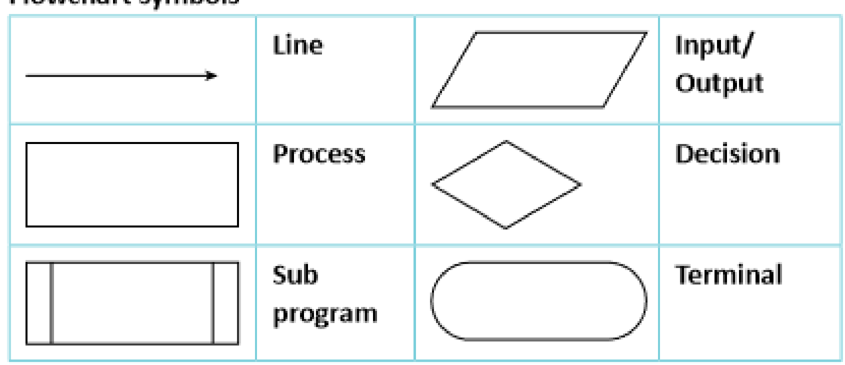
J277 - 2.1 Algorithms

2c. Content of Computational thinking, algorithms and programming (J277/02)

2.1 – Algorithms

Sub topic	Guidance
2.1.1 Computational thinking	
☐ Principles of computational thinking: ○ Abstraction ○ Decomposition ○ Algorithmic thinking	**Required** ✓ Understanding of these principles and how they are used to define and refine problems
2.1.2 Designing, creating and refining algorithms	
☐ Identify the inputs, processes, and outputs for a problem ☐ Structure diagrams ☐ Create, interpret, correct, complete, and refine algorithms using: ○ Pseudocode ○ Flowcharts ○ Reference language/high-level programming language ☐ Identify common errors ☐ Trace tables	**Required** ✓ Produce simple diagrams to show: ▪ The structure of a problem ▪ Subsections and their links to other subsections ✓ Complete, write or refine an algorithm using the techniques listed ✓ Identify syntax/logic errors in code and suggest fixes ✓ Create and use trace tables to follow an algorithm **Flowchart symbols**



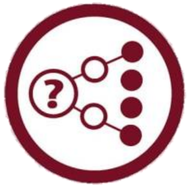
J277 - 2.1 Algorithms

Key techniques for computational thinking

Abstraction

Representing 'real world' problems in a computer using variables and symbols and removing unnecessary elements from the problem.

Decomposition



Breaking down a large problem into smaller sub-problems.

Algorithmic thinking

Identifying the steps involved in solving a problem.

Example: Find the quickest route by car between two places.

Details to ignore	Details to focus on
Distance crow flies	Shortest route along the roads
Road names	Traffic information

What is the length of each route?
What are the speed limits on each route?

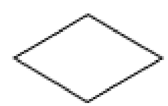
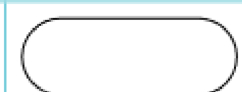
1. List all potential routes.
2. Find lengths of each route.
3. Calculate time for each route.
4. Find route with shortest time.

2c. Content of Computational thinking, algorithms and programming (J277/02)

2.1 – Algorithms

Sub topic	Guidance
2.1.1 Computational thinking	
☐ Principles of computational thinking: ○ Abstraction ○ Decomposition ○ Algorithmic thinking	**Required** ✓ Understanding of these principles and how they are used to define and refine problems
2.1.2 Designing, creating and refining algorithms	
☐ Identify the inputs, processes, and outputs for a problem ☐ Structure diagrams ☐ Create, interpret, correct, complete, and refine algorithms using: ○ Pseudocode ○ Flowcharts ○ Reference language/high-level programming language ☐ Identify common errors ☐ Trace tables	**Required** ✓ Produce simple diagrams to show: ▪ The structure of a problem ▪ Subsections and their links to other subsections ✓ Complete, write or refine an algorithm using the techniques listed ✓ Identify syntax/logic errors in code and suggest fixes ✓ Create and use trace tables to follow an algorithm

Flowchart symbols

	**Line**		**Input/ Output**
	**Process**		**Decision**
	**Sub program**		**Terminal**

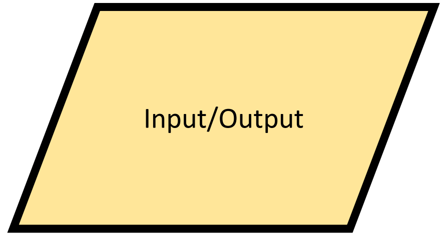
Identify the input, processes and outputs for a problem

An input is:	Any information or data which goes into a system.
A process is:	Anything which happens to information or data during a programs execution e.g. performing calculations or conversions.
An output is:	Any information of data which leaves a system.

