Unit 2: Computational thinking

1. Introduction

Computational thinking is what allows us to go from the quick and simple little programs we've been writing, to solving more meaningful problems.

When discussing Computational Thinking, there are four thinking skills commonly associated with this.

- Decomposition - Can I divide this into sub-problems?
- Pattern recognition - Can I find repeating patterns?
- Abstraction - Can generalise this to make an overall rule?
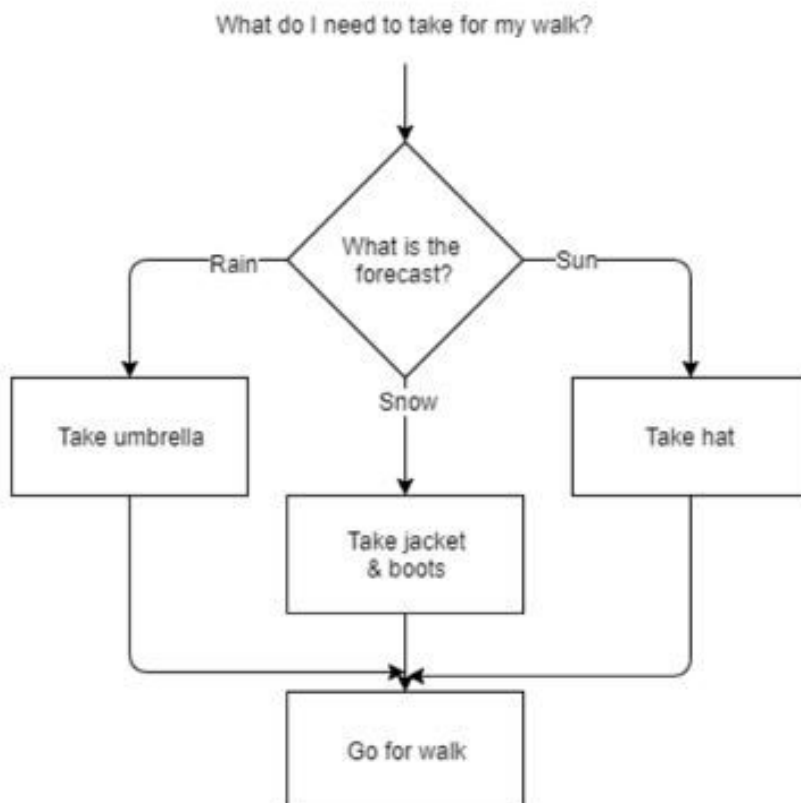- Algorithm design - Can I document the programming steps for any of this?

To demonstrate Computational thinking in action, study the walk through in these slides:

- Slides: https://pbaumgarten.com/igcse-compsci/igcse-2-computational-thinking-slides.pdf
- Video with narration of the process: https://youtu.be/2bvt6PCBVPo

Finally, some other general advice I would give to new programmers is:

- Just start - A blank screen can be overwhelming
- Don't start at the start - Programs are not novels, you read and write a program by jumping around
- Start with something you know - Build the user interface and work back from there
- Don't be afraid to Google - Prioritise results from forums such as stackoverflow
- Test & print a lot

2. Introducing flowcharts

What do I need to take for my walk?

```
                    Is forecast
        Yes ────    for rain?
         │          
         ▼          │ No
   Take umbrella    ▼
         │       Go for walk
         └──────►
```

What do I need to take for my walk?

```
                    What is the
      Rain ────     forecast?     ──── Sun
        │                                │
        ▼            │ Snow               ▼
  Take umbrella      ▼                 Take hat
        │        Take jacket              │
        │        & boots                  │
        │            │                    │
        └────────►◄──┴────────◄───────────┘
                 Go for walk
```

