IGCSE Comp Sci: Arrays & Iteration Workbook

Eason Qin (eason@ezntek.com)

February 2025

1 Introduction

VERY IMPORTANT! This workbook only covers Iteration, Arrays (1 and 2D), This is not comprehensive (yet).

I'll try my best to keep this short.

This document is designed to help you with Iteration, 1 and 2D arrays, functions and procedures in your IGCSE Comp Sci syllabus. I have been asked to write worksheets on this topic by my teacher, and I will try my best.

THIS WORKSHEET PACK IS NOT FOR THE FAINT-HEARTED. They will not be easy at the end. These are IB Comp Sci style; no silly MCQs, no silly short or long paragraph essay answers/definitions; you will only be solving mini problems to hone your skill. You may also need to justify your problem-solving skills. For the best effect, *Do the worksheet twice, in Pseudocode on this sheet first and in Python to see it in action.* Please use the Python to Pseudocode reference for help.

NOTE: everything in angle brackets (`< >`) like (`<counter>`) is a *placeholder* for something; the thing you are to replace the placeholder with is described on the inside. For that example, instead of writing `<counter>`, you write `MyCounter`, or something similar. This document is also licensed under the CC BY-SA-NC 4.0 Int'l License.

Have fun, dear traveller.

1.1 Prerequisites

You need to know:

- comments (`// this is a comment`)
- variables, constants, and pseudocode types
- if statements (conditionals)
- types of comparisons (`>= <= > < = <>` etc.)

2 Iteration

Iteration in programming allows you to **execute multiple statements repeatedly**, based on a *condition*. In practice, this may mean printing your name 20 times or explaining to someone a concept repeatedly until they understand.

In pseudocode, there exists many types of loops.

2.1 For Loops

For loops look as follows:

```
FOR <counter> <- <begin> TO <end>
    <your code goes here>
NEXT <counter>
```

These loops are also called *count-controlled*, because a *counter* ***controls*** the behavior of the loop. *The loop runs as long as if the counter is not at the ending value.*

To illustrate, here's an example:

```
FOR Counter <- 1 TO 5
    OUTPUT "Eason Qin"
NEXT Counter
```

Note: the inside of the loop is also known as the body.

The code above prints the text `Eason Qin` to the console 5 times. A counter is kept, and the value goes from 1 to 5. Each time the code inside the loop runs, the counter is set to the current value; on the first time the code is run[1] it will be the beginning value of 1, then after `OUTPUT "Vedaant Atreya"` it will be 2, then 3, then 4, then 5.

In fact, we can even output the counter to prove that this is happening.

```
FOR Counter <- 1 TO 5
    OUTPUT Counter, " Eason Qin"
NEXT Counter
```

Which outputs the following text:

```
1 Eason Qin
2 Eason Qin
3 Eason Qin
4 Eason Qin
5 Eason Qin
```

[1] First Iteration.

For loops also support *steps*. This allows a for loop's counter to increment by a number other than one. If i wanted to print all the odd numbers till 10:

```
FOR Counter <- 1 TO 10 STEP 2
    OUTPUT Counter
NEXT Counter
```

Which outputs:

```
1
3
5
7
9
```

Note that 11 is not printed, as 11 is past the end, which is 10.

2.1.1 Exercises

For the following problems, write pseudocode.

1. Write code to print out *your* name 5 times.

2. Write code to print the numbers 3 till 7 with a for loop.

3. Write code to print all the even numbers between 1 to 10. *Hint: don't begin at one.*

4. Print the times table for one number. If the number is 5, it should output 5, 10, 15, 20 etc.

5. Write code to print numbers between 1 to 10 backwards, i.e. `10, 9, 8, 7,` etc.

2.2 Nested Loops

Nested loops in essence are loops in loops; they allow you to repeat statements repeatedly.

The following pseudocode illustrates this.

```
FOR Outer <- 1 TO 10
    // The inner loop.

    FOR Inner <- 1 TO 3
        OUTPUT "Inner"
    NEXT Inner

    OUTPUT "Finished iteration ", Outer
NEXT Outer
```

Which produces this output:

```
Inner
Inner
Inner
Finnished iteration 1
Inner
Inner
Inner
Finnished iteration 2
Inner
Inner
--- output truncated ---
```

Note: truncated means "end cut off".