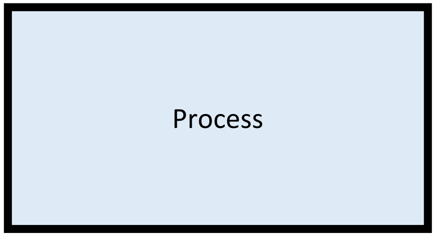
Title of program	**What does it do?**	**Inputs**	**Processes**	**Outputs**
Temperature Converter	Converts the temperature from Celsius to Fahrenheit	Temperature in Celsius (e.g., 25 degrees)	Convert the Celsius temperature to Fahrenheit	Temperature in Fahrenheit (e.g., 77 degrees)
Addition Calculator	Add 2 numbers together	Two numbers (e.g., 5 and 3)	Add the two numbers together	The sum of the two numbers (e.g., 8)
BMI Calculator	Works out a person's BMI	Person's weight (in kg) and height (in meters)	Calculate the Body Mass Index (BMI) using the weight and height	The calculated BMI value (e.g., 23.4)
File Sorter	Sorts files into alphabetical order	List of unsorted filenames (e.g., ["file3.txt", "file1.txt", "file2.txt"])	Sort the filenames alphabetically	Sorted list of filenames (e.g., ["file1.txt", "file2.txt", "file3.txt"])

Flow diagram symbols

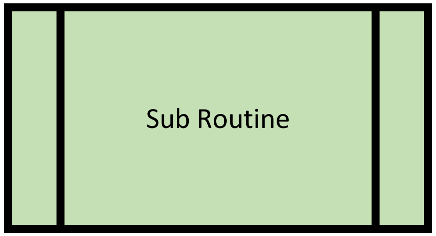
This shape represents the start or end of the process.



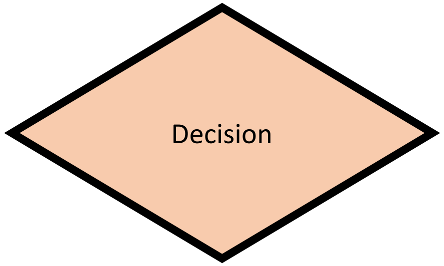
This shape represents the input or output of data.



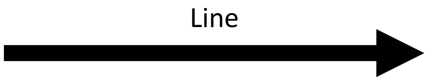
This shape represents something being initialised, processed or calculated.



This shape represents a subroutine call that will relate to a separate non-linked flow chart.



This shape represents a decision with yes or no, true or false that results in two lines for the two outcomes.



An arrow represents control passing between connected shapes.

Pseudocode

Pseudocode uses short English words/statements to describe an algorithm.

It would generally look a little more structured than just writing English sentences.

However it is very flexible.

It is less precise than using a reference language, or a programming language.

```
IF Age is equal to 14 THEN
    Stand up
ELSE Age is equal to 15 THEN
    Clap
ELSE Age is equal to 16 THEN
    Sing a song
ELSE
    Sit on the floor
END
```

Exam reference language

Output:

```
print("Hello")
```

Input:

```
num = input("Enter a number")
```

Selection:

```
if num == 2 then
        ...
elseif num < 4 then
        ...
endif
```

FOR Loops

```
for i = 1 to 10
      ...
next i
```

WHILE Loops

```
while (i != 11)
      ...
endwhile
```

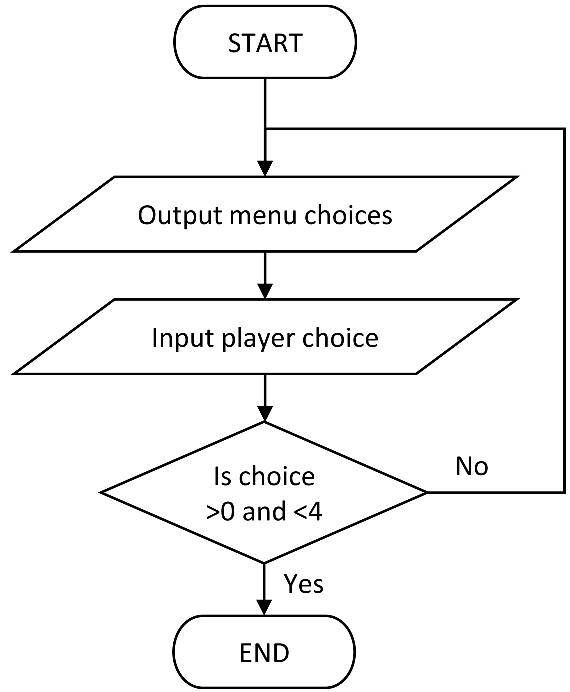
```
do
      ...
until i > 10
```

How to produce algorithms using flow diagrams

An algorithm for an RPG game displays 3 choices from a menu and allows the user to enter their choice.