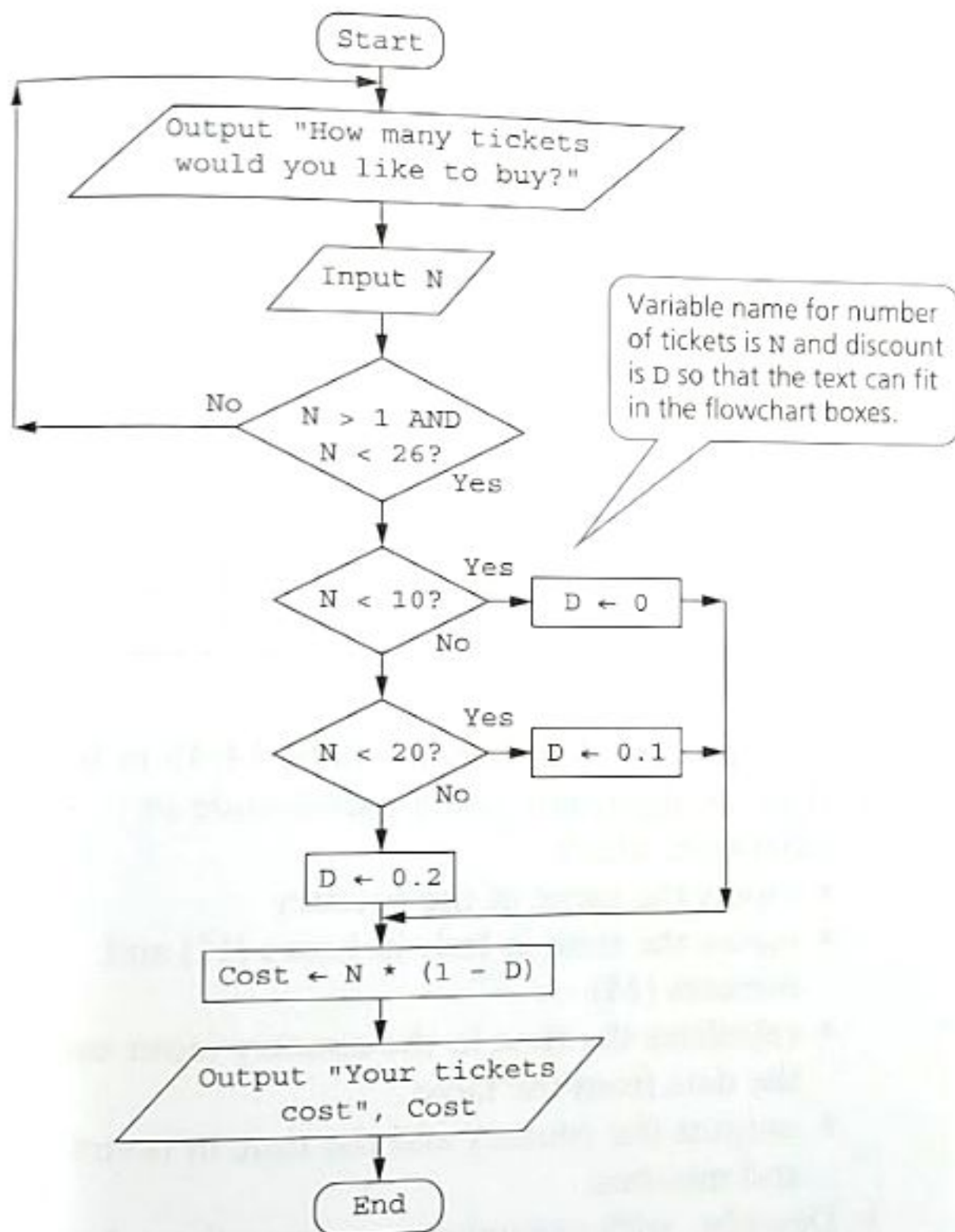| Symbol	Meaning
Terminator	Start or end point of a program.
Process	A task or action that needs to be performed.
Arrows	The "flow" indicating what part of the program should be executed next.
Decision	A decision is required to move forward. This could be a binary, this-or-that choice or a more complex decision with multiple choices. Make sure that you capture each possible choice within your diagram.

```
                    ┌──────────┐
                    │  Start   │
                    └──────────┘
                         │
        ┌────────────────▼──────────────────┐
        │   Output "How many tickets        │
        │    would you like to buy?"        │
        └────────────────┬──────────────────┘
                         │
                    ╱────▼────╲
                   │ Input N   │
                    ╲─────────╱
                         │
      No          ╱──────▼──────╲
    ◄─────────────   N > 1 AND   ╲        ┌────────────────────────────┐
                  ╲  N < 26?    ╱         │ Variable name for number   │
                   ╲──────┬────╱          │ of tickets is N and discount│
                         │ Yes            │ is D so that the text can fit│
                         │                │ in the flowchart boxes.    │
                         │                └────────────────────────────┘
                   Yes
              ╱──────▼──────╲        ┌──────────┐
             ╲  N < 10?    ╱─────────►  D ← 0   │───┐
              ╲──────┬────╱          └──────────┘   │
                  │ No                              │
              ╱───▼────────╲     Yes  ┌──────────┐  │
             ╲  N < 20?    ╱─────────►  D ← 0.1  │──┤
              ╲──────┬────╱          └──────────┘  │
                  │ No                             │
           ┌──────▼──────┐                         │
           │  D ← 0.2    │                         │
           └──────┬──────┘                         │
                  │◄────────────────────────────────┘
         ┌────────▼─────────────┐
         │ Cost ← N * (1 - D)   │
         └────────┬─────────────┘
                  │
      ┌───────────▼───────────────┐
      │  Output "Your tickets     │
      │    cost", Cost            │
      └───────────┬───────────────┘
                  │
             ┌────▼────┐
             │   End   │
             └─────────┘
```
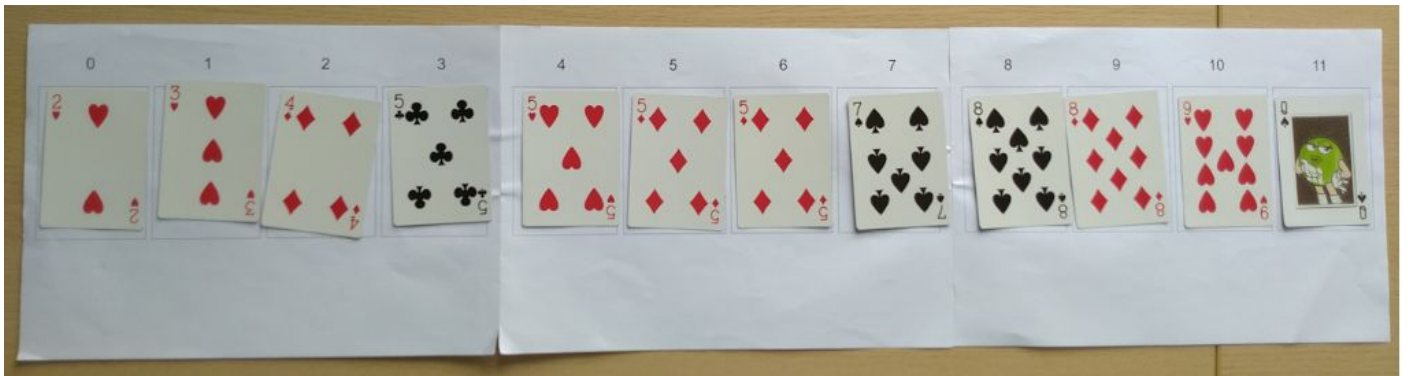
Figure 10.10 Flowchart for Example 1

Create flowcharts exercise

You will be given a paper "array" and some playing cards. Shuffle the cards and arrange them as follows.