2.2.1 Exercises

For the following problems, write pseudocode.

1. Print the numbers 1-5, then 2-5, 3-5, and then 4-5. *hint: you can use a variable like the counter as the begin/end value.*

2. Using your answer to exercise 2.1.1 question 4, print the times tables for all numbers, 1 through 12.

2.3 While Loops

While loops are similar to for loops. They typically look like this:

```
WHILE <condition> DO
    <code>
ENDWHILE
```

They allow you to repeat statements inside the loop body, like for loops. However, while loops give you more flexibility, as you can have a custom condition that dictates when the loop should run.

Consider the following example:

```
FOR Counter <- 1 TO 5
    OUTPUT "I love programming!"
NEXT Counter
```

With a while loop, it would look as follows:

```
1 DECLARE Counter:INTEGER
2 Counter <- 1
3
4 WHILE Counter <= 5 DO
5     OUTPUT "I love programming!"
6     Counter <- Counter + 1
7 ENDWHILE
```

Confusing, right?

Well, lines 1 and 2 in the prior example simply sets the beginning value of the counter, as the for loop would do; it would set counter to 1. Then, while the counter is less than or equal to 5, it prints `I love programming!`. Line 6 is actually where the magic happens; the counter is incremented by one, just as the for loop does.

In fact, this is how for loops under the hood; they are almost identical to while loops in behavior; it's just that while loops give you more control over the condition.

```
DECLARE UserNumber:INTEGER
DECLARE Result:INTEGER

UserNumber <- 0

// the <> symbol means not equal to.
WHILE UserNumber <> -1 DO
    OUTPUT "Please enter a number, or -1 to stop."
    INPUT UserNumber

    OUTPUT "Your number is: ", UserNumber
ENDWHILE
```

In the prior example, we use a while loop to do something *while* the condition holds true. Breaking down the code:

- Initially, `UserNumber` is set to 0.
- The loop checks if the user number is not equal to one.[2]
- If it holds true, execute the code. The code asks the user to enter a number. The code outputs it again.
- Now `UserNumber` may have changed, which it has; so the condition is checked again. If the condition does not hold true anymore (the user entered `-1`), the loop exits, and so does the program. If not, the code is ran again until the condition must be checked.

Therefore, we can deduct that this loop is controlled by a condition and not a counter (although it can!); giving us more flexibility.

2.3.1 Exercises

1. Write code to count from 1 to 10, but with a while loop.

____

____

____

____

____

2. Write code that counts from 20 to 0 in 2s, backwards (i.e, `20, 18, 16`, etc.) with a while loop.

____

____

[2]Advanced learners: would this mean that this is a post-condition or pre-condition loop?

3. Write code that greets users. Ask for users' names and say **`Hello, User!`**. If the user entered **`exit`**, quit the loop. Use a while loop.

4. Write code that checks if numbers are valid. If the number is greater than 5 and less than 20, tell the user they were right, and keep asking. If the user entered anything outside that range, quit the loop. Use a while loop, and optionally a variable that keeps track of if the user was correct or not.

2.4 Repeat-until loops

Note: these do not exist in Python; one may argue Python killed the idea of needing post-condition loops entirely. I argue post-condition loops are good, but I digress.

These are actually very similar to while loops, and the name is quite intuitive[3]. They look as follows:

```
REPEAT
    <code>
UNTIL <condition>
```

There is a key difference between a while and repeat-until loop, however. Consider the example from before:

```
DECLARE UserNumber:INTEGER
DECLARE Result:INTEGER

UserNumber <- 0
WHILE UserNumber <> -1 DO
    OUTPUT "Please enter a number, or -1 to stop."
    INPUT UserNumber

    OUTPUT "Your number is: ", UserNumber
ENDWHILE
```

The code here:

- Declares and initializes the variables.
- Checks the condition.
- Runs the code in the body if its true.

