TEMASEK
JUNIOR COLLEGE

Temasek Junior College
JC H2 Computing
Problem Solving & Algorithm Design 6
Arrays

1 Arrays

- Traditionally a **static data structure** made up of a number of variables or data items that all have the **same data type** (fixed-length structures) and are **accessed by the same name**.

- It is good practice to explicitly state what the lower bound of the array (i.e. the index of the first element) is. This is because the index defaults to either 0 or 1 in different systems. Generally, a lower bound of 1 will be used.

- Square brackets `[]` are used to indicate the array indices.

- The individual data items that make up the array are referred to as the **elements** of the array. Elements in the array are distinguished from one another by the use of an index or subscript, enclosed in parentheses, following the array name.

  - The subscript or index indicates the position of an element within the array.

  - The subscript or index may be a number or a variable. It may be used to access any item within the valid bounds of an array.

- Arrays are internal data structures, i.e. they are required only for the duration of the program in which they are defined.

- Arrays are a very convenient mechanism for storing and manipulating a collection of similar data items in a program.

- One- and two-dimensional arrays are declared as follows (where L, L1, L2 are lower bounds and U, U1, U2 are upper bounds):

Declaring One-Dimensional (1D) Array

```
DECLARE <identifier> : ARRAY[<L>:<U>] OF <data type>
```

e.g. `DECLARE StudentAges : ARRAY[1:10] OF INTEGER`

Possible interpretation:

	StudentAges – One dimension									
Address	**1001**	**1003**	**1005**	**1007**	**1009**	**1011**	**1013**	**1015**	**1017**	**1019**
Index	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
Row or Column	17	14	15	13	17	11	18	16	19	14

Declaring Two-Dimensional (2D) Array

```
DECLARE <identifier> : ARRAY[<L1>:<U1>,<L2>:<U2>] OF <data type>
```

e.g. `DECLARE Attendance : ARRAY[1:5,1:4] OF CHAR`

Possible interpretation:

	Attendance – Two Dimension			
Row \ Column	1	2	3	4
Address	**1001**	**1002**	**1003**	**1004**
1	[1, 1]	[1, 2]	[1, 3]	[1, 4]
Address	**1005**	**1006**	**1007**	**1008**
2	[2, 1]	[2, 2]	[2, 3]	[2, 4]
Address	**1009**	**1010**	**1011**	**1012**
3	[3, 1]	[3, 2]	[3, 3]	[3, 4]
Address	**1013**	**1014**	**1015**	**1016**
4	[4, 1]	[4, 2]	[4, 3]	[4, 4]
Address	**1017**	**1018**	**1019**	**1020**
5	[5, 1]	[5, 2]	[5, 3]	[5, 4]

	Attendance – with values			
Row \ Column	1	2	3	4
1	Y	Y	Y	Y
2	Y	X	X	Y
3	X	Y	Y	X
4	Y	Y	Y	Y
5	X	X	X	Y

Attendance[3, 2] = 'Y'

Attendance[J, K] = 'X'

1.1 Operations on Arrays

- to load initial information into an array which is later required for processing,
- to process the elements of the array,
- to store the results of processing into the elements of an array, or
- to store information, which is then written to a report.

Thus, the most typical operations performed on arrays are:

- loading a set of initial values into the elements of an array,
- processing the elements of an array,
- searching an array, using a linear or binary search, for a particular element, and
- writing out the contents of an array to a report.

1.2 Using Arrays

With reference to the 1D-array `StudentAges` and 2D-array `Attendance`, do you know what the pseudocodes below mean?

```
StudentAges[1] ← 17
Attendance [2, 3] ← 'X'
StudentNames[n+1] ← StudentNames[n]
```

Example 1 – Arrays with Loops
The following test scores are given: 65, 71, 82, 63, 90, 58, 66, 67, and 68.
Use the array `TestScores` to store the scores.
Write a program in pseudocode to calculate the average and the number of test scores that were above average and below average.

[Solution]

(A) *Array Interpretation*

	TestScores – One dimension								
Address	**1001**	**1003**	**1005**	**1007**	**1009**	**1011**	**1013**	**1015**	**1017**
Index	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Row or Column	65	71	82	63	90	58	66	67	68

(B) *Pseudocode*

```
CONSTANT ArraySize = 9
DECLARE TestScores : ARRAY[1:ArraySize] OF INTEGER
DECLARE Count, Sum, Average, LargeCount, SmallCount : INTEGER

// Initialization
Sum ← 0
AboveCount ← 0
BelowCount ← 0

// Read the test scores and store into element of array TestScores
FOR Count ← 1 TO ArraySize
  PRINT "Enter score of number" & Count & ":"
  READ TestScores[Count]
  Sum = Sum + TestScores[Count]  // Add each score to Sum
ENDFOR

Average = Sum/ArraySize  // Calculate average and store into Average

// Scanning all elements in array TestScores
FOR Count ← 1 TO ArraySize
      // Check whether the current element is greater than Average
      IF TestScores[Count] > Average THEN
            AboveCount ← AboveCount + 1  // Increment AboveCount by 1
      ENDIF
      // Check whether the current element is smaller than Average
      IF TestScores[Count] < Average THEN
            BelowCount ← BelowCount + 1  //Increment BelowCount by 1
      ENDIF
ENDFOR

PRINT "Average: " , Average
PRINT "The number of test scores that were above average: ", AboveCount
PRINT "The number of test scores that were below average: ", BelowCount
```

