Algorithms

An **algorithm** is a sequence of ordered instructions that are followed step-by-step to solve a problem. This does *not* need to be on a computer.

Decomposition is the breaking down of a complex problem into smaller more manageable problems that are easier to solve.

Abstraction allows us to remove unnecessary detail from a problem leaving us with only the relevant parts of a problem thereby making it easier to solve.

Algorithm Efficiency More than one algorithm can be used to solve the same problem. Normally we use the algorithm that solves the problem in the quickest time with the fewest operations or makes use of the least amount of memory.

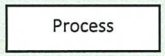
Dry run testing is carried out using **trace tables.** The purpose of the trace tables is for the programmer to track the value of the variables and outputs at each step of the program and to track how they change throughout the running of the program.

Flowchart Symbols

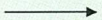
We can represent algorithms using flowcharts

Start and Stop

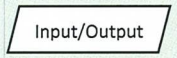
Process – An operation that the algorithm performs



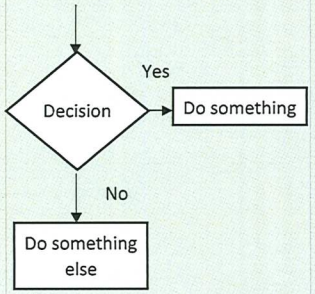
Connector – Links all the other symbols together



Input and Output of data that is read in and written out



Decision is the same as a selection (if then ... else)




```
IF answer is "yes" THEN
      do something
ELSE IF answer is "no"
      do something else
ENDIF
```

Pseudocode

We can represent algorithms using pseudocode

	Example	Python equivalent
Variable assignment	a ← 10	a = 10
Constant assignment	constant PI ← 3.142	PI = 3.142
Input	a ← USERINPUT	a = input()
Output	OUTPUT "Bye"	print("Bye")
Arithmetic Operators		
Add	+	+
Multiply	*	*
Divide	/	/
Subtract	-	-
Integer division	a ← 7 DIV 2	a= 7 // 2
Modulus (remainder)	a ← 7 MOD 2	a = 7 % 2
Relational Operators		
Less than	<	<
Greater than	>	>
Equal to	=	==
Not equal to	≠ or <>	!=
Less than or equal to	≤	<=
Greater than or equal to	≥	>=
Boolean Operators		
AND	AND	AND
OR	OR	OR
NOT	NOT	NOT
Selection		

if ..	IF i > 2 THEN
	j ← 10
	ENDIF
		if i > 2:
		j=10

if .. else ...	IF i > 2 THEN
	j ← 10
	ELSE
	j ← 3
	ENDIF
		if i > 2:
		j=10
		else:
		j=3

if ... else if ... else	IF i ==2 THEN
	j ← 10
	ELSE IF i==3
	THEN
	j ← 3
	ELSE
	j ← 1
	ENDIF
		if i ==2:
		j=10
		elif i==3:
		j=3
		else:
		j=1

Iteration		
While loops	a ← 1 WHILE a < 4 OUTPUT a a ← a + 1 ENDWHILE	while a<4: print(a) a=a+1
For loops	FOR a ← 0 TO 3 OUTPUT a ENDFOR a ← 1	for a in range(3): print(a)
Repeat loops	REPEAT OUTPUT a a ← a + 1 UNTIL a←4	
Subroutines		
procedure	SUB hello() OUTPUT "hello" ENDSUB	def hello(): print("hello")
Function (with paramerters and return)	SUB add(n) a ← 0 FOR a ← 0 TO n a ← a + n ENDFOR RETURN a ENDSUB	def add(n): a=0 for a in range(n+1): a=a+n return a
Built-in functions		
Length of array	LEN(a)	len(a)
Random integer	RANDOM_INT(0, 9)	import random random.randint(0,9)

Searching Algorithms

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 the item 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 the list 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 if the target item is near the start of the list.

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 that they match, so the algorithm returns the index position and stops.