However, we could write this with a repeat-until loop:

```
DECLARE UserNumber:INTEGER
DECLARE Result:INTEGER

REPEAT
    OUTPUT "Please enter a number, or -1 to stop."
    INPUT UserNumber

    OUTPUT "Your number is: ", UserNumber
UNTIL UserNumber = -1
```

There are 2 key differences here.

[3]It repeats until something happens.

- *The code runs at least once*; regardless of the condition, it must hit the line where the condition is checked (line 9 in this case).
- The condition is evaluated at the end. What this means is that the code is ran, *then the condition is checked AFTERWARDS*. This is unlike the while loop, where the condition is checked first.
- The condition is *negated*, or flipped. The repeat-until loop does something *until the condition is true*, i.e. if the loop is running, the condition must not be true, or else the until part would be satisfied!

This is also where a large advantage of a repeat-until shows. In the while loop example, on line 4, the variable must be set to 0, as the condition that needs the variable is checked, the user did not enter an input yet. However, with a repeat-until, the code is ran at least once, so the user has a chance to input something, only then will the variable be used.

2.4.1 Exercises

1. Write code to let the user guess your home country. Save your home country in a constant. Keep letting the user try until their answer matches yours. Use a repeat-until loop.

2. Write code to let the user enter the temperatures in their city. Let the user enter values until they enter a temperature too hot, i.e. above 40 degrees celsius. When it is too hot, also output ”Wow, your city’s temperature is so hot!” before the code stops executing. You may use a variable to keep track of the validity of the user’s input. Use a repeat-until loop.

3 Arrays

Arrays are data structures[4] whereby you can store lists of information. The cats in your house or the people, a row of tall lockers and the names of their owners, or a list of data you need to sort.

3.1 Important Information

In pseudocode:

- Arrays in pseudocode are ***static***. This means that the length *cannot change*. No appending, no removing, etc.
- You must declare the beginning and end of the array. All the values will be set to uninitialized; i.e. invalid.
- You can freely index any element despite if it is initialized or not and write to it.

and in Python:

- They are called lists[5].
- They are dynamic, meaning you can change the length of the array. You can add items after the end to change the length and remove them.
- You do not need to and in fact cannot declare the beginning and end of the list.
- You cannot freely index any element. It must be initialized first, or at least be set to `None`.

NOTE: I will refer to arrays and lists as the same thing in pseudocode, interchangeably.

3.2 1D arrays

3.2.1 Declaration, Indexing, and Population

Declaring an array is trivial:

[4] A structure by which you can store data!

[5] There are arrays that behave like pseudocode arrays in Python in a library; but please do not use them. They are archaic, primitive and provide no benefit over lists.

```
DECLARE <variable>:ARRAY[<begin>:<end>] OF <type>
```

And as an example:

```
// I want 3 cats, so I give space for elements between 1 and 3.
DECLARE CatNames:ARRAY[1:3] OF STRING

DECLARE Scores:ARRAY[1:8] OF INTEGER
```

When you work with arrays, you also have to access the data stored within them. This process is known as *indexing*.

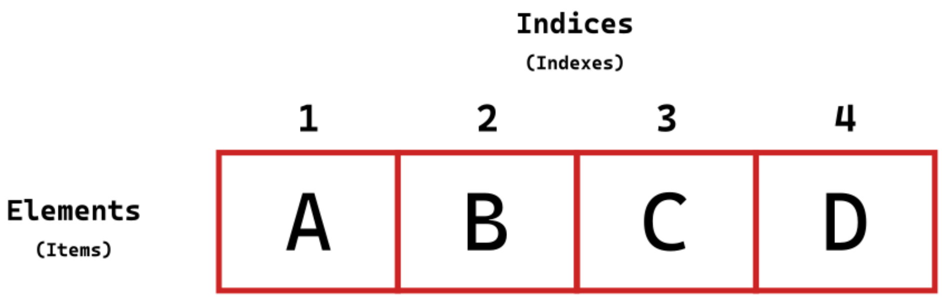

Figure 1: A breakdown and representation of an array.

In figure 1, the index is the nth element. Indices in other languages like Python begin at 0, and for a very good reason[6]. However, you just need to remember that indices start at 1 in pseudocode.