1. Play game
2. Change character
3. Quit

The user input is validated so only the numbers 1-3 can be entered.



Interpret, correct, refine or complete algorithms.

An algorithm for an RPG game displays 3 choices from a menu and allows the user to enter their choice.

1. Play game
2. Change character
3. Quit

The user input is validated so only the numbers 1-3 can be entered.

```
do
    print(“1. Play game”)
    print(“2. Change character”)
    print(“3. Quit”)

    input(int(choice))

until choice<1 OR choice>3
```

Identifying common errors and suggesting fixes

```
def calculate_average(numbers):
    total = sum(numbers)
    average = total / len(numbers) + 1  # Logic error: Adding 1 to the avera
    return average

# Example usage
number_list = [5, 10, 15, 20, 25]
result = calculate_average(number_list)
print("The average is:", result)
```

The error is on:

`average = total / len(numbers) + 1`

The type of error is:

Logical

In order to fix this error:

Instead of calculating the correct average, the program mistakenly adds 1 to the average value.

`average = total / len(numbers)`

Identifying common errors and suggesting fixes

```
def print_message:
    message = "Hello, world!"
    print(message)

# Call the function
print_message()
```

The error is on:	`Def print_message`
The type of error is:	Syntax
In order to fix this error:	Includes the required parentheses after the function name. `def print_message():`

Trace tables

In this example, the trace table represents the input values **a**, **b**, and **c**, as well as the expected result for each combination. The Python code defines a function called **calculate_result** that takes three parameters: **a**, **b**, and **c**.

The logic in the code checks different conditions using **if**, **elif**, and **else** statements to determine the appropriate result based on the given inputs. The function then returns the calculated result.

The example usage section calls the **calculate_result** function with different sets of input values, and the results are stored in **result1**, **result2**, and **result3**. Finally, the program prints out the calculated results.

```python
def calculate_result(a, b, c):
    if a > b:
        result = 0
    elif b > c:
        result = a - b
    else:
        result = -1
    return result

# Example usage
result1 = calculate_result(2, 3, 4)
result2 = calculate_result(5, 2, 3)
result3 = calculate_result(1, 1, 1)

print("Result 1:", result1)
print("Result 2:", result2)
print("Result 3:", result3)
```

a	b	c	Result
2	3	4	0
5	2	3	5
1	1	1	-1

J277 - 2.1 Algorithms

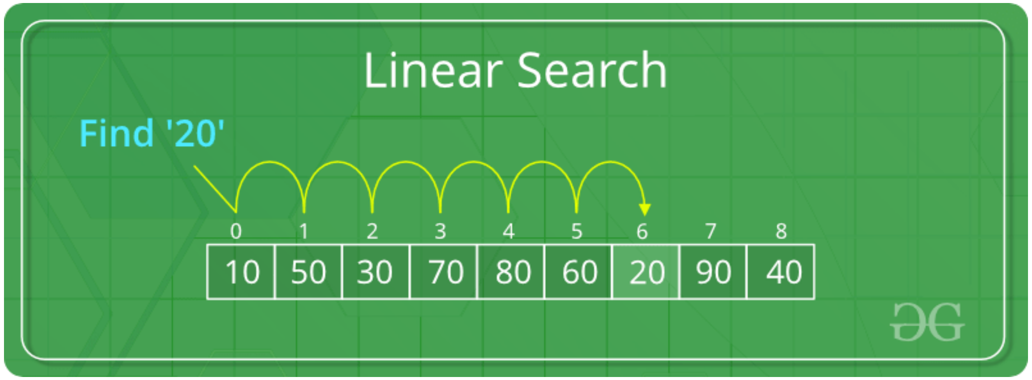
2.1.3 Searching and sorting algorithms	
☐ Standard searching algorithms: ○ Binary search ○ Linear search ☐ Standard sorting algorithms: ○ Bubble sort ○ Merge sort ○ Insertion sort	**Required** ✓ Understand the main steps of each algorithm ✓ Understand any pre-requisites of an algorithm ✓ Apply the algorithm to a data set ✓ Identify an algorithm if given the code for it **Not required** ✗ To remember the code for these algorithms

Linear search

Explanation of a linear search:

Each item in the list is checked in order. Only works on an ordered list.