The 0 through 11 across the top represent the index numbers for each element of the array.
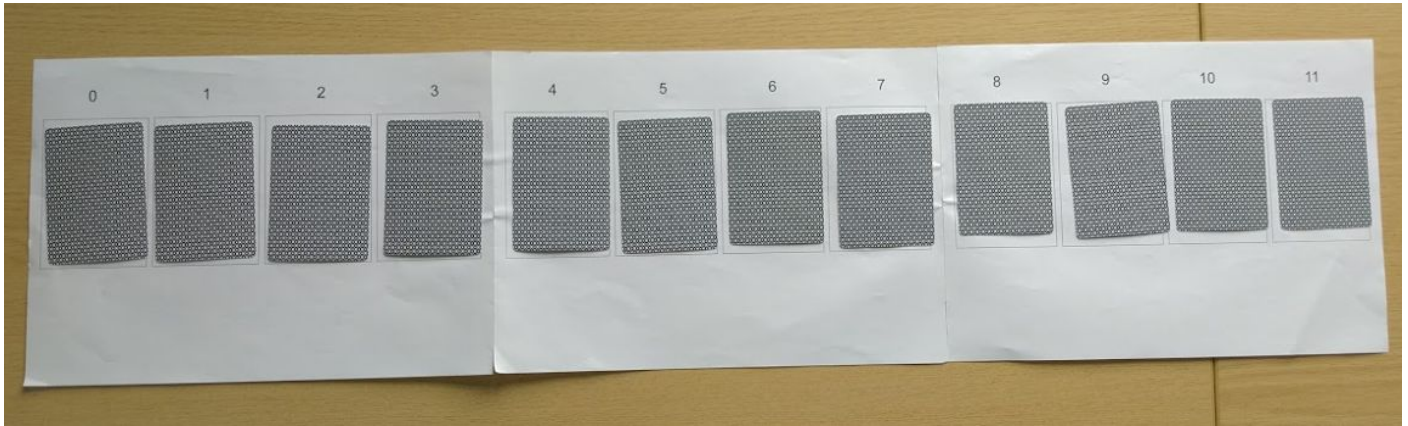
Your overall objective is to create a flowchart procedure, that will place all cards into sorted order, eg:
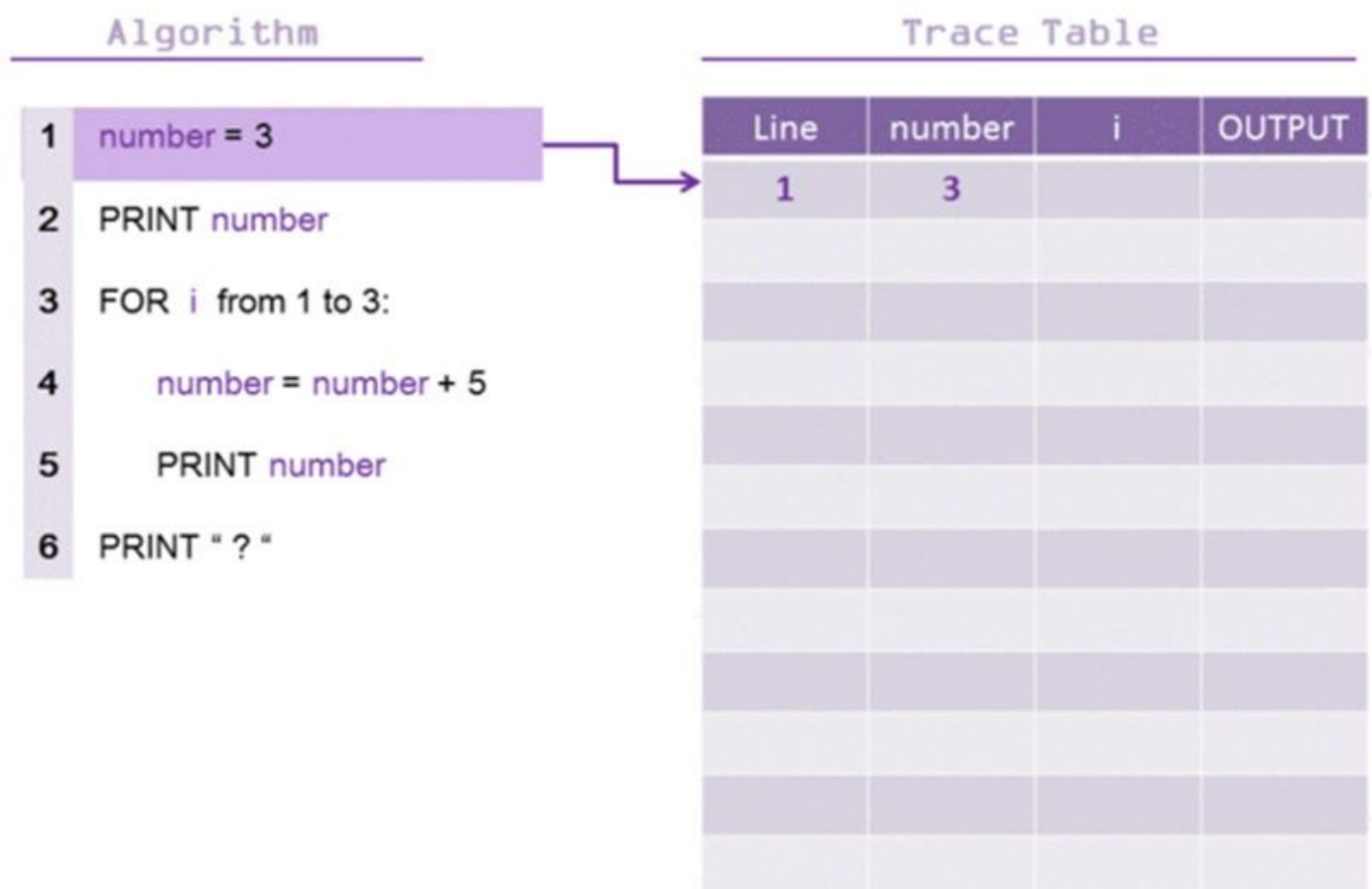

Constraints:
1. Working in pairs
2. Only 2 cards may be face up at a time
3. Ignore suits
4. Treat Jack as 11, Queen as 12, King as 13, etc
5. Make your instructions as "code like" as possible but some English phrases acceptable