To index that array (let's call it **`Array`**), one may write **`Array[<index>]`**. In code, it looks like

```
OUTPUT Array[2] // gets the second item in the array.
```

In fact, you may also write to the array this way:

```
Array[2] <- 'F' // sets the second element to the character F.
```

To wrap this new knowledge up, let us create an array of food items.

[6]Advanced learners: this is due to its heritage in older languages like C. Arrays are actually not values that stand by themselves, as the size isnt promised to be some value. Attempting to remove/modify the array will cause memory to overlap, which is very bad. Instead, an array is simply a *location in memory where data is*. The location in memory will have the array's data; uniformly sized data that are equally spaced out. Therefore, if you were to get the first item of a list at address 4000, the first item would be 4000+0. However, the second one would be 1 address away, so 4000+1, then 4000+2, etc. This is why arrays begin at 0; it is simply the *offset* from the beginning of the memory address.

```
DECLARE FoodItems:ARRAY[1:5] OF STRING
```

We must fill it up with items. In programming terms, this is known as *populating*.

```
FoodItems[1] <- "Burger"
FoodItems[2] <- "Chilli Crab"
FoodItems[3] <- "Paneer Tikka"
FoodItems[4] <- "Fried Rice"
FoodItems[5] <- "Beans on Toast"
```

Now, there are items in the list and we can freely index it now!

3.2.2 Iterating through an array

Revisit section 2.1 on for loops.

```
FOR Counter <- 1 TO 5
    OUTPUT Counter
NEXT Counter
```

For loops allow us to count through values between 1 and something. Conveniently, array indexes are numbers. **Do you see the pattern?**

YES! We can use for loops to go through the items in a list!

```
FOR Counter <- 1 TO 5 // our list is 5 items long
    OUTPUT FoodItems[Counter]
NEXT Counter
```

Will output:

```
Burger
Chilli Crab
Paneer Tikka
Fried Rice
Beans on Toast
```

Armed with this information, let us practice.

3.2.3 Exercises

Write pseudocode for all below questions.

1. With the array we declared before, loop through it backwards and print **`Do you like <the food item>?`**

2. Create a constant **`FoodsLength`** and set it to 10. Create a new array **`FoodsLength`** items long called **`FavoriteFoods`**.

3. Move all the items in the old array to the new array, without copying the values directly. Only use indexing to move the values.

4. Redo question 3, but you may not write more than 3 lines. Use the correct loop for this.

5. In the most logical and optimized manner, ask the user to populate the 6th to the 10th index in the new array through user input and a loop of your choosing. Do not write more than 1 input statement. *hint: counters and indexing!*

6. Iterate through your new list and print out each of the items in the list.

3.3 2D arrays

2D arrays are not dissimilar from plain 1D arrays. In essence, they are still arrays; and the primitive definition of a 2D array would be an array of arrays.

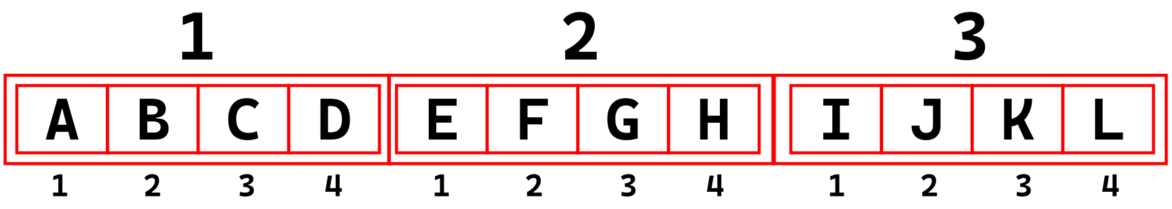

Figure 2: A (bad) visual representation of a 1D array.

Declaring them can be done as follows:

```
//                          COLUMNS       ROWS
DECLARE <variable>:ARRAY[<begin>:<end>,<begin>:<end>] OF <type>
```

Confusing, right?

In this sense, 2D arrays will not be very useful. Arrays in arrays, pretty ridiculous, right?

But no, imagine a situation where you are using an array of names to represent peoples' lockers, and you store the name of each student.

Figure 3: American High School Lockers.

