1
            else
                last = midpoint – 1
            end if
        end if
    end while
    return index     #index = -1 if key not found
```

SORTING

9

Computing @ Turton

Bubble sort - An example of a computer algorithm is bubble sort. This is a simple algorithm used for taking a list of jumbled up numbers and putting them into the correct order.

Method:

1. Look at the first number in the list.
2. Compare the current number with the next number.
3. Is the next number smaller than the current number? If so, swap the two numbers around. If not, do not swap.
4. Move to the next number along in the list and make this the current number.
5. Repeat from step 2 until the last number in the list has been reached.
6. If any numbers were swapped, repeat again from step 1.
7. If the end of the list is reached without any swaps being made, then the list is ordered and the algorithm can stop.

When showing the bubble sort in action only re-write the list of numbers/data if you have made a swap!

working example on list of numbers (6, 1, 8, 2, 4):

6 1 8 2 4	6 and 1 **needs** swapping. Re-write the list after the swap.
1 6 8 2 4	6 and 8 doesn't need swapping. 8 and 3 **does** need swapping.
1 6 2 8 4	8 and 4 **needs** swapping. Been through list once. **FIRST PASS!**
1 6 2 4 8	After first pass we know last element in correct place. (8)
1 2 6 4 8	Make **second pass**. 6 and 2 were swapped. 6 and 4 needs swapping.
1 2 4 6 8	**List is now sorted!**

The algorithm knows the list is in order when it goes through a pass without making any swaps!

✓ Easy to implement, more popular
✓ Elements are swapped in place without using additional temporary storage.

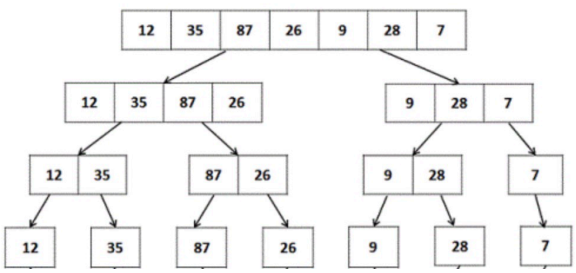
✗ Inefficient with large lists, time-consuming
✗ The more elements stored, the more processing is required

Merge sort - The idea behind this method is the insight that it is quicker to sort two small lists then merge them together, rather sort one big list in the first place. **Two stage sort.**

Method:

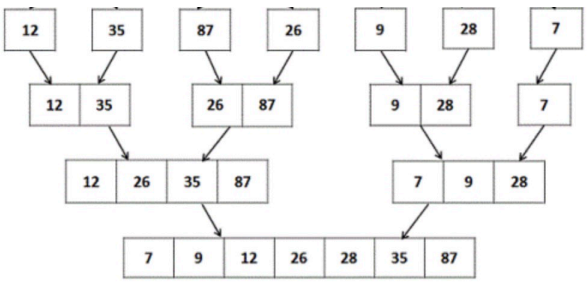
1. If the sub-list is 1 in length, then that sub-list has been fully sorted
2. If the list is more than 1 in length, then divide the unsorted list into roughly two parts. (An odd numbered length list can't be divided equally in two)
3. Keep dividing the sub-lists until each one is only 1 item in length.
4. Now merge the sub-lists back into a list twice their size, at the same time sorting each items into order
5. Keep merging the sub-lists until the full list is complete once again.
6. So the idea is to keep dividing the list and then merge the items back again.

Stage 1 – *Dividing*



Keep dividing the list into sub-lists until there is only 1 item in each sublist.

Stage 2 – *Re-merging*



Now merge the sub-lists back into a list twice their size, at the same time **sorting each items into order.**

✓ Good for sorting slow-access data e.g. tapes
✓ Efficient for sorting data that is accessed sequentially e.g. hard disk

✗ In many implementations if the list is long, then it needs more memory space to handle the sort and all the sub-lists.

***Remember** in a BUBBLE sort the computer knows the list is in order if it goes through a pass without making any swaps.*

Boolean logic

10

Data representation

11

Representing images and sound

12

Data compression

13

- cpu

Ethical, legal and environmental issues

19