Example 2 – Arrays with Loops
Shown below is a segment of an algorithm.

```
set S = 0
set E = 7
set F = False
set P = -1

input SV

repeat
  set M = (S + E) DIV 2

  if SV = Array[M] then
    set F = True
    set P = M
  endif

  if SV > Array[M] then
    set S = M + 1
  endif

  if SV < Array[M] Then
    set E = M - 1
  endif

until (F = True) OR (E < S)
```

Test Data

Array	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
	13	27	33	49	55	67	75	79

SV = 75

Note
DIV is integer division (floor division). It gives the quotient that is obtained from division.
e.g. 7 DIV 2 = 3

Complete the table below to show how each variable changes when the algorithm is performed on the test data given above.

S	E	M	Array[M]	F	P
0	7	3	49	False	-1
4	7			False	-1

[Solution]

Example 3 – Arrays with Loops: Manipulating Index
The Fibonacci numbers are generated by first letting the first two numbers of the sequence be equal to 1. Subsequently, each number may be found by taking the sum of the previous two elements in the sequence. Hence you get 1, 1, 2, 3, 5, 8, 13, etc.

Write a program in pseudocode that fulfils the following requirements:

- Prepare two lists: one with the first 10 Fibonnacci numbers and the other with the next 10 Fibonacci numbers.
- For each Fibonacci number from 2nd to the 19th, find the difference between the square of the element and the product of the elements immediately preceding and following. In other words, print `F[K]**2 - F[K - 1] * F[K + 1]`.

```
CONSTANT ArraySize = 10
DECLARE F1, F2 : ARRAY[1: ArraySize] OF INTEGER
DECLARE Count: INTEGER

// Creating the first list with the first 10 Fibonacci numbers

// First two numbers of the sequence F1 equal 1
F1[1] ← 1, F1[2] ← 1
```

F1[1]	F1[2]	F1[3]	F1[4]	F1[5]	F1[6]	F1[7]	F1[8]	F1[9]	F1[10]
1	1								

F2[1]	F2[2]	F2[3]	F2[4]	F2[5]	F2[6]	F2[7]	F2[8]	F2[9]	F2[10]

```
// Each subsequent F1 may be found by taking the sum of the previous two elements in the sequence
FOR Count ← 3 TO ArraySize
  F1[Count] ← F1[Count - 2] + F1[Count - 1]
NEXT
```

F1[1]	F1[2]	F1[3]	F1[4]	F1[5]	F1[6]	F1[7]	F1[8]	F1[9]	F1[10]
1	1	2	3	5	8	13	21	34	55

F2[1]	F2[2]	F2[3]	F2[4]	F2[5]	F2[6]	F2[7]	F2[8]	F2[9]	F2[10]

```
// Creating the second list with the next 10 Fibonacci numbers

// Assignment to first element to F2 by last 2 elements of F1.
F2[1] ← F1[ArraySize - 1] + F1[ArraySize]
```

F1[1]	F1[2]	F1[3]	F1[4]	F1[5]	F1[6]	F1[7]	F1[8]	F1[9]	F1[10]
1	1	2	3	5	8	13	21	34	55

F2[1]	F2[2]	F2[3]	F2[4]	F2[5]	F2[6]	F2[7]	F2[8]	F2[9]	F2[10]
89									

```
//Second element of F2 should come from last element of F1 and first element of F2
F2[2] ← F1[ArraySize] + F2[1]
```

F1[1]	F1[2]	F1[3]	F1[4]	F1[5]	F1[6]	F1[7]	F1[8]	F1[9]	F1[10]
1	1	2	3	5	8	13	21	34	55

F2[1]	F2[2]	F2[3]	F2[4]	F2[5]	F2[6]	F2[7]	F2[8]	F2[9]	F2[10]
89	144								

```
// Each subsequent F2 may be found by taking the sum of the previous two elements in the sequence
FOR Count ← 3 TO ArraySize
  F2[Count] ← F2[Count - 2] + F2[Count - 1]
NEXT
```

F1[1]	F1[2]	F1[3]	F1[4]	F1[5]	F1[6]	F1[7]	F1[8]	F1[9]	F1[10]
1	1	2	3	5	8	13	21	34	55

F2[1]	F2[2]	F2[3]	F2[4]	F2[5]	F2[6]	F2[7]	F2[8]	F2[9]	F2[10]
89	144	233	377	610	987	1597	2584	4181	6765

```
// PRINT F1[K]**2 - F1[K - 1] * F1[K + 1] FROM K = 2 UNTIL K = 9
FOR Count ← 2 TO ArraySize - 1
  PRINT F1[Count] * F1[Count] - F1[Count - 1] * F1[Count + 1]
NEXT
```