(Lockers are to be imagined as the ones depicted in figure 3.)

However, imagine if your school replaced the lockers with the ones similar to the ones at our school. Instead of being a linear list of lockers, it would be more like a grid, 3 lockers high. Now you need to store the names of the students for the lockers; what do you do?

The smart solution would be to store the names in a list for a row, and store 3 rows in a list itself. Boom, 2D array!

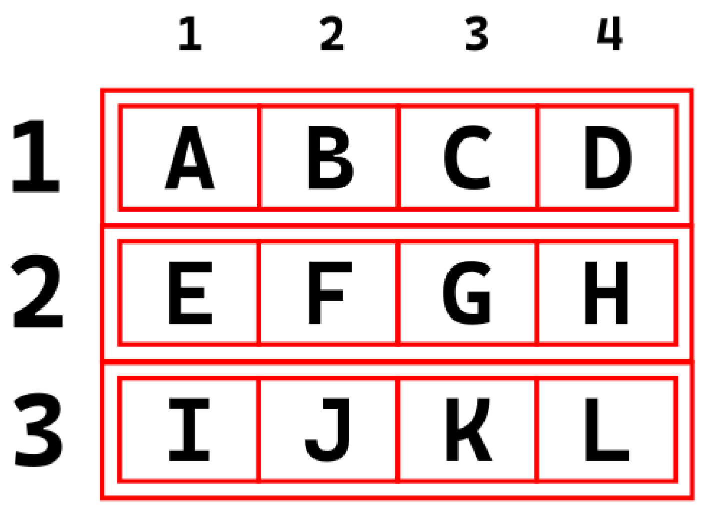

Figure 4: 2D arrays represented with a matrix.

The objective of my explanation is not to explain why 2D arrays are so useful for lockers. It's that 2D arrays give you matrix superpowers in your code; suddenly you can store tables and grids in your code. From a tic tac toe[7] board, to battleships; lockers, apartments and even student scores (foreshadowing), 2D arrays gives you a lot of opportunities.

3.3.1 Declaration and Population

Let us declare the 2D array we described.

```
DECLARE Lockers:ARRAY[1:8,1:3] OF STRING
```

Indexing a 2D array is as simple as providing 2 indexes. The first of which is the row, and the second of which is the column. This may seem very, very counterintuitive, but looking back at figure 2, 2D arrays are stored rows-first, so to access a cell in the grid, you must get the correct row first. This is contrary to math, where coordinates on the cartesian plane are specified as `(x, y)`, such as `(5, 2)`. To get the cell at `(5, 2)` in math terms, we write:

```
OUTPUT Lockers[2,5]
```

And to populate it, we do a very similar thing to a 1D array.

```
Lockers[2,5] <- "Jonathan"

// et cetera, fill it up however you require.
```

[7]Noughts and Crosses for those who are in/from the UK!

3.3.2 Iteration

Remember when we used a single for loop to loop through a 1D array? Now, guess what you do to iterate through a 2D array.

...if you guessed a nested loop, you are correct. The inner loop here would loop through one row, and the outer loop controls how many rows are iterated through.

```
1  DECLARE CurrentName:STRING
2  CurrentName <- "" // it is always good practice to initialize your
3                    // variable after you declare it!
4
5  FOR Row <- 1 TO 3
6      FOR Column <- 1 TO 8
7          CurrentName <- Lockers[Row,Column]
8
9          OUTPUT "The student at row ", Row, " column ",
10                 Column "'s name is", CurrentName
11     NEXT Column
12
13     OUTPUT "Done with row ", Row
14 NEXT Row
```

The code example is complicated but it is straightforward when you read it very carefully:

- The first 2 lines declares and initializes a variable to make life easier.
- Line 4 begins a loop that loops through the 3 rows we declared.
- Line 5 begins a loop *inside* the previous loop that goes through each column.
- We use the variable on line 6 so we dont have to type `Lockers[Row,Column]` in the output line. As mentioned previously, it simplifies the task.
- Line 8 looks intimidating, but it just formats the string so it looks something like, **`The student at row 2 column 5's name is Jonathan`**.
- Line 11 prints out some text to indicate the row is done.

