Algorithms And Pseudocode

A compilation by C. Thompson

2019

Table of Contents

What is an Algorithm

An Algorithm, (after Al Kho-war-iz-mi a 9th century Persian mathematician) is an ordered sequence of unambiguous and well-defined instructions that performs some task and halts/stops in a finite time.

Let's examine the four parts of this definition more closely

- an ordered sequence means that you can number the steps (it's socks then shoes!)
- unambiguous and well-defined instructions mean that each instruction is clear, do-able, and can be done without difficulty
- performs some task
- halts in finite time (algorithms terminate!). If a planned solution does not finish, then it is **NOT** an algorithm.

Algorithms can be executed by a computing agent (see “Hidden Figures” (Anon., 2016)) which is not necessarily a computer.

Once an algorithm has been designed and perfected, it must be translated – or programmed – into code that a computer can read.

We create programs to implement algorithms. Algorithms consist of steps, where programs consist of statements. It is not necessary to code an algorithm. We use algorithms in everyday life.

Three Categories of Algorithmic Operations

Algorithmic operations are ordered in that there is a first instruction, a second instruction etc. However, this is not enough. An algorithm must have the ability to alter the order of its instructions. An instruction that alters the order of an algorithm is called a control structure or **construct**.

There are (fundamentally) three control structures as listed below.

1. **SEQUENCE** or sequential operations - instructions are executed in order
2. **SELECTION** or conditional ("question asking") operations - a control structure that asks a true/false question and then selects the next instruction based on the answer
3. **ITERATION** or repetition operations (loops) - a control structure that repeats the execution of a block of instructions

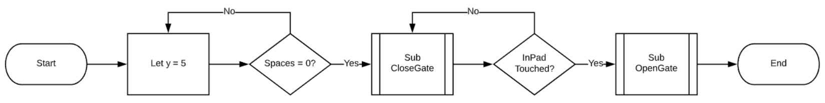
Unfortunately, not every problem or task has a "good" algorithmic solution. There are unsolvable problems:

- no algorithm can exist to solve the problem (Halting Problem)
- "hard" (intractable) problems - algorithm takes too long to solve the problem (Traveling Salesman Problem)
- problems with no known algorithmic solution

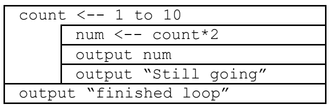
Representing Algorithms

There are many ways to represent an algorithm. Common methods for computing related solutions include:

- Flow Charts (e.g. Flowol)



- Nassi-Schneiderman Charts



```
count <-- 1 to 10
        num <-- count*2
        output num
        output "Still going"
output "finished loop"
```

- Pseudocode
  - natural language constructs modelled to look like statements available in many programming languages
  - There is no universal "standard" for writing pseudocode. The QCAA syllabus has some reference to using pseudocode and that has been included over the page.
  - Pseudocode is simply a list of instructions to perform some task. In this course we will enforce **four** standards for good pseudocode:

1. All algorithms will always start with a **`BEGIN`** and finishes with an **`END`**. Additional modules such as functions or procedures also contain these key words.
2. Each instruction should be unambiguous (that is the computing agent, in this case the reader, is capable of carrying out the instruction) and effectively computable (do-able).
3. Completeness. Nothing is left out.
4. Key words are written in **`BOLD UPPERCASE`**

Conventions for writing pseudocode (QCAA, n.d.)

KEYWORDS are written in bold capitals and are often words taken directly from programming languages. For example, **`IF`**, **`THEN`** and **`ELSE`** are all words that can be validly used in most languages.
Keywords do not have to be valid programming language words as long as they clearly convey the intent of the line of pseudocode.

Statements that form part of a repetition loop (**`FOR`** or **`WHILE`**) are indented by the same amount to indicate that they form a logical grouping.

In a similar way, **`IF,`** **`THEN`** and **`ELSE`** statements are indented to clearly distinguish the alternative processing paths.