F1[1]	F1[2]	F1[3]	F1[4]	F1[5]	F1[6]	F1[7]	F1[8]	F1[9]	F1[10]
1	1	2	3	5	8	13	21	34	55

F2[1]	F2[2]	F2[3]	F2[4]	F2[5]	F2[6]	F2[7]	F2[8]	F2[9]	F2[10]
89	144	233	377	610	987	1597	2584	4181	6765

```
// PRINT F1[K]**2 - F1[K - 1] * F1[K + 1] WHEN K = 10
PRINT F1[ArraySize] * F1[ArraySize] - F1[ArraySize-1] * F2[1]
```

F1[1]	F1[2]	F1[3]	F1[4]	F1[5]	F1[6]	F1[7]	F1[8]	F1[9]	F1[10]
1	1	2	3	5	8	13	21	34	55

F2[1]	F2[2]	F2[3]	F2[4]	F2[5]	F2[6]	F2[7]	F2[8]	F2[9]	F2[10]
89	144	233	377	610	987	1597	2584	4181	6765

```
// PRINT F2[K]**2 - F2[K - 1] * F2[K + 1] WHEN K = 1
PRINT F2[1] * F2[1] - F1[ArraySize] * F2[2]
```

F1[1]	F1[2]	F1[3]	F1[4]	F1[5]	F1[6]	F1[7]	F1[8]	F1[9]	F1[10]
1	1	2	3	5	8	13	21	34	55

F2[1]	F2[2]	F2[3]	F2[4]	F2[5]	F2[6]	F2[7]	F2[8]	F2[9]	F2[10]
89	144	233	377	610	987	1597	2584	4181	6765

```
// PRINT F2[K]**2 - F2[K - 1] * F2[K + 1] FROM K = 2 UNTIL K = 9
FOR Count ← 2 TO ArraySize - 1
  PRINT F2[Count] * F2[Count] - F2[Count - 1] * F2[Count + 1]
NEXT
```

F1[1]	F1[2]	F1[3]	F1[4]	F1[5]	F1[6]	F1[7]	F1[8]	F1[9]	F1[10]
1	1	2	3	5	8	13	21	34	55

F2[1]	F2[2]	F2[3]	F2[4]	F2[5]	F2[6]	F2[7]	F2[8]	F2[9]	F2[10]
89	144	233	377	610	987	1597	2584	4181	6765

1.3 Dry Running (Desk Checking) of Algorithms II

In **Problem Solving and Algorithm Design 3 – Dry Running (Desk Checking) of Algorithms**, we took a brief look at how trace tables can be used dry run an algorithm to test its correctness. In fact, dry running (desk checking) an algorithm is a necessary step after a solution algorithm has been constructed. The solution algorithm must be **tested for correctness** because most major errors occur during the development of the algorithm, and if not detected, these errors would be passed on to the program when it is written. In this section, we shall look at the dry running (desk checking) of algorithms in greater detail.

1.3.1 Key Idea

Dry running (desk checking) involves tracing through the logic of the algorithm with some chosen test data. That is, you walk through the logic of the algorithm exactly as a computer would, keeping track of all major variable values on a sheet of paper.

1.3.2 Selecting Test Data

```
Test plan
  {
     {Test case 1
        [Test data]}
     {Test case 2
        [Test data]}
     ...
  }
```

When selecting test data to dry run (desk check) an algorithm, it is important to look at the **program specification** and **choose simple test cases** only, based on the **requirements of the specification** not the algorithm

1.3.3 Steps in Dry Running (Desk Checking) an Algorithm

(1) Choose simple input test cases that are valid. Two or three test cases are usually sufficient.

(2) Establish what the expected result should be for each test case. This is one of the reasons for choosing simple test data in the first place: it is much easier to determine the total of 10, 20 and 30 than 3.11, 1.78 and 9.89!

(3) Construct a table of the relevant variable names within the algorithm on a piece of paper.

(4) Walk the first test case through the algorithm, keeping a step-by-step record of the contents of each variable in the table as the data passes through the logic.

(5) Repeat the walk-through process using the data for the other test cases, until the algorithm has reached its logical end.

(6) Check that the expected result established in Step 2 matches the actual result developed in Step 5.

By desk checking an algorithm you are attempting to **detect early errors**. Desk checking will **eliminate most errors**, but it **still cannot prove** that the algorithm is 100% correct.