Your tasks:
1. Document your procedure as a flowchart.
2. Swap your procedure with another group.
3. Test the other groups' procedure. Did they come up with the same method as you or is it different? What good ideas did they have? What suggestions can you make to improve theirs? Swap your feedback with the other group.
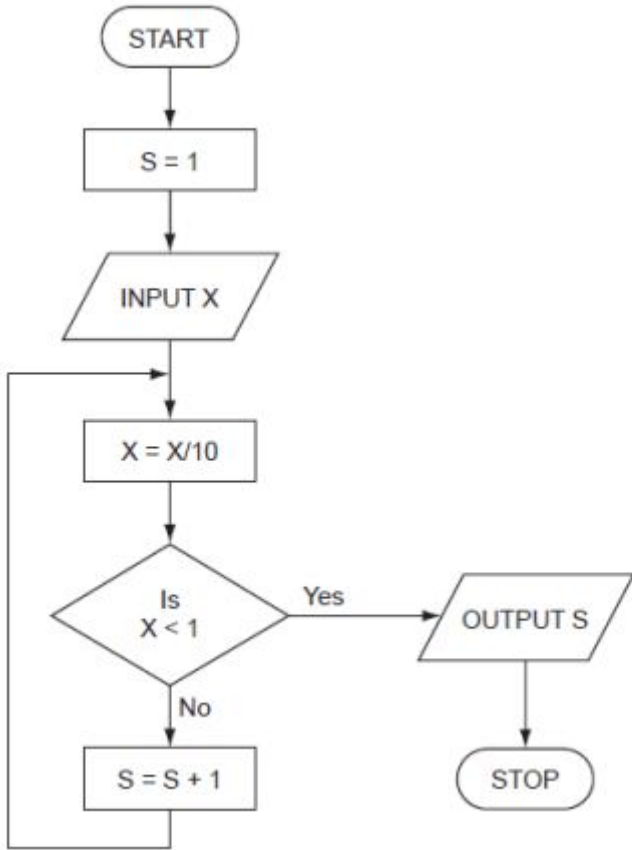4. Finalise your procedure after receiving feedback.

3. Tracing flowcharts

Learning objectives

- A trace table is a manual mechanism of debugging an algorithm

Introduction

Read 9.5 Using trace tables and
Read 9.6 Identifying and correcting errors

Discussion & learning items

Trace tables are a manual tool for testing and correcting an algorithm.

They are a table where you manually walk through each line of your algorithm, calculating the value of each variable as you go. Every variable gets its own column, and you add a new row for every new step of program execution.

Algorithm

1   number = 3

2   PRINT number

3   FOR i from 1 to 3:

4   number = number + 5

5   PRINT number

6   PRINT " ? "

Trace Table

Line	number	i	OUTPUT
1	3		
2			3
3		1	
4	8		
5			8
3		2	
4	13		
5			13
3		3	
4	18		
5			18
6			?

Problem 1

Create a trace table using the following inputs: 48, 9170, -800.

INPUT	S	X	OUTPUT

START

INPUT number

product = 1

value = number

product = value * product

value = value – 1

is value = 0?   No

Yes

OUTPUT number, product

STOP

Complete the trace table for the input value of 5.

INPUT	Number	Product	Value	OUTPUT
5	5	1	5	
		5	4	
		20	3	
		60	2	
		120	1	
		120	0	5, 120

```
                    ┌─────────────┐
                    │    START    │
                    └──────┬──────┘
                           │
                    ┌──────▼──────┐
                    │  LIMIT = 4  │
                    │ FLAG = TRUE │
                    └──────┬──────┘
                           │
                    ┌──────▼──────┐      No
                    ◇ Is FLAG = TRUE? ◇──────────►  END
                    └──────┬──────┘
                           │ Yes
                    ┌──────▼──────┐
                    │ FLAG = FALSE │
                    │ COUNTER = 0 │
                    └──────┬──────┘
                           │
           Yes      ┌──────▼──────┐
        ◄───────────◇ Is COUNTER > ◇
                    │   LIMIT - 1? │
                    └──────┬──────┘
                           │ No
                    ┌──────▼──────┐
                    │      Is      │   Yes
                    ◇ VALUES[COUNTER] > ◇──────►  TEMPORARY = VALUES[COUNTER]
                    │ VALUES[COUNTER + 1]? │
                    └──────┬──────┘               VALUES[COUNTER] = VALUES[COUNTER + 1]
                           │ No
                           │                      VALUES[COUNTER + 1] = TEMPORARY
                           │
                           │                      FLAG = TRUE
                           │
                    ┌──────▼──────┐
                    │ COUNTER = COUNTER + 1 │
                    └─────────────┘
```

START

LIMIT = 4
FLAG = TRUE

Is FLAG = TRUE?   No   END

Yes

FLAG = FALSE
COUNTER = 0

Is COUNTER >
LIMIT - 1?   Yes

No

Is
VALUES[COUNTER] >
VALUES[COUNTER + 1]?   Yes   TEMPORARY = VALUES[COUNTER]

No

VALUES[COUNTER] = VALUES[COUNTER + 1]

VALUES[COUNTER + 1] = TEMPORARY

FLAG = TRUE

COUNTER = COUNTER + 1

Line	LIMIT	FLAG	COUNTER	TEMPORARY	VALUES			
					0	1	2	3
					19	32	11	21

4. Tracing pseudocode

Learning objectives

- Pseudocode is a manner of documenting a computer algorithm in a way that is programming-language agnostic, so it can be understood by any programmer no matter their language of speculation.

Introduction

Read 9.7
Read all of chapter 10

Discussion & learning items

What is pseudo code?

"Pseudo" is defined as "not actually but having the appearance of; pretended; false or spurious; sham; almost, approaching, or trying to be." ([reference](#)).

So pseudo-code is pretend-code. It is not a genuine, real programming language. The intent behind the idea is that it is a generic, language agnostic tool that can be used to communicate algorithms between programmers regardless of what language they may be specialists in. That is, something that can be read and understood by all programmers because it is so clear and simple.