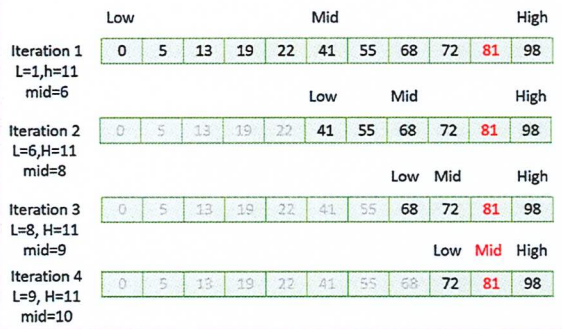
Pseudocode

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

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.
- The 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.

Example: Binary search in operation to find 81



Pseudocode

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

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 searchers through the whole list • very inefficient for long lists
Binary Search	• much quicker than linear search, because it halves the search zone each step	• The list need to be ordered

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.

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**	

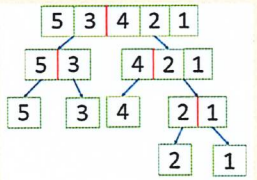
Bubble sort Pseudocode

```
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

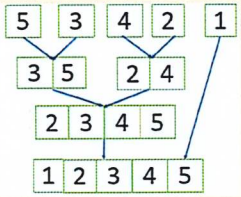
- Merge sort is a type of divide and conquer algorithm.
- There are two steps: divide and combine
- Merge sort works by dividing the unsorted list sublists. It keeps on doing this until there is 1 item in each list.
- Pairs of sublists are combined into an ordered list containing all items in the two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

Step 1: Divide



Keep dividing until there is only 1 item in each list

Step2: Combine



1. The first items in 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 sublist are them copied in order to the parent list.

Merge sort Versus Bubble sort

	Advantages	Disadvantages
Bubble sort	Very simple and robust algorithm	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 than bubble sort especially when the number of elements is large	More complex to understand Step 1: Divide Step 2: Combine

Programming - Python

Comment – Text within the code that is ignored by the computer. A Python comment is preceeded by a #.

```
# This is an example of a comment
```

Output – Processed information that is sent out from a computer

Python	Pseudocode

`print("Hello World!")`
Hello World!
`print("Hello", "World!")`
Hello World!
`print("Hello"+"World!")`
HelloWorld!
`print("Hello\nWorld!")`
Hello
World!
	`OUTPUT "Hello World"`

Input – Data sent to a computer to be processed

`print("Enter name")`	`OUTPUT "Enter name"`
`name=input()`	`name ← USERINPUT`
`print("Hello", name)`	`OUTPUT "Hello", name`
`print("Enter age")`	`OUTPUT "Enter age"`
`age=int(input())`	`age ← USERINPUT`

Assignment - The allocation of data values to variables, constants, arrays and other data structures so that the values can be stored.

- *Variable* – Value that can change during the running of a program. By convention we use lower case to identify variables (eg `a=12`)
- *Constant* – Value that remains unchanged for the duration of the program. By convention we use upper case letters to identify constants. (e.g. `PI=3.141`)

Data Types

Integer	`age = 12`	`age ← 12`
Float (real) number	`height = 1.52`	`height ← 12`
Character	`a = 'a'`	`a ← 'a'`
String – multiple characters	`name = "Bart"`	`name ← "Bart"`
Boolean (true/false)	`a = True` `b = False`	`a ← True` `b ← False`

Arithmetic Operators

Add	`7 + 2`	`= 9`	`7 + 2`
Subtract	`7 - 2`	`= 5`	`7 - 2`
Multiply	`7 * 2`	`= 14`	`7 * 2`
Divide	`4 / 2`	`= 2`	`4 / 2`
power	`2 ** 3`	`= 8`	`2 ** 3`
Integer division	`7 // 2`	`= 3`	`7 DIV 2`
Modulus (remainder)	`7 % 2`	`= 1`	`7 MOD 2`

Relational Operators – Allows the Comparison of values

Less than	`<`	<	`7<2`	`-> False`
Greater than	`>`	<	`7 > 2`	`-> True`
Equal to	`==`	==	`7==2`	`-> False`
Not equal to	`!=`	≠ or <>	`7!=2`	`-> True`
Less than or equal to	`<=`	≤	`7<=2`	`-> False`
Greater than or equal to	`>=`	≥	`7>=2`	`-> True`

Boolean Operators

AND	and	`7 < 2 and 1 < 2`	`-> False`
OR	or	`7 < 2 or 1 < 2`	`-> False`
NOT	not	`not 7 < 2`	`-> True`

Sequencing represents a set of steps. Each line of code will have some operation and these operations will be carried out in order line-by-line

Using + operator for adding