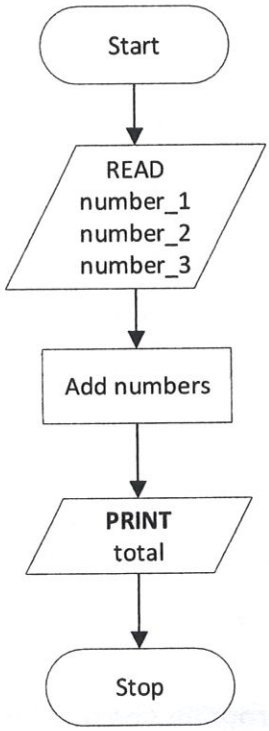
Example 4 – Adding Three Numbers

Let us revisit the following problem previously encountered in **Problem Solving and Algorithm Design 2 – Program Flowcharting**. We shall now look at the pseudocode of the algorithm as well as the dry run (desk check) process.

A program is required to read three numbers, add them together and print their total.

(A) ***Solution Algorithm***

```
READ number_1, number_2, number_3
total ← number_1 + number_2 + number_3
PRINT total
```



(B) ***Dry Run (Desk Check)***

(1) Prepare a **test plan**.

(2) In the test plan, prepare the **test cases**.

(3) Choose two sets of input **test data**.
- **(a)** 10, 20 and 30
- **(b)** 40, 41 and 42

Test Plan			
	Test Case 1	**Test Case 2**	
INPUT	Data Set 1	Data Set 2	
number_1	10	40	Test
number_2	20	41	Test
number_3	30	42	Test

(4) Establish the expected result for each test case.

	Test Case 1	**Test Case 2**
total	60	123

(5) Set up a table of the relevant variable names and pass each test data set through the solution algorithm, statement by statement.

Statement		**number_1**	**number_2**	**number_3**	**total**	**PRINT**
First Pass	READ	10	20	30		
	total				60	
	PRINT					Y
Second Pass	READ	40	41	42		
	total				123	
	PRINT					Y

(6) Check that the expected result (60 and 123) matches the actual result (the total column in the table)

Example 5 – Counting – Applications: All forms of Counting
Given a set examination marks (in the range 0 to 100) for *n* students, make a count of the number of students that passed the examination. A pass is awarded for all marks of 50 and above.

(A) *Solution Algorithm*

Using pre-condition loop

```
1   Prompt then Read the number of marks (n) to be processed.
2   Initialise count to zero, input count to 1.
3   DOWHILE input_count ≤ n
4     READ next_mark
5     IF next_mark ≥ 50 THEN
6       Add 1 to count
7     ENDIF
8     Add 1 to input_count
9   ENDDO
10  PRINT total number of passes (count)
```

Use post-condition loop

```
1   Prompt then Read the number of marks (n) to be processed.
2   Initialise count to zero, input_count to 1.
3   REPEAT
4     READ next_mark
5     IF next_mark ≥ 50 THEN
6       Add 1 to count
7     ENDIF
8     Add 1 to input_count
9   UNTIL input_count > n
10  PRINT total number of passes (count)
```

(B) *Dry Run (Desk Check)*

(1) Prepare a **test plan**.

(2) In the test plan, prepare the **test cases**.

(3) Choose two sets of input **test data**.

Test Plan			
	Test Case 1	**Test Case 2**	
INPUT	Data Set 1	Data Set 2	
n	3	5	
next_mark	60	60	Test Data
next_mark	70	50	Test Data
next_mark	80	45	Test Data
		78	
		64	

(4) Establish the expected result for each test case.

	Test Case 1	Test Case 2
count	3	4

(5) Set up a table of the relevant variable names and pass each test data set through the solution algorithm, statement by statement.

Tracing for Test Case 1 using Pre-Condition Loop							
Statement	**n**	**count**	**input_count**	**input_count ≤ n**	**next_mark**	**next_mark ≥ 50**	**PRINT**
1 Prompt	3						
2 Initialise		0	1				
3 DOWHILE				T			
4 READ					60		
5 IF						T	
6 Add		1					
7 ENDIF							
8 Add			2				
9 ENDDO							
3 DOWHILE ...		2	3	T	70	T	
9 ENDDO							
3 DOWHILE ...		3	4	T	80		
9 ENDDO							
3 DOWHILE				F			
10 PRINT							Y

(6) Check that the expected result (3 and 4) matches the actual result (the count column in the table).