The end of **`WHILE`** loop and **`IF`**, **`THEN`** and **`ELSE`** statements are explicitly indicated by the use of **`ENDWHILE`** and **`ENDIF`** at the appropriate points.

Pseudocode should clearly indicate what is happening at each step, including formulas of calculations.
For example:
`CALCULATE` `net` is not as clear as **`CALCULATE`** `net = gross – tax`.
Programmers prefer to use a more abbreviated version in which memory cells used to store the input are given program-like names; `num1 num2`
For example:

```
    INPUT num1
    INPUT num2
```

is preferable to

```
    INPUT first number
    INPUT second number
```

Pseudocode is best understood by looking at examples. Each example below demonstrates one of the control structures used in algorithms: SEQUENCE, SELECTION, or ITERATION operations. Also listed are all variables used at the end of the pseudocode.

SEQUENCE

SEQUENCE is a series of commands executed linearly or in sequence without any option to deviate down a different pathway. Every algorithm will most likely have elements of sequence regardless of what other constructs may be employed.

Example #1 - Computing Sales Tax: Write an algorithm in pseudocode to determine the task of computing the final price of an item after figuring in sales tax. Note the three types of instructions: **`INPUT`** (get), process/**`CALCULATE/ASSIGN`** (←) and **`OUTPUT`** (display)

```
BEGIN
INPUT itemCostPrice
INPUT salesTaxRate
CALCULATE salesTaxAmt ← itemCostPrice * salesTaxRate
CALCULATE itemSalePrice ← itemCostPrice + salesTaxAmt
OUTPUT itemSalePrice
END
```

Variables:

```
itemCostPrice = the cost price of an item (real)
salesTaxRate = the tax rate (GST)(real)
salesTaxAmt = calculated tax (Item cost times the Tax Rate)(real)
itemSalePrice = the cost price + the sales tax (real)
```

Line 3 above is an example of an **ASSIGNMENT**; that is, we **assign** a value to, (in this case), a variable called `salesTaxAmt`. The value **assigned** is result of the calculation on the right side of the equals. When we use **`INPUT`** as in lines 1 and 2, we are also assigning values to the variables `itemCostPrice` and `salesTaxRate`.

We extract and list all variables used in our pseudocode. This will be useful when translating pseudocode into a programming language. Note that variable names should be written in `camelCase` which always begins with a lower-case letter. If the variable name is “composed” of two or more words, no space is used, and each subsequent word begins with a capital or upper-case letter.

Notice after each variable description, there is the word (e.g. real), indicating that a particular variable will store data containing (potentially) decimal places, in other words “real” numbers. In Python, real numbers are described as **Float** or floating-point decimals.

A list of common data types you will use in writing pseudocode are:

- Real – number containing decimal places
- Integer – whole numbers only
- String – a group of characters (could be letters, numbers, combination of both and other characters)
- Boolean – has only the value of either True or False. Often used to control a loop or selection

There are some other less commonly used data types which will be referenced as required.

Activity 1

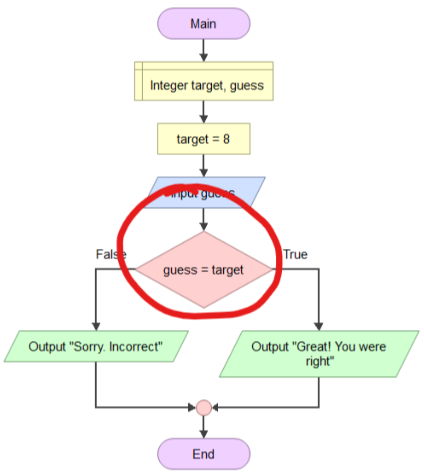
Determine suitable variable names for the following situations. For each variable, determine the most suitable data type:

A. stores a loop counter value for computer game
B. stores a pet's name and their owner's name for a Vet surgery
C. stores member number, registration number, owner's name, owner's address for a car, number of cylinders, engine size, original car colour for the RACQ breakdown service.