The following is an example of pseudo-code you will see in this course. You should be able to read it, understand what it is conveying and convert it into your own programming language without difficulty.

```
PRINT "I can count! What number should I count up to?"
INPUT Target
N ← 1
WHILE N < Target
    PRINT N
    N ← N + 1
PRINT "Told you I could do it!"
```

A note about pseudo code: What it is supposed to be versus what it is in the iGCSE course... While the idea of pseudo-code is supposed to be a language-agnostic, generic undefined tool for documenting algorithms, that is not compatible with the notion of exam marking schemes. As a result, pseudo code for our course is actually quite prescriptively defined. While you don't have to be syntactically perfect, you are expected to write pseudocode that is very clear and to the same level of detail as defined within the syllabus. In other words, while not a real programming language, you do need to learn it and practice it for it will appear in your exams.

The official syntax of pseudo code for the iGCSE course:
- Chapter 10, page 134-139 of the textbook.

Tracing pseudocode exercises

Practice your knowledge of trace tables and pseudo code.

Problem 1.

Complete a trace table to determine the final state of the variables in this algorithm.

```
sum ← 0
N ← 10
WHILE N < 40
    sum ← sum + N
    OUTPUT N, SUM
    N ← N + 5
END WHILE
```

Problem 2.

Use a trace table to determine what is printed by the following when N = 5.

```
IF (N = 1) OR (N = 2) THEN
    H ← 1
ELSE
    F ← 1
    G ← 1
    FOR J ← 1 TO N-1 DO
        H ← F + G
        F ← G
        G ← H
        OUTPUT H
    END FOR
END IF
OUTPUT F, G
```

Problem 3.

Use a trace table to determine what is printed by the following.

```
Sum ← 0
N ← 20
WHILE N < 30
    Sum ← Sum + N
    OUTPUT N, SUM
    N ← N + 3
END WHILE
```

Problem 4.

Use a trace table to determine what is printed by the following.

```
Count ← 1
X ← 2
WHILE Count < 25
    X ← X + 2
    OUTPUT Count, X
    Count ← Count + 5
END WHILE
```

Problem 5.

Complete a trace table for the following given that the input will be 6.

```
INPUT X
FOR M = 1 TO X
    Y ← X - M
    Z ← 5 * Y - M
END FOR
OUTPUT Z
```

Problem 6.

Create a trace table for the following algorithm to determine its purpose. You may need to "run" this two or three times to determine its purpose. Suggested inputs:

- First run: 3, 6
- Second run: 5, 4
- Third run: 4, 10

```
OUTPUT "Enter the first integer: "
INPUT x
OUTPUT "Enter the second integer: "
INPUT y
z ← 0
WHILE x > 0
    IF x mod 2 = 1 THEN
        z ← z + y
    ENDIF
    x ← x div 2
    y ← y * 2
ENDWHILE
OUTPUT "Answer =", z
```

Problem 1 to 5:   https://retrosnob.files.wordpress.com/2013/11/trace-tables-chapter.pdf
Problem 6:   https://www.tes.com/teaching-resource/trace-table-activities-for-gcse-computer-science-11782527

5. Writing and tracing pseudocode

From the official course document, <u>0984_Pseudocode_Guide_for_Teachers_(for_examination_from_2019)</u>:

> *Learners are not required to follow this guide in their examination answers or any other material they present for assessment. By definition, pseudocode is not a programming language with a defined, mandatory syntax.* ***Any pseudocode presented by candidates will only be assessed for the logic of the solution presented – where the logic is understood by the Examiner, and correctly solves the problem addressed, the candidate will be given credit*** *regardless of whether the candidate has followed the style presented here. Using a recommended style will, however, enable the candidate to communicate their solution to the Examiner more effectively.*

Writing and tracing pseudocode exercises

For each of the following:
1. Design an algorithm with pseudo-code
2. Conduct a trace table to test your algorithm, make any corrections required
3. Swap and test algorithms with a partner in the room

Exercises:

1. Design an algorithm… where the user inputs three numbers, and then outputs them in order from lowest to highest value.

    For example: Given inputs 8, 3 and 10; the program will output 3, 8, 10.

2. Design an algorithm… where the user inputs three peoples names, and the program prints the name of the person whose name is longest (contains the most characters).

    For example: Given inputs: Leah, Han, Chewbacca; the program will output Chewbacca.

3. Design an algorithm… where the user continually inputs a number, stopping when -1 is provided. For each number, the count and the sum of the numbers provided is kept and output at the end.

    For example, given inputs 5, 8, 3, 11, -1; the program outputs 4 (the count) and 27 (the sum).

4. Design an algorithm… that counts numbers. Have the user input two positive integers, and the program counts up by increments of 1 from the smaller number up to but not including the larger number.

    For example, given inputs 10 and 20, the program outputs 10, 11, 12, 13, 14, 15, 16, 17, 18, 19.

Optional extra: 70 pseudocode practice questions:

- https://pbaumgarten.com/igcse-compsci/distribute/pseudocode-70-questions.pdf
  (I have the solutions for these for you to self-check)

6. Mini-quiz

20 minute quiz on flowchart, pseudocode and trace tables

20 minute lesson on reading text files, introducing the hang person problem

7. Validation (or: Never trust the user!)

Learning objectives
- Appreciation for the need to test that our algorithms do not fail when presented with unexpected data
- Consider the importance of validating input from the user before accepting it into your program
- Implement input validation checks for some given problems

Textbook
- Pre-reading from textbook: 9.3 Test data

Introduction

A maxim of computer science is **do not trust the user!** If they can misinterpret your instructions (or intentionally try to work around them), they will. The corollary of that is to **always parse your inputs!**

As our programs become more complicated, having a proper testing regime becomes increasingly important. It is no longer good enough to run our program with a "typical" value, get the expected result and consider it "working".

We do this through:

- Anticipate - What are possible incorrect or invalid inputs that could be received here
- Validate - Write code to identify incorrect/invalid inputs and prevent them from being used by our program
- Test - Use sample data to ensure your validation code works as intended

Anticipate

When seeking to anticipate incorrect or invalid inputs, there are commonly four types of input data we want to validate and test with our program:

- Normal data: This is the data you have designed your program to receive as input..
- Erroneous & abnormal data: Data your program should reject if received as input.
- Extreme data: The extreme values of what is valid to receive as an input.
- Boundary data: Data on the edge of what your program may receive as inputs. This tends to overlap with the extreme data. You include data just inside the boundary (test it is accepted) and data just outside the boundary (test it is rejected).