Note

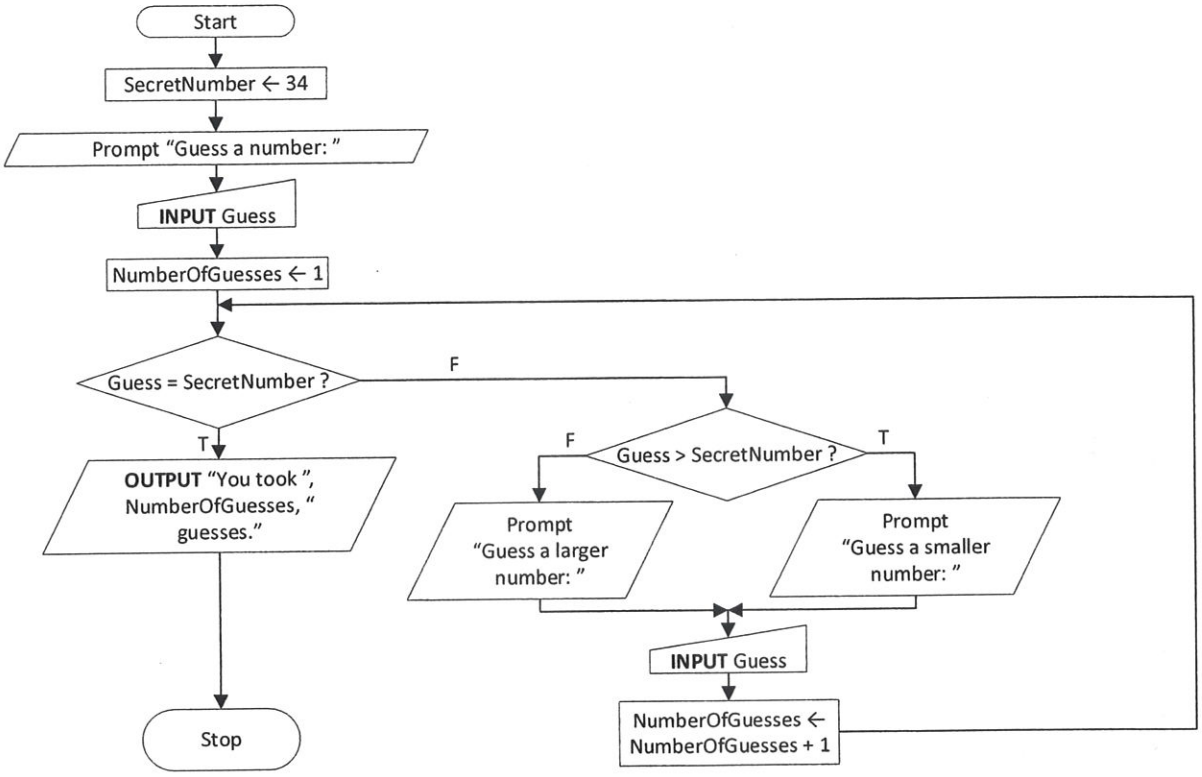
- In an actual scenario involving the use of the test plan, subsequent passes after the first should also be expanded on a step by step basis.

Questions

- Perform the dry run (desk check) for Test Case 2 using the Pre-Condition Loop.
- Perform the dry run (desk check) using the Post-Condition Loop.

Example 6 – Number Guessing Game (Reference only)

Let us revisit the algorithm of the number-guessing game previously encountered in **Problem Solving and Algorithm Design 3 – Dry Running (Desk Checking) of Algorithms**. We shall now look at the pseudocode of the algorithm as well as the dry run (desk check) process in detail.



(A) *Pseudocode*

```
1   SecretNumber ← 34
2   PRINT "Guess a number: "
3   INPUT Guess
4   NumberOfGuesses ← 1
5   REPEAT
6     IF Guess = SecretNumber
7       THEN
8         OUTPUT "You took " & NumberOfGuesses & " guesses. "
9       ELSE
10        IF Guess > SecretNumber
11          THEN
12            PRINT "Guess a smaller number: "
13            INPUT Guess
14          ELSE
15            PRINT "Guess a larger number: "
16            INPUT Guess
17        ENDIF
18        NumberOfGuesses ← NumberOfGuesses + 1
19    ENDIF
20  UNTIL Guess = SecretNumber
```

(B) *Dry Run (Desk Check)*

Let us revisit the trace table of the dry run for the following set of test data: 5, 55, 30, 42, 35, 33 and 34.

SecretNumber	Guess	NumberOfGuesses	Guess = SecretNumber	Guess > SecretNumber	Message
34	5	1	FALSE	FALSE	... larger ...
	55	2	FALSE	TRUE	... smaller ...
	30	3	FALSE	FALSE	... larger ...
	42	4	FALSE	TRUE	... smaller ...
	36	5	FALSE	TRUE	... smaller ...
	33	6	FALSE	FALSE	... larger ...
	34	7	TRUE		... 7 guesses

Observe that the above trace table compacts the tracing of the algorithm in each pass into a single row. Nevertheless, each row is still generated by a statement by statement tracing.

Example 7 – Problem Solving (Armstrong Numbers)
Numbers that are equal to the sum of the cubes of their digits are known as Armstrong numbers e.g. 153 is an Armstrong number, since

$$153 = 1^3 + 5^3 + 3^3$$

Write the algorithm that outputs all Armstrong numbers from 1 to 2000 in pseudocode.

[Solution]

(A) *Analysis*

- To determine whether a number is an Armstrong number, we must take each of the digits making up the number (e.g. 1, 5 and 3) and then calculate the sum of the cubes of those digits.

- For any number, the digit in the ones place is the remainder when the number is divided by 10 e.g. if `NUM` is number, we obtain the digit in the ones place by

```
Quotient = INT(NUM / 10)
Remainder = NUM - 10 * Quotient
```

  The `Remainder` is now the ones digit.