SELECTION

SELECTION involves control flowing along different pathways dependent upon the value of a decision.

In a flowchart, this is often shown as a decision box (a diamond) with two pathways flowing from it and the choice being the answer to the decision box question.



Example #2 - Computing Weekly Wages: Gross pay depends on the pay rate and the number of hours worked per week. However, if you work more than 40 hours, you get paid time-and-a-half for all hours worked over 40. Create an algorithm in pseudocode to compute the gross pay given pay rate and hours worked. Hours worked can only be recorded as whole hours.

```
BEGIN
INPUT hoursWorked
INPUT payRate
IF hoursWorked ≤ 40 THEN
   CALCULATE grossPay ← payRrate * hoursWorked
ELSE
   CALCULATE grossPay ← payRate * 40 + 1.5 * payRate * (hoursWorked – 40)
ENDIF
DISPLAY grossPay
END
```

Variables:

```
hoursWorked = number of hours worked (integer)
payrate = rate of pay per hour (real)
grossPay = gross pay (real)
```

This example introduces the conditional or **SELECTION** control structure. On the basis of the true/false question asked in line 3, we execute line 4 if the answer is True; otherwise if the answer is

False we execute the lines subordinate to line 5 (i.e. line 6). In both cases we resume the pseudocode at line 8.

Note the use of `ENDIF` to signify the end of the SELECTION construct. Thus, everything between the `IF` statement and the `ENDIF` statement is part of the SELECTION construct.

The condition tested (`hoursWorked ≤ 40`) always results in either a True or False value. Such conditions are called **BOOLEAN** conditions (which, like Boolean data types can only be either True or False).

Boolean variables can also form part of a Boolean condition. The following examples of Selection conditions are all equivalent.

```
IF continue = True THEN
    OUTPUT ('Game will continue')
ENDIF
```

Is equivalent to …

```
IF continue THEN
    OUTPUT ('Game will continue')
ENDIF
```

Note that both examples do not include an ELSE statement. Sometimes there is no option suitable if the Boolean condition is false.

Using `IF` and `ELSE` gives two possible choices (paths) that a program can follow. However, sometimes more than two choices are wanted. To do this, the statement `ELSE IF` is used.

This simple algorithm (BBC, 2019) prints out a different message depending on how old you are. Using `IF`, `ELSE` and `ELSE IF`, the steps are:

```
BEGIN
INPUT age
IF age ≥ 60 THEN
    OUTPUT ('You are aged to perfection!')
ELSE
    IF age = 50 THEN
        OUTPUT ('Wow, you are half a century old!')
    ELSE
        OUTPUT ('You are a spring chicken!')
    ENDIF
ENDIF
END
```

Note that there must be an endif for every if used and indentation is critically important. You can see that lines 2, 4 and 10 are aligned and 5, 7 and 9 are indented and aligned as these represent the inner **NESTED** condition. Fundamentally, there is no limit to how many nested conditions you can develop.

In situations where there is a series of `IF .. ELSE IF ..` statements such as below:

```
BEGIN
INPUT grade
IF grade ≥ 100 THEN
    OUTPUT (“Perfect Score”)
ELSE IF grade > 89 THEN
    OUTPUT (“Grade = A”)
ELSE IF grade > 79 THEN
    OUTPUT (“Grade = B”)
ELSE IF grade > 69 THEN
    OUTPUT (“Grade = C”)
ELSE IF grade > 59 THEN
    OUTPUT (“Grade = D”)
ELSE
    OUTPUT (“Grade = F”)
ENDIF
END
```

We can see that as control flows from one **IF** statement to the next, evaluating the value of grade at each statement. When this situation arises (i.e. **evaluating the same expression**) we can replace the **IF.. THEN.. ELSE IF ..** with a **CASE** statement (sometimes in pseudocode known as a **SWITCH** statement). (Parkland College Business/Computer Science & Technologies, 2010)

```
BEGIN
INPUT grade
CASE OF
    grade ≥ 100
        OUTPUT (“Perfect Score”)
    grade > 89
        OUTPUT (“Grade = A”)
    grade > 79
        OUTPUT (“Grade = B”)
    grade > 69
        OUTPUT (“Grade = C”)
    grade > 59
        OUTPUT (“Grade = D”)
```

```
 ELSE
        OUTPUT ("Grade = F")
END CASE
END
```

Note that not all languages implement a **CASE** statement and you must use the **IF..THEN..ELSE IF** control structure.