Example: An alarm app that requires you to enter a time for the alarm in 24 hour format, hh:mm.

Data type	Examples
Normal data	06:30, 07:00, 16:00
Erroneous data	09:00am, Seven, -17:00, 4 o'clock, 25:00, 8:75, 5pm
Extreme data	00:00, 23:59
Boundary data	00:00, 23:59, 24:00, 05:60

(Of course, you could modify the app's requirements so that it can correctly interpret some of the "erroneous data" so it can become "normal data")

The question then becomes: how do you want your app to respond to the various types of data? And does it respond in the manner you intend? This is what is required of a proper testing regime.

It may also be necessary to state some assumptions when devising your test data if the problem scenario is not sufficiently clear.

Validation strategies

Pre-read from textbook: 9.4 Validation and verification

Validation checks are the process of programmatically checking the input given to your program to ensure it meets basic criteria. The idea is to check what the user has input to detect possible mistakes before accepting it within your program.

Some of the typical checks involved are:

- Range checks - Is the input value within the permitted minimum and maximum range?
- Length checks - Is the input value within the permitted size length? eg: perhaps you only allow up to 30 characters for an email address. You will have encountered these kinds of limits on apps and websites many times, why do they exist? Typically it's because these systems save their data into a database, and databases require you to specify the size of each field so it can allocate the appropriate amount of disk space.
- Type checks - Is the input value of the correct type? ie: if you ask for an integer, but are given a string or a float.
- Character checks - Does the input value only contain permitted characters? Sometimes special punctuation symbols can cause problems.
- Format checks - Is the style of the input value according to expected rules? eg: dd/mm/yyyy format for a date.
- Presence checks - Has the input been provided if it is required? eg: not left blank.
- Algorithmic checks - We will consider this next lesson.

Validation exercise

The anticipate, validate and test process can also be used to uncover programming errors. A great example of the need for this kind of testing is from our "Making decisions" problem set question 2. About 40% of students submitted something like the following. If this code was properly tested, an error would be discovered.

Create a set of testing data (normal, erroneous, boundary) to seek to identify how this program does not behave in the manner you might expect.

```
side_a = int(input("Input side A of your triangle: "))
side_b = int(input("Input side B of your triangle: "))
side_c = int(input("Input side C of your triangle: "))

if side_a^2 + side_b^2 == side_c^2:
  print("This is a pythagorean triple.")
else:
  print("This is not a pythagorean triple :(")
```

Validation questions

1. The following pseudo code performs a validation check.

```
PRINT "Input a value between 0 and 100 inclusive"
INPUT Value
WHILE Value < 0 OR Value > 100
    PRINT "Invalid value, try again"
    INPUT Value
ENDWHILE
PRINT "Accepted: ", Value
```

(a) What is the name for this type of check? [1]
(b) Describe what is happening in this check? [2]

2. A programmer has written a routine to check that prices are below $10.00. These are the values used as test data:

      10.00   9.99   ten   -$2.00

(a) Explain why each was chosen.

3. A programmer has identified several fields entered by the user that require validation. Identify the validation checks suitable for each input, and provide test data that could be used for that validation.

	Validation checks suitable	Test data for validation
Email address		
Password		
HK phone number		
Date		
Currency		

For more: Activity 9.5, 9.6, 9.7, 9.8 (Textbook pages 119, 120).

8. Validation programming

Earlier we looked at the idea of an alarm clock app. The following is my attempt to validate the input as a valid 24 hour time.

```python
def check_valid_time(time):
    # Presence check
    # Length check - must be 5 characters
    if len(alarm) != 5:
        return False
    # Format check - look for colon character
    if alarm[2:3] != ":":
        return False
    # Type check - Must be numbers
    # Format check - numbers in correct positions
    if not alarm[0:2].isnumeric():
        return False
    if not alarm[3:5].isnumeric():
        return False
    # Range checks
    hours = int(alarm[0:2])
    minutes = int(alarm[3:5])
    if hours < 0 or hours > 23:
        return False
    if minutes < 0 or minutes > 59:
        return False
    # All checks passed
    return True

# Testing code
alarm = input("Enter alarm time as hh:mm >> ")
ok = check_valid_time(alarm)
print(ok)
```

Previously, you created testing data for a range of common input scenarios. They were:
- Currency
- Dates
- Hong kong mobile phone number (hint: check wikipedia for the rules of what constitutes valid mobile phone numbers in HK)
- Email address input

Your task: Select TWO of the above scenarios and create Python functions that would seek to validate their input.

9. Validation algorithms

Once you've validated the user input with range checks, format checks, length checks and so forth, the user may have still entered incorrect data!

Humans are rather error-prone when recording numbers. This is because there aren't set rules for noticing when you have incorrectly written a number like a spelling error would make obvious for written words.

For some special types of data, such as bank account numbers, it might be worth the extra hassle of one more additional check.

This additional check involves creating an algorithmic rule that allows the data to be used to check itself. This occurs by adding what is known as a **check digit**. This check digit can be automatically calculated from the rest of the number. If the calculation works, then there is a much greater likelihood that the number has been correctly entered (though mistakes can still occur).

Check digits

Check digits are used whenever it is considered important to minimise data entry errors of numbers.

Three common numbers you may be familiar with that contain check digit algorithms are:

- ISBN - The last digit of the ISBN is a check digit. The algorithm is explained on page 122 of the textbook.

- Credit cards - Most common credit cards use the LUHN algorithm to generate a check digit. The algorithm is documented at....
    - https://pbaumgarten.com/problems/problem-luhn.pdf

- HKID - The Hong Kong ID number is another check digit algorithm. The number in the bracket is generated by the algorithm. The algorithm is documented at...
    - https://access-excel.tips/hkid-check-digit/

Check digits exercise

Working in pairs or individually, select one of the problems: ISBN, LUHN or HKID.