This is a very challenging topic; but if you made it this far; give yourself a pat on the back (or ask your friend :))

Exercises next; they will not get easier than before :)

3.3.3 Exercises

As usual, write all questions in pseudocode.

1. Modify the code block above to loop through the rows and columns backwards. Simply state the lines that need to be rewritten.

2. Declare a 5x5 2D array to store students' exam scores, betewen 0 and 100. Choose the correct data type.

3. Justify why you picked the type you picked for question 2. Include the boundaries/the type of data the data type supports.

4. Ask the user to populate each cell. Tell the user which cell they are populating. Only allow numbers between 5 and 10 to be input into the table. Else, tell the user to tell again and do so until the number is between 5 and 10.

3.4 Parallel Arrays (2D arrays)

This is a short section explaing what parallel arrays are.

The naïve definition of parallel arrays are arrays that go in parallel, one index are associates with another.

Unclear, right?

Now imagine this dataset:

- I have a list of names (`DECLARE ARRAY[1:5] OF STRING`).
- I have a bunch of exam scores that correspond to the students in the array. Each student has 3 exam scores for Science, Math and English.

Judging by the activity we just did, can you guess how we can store the scores? Yes! a 2D array. Since the 3 exam scores are repeating for each student, we can have a 2D array like the following:

Index	1	2	3
	Science	**Math**	**English**
1	90	95	70
2	80	80	90
3	65	75	80
4	25	30	30
5	100	65	80

Figure 5: The data set in question. We will call it ***`StudentScores`***
.

and student names in a normal array. Then, you can have a setup as follows:

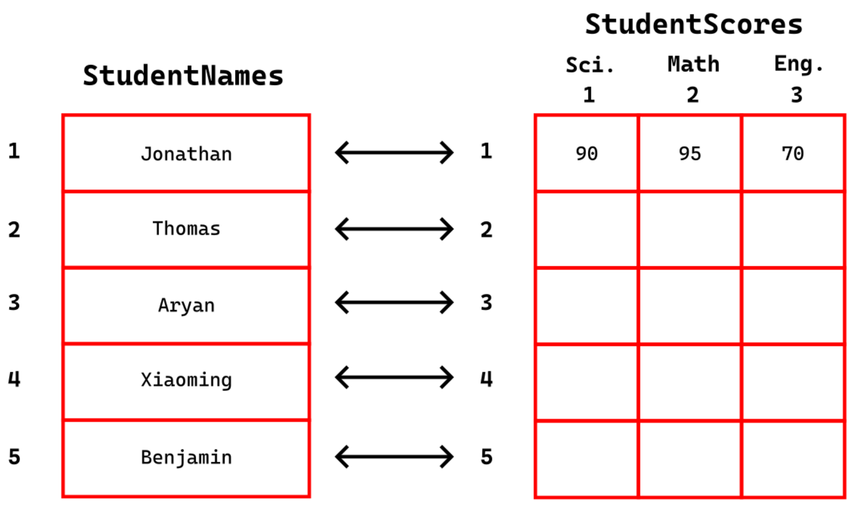

Figure 6: parallel arrays.

Figure 6 shows the data (unfortunately truncated, as I do not want to copy all the data in) in terms of parallel arrays. Now, you may be able to see the etymology;

parallel refers to the nature of the arrays where each of the arrays are correlated by the index. You can simply *define* and even assert that whichever name you insert into the names array, the right must correspond to the results. Therefore, when writing code, you can use the index to bind the arrays together in some regard; they suddenly become related and associated by the index and you now have 2 arrays in parallel by which you can store complex data.

In terms of code, this means you could, say print the names and scores together, calculate the average score and manipulate the name string all while the data is tied together, In fact,

3.4.1 Exercises

As always, write all below answers in pseudocode.

1. Write code to print out the parallel arrays. Output the name of the student and their corresponding scores.

2. Write code to print out the student's name, but this time, the average of their scores as well.

3. Write code to determine who has the lowest average in the class. You may keep an extra variable and conditionally set it somehow. Print out the lowest scoring student's name and average at the end.