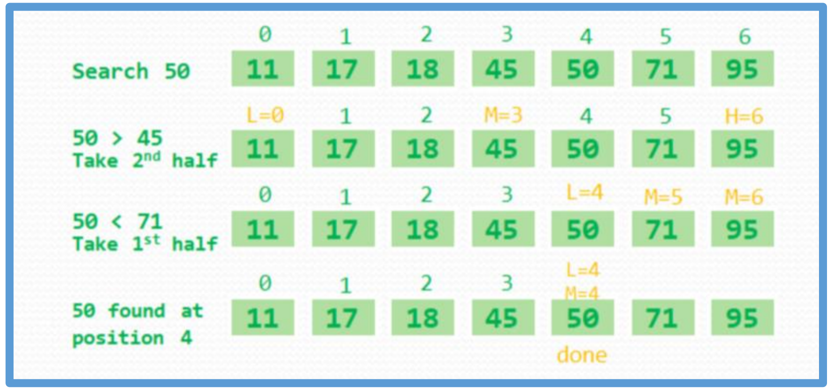
- Check the first value
- IF it is the value you are looking for
  - Celebrate and stop
- ELSE move to and check the next value
- REPEAT UNTIL you have checked all the elements and not found the value you are looking for



Binary search

Explanation of a binary search:

> Calculate the mid point. Check if that is the item to find. If not, if it is lower than the midpoint, repeat on the left half of the list, or repeat on the right half of the list.



The list needs to be in order.
Take the middle value.
Compare to the value you are looking for.
IF it is the value you are looking for.
- Celebrate, and stop.

ELSEIF it is larger than the one you are looking for.
- Take the values to the **left** of the middle value.

IF it is smaller than the one you are looking for.
- Take the values to the **right** of the middle value.

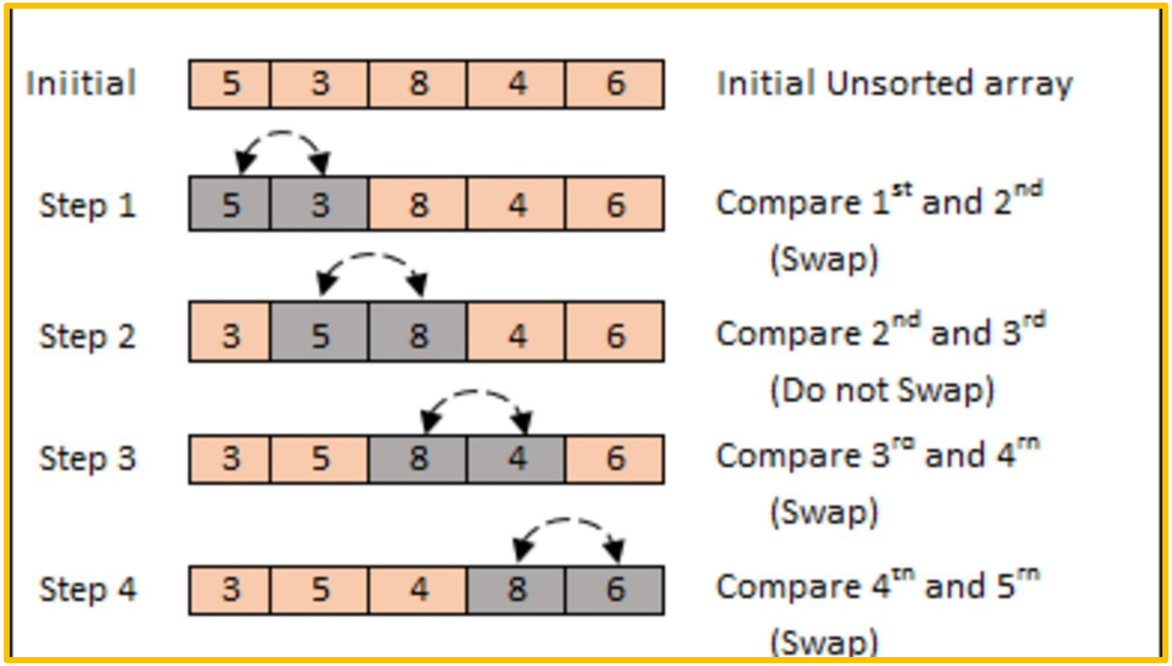
Repeat with the new list.