1. Create a table to devise test data (normal, erroneous, extreme, boundary)
2. Decide on the input checks you will use (range, length, type, character, format, presence)
3. Use computational thinking to decide how you will calculate the check digit (decomposition, abstraction, pattern recognition, algorithm design)
4. Code it in Python
5. Use your test data - do your checks behave as they should?
6. Submit your written work (tests and checks) and Python programming.

10. Verification strategies

Learning objectives:
- Verification is an additional measure that can be used to ensure the accuracy of data before using it for processing.
- Verification involves having the user check the data

Introduction

What is verification and how does it contrast with validation?

- Validation checks is about the program conducting its own checks to identify erroneous data entry.
- Verification checks are about involving the user to manually check the input themselves and confirm it as correct.

Be aware this distinction between validation and verification can sometimes get a bit confused in industry and when reading resources online. The above difference is how the iGCSE course defines them so it is what you need to know. As wikipedia puts it so well, "In practice... the definitions of verification and validation can be inconsistent. Sometimes they are even used interchangeably". Be aware of this issue when searching online.

How to verify?

Verification is about ensuring the user has typed what they intended to type. There are a few ways a programmer could do this. Some common strategies include:

- Double entry - Have the user enter the information twice. This is used a lot with passwords and email addresses.
- Screen/visual check - Show the user the information entered and ask them to click to confirm it is correct. This is used a lot with online shopping prior to finalising a purchase.
- Check digit (notice that this can be used for validation and verification)

Verification exercise

As an exercise in using verification, I have provided the code to create a Windows application that will ask for email and password fields to be entered twice. This makes an ideal excuse to expose you to what is involved in creating a Windows style GUI with Python.

The provided code will get the windows GUI system up and running, but you will need to finish the `clicked()` function to perform the verification step.

https://pbaumgarten.com/igcse-compsci/distribute/unit-2-tkinter-verification-demo-app.pdf
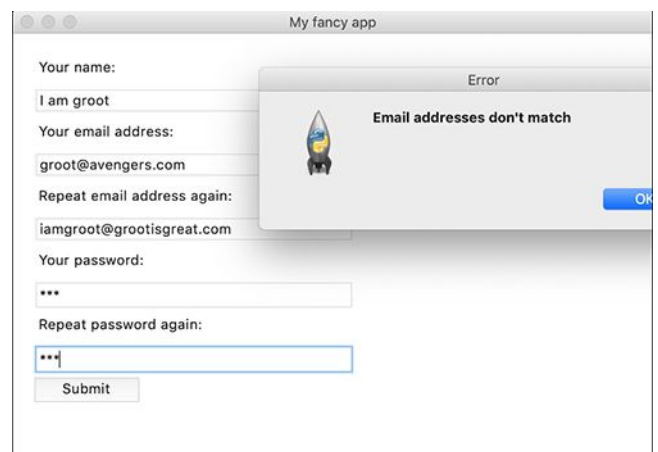
11. Designing and structuring clean code

Learning objectives

- Introduction to the idea of top-down design and structure diagrams
- Practice identifying subsystems and document it on a structure diagram
- Appreciation of the need to document algorithms

Introduction

Pre-reading from textbook

- 9.1 Introduction
- 9.2 Algorithms

Effective naming schemes

- Meaningful variable names - nouns
- Meaningful function names - verbs

Good use of consistent, meaningful naming schemes is part of the igcse mark scheme.

Effective code styles

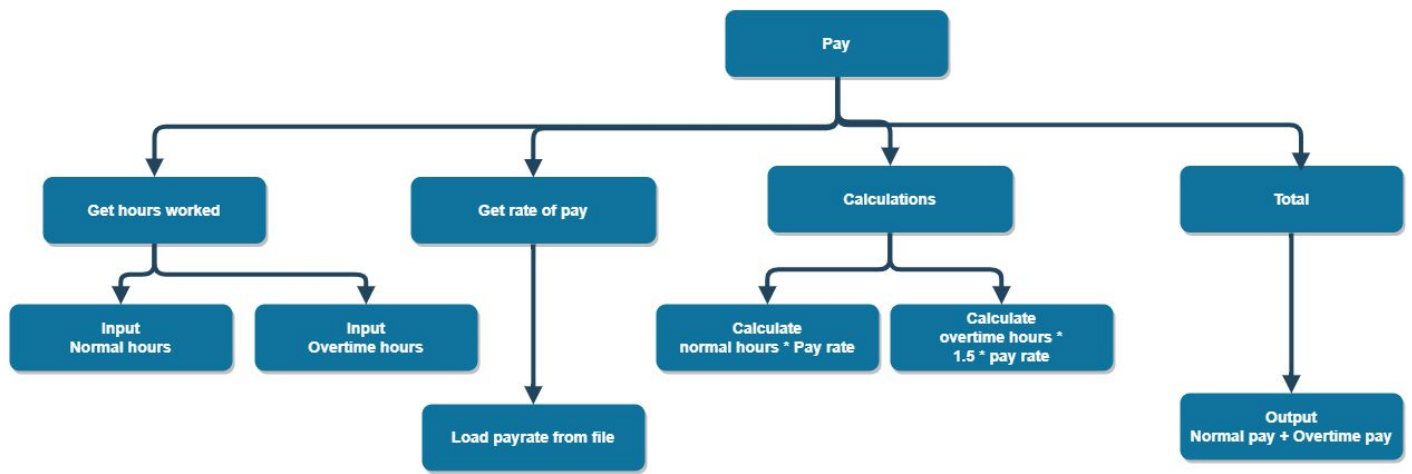- Consistent styling
- In code comments

Good use of a consistent layout style and in-code comments is part of the igcse mark scheme.

Top down design

Top-down design links back to the idea of decomposition. To quote from the textbook, "*each computer system can be divided up into a set of subsystems. Each sub-system can be further divided into subsystems and so on until each sub-system just performs a single action*" (p115).

This idea of dividing a system into its subsystems is known as top-down design, where at the apex you have your overall project or system, and then you create a hierarchical tree of subsystems from that.