ITERATION

ITERATION occurs when control flows around a loop construct. In other words, dependent upon the value of a loop test condition, control will flow down a set of instructions until it reaches the end of those within the loop and then revert to the top of the loop to execute again (if appropriate). Subordinate commands within the loop are identified by being indented under the loop test condition.

Example #3 - Computing a Quiz Average: Write a pseudocode solution to calculate your quiz average from a number of quizzes.

```
BEGIN
INPUT numberOfQuizzes
INIT sum ← 0
INIT count ← 0
WHILE count < numberOfQuizzes DO
    INPUT quizGrade
    CALCULATE sum ← sum + quizGrade
    CALCULATE count ← count + 1
ENDWHILE
CALCULATE average ← sum / numberOfQuizzes
DISPLAY average
END
```

Variables:

```
numberOfQuizzes = number of quizzes undertaken (integer)
sum = running total of quiz scores (real)
count = loop counter measuring times of times through loop (integer)
quizGrade = score on a particular quiz (real)
average = the average of all quizzes (real)
```

Note that `sum` and `count` are assigned the value zero (0) in lines 2 and 3. We call this **initialising** the value of the variables. What it means we set an initial or starting value to a known value. We identify initializing by adding the keyword **INIT** before the assignment.

This example introduces an **ITERATION** or loop control statement. As long as the condition in line 4 is True, we execute the subordinate operations 5-7. When the condition becomes False, we resume the pseudocode at line 9.

This is an example of a **pre-test** or `WHILE DO` iterative control structure (executing the inner block of instruction while the Boolean condition is True. There is also a **post-test** or `REPEAT UNTIL` iterative control structure which executes a block of statements until the condition tested at the end of the block is True. Not all programming languages (e.g. Python) implement a `REPEAT UNTIL` construct, however it can be used in pseudocode. In nearly all cases a `REPEAT UNTIL` can be rewritten as a `WHILE DO`.

Both `WHILE DO` and `REPEAT UNTIL` are formally called **INDEFINITE** iteration (with `WHILE DO` specifically a pre-tested indefinite iterative loop and `REPEAT UNTIL` being a post-tested indefinite iterative loop). They are called **INDEFINITE** as the end of the loop depends entirely upon a Boolean condition that may be different each time the program is executed.

There is also a FIXED iterative construct used when the loop instruction must be executed a known or FIXED number of times. When the number of iterations is known, a `FOR DO` loop is used. This is called **DEFINITE** (or known) iteration.

The above example could easily be redeveloped as a **DEFINITE** loop as follows:

```
BEGIN
INPUT numberOfQuizzes
INIT sum ← 0
FOR count ← 1 to numberOfQuizzes DO
    INPUT quizGrade
    CALCULATE sum ← sum + quizGrade
ENDFOR
CALCULATE average ← sum / numberOfQuizzes
DISPLAY average
END
```

In this example, as the number of quizzes is input early in the pseudocode sequence, there is a known fixed number of iterations or loops that will be accessed. Line 3 tells the loop to begin at 1 and continue to the number of quizzes entered.

So, the above example is probably best designed as a fixed or DEFINITE iterative loop, i.e. a `FOR DO` loop. Let's look at some other examples that would specifically be best designed as an INDEFINITE iterative loop.

(Thompson and Shuttlewood, 2008) Example #4: A girl has saved $100 to spend on Christmas presents. She requires an algorithm which will accept the value of presents that are to be purchased and show the amount of money remaining. The algorithm will stop when the value of the present exceeds the money remaining.