```
a = 1
b = 2
c = a + b
print(c)    -> 3
```

```
a ← 1
b ← 2
c ← a + b
OUTPUT c
```

Using + operator for concatenation

```
a = 'Hello '
b = 'World'
c = a + b
print(c) -> Hello World
```

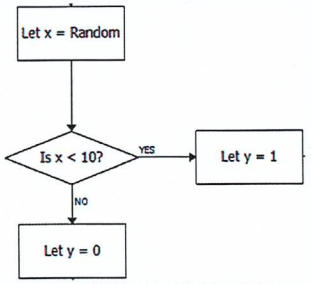
```
a ← 'Hello '
b ← 'World'
c ← a + b
OUTPUT c
```

Random number

Random integer	`import random` `random.randint(0,9)`	`RANDOM_INT(0,9)`
Choice	`random.choice('a','b','c')`	
Random value from 0 to 1	`random.random()`	

Selection represents a decision in the code according to some condition. The condition is met then the block of code is executed otherwise it is not. Often alternative blocks of code are executed according to some condition.

```
x=RANDOM_INT()
IF  x < 10 THEN
 y=1
ELSE
 y=0
ENDIF
```



IF ...	`IF i > 2 THEN`
	`j ← 10`
	`ENDIF`
		`if i > 2:`
		`j=10`

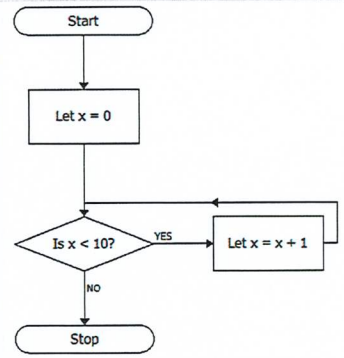
IF ... ELSE ...	`IF i > 2 THEN`
	`j ← 10`
	`ELSE`
	`j ← 3`
	`ENDIF`
		`if i > 2:`
		`j=10`
		`else:`
		`j=3`

IF ... ELSE IF ... ELSE	`IF i ==2 THEN`
	`j ← 10`
	`ELSE IF i==3`
	`j ← 3`
	`ELSE`
	`j ← 1`
	`ENDIF`
		`if i ==2:`
		`j=10`
		`elif i==3:`
		`j=3`
		`else:`
		`j=1`

Iteration Sometimes we wish the code to repeat a set of instructions

`WHILE` loops are used when the we do not know beforehand the number of iterations needed and this varies according to some condition.