4. **Yay! No more exercises..from me. Do the last question of a Comp Sci past paper 2 from recently; ask your computer science teacher for a copy (you should already have one) :)**

4 The End?

Congratulations on coming thus far. I'm proud of you, and keep up the hard work.

This is not the end (you will never stop learning new things in computer science because new things are always being made!), in terms of IGCSE, you still have to know Functions, Procedures, some builtin functions (referred to as *Library Routines*), and some algorithms (sometimes referred to as *standard methods of solution*) like sorting, searching, totalling, finding the minimum/maximum, etc.

Attached at the end should be an answer key; although if you have the printed copy, you may not have it.

5 Answers

NOTE: Please mark logically. These are general solutions; solutions can take many forms. Do not be worried if your exact output format looks dissimilar from these answers. Allow slight variations in formatting, identifiers, used variables, indentation, etc.

Exercise 2.1.1

From: 2.1.1

Q1

```
FOR Counter <- 1 TO 5
    OUTPUT "Name"
NEXT Counter
```

Q2

```
FOR Counter <- 3 TO 7
    OUTPUT Counter
NEXT Counter
```

Q3

```
FOR Counter <- 2 TO 10 STEP 2
    OUTPUT Counter
NEXT Counter
```

Q3

```
FOR Counter <- 1 TO 12
    OUTPUT 5 * Counter
NEXT Counter
```

Q5

```
FOR Counter <- 10 TO 1 STEP -1
    OUTPUT Counter
NEXT Counter
```

Exercise 2.2.1

From 2.2.1

Q1

```
FOR Begin <- 1 TO 4 // Question specifies 4-5.
    FOR Counter <- Begin TO 5
        OUTPUT Counter
    NEXT Counter
NEXT Begin
```

Q2

```
FOR Base <- 1 TO 12
    OUTPUT "Times tables for ", Base
    FOR Multiplier <- 1 TO 12
        OUTPUT Base * Multiplier
    NEXT Multiplier
NEXT Base
```

Exercise 2.3.1

From 2.3.1

NOTE: For these questions, allow both exclusive and inclusive ranges.

Q1

```
DECLARE Counter:INTEGER
Counter <- 1

WHILE Counter <= 10 DO
    OUTPUT Counter
    Counter <- Counter + 1
ENDWHILE
```

Q2

```
DECLARE Counter:INTEGER
Counter <- 20

WHILE Counter >= 0 DO
    OUTPUT Counter
    Counter <- Counter - 2
ENDWHILE
```

Q3

```
DECLARE Name:STRING
Name <- "" // optional

WHILE Name <> "exit" DO
    OUTPUT "what is your name"
    INPUT Name

    IF Name <> "exit"
      THEN
        OUTPUT "Hello, ", Name
    ENDIF
ENDWHILE
```

Q4

Accept code that uses the variable and doesn't. The logic and condition must be identical. Allow `BREAK` statements despite their nonexistence in the specification of pseudocode as logic is the same.

```
DECLARE Num:INTEGER
DECLARE Valid:BOOLEAN

Valid <- TRUE
Num <- 0 // optional

WHILE Valid DO
    OUTPUT "enter a number"
    INPUT Num

    IF Num > 5 AND Num < 20
      THEN
        OUTPUT "Your input is valid"
      ELSE
        Valid <- FALSE
    ENDIF
ENDWHILE
```

Exercise 2.4.1

From 2.4.1

Q1

```
DECLARE Guess:STRING
CONSTANT HomeCountry <- "China"

REPEAT
    OUTPUT "Guess my home country!"
    INPUT Guess

    // the IF statement is technically optional
    IF Guess <> HomeCountry
      THEN
        OUTPUT "Not quite right..."
    ENDIF
UNTIL Guess = HomeCountry
```

Q2

```
DECLARE Temperature:INTEGER

REPEAT
    OUTPUT "Tell me the temperature in your city..."
    INPUT Temperature

    // the IF statement is technically optional
    IF Temperature < 40
      THEN
        OUTPUT "Not bad..."
    ENDIF
UNTIL Temperature > 40

OUTPUT "Wow! your city's temperature is so hot!"
```