This example is best done as a pre-tested **INDEFINITE** loop (`WHILE DO`) because:

1. We must test if she has enough for her first purchase before allowing her to purchase and subtract that amount from her savings. If she tries to purchase something greater than the initial $100 then the algorithm must flag that with the appropriate message. Hence, we are **pre-testing**.
2. It is **INDEFINITE** because we do not know beforehand how many times she will be able to purchase presents as it is based upon the amount left and the cost of each presents; factors unknown at the beginning.

```
BEGIN
INIT money ← 100.00
INPUT presentCost
WHILE presentCost ≤ money DO
   CALCULATE money ← money - presentCost
   OUTPUT money
ENDWHILE
INPUT presentCost
OUTPUT ('Not enough money left for that purchase')
END
```

Variables:

```
   money = initial money amount available (real)
   presentCost = the value of each present to be purchased (real)
```

Example #5: A boy is saving his pocket money he earns from odd jobs around the home to buy a skateboard. An algorithm is required to record the money that is saved and output a message when the target is reached.

This example is best done as a post-tested **INDEFINITE** loop (`REPEAT UNTIL`) because:

1. The solution must first accept an amount of saving and add this to the total saved. It must then compare the total saved to the target amount and send an appropriate message when reached. Since the test condition happens after the initial deposit, we are **post-testing**.
2. It is **INDEFINITE** because we do not know beforehand how many times he needs to deposit money as it is based upon if he has reached the total required and how much he deposits each time; factors unknown at the beginning.

```
BEGIN
INPUT targetAmount
INIT total ← 0
REPEAT
   INPUT money
   CALCULATE total ← total + money
UNTIL total ≥ targetAmount
```

```
OUTPUT ('Target has been reached')
END
```

Variables:

```
    money = money deposited (real)
    targetAmount ← target amount required for skateboard purchase (real)
    total ← running total of savings
```

Now, it was mentioned earlier, that some programming languages (such as Python) do not allow for a post-tested iterative construct such as **REPEAT UNTIL**. In most cases, these can be re-written as pre-tested **WHILE DO** constructs. The above example can be rewritten by pre-testing the `total` against the `targetAmount` before entering the loop the first time. Because initially the boy will not have the required savings, control will enter the loop.

```
BEGIN
INPUT targetAmount
INIT total ← 0
WHILE total < targetAmount DO
    INPUT money
    CALCULATE total ← total + money
ENDWHILE
OUTPUT ('Target has been reached')
END
```

MODULES

Very often, there is a need to create algorithmic sections that are required to be executed several times at various times. While the instructions can be re-written where appropriate, there is a definite waste of resources in doing so. A more efficient way is to create the segment as a Module or Method. Modules or Methods are used to group segments for a specific purpose.

Most languages identify two types of modules; **FUNCTIONS** and **PROCEDURES**. Some languages group them together. For the purposes of these notes, one way to distinguish between them is to look at what they do. Mostly, a **procedure** will be a selection of code to perform a specific task (perhaps print a formatted receipt of sales). On the other hand, a **function** is a section of code that is called upon to perform some operation and return a single value to the part that called it (perhaps determine the GST on a sale). (StackExchange, 2013)

Functions and Procedures are CALLED by the main pseudocode when required. We can see their use in the following example.

Example #6 – an algorithm is required to calculate GST on goods purchased and print a receipt showing total cost (inc GST) as well as the GST separately.

```
FUNCTION calculateGST (saleCost)
    CALCULATE gst ← saleCost * 0.10
    RETURN gst
```

```
END FUNCTION
PROCEDURE printReceipt
    OUTPUT ('Store Headings')
    REPEAT
            OUTPUT itemName
            OUTPUT itemCost
            OUTPUT total
            OUTPUT gst
    UNTIL no more items
END PROCEDURE


INIT total ← 0
INIT gst ← 0
WHILE more items DO
    INPUT itemCost
    CALL calculateGST (itemCost)
    CALCULATE Total ← total + itemCost + gst
END WHILE
CALL printReceipt
```

QCAA Note

The document “Supporting resource: Representing algorithms with pseudocode” released by QCAA in late 2019 suggests that modules (e.g. functions and procedures) should be described in algorithms as follows:

Procedures, subroutines, methods or functions:

```
BEGIN name
   statements
END name
```

Where **NAME** is the name of the module. This document assumes either methodology is acceptable.