- To get the tens digit, the calculation is repeated using

```
Quotient_1 = INT(Quotient / 10)
Remainder = Quotient - 10 * Quotient_1
```

  The `Remainder` is now the tens digit.

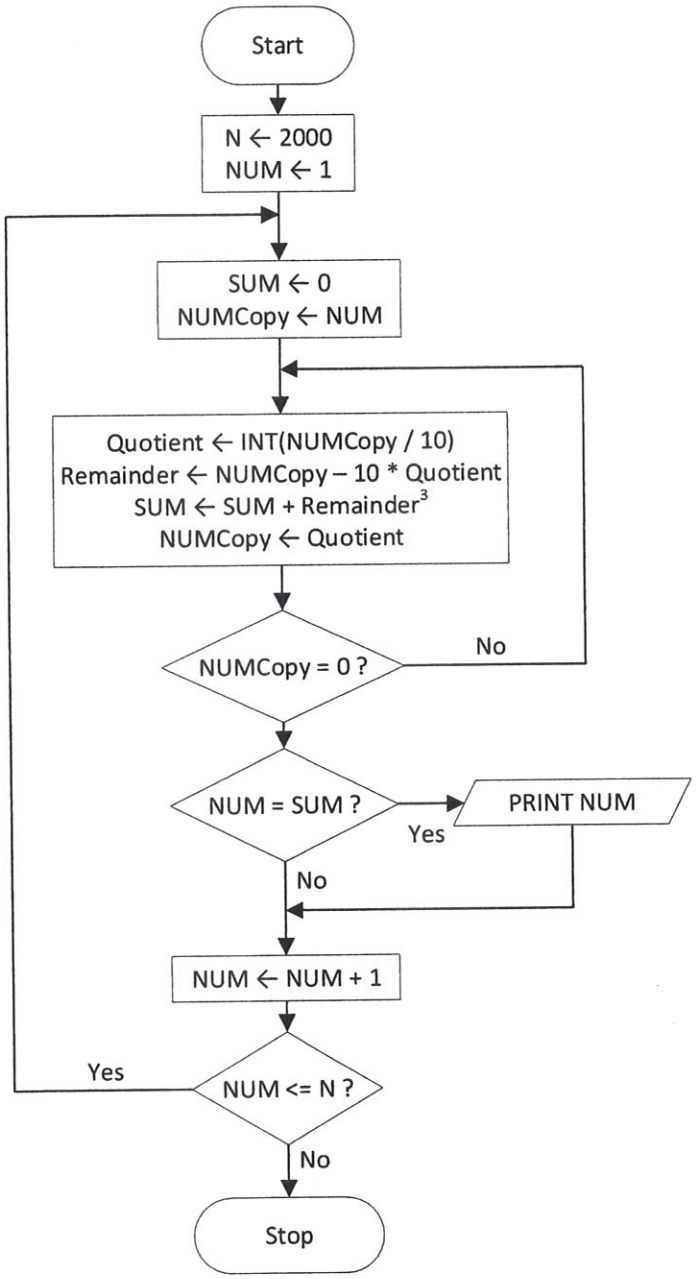
- This same process is repeated until we get a zero quotient.

- In this question, since the upper limit is 2000, we will never exceed four digits.

  Question:
  Can you suggest an alternative method to obtain the quotients and remainders?

(B) *Flowchart and Pseudocode*

- Rather that calculating `Quotient_1`, `Quotient_2`, `Quotient_3` etc., the operation may be carried out as illustrated in the flowchart shown.



- Part of the pseudocode is given below. Complete the pseudocode using the flowchart as a reference.

```
Set NUMCopy = NUM and SUM = 0
Quotient = INT (NUMCopy / 10)
Reminder = NUMCopy - 10 * Quotient
Set SUM = S + (Reminder * Reminder * Reminder)
Set NUMCopy = Quotient  // for the next iteration
```

(C) *Dry Run (Desk Check)*

- Perform a dry run for the algorithm with the following set of test values: 153, 792, 937 and 1436.

Example 8

Below is an incomplete algorithm that attempts to produce the multiplication table for a given positive integer.

```
Algorithm Multiplication Table

Multiplier is integer {input by user}
Product is integer {used to store current answer}
i is integer

startmainprog

  input Multiplier

  if Multiplier < 1 then
    output "Number input must be greater than zero"
  else
    for i = 1 to 12
      set Product =
      output Product
    endfor
  endif

endmainprog
```

(a) Write down an example of a meaningful identifier from the algorithm.

Mu
Mu
Pr

(b) Why should algorithms be written with meaningful identifiers?

To r

(c) Explain the role of the following line of code from the algorithm

Ro
po

or

(d) Identify the incomplete part(s) of the algorithm. Complete the part(s) so that the algorithm produces a multiplication table for a given positive integer

Th

se

(e) Perform a dry run (desk check) for the algorithm for the `multiplier = 5`.