```
x = 0
while (x < 10):
      x = x + 1
```



`while True:`
`print("Hello World")`
	`WHILE TRUE`
	`OUTPUT "Hello World"`
	`ENDWHILE`

`a=0`
`while a<4:`
`print(a)`
`a=a+3`
	`a ← 0`
	`WHILE a < 4`
	`OUTPUT a`
	`a ← a + 3`
	`ENDWHILE`

`FOR` loops are used when we know before hand the number of iterations we wish to make.

`for a in range(3):`
`print(a)`
	`FOR a ← 0 TO 3`
	`OUTPUT a`
	`ENDFOR`

Nested structures - Use constructs (e.g. `WHILE, FOR, IF`) inside another.

use a nested FOR loop to print out a grid	`for i in range (10):`
	`  for i in range (10):`
	`  print ("x ",end="")`
	`  print()`

Use a nested while and if to print out only even numbers	`i=0`
	`while i<51:`
	`  if (i%2==0):`
	`  print(i)`
	`  i=i+1`

Lists

Create a list	`shapes=["square","circle"]`
Access element by index pos	`shapes[1] -> circle`
Append item to list	`shapes.append("triangle")`
Remove item from list	`shapes.remove("circle")`
Remove item from list by index	`shapes.pop(1)`
Insert item into list	`shapes.insert(2,"rectangle")`
Number of elements in a list	`len(shapes)`
Get index pos of item in list	`shapes.index("triangle")`

Concatenating lists	`shapesGroup1["square","circle"]`
	`shapesGroup2=["triangle"]`
	`shapes=shapesGroup1+shapesGroup2`

Loop through list	`for i in range(len(shapes)):` `  print(shapes[i])`
Reverse elements in a list	`shapes.reverse()`
Order elements in a list	`shapes.sort()`

2D lists - A list if lists

Create a 2D list	`d = [ [23, 14, 17], [12, 18, 37], [16, 67, 83]]`

Another way to create a 2D list	`a = [23, 14, 17]`
	`b = [12, 18, 37]`
	`c = [16, 67, 83]`
	`d = [a,b,c]`

Access element by index position	`d[1][2] -> 37`

Strings

Get length of a string	`len("Hello")`	`LEN("Hello")`
Character to character code	`ord("a") -> 97`	`ORD("a")`
Character code to character	`chr(101) -> 'e'`	`CHR(101)`
String to integer	`a=int("12")`	`a=INT("12")`
String to float	`a=float("12.3")`	`a=FLOAT("12.3")`
integer to string	`a=str(12)`	`a=STR(12)`
real to string	`a=str(12.3)`	`a=STR(12.3)`

Concatenation -merge multiple strings together	`a="hello "`
	`b="world"`
	`c=a+b`
	`print(c) ->`
	`hello world`

Return the position of a character If there is more than 1 of the same character the position of the first character is returned.	`student = "Hermione"` `student.index('i')`
Find the character at a specified position	`student = "Hermione"` `print(student[2]) -> r`

sub strings - select parts of a string

Example	`student="Harry Potter"`	
Output the first two characters	`print(student[0:2])`	Ha
Output the first three characters	`print(student[:3])`	Har
Output characters 2-4	`print(student[2:5])`	Rry
Output the last 3 characters	`print(student[-3:])`	Ter
Output a middle set of characters	`print(student[4:-3])`	y Pot

*A negative value is taken from the end of the string.

Subroutines are a way of managing and organising programs in a structured way. This allows us to break up programs into smaller chunks.

- Can make the code more modular and more easy to read as each function performs a specific task.
- Functions can be reused within the code without having to write the code multiple times.

- **Procedures** are subroutines that do not return values
- **Functions** are subroutines that have both input and output

Procedure: No input parameters or return	`SUB greeting()`
	`  OUTPUT "hello"`
	`ENDSUB`
		`def greeting():`
		`  print("hello")`
		
		`call: greeting()`

Procedure: One input parameter, no return	`SUB greeting(name)`
	`  OUTPUT "Hello",name`
	`ENDSUB`
		`def greeting(name):`
		`  print("Hello",name)`
		
		`greeting("grey")`

Function: 1 input parameter, and 1 return value	`SUB add(n)`
	`  a ← 0`
	`  FOR a ← 0 TO n`
	`  a ← a + n`
	`  ENDFOR`
	`  RETURN a`
	`ENDSUB`
		`def add(n):`
		`  a=0`
		`  for a in range(n+1):`
		`  a=a+n`
		`  return a`

Function: Two input parameters, and 1 return value	`SUB (num1,num2)`
	`  sum=num1+num2`
	`  return sum`
		`def add(num1,num2):`
		`  sum=num1+num2`
		`  return sum`
		
		`greeting(1,2)`

The **scope** of a variable determines which parts of a program can access and use that variable.

A **global variable** is a variable that can be used anywhere in a program. The issue with global variables is that one part of the code may inadvertently modify the value because global variables are hard to track.

A **local variable** is a variable that can only be accessed within a certain block of code typically within a function. Local variables are not recognized outside a function unless they are returned. There is no way of modifying or changing the behavior of a local variable outside its scope.

Global variables need to defined throughout the running of the whole program. This is an inefficient use of memory resources. Local variables are defined only when they are needed an so have less demand on memory. Local variables only exist within the subroutine.

Reading and writing files

Open file Whatever we are doing to a file whether we are reading, writing or adding to or modifying a file we first need to open it using:

```
open(filename,access_mode)
```

There are a range of access mode depending on what we want to do to the file, the principal ones are given below:

Access Mode	Description
r	Opens a file for reading only
w	Opens a file for writing only. Create a new file if one does not exist. Overwrites file if it already exists.
a	Append to the end of a file. Create a new file if one does not exist.

Reading text files

read – Reads in the whole file into a single string	`f=open("filetxt","r")`
	`print(f.read())`
	`f.close()`

readline – Reads in each line one at a time	`f=open("file.txt","r")`
	`print(f.readline())`
	`print(f.readline())`
	`print(f.readline())`
	`f.close()`

readlines – Reads in the whole file into a list	`f=open("file.txt","r")`
	`print(f.readlines())`
	`f.close()`

Writing text files

Write in single lines at a time	`file=open("days.txt",'w')`
	`file.write("Monday\n")`
	`file.write("Tuesday\n")`
	`file.write("Wednesday\n")`
	`file.close()`

Write in a list	`say=["How\n","are\n","you\n"]`
	`file=open("say.txt",'w')`
	`file.writelines(say)`
	`file.close()`

Data Validation Routines

Check if an entered string has a minimum length	`OUTPUT "Enter String"`
	`s ← USERINPUT`
	`IF LEN(S) > 5 THEN`
	`  OUTPUT "STRING OK"`
	`ELSE`
	`  OUTPUT "TOO SHORT"`
	`ENDIF`

Check is a string is empty	`OUTPUT "Enter String"`
	`s ← USERINPUT`
	`IF LEN(S) == 0 THEN`
	`  OUTPUT "EMPTY STRING"`
	`ENDIF`

Check if data entered lies within a given range	`OUTPUT "Enter number" s num ←`
	`USERINPUT`
	`IF num > 1 AND num < 10`
	`  OUTPUT "Within range"`
	`ENDIF`

Authentication Routine

```
OUTPUT "Enter Username"
username ← USERINPUT
OUTPUT "Enter Password"
password ← USERINPUT

WHILE username != "bart" OR password !="abc"

 OUTPUT "Login failed"
 OUTPUT "Enter Username"
 username ← USERINPUT
 OUTPUT "Enter Password"
 password ← USERINPUT

ENDWHILE

OUTPUT "Login Successful"
```

Debugging

Syntax errors – Errors in the code that mean the program will not even run at all. Normally this is things like missing brackets, spelling mistakes and other typos.

Runtime errors – Errors during the running of the program. This might be because the program is writing to a memory location that does not exist for instance. eg. An array index value that does not exist.

Logical errors - The program runs to termination, but the output is not what is expected. Often these are arithmetic errors.

Test data

Code needs to be tested with a range of different input data to ensure that it works as expected under all situations. Data entered need to be checked to ensure that the input values are:

- within a certain range
- in correct format
- the correct length
- The correct data type (eg float, integer, string)

The program is tested using normal, erroneous or boundary data.

Normal data - Data that we would normally expect to be entered. For example for the age of secondary school pupils we would expect integer values ranging from 11 to 19.

Erroneous data - Data that are input that are clearly wrong. For instance, if some entered 40 for the age of a school pupil. The program should identify this as invalid data but at the same time should be able to handle this sensibly which returns a sensible message and the program does not crash.

Boundary data - Data that are on the edge of what we might expect. For instance if someone entered their age as 10, 11, 19 or 20.