Notice that the function `calculateGST` is called from within the **WHILE DO** loop a number of times (while there are items sold) and each time the `itemCost` is sent to the function (and referenced in the function as `saleCost`) to calculate the GST amount for that item. The value of the GST is then returned to the main part of the algorithm and used to determine the running total. This continues until no more items are sold. The loop completes and control is then passed to a procedure call `printReceipt` for final printing of the sales receipt.

The variable in the function `calculateGST` (`saleCost`) is called a **parameter** and it receives its value from the main part of the pseudocode that **CALLS** the function. The value passed is called an **argument**.

Finally, there is some contention that Functions and Procedures are one and the same and in a sense this is correct. However, if we distinguish them by their ability (or not) to return a value then we can make a valid comparison and recognise their uses.

Pseudocode Exercises

NB – in any question where you are required to complete or develop an algorithm in pseudocode, complete the question by listing the variables, their description (as above) and data types.

1. Trace the following program plans and describe the actions that each performs:

```
a)  INPUT age
    CALCULATE lucky ← 365 * age
    OUTPUT lucky
b)  INPUT number
    CALCULATE a ← number + 10
    CALCULATE b ← number * 10
    CALCULATE c ← a + b
    OUTPUT c
c)  INPUT litres
    INPUT rate
    CALCULATE cost ← litres * rate
    OUTPUT cost
```

2. Carefully read the following algorithm. Describe what actions it performs.

```
INPUT age

INPUT height

INPUT weight

CALCULATE lucky ← age + height – weight

OUTPUT lucky
```

3. Brian's Curtains charge $13 per metre of material and a fixed charge of $25 when making curtains for customers. A program is to be created to perform the calculations for the shop owner.
   Copy and complete the following algorithm and design an interface appropriate to the problem.

```
INPUT length

CALCULATE cost ← ________ * 13 +

OUTPUT ______
```

4. An insurance company pays its employees a salary of $100 per week plus $15 per hour for every hour worked during the week.
   a) Plan a program to calculate the amount of pay owed to employees.
   b) List the variables, what they are to be used for and the data types for each.

5. A contractor who erects fences charges his customers $104 per metre for materials and $25 per metre to erect the fence. He wishes to have a computer program to assist in the calculation of his accounts. Develop an algorithm in pseudocode to be used for this purpose. List all relevant variables, descriptions and data types.

6. A painter requires a program to calculate the number of litres of paint needed for a job. One litre of paint will cover 16 square metres of wall. Plan a solution that will accept the length and height of the wall, determine the number of square metres to be painted and then show the number of litres of paint needed.

7. Read the following algorithm and describe the task it performs.

```
total ← 0
INPUT people
FOR count ← 1 to people DO
    INPUT donation
    CALCULATE total ← total + donation
ENDFOR
OUTPUT total
```

8. A program is needed to find the total number of runs scored by the 11 players in a cricket team. Complete the following algorithm to perform this task.

```
total ← 0
FOR player ← ___________________________
    INPUT _________
    CALCULATE total ←
ENDFOR
OUTPUT ________
```

9. Read the following algorithm and describe the task it performs.

```
INPUT password
WHILE password <> 'xyz' DO
  OUTPUT 'Wrong Password, try
  again'
  INPUT password
ENDWHILE
OUTPUT 'Access Permitted'
```