This hierarchy can be drawn into a chart, which is known as a **structure diagram**. An example is provided below. There is also an example for an alarm app in the textbook on page 116.

```
                                    ┌──────────────┐
                                    │     Pay      │
                                    └──────┬───────┘
        ┌──────────────────┬──────────────┴──────────────┬──────────────────┐
        ▼                  ▼                              ▼                  ▼
┌──────────────┐  ┌──────────────┐            ┌──────────────┐    ┌──────────────┐
│ Get hours    │  │ Get rate of  │            │ Calculations │    │    Total     │
│ worked       │  │ pay          │            │              │    │              │
└──────┬───────┘  └──────┬───────┘            └──────┬───────┘    └──────┬───────┘
   ┌───┴────┐            ▼                   ┌────────┴────────┐         ▼
   ▼        ▼   ┌────────────────┐           ▼                 ▼  ┌────────────────┐
┌───────┐ ┌───────┐ │ Load payrate   │ ┌────────────┐ ┌──────────────┐ │ Output         │
│ Input │ │ Input │ │ from file      │ │ Calculate  │ │ Calculate    │ │ Normal pay +   │
│ Normal│ │Overtime│└────────────────┘ │ normal     │ │ overtime     │ │ Overtime pay   │
│ hours │ │ hours  │                    │ hours *    │ │ hours *      │ └────────────────┘
└───────┘ └───────┘                    │ Pay rate   │ │ 1.5 * pay    │
                                        └────────────┘ │ rate         │
                                                        └──────────────┘
```

The main question that might arise is "how do I know when I need to divide a system into new sub-systems"? For the purposes of this course, it is fairly safe to think of each box as a function. If it makes sense to write new functions, you are creating new sub-systems.

12. Edge cases are people too

Validating your inputs is crucial - just make sure you don't contribute to the further marginalisation of people in the process!

"The humans who sit on the margins of our products are the same humans who sit on the margins of society."

As companies push for scale and growth at breakneck pace they are weaponizing technology against groups that fall outside of their defined happy path.

The "happy path design" is not human-centered, it is business-centered. It's good for businesses because it allows them to move fast.

Designing for speed has trained us to ignore edge cases, and the overwhelming prevalence of homogenous teams made up of the least vulnerable among us (read: dudes) has conditioned us to center their life experience in our design process.

Today we are building massive platforms, with massive reach and impact, yet they are massively fragile. If we are honest with ourselves, these platforms represent a failure of design. Their success hinges on an intentional disregard for human complexity, and society pays the price for it.

Facebook claims to have two billion users…1% of two billion is twenty million. When you're moving fast and breaking things 1% is well within the acceptable breaking point for rolling out new work. Yet it contains twenty million people. They have names. They have faces. Technology companies call these people edge cases, because they live at the margins. They are, by definition, the marginalized.

While it may never have been explicitly stated, the inherent bias in our product development approach has meant that our underlying first principle has been to ignore human complexity.

Reference: https://modus.medium.com/who-pays-the-price-of-happy-path-design-36047f00c044
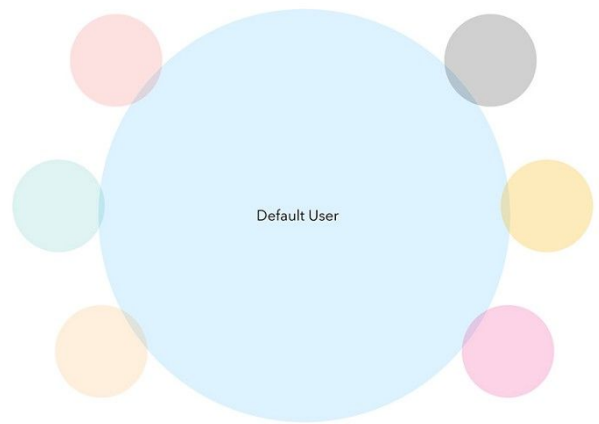
Fascinating reading

Falsehoods programmers believe about names (with examples)
- https://shinesolutions.com/2018/01/08/falsehoods-programmers-believe-about-names-with-examples/

Falsehoods programmers believe about prices (& currency)
- https://gist.github.com/rgs/6509585

Falsehoods programmers believe about gender
- https://gist.github.com/garbados/f82604ea639e0e47bf44

Falsehoods programmers believe about time
- https://infiniteundo.com/post/25326999628/falsehoods-programmers-believe-about-time

Validating email addresses
- http://mirroronet.pl/pub/mirrors/video.fosdem.org/2018/K.1.105%20(La%20Fontaine)/email_address_quiz.webm (poor quality audio but really interesting)
- After showing crazy examples of valid/not valid email addresses, his bottom line is that the only authoritative way to validate is to send a test email containing a response code or link (~ 9:30).

Falsehoods programmers believe about maps (and coordinates and geometry)
- http://www.atlefren.net/post/2014/09/falsehoods-programmers-believe-about-maps/
- https://wiesmann.codiferes.net/wordpress/?p=15187

Discussion questions

There are many subtle complexities that can be easily washed over in the rush to bring a product to market:
- Names
- Gender & sex
- Language
- Assumed wealth (assuming they have a phone number for instance)

1. What are some ways in which software could marginalise people by making assumptions about the above categories?

2. What other categories can you identify in which software marginalises people? Examples?

3. What steps could you take as a programmer to help ensure your products don't further marginalise people?

13. Debuggers

Introduce the debugger in VS Code

- https://code.visualstudio.com/docs/editor/debugging

Exercises

- TBA

14. Review

Six terms associated with programming and six descriptions are listed.

Draw a line to link each term with its most appropriate description.

Term	Description
Top-down design	Pre-written code to include in your own program to carry out a common task.
Structure diagram	Shows the steps representing an algorithm using various shapes of boxes.
Flowchart	Shows the hierarchy of the different components which make up a system.
Pseudocode	Shows the values of variables as you manually test your program.
Library routine	Breaks down a system into successively smaller pieces.
Trace table	Describes a program using a simplified high-level notation.

[5]

Some review questions can be found at:

- Chapter 9 review questions: Page 132-133.
- Chapter 10 review questions: Page 144-145.