Bubble sort

Moving through a list repeatedly, swapping elements that are in the wrong order.

1. Take the first element and second element from the list
2. Compare them
3. IF element 1 > element 2 THEN
   - Swap then
4. ELSE
   - Do nothing
5. **Repeat**: Move along the list to the next pair
   - IF no more elements: Goto 1
   - ELSE: Goto 2

   Until: you have moved through the entire list and **not** made any changes

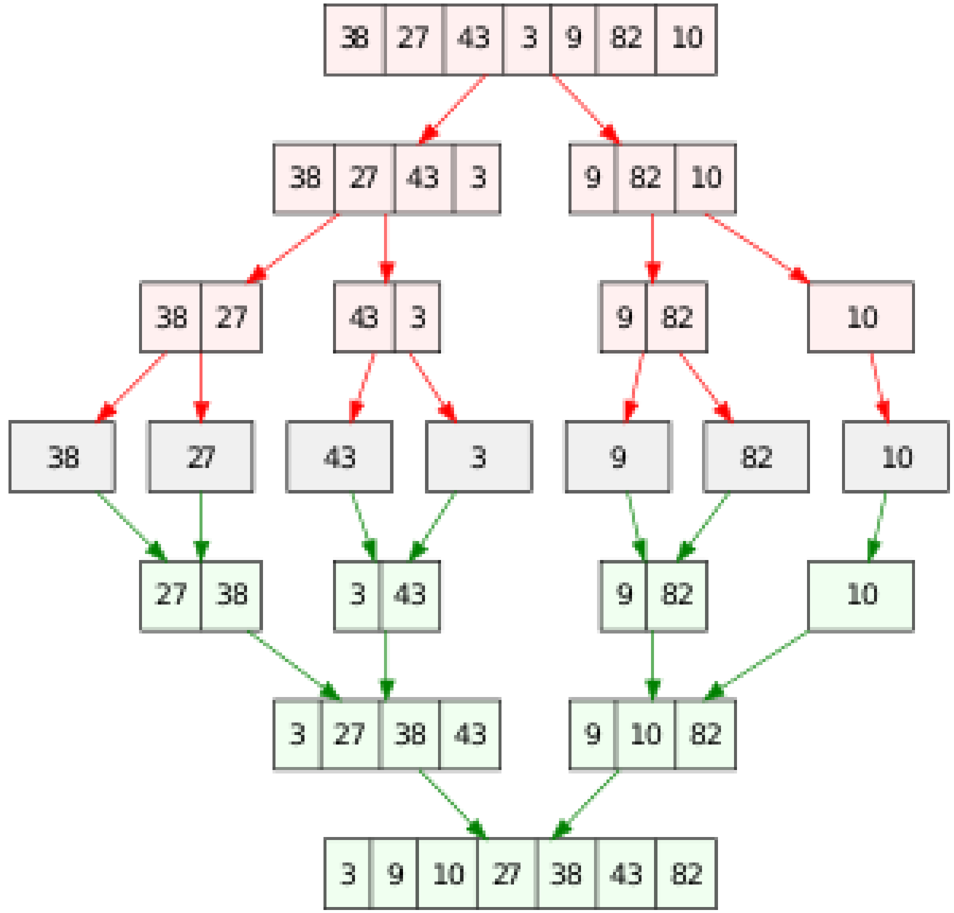


Continue until there are no more swaps.

Merge sort

A list is split into individual lists, these are then combined (2 lists at a time).

1. Split all elements into individual lists.
2. Compare the first element in both lists.
3. Put the smallest into a new list.
4. Compare the next element of 1 list with the second element of the 2nd list.
5. Put the smallest into a new list.
6. Repeat until merged.



J277 - 2.1 Algorithms

Insertion sort

Each items is take in turn, compare to the items in a sorted list and placed in the correct position.

1. Element 1 is a ‘sorted’ list.
2. The rest of the elements are an ‘unsorted’ list.
3. Compare the first element in the ‘unsorted’ list to each element in the sorted list.
4. IF it is smaller, put it in in front of that element (move the others along).
5. ELSEIF it is larger, compare with the next.
6. ELSEIF there are no more elements in the ‘sorted’ list put it in the final position.
7. REPEAT UNTIL all element in the ‘unsorted’ list are in the ‘sorted’ list.