Exercise 3.2.3

From 3.2.3

Q1

```
FOR Counter <- 5 TO 1 STEP -1
    OUTPUT "Do you like ", FoodItems[Counter], "?"
NEXT Counter
```

Q2

```
CONSTANT FoodsLength <- 10
DECLARE FavoriteFoods:ARRAY[1:FoodsLength] OF STRING
```

Q3

```
FavoriteFoods[1] <- FoodItems[1]
FavoriteFoods[2] <- FoodItems[2]
FavoriteFoods[3] <- FoodItems[3]
FavoriteFoods[4] <- FoodItems[4]
FavoriteFoods[5] <- FoodItems[5]
```

Q4

```
FOR Counter <- 1 TO 5
    FavoriteFoods[Counter] <- FoodItems[Counter]
NEXT Counter
```

Q5

Students may loop through 1-5 and use `Counter + 5` as the index, generating 6-10. Students may also use FoodsLength instead of 10.

```
DECLARE Food:STRING

FOR Counter <- 6 TO 10
    OUTPUT "Enter a food: "
    INPUT Food
    FavoriteFoods[Counter] <- Food
NEXT Counter
```

Q6

Students may use FoodsLength instead of 10.

```
FOR Counter <- 1 TO 10
    OUTPUT FavoriteFoods[Counter]
NEXT Counter
```

Exercise 3.3.3

From 3.3.3

Q1

```
Line 5: FOR Row <- 3 TO 1 STEP -1
Line 6: FOR Column <- 8 TO 1 STEP -1
```

Q2

```
DECLARE ExamScores:ARRAY[1:5,1:5] OF INTEGER
```

Q3

- INTEGER
- Boundaries depend on the implementation, but accept all negative and positive non-decimals
- Since the data does not need to store decimals, nor text nor characters, INTEGER makes the most sense.

Q4

```
DECLARE Num:INTEGER
Num <- 0 // optional

FOR Row <- 1 TO 5
    FOR Column <- 1 TO 5
        // also allow while loops.

        REPEAT
            OUTPUT "enter a number: "
            INPUT Num

            IF Num < 5 AND NUM > 10
              THEN
                OUTPUT "invalid input! try again..."
            ENDIF
        UNTIL Num >= 5 AND NUM <= 10
    NEXT Column
NEXT Row
```

Exercise 3.4.1

From 3.4.1

Q1

```
FOR Counter <- 1 TO 5
    OUTPUT "Name: ", StudentNames[Counter]

    // as long as if the student attempts to output something, the answer
    // should be accepted. the objective is to test the understanding of
    // the concept of parallel arrays bind data through the index.
    //
    OUTPUT "Science: ", StudentGrades[1]
    OUTPUT "Math: ", StudentGrades[2]
    OUTPUT "English: ", StudentGrades[3]
NEXT Counter
```

Q2

If the student attempts to calculate the median/mode, give them a pat on their back for their effort and rebeliousness.

```
DECLARE Total:INTEGER
DECLARE Average:REAL // make sure it is REAL, do not accept INTEGER
                     // this variable is also optional

FOR Counter <- 1 TO 5
    OUTPUT "Name: ", StudentNames[Counter]

    // A for loop is appreciated, but not required.
    // Here, additions are wrapped to the next line.
    Total <- StudentGrades[Counter,1]
             + StudentGrades[Counter,2]
             + StudentGrades[Counter,3]
    Average <- Total / 3

    OUTPUT "Average grade is: ", Average
NEXT Counter
```

Q3

```
DECLARE Total:INTEGER
DECLARE Average:REAL
DECLARE LowestAverage:REAL
DECLARE LowestAverageName:STRING

LowestAverage <- 0 // LowestAverage will be accessed without
                   // being initialized in the loop.
                   // The student must initialize this.
```

```

FOR Counter <- 1 TO 5
    Total <- StudentGrades[1] + StudentGrades[2] + StudentGrades[3]
    Average <- Total / 3

    IF Average < LowestAverage
      THEN
        LowestAverageName <- StudentName[Counter]
        LowestAverage <- Average
    ENDIF
NEXT Counter

OUTPUT "The lowest scorer was ", LowestAverageName
OUTPUT "They earned a ", LowestAverage, " average."
```