10. Read the following algorithm and describe the task it performs.

```
INPUT passwd
count ← 1
WHILE passwd <> 'xyz' and count < 3 DO
  OUTPUT ('Wrong Password, try again')
  INPUT passwd
```

```
        CALCULATE count ← count + 1
    ENDWHILE
    IF passwd = 'xyz' THEN
        OUTPUT ('Access Permitted')
    ELSE
        OUTPUT ('Access Denied')
    ENDIF
```

11. Read the following algorithm and describe the task which it performs.

```
INPUT age
INPUT height
IF age > 15 and height < 180 THEN
    OUTPUT ('Unsuitable for basketball')
ELSE
    OUTPUT('Consider basketball')
```

12. A algorithm has to read in the following information on a person: AGE, SEX, HEIGHT and SPORT. Create a test expression to identify the following people:
    a) Males over 16;
    b) People over 15 or taller than 165cms;
    c) Football players over 15;
    d) Males or females over 16 and less than 150 cms tall;
13. In JCs Computing Company, there are several junior salespeople who are paid less wages than the seniors. Employees under the age of 18 are paid $10.75 per hour while others are paid $15.25 per hour. A solution is needed which will calculate the wages due to an employee.

    Copy and complete the following algorithm for such a program.

```
INPUT age
INPUT hoursWorked
IF _________________________ THEN
      CALCULATE wage ← ___________________
_____
      _________________________________

OUTPUT wage
```

14. An algorithm is needed to calculate the amount and cost of materials which are needed when framing a picture. The program must accept the dimensions of the frame and calculate the cost of the frame and glass given that frames cost $7.25/m each and glass costs $4.50/sq m

When a procedure is defined which returns a single value of a simple data type, for example the cost of framing as in the above problem, it is better to use a particular form of procedure called a ***function.*** The use of functions provides some distinct advantages to the user.

For this solution you will need to create a **`FUNCTION`** `frame` which will accept two parameters `size1` and `size2` (from the main algorithm `length` and `width`) and **`RETURN`** `result` (determined by the formula `2*(size1 + size2)*`**`FRAMECOST`**) and a **`FUNCTION`** `glass` which also accepts two parameters, `size1` and `size2` (from the main algorithm `length` and `width`) and **`RETURN`** result (determined by the formula `(size1 * size2)*`**`GLASSCOST`**).

`FRAMECOST` and **`GLASSCOST`** are constants initialized at the start.

```
FUNCTION frame (size1, ______)

CALCULATE amount ← 2*(_____ + _______)

__________ result ← amount * __________

RETURN ___________

END FUNCTION


FUNCTION ______________ (size1, ______)

CALCULATE area ← (_____ * _______)

__________ result ← area * __________

RETURN ___________

_________________


INIT FRAMECOST ← 7.25

INIT _________ ← 4.5

INPUT length

INPUT width

____________ totalCost ← CALL frame (length, _____) + CALL
________(_____,_____)

OUTPUT totalCost
```

Bibliography

BBC, 2019. *Bitesize.* [Online]
Available at: https://www.bbc.com/bitesize
[Accessed 1 March 2019].

Cal Poly College of Engineering, 2003. *Pseudocode Standard.* [Online]
Available at: http://users.csc.calpoly.edu/~jdalbey/SWE/pdl_std.html
[Accessed 4 March 2019].

Parkland College Business/Computer Science & Technologies, 2010. *Case Structure Pseudocode.* [Online]
Available at: http://virtual.parkland.edu/kcouch/CIS122/Week8/case_psuedocode.htm
[Accessed 4 March 2019].

QCAA, n.d. *Digital Solutions 2019 v1.2.* [Online]
Available at:
https://www.qcaa.qld.edu.au/downloads/portal/syllabuses/snr_digital_solutions_19_syll.pdf
[Accessed 1 Mar 2019].

StackExchange, 2013. *Pseudocode: \Function vs. \Procedure?.* [Online]
Available at: https://tex.stackexchange.com/questions/145736/pseudocode-function-vs-procedure
[Accessed 4 Mar 2019].

Thompson and Shuttlewood, 2008. *Algorithms, Programming and Delphi.* Mackay: s.n.