Tutorial 6 Q2, Q3, Q4, Q6

Tutorial 6

1 The following are the subject results of 5 students which are stored in a 2-dimensional array as follows:

```
DECLARE SUBJRESULTS: ARRAY[1:5,1:6] OF INTEGER
```

65	68	73	85	82	87
74	87	90	88	87	88
88	97	91	92	90	89
91	83	78	89	79	87
65	76	67	50	60	66

Design the pseudocode with **REPEAT-UNTIL** loop to generate the following report:

65	68	73	85	82	87	AVG = 76.67
74	87	90	88	87	88	AVG = 85.67
88	97	91	92	90	89	AVG = 91.17
91	83	78	89	79	87	AVG = 84.50
65	76	67	50	60	66	AVG = 64.00

2 *A program is required **print** a series of names and exam scores for students enrolled in a Computing course. The class average is to be **computed** and **printed** at the end of the report. Scores can range from 0 to 100. Data are given and stored in two 1-dimensional arrays `STUName` and `Score`.

Design the pseudocode with **DO-WHILE** loop to print examination scores.

Defining diagram

Store	Process	Output
Assign data to `STUName` and `Score`	Print student details	`STUName` and `Score`
	Compute average score	
	Print `average_score`	`average_score`

Test Data

JOYCE TAN WAN LIN	65
NG HUI TING	45
NUR RAMIZAH	77
TANG KUAN YEE	67
KAW TECK LIN	30
KUNG GUANGJUN	90
LOE CHUAN YUN	71
OH JIEYI JOEL	68
PANG YINGXIANG	88
TAY KAI ZHONG	56

3 *Design the pseudocode with nested loop to generate the data given and store them in a 2-dimensional array `TRIValue` as follows:

```
DECLARE TRIValue: ARRAY [1:10, 1:10] OF INTEGER
```

(a) with nested **FOR-ENDFOR** loop

Test data to be generated:

Row ↓ / Col →	1	2	3	4	5	6	7	8	9	10
→1	1									
2	1	2								
3	1	2	3							
4	1	2	3	4						
5	1	2	3	4	5					
6	1	2	3	4	5	6				
7	1	2	3	4	5	6	7			
8	1	2	3	4	5	6	7	8		
9	1	2	3	4	5	6	7	8	9	
10	1	2	3	4	5	6	7	8	9	10

The pseudocode will print a report as follows:

1										1
1	2									3
1	2	3								6 ←Sum of row
1	2	3	4							10
1	2	3	4	5						15
1	2	3	4	5	6					:
1	2	3	4	5	6	7				:
1	2	3	4	5	6	7	8			:
1	2	3	4	5	6	7	8	9		:
1	2	3	4	5	6	7	8	9	10	55
Sum of columns→	10	18	24	28	30	:	:	:	18	10

(b) with nested **WHILE** loop

Test data to be generated:

									1
								2	1
							3	2	1
						4	3	2	1
					5	4	3	2	1
				6	5	4	3	2	1
			7	6	5	4	3	2	1
		8	7	6	5	4	3	2	1
	9	8	7	6	5	4	3	2	1
10	9	8	7	6	5	4	3	2	1

The pseudocode will print a report as follows:

									1	1	
								2	1	3	
							3	2	1	6	←Sum of row
						4	3	2	1	10	
					5	4	3	2	1	15	
				6	5	4	3	2	1	:	
			7	6	5	4	3	2	1	:	
		8	7	6	5	4	3	2	1	:	
	9	8	7	6	5	4	3	2	1	:	
10	9	8	7	6	5	4	3	2	1	55	
Sum of columns→	10	18	24	28	30	:	:	:	18	10	

(c) with nested **REPEAT-UNTIL** loop

Test data to be generated

10	9	8	7	6	5	4	3	2	1
	9	8	7	6	5	4	3	2	1
		8	7	6	5	4	3	2	1
			7	6	5	4	3	2	1
				6	5	4	3	2	1
					5	4	3	2	1
						4	3	2	1
							3	2	1
								2	1
									1

The pseudocode will print a report as follows:

	55	10	9	8	7	6	5	4	3	2	1
Sum of row→	45		9	8	7	6	5	4	3	2	1
	:			8	7	6	5	4	3	2	1
	:				7	6	5	4	3	2	1
	:					6	5	4	3	2	1
	:						5	4	3	2	1
	:							4	3	2	1
	6								3	2	1
	3									2	1
	1										1
		10	18	24	28	30	:	:	:	18	10

←Sum of columns

(d) with any one of the following nested **DO-WHILE / REPEAT-UNTIL / FOR-ENDFOR** loop

Test data to be generated:

10	9	8	7	6	5	4	3	2	1
	9	8	7	6	5	4	3	2	
		8	7	6	5	4	3		
			7	6	5	4			
				6	5				

The pseudocode will print a report as follows:

	55	10	9	8	7	6	5	4	3	2	1
Sum of row→	44		9	8	7	6	5	4	3	2	
	33			8	7	6	5	4	3		
	22				7	6	5	4			
	11					6	5				
		10	18	24	28	30	:	:	:	4	1

←Sum of columns

(e) with nested **FOR-ENDFOR** loop

Test data to be generated:

1									
2	1								
1	2	3							
4	3	2	1						
1	2	3	4	5					
6	5	4	3	2	1				
1	2	3	4	5	6	7			
8	7	6	5	4	3	2	1		
1	2	3	4	5	6	7	8	9	
10	9	8	7	6	5	4	3	2	1

The pseudocode will print a report as follows:

1										1
2	1									3
1	2	3								6
4	3	2	1							10
1	2	3	4	5						15
6	5	4	3	2	1					:
1	2	3	4	5	6	7				:
8	7	6	5	4	3	2	1			:
1	2	3	4	5	6	7	8	9		:
10	9	8	7	6	5	4	3	2	1	55
35	33	:	:	:	:	:	:	:	1	

←Sum of row

Sum of columns→

4 *`Calculate_postage_rate` with **searching of element**

Design an algorithm that will read an input weight for a local mail, search an array of weight of mails and retrieve a corresponding postage rate.

In this algorithm, two paired arrays, each containing 7 elements have been established and initialized. The array `mail_weights` contains a range of weights of mails in grams and the array `postage_rates` contains a corresponding array of postage rates in cents as follows:

LOCAL MAIL RATES		
Rates for mail items less than 2kg	For Letters, Postcards, Printed Papers and Small Packets	
	Weight Step Not Over	**Standard Mail**
	Regular	
	`mail_weights`	**`postage_rates`**
	20g	26¢
	40g	32¢
	Large	
	100g	50¢
	250g	80¢
	500g	$1.00
	1kg	$2.55
	2kg	$3.35

	Array `mail_weights`
1	20
2	40
3	100
4	250
5	500
6	1000
7	2000

Array `postage_rates`
26
32
50
80
100
255
335

(A) *Defining Diagram*

- The defining diagram is as follows:

Input	Processing	Output
Entry mail's weight	Prompt for entry mail's weight Get `entry_weight` Search `mail_weights` array Compute postage rate Display postage rate	 postage_rate or error_message

(B) *Requirements and Control Structures*

- Two arrays: `mail_weights` and `postage_rates` (already initialized)

```
DECLARE mail_weights  : ARRAY[1:?] OF INTEGER
DECLARE postage_rates : ARRAY[1:?] OF INTEGER
```

- A **DO-WHILE** loop to search the `mail_weights` array and hence retrieve the postage rate from the `postage_rates` array.

- Handling invalid input

(C) *Solution Algorithm*

Copy and complete the algorithm below:

```
Calculate_postage_rate

  Set max_number_elements to 7
  Set index to 1
  Prompt for entry mail's weight
  Get entry_weight

  ...

ENDCalculate_postage_rate
```

5 `Calculate_postage_rate`

Design an algorithm that will read an input weight for a local mail, search an array of weight of mails and retrieve a corresponding postage rate.

The array `postage_rates` is a two-dimensional array. It is two-dimensional because the calculation of the postage rate of mailing an article depends on two values; the mail's weight and the zone to which it is to be mailed, namely 1, 2, 3 or 4.

LOCAL MAIL RATES					
Rates for mail items less than 2kg	For Letters, Postcards, Printed Papers and Small Packets.				
	Weight Step	Standard Mail			
	mail_weights	**postage_rates**			
		ZONE			
		1	2	3	4
		Singapore	Asian	Europe	America
	Regular				
	20g	26¢	26¢	46¢	36¢
	40g	32¢	32¢	52¢	42¢
	Large				
	100g	50¢	70¢	80¢	80¢
	250g	80¢	90¢	$1.00	$1.00
	500g	$1.00	$1.50	$2.50	$1.80
	1kg	$2.55	$3.55	$5.00	$4.00
	2kg	$3.35	$4.35	$10.00	$8.00

Array mail_weights	
1	20
2	40
3	100
4	250
5	500
6	1000
7	2000

row_index (ZONE) \ column_index	2D-Array postage_rates(row_index, column_index)						
	1	2	3	4	5	6	7
1	(1,1) 26	32	50	80	100	255	(1,7) 335
2	26	32	70	90	150	355	435
3	46	52	80	100	250	500	1000
4	(4,1) 36	42	80	100	180	400	(4,7) 800

- Assume data has been loaded and stored in arrays:

```
DECLARE mail_weights: [1:] OF INTEGER
DECLARE postage_rates: [1:?, 1:?] OF INTEGER
```

- Complete the pseudocode below with searching of a two-dimensional array (indicated by ...). The code must fulfil the following requirements:
  - Use the inputs: `weight_of_mail`, `mail_zone`.
  - Must include code for handling of **invalid** input
  - Use the output: `postage_charge`

Calculate_postage_charge_by_zone

Prompt for weight_of_mail, mail_zone
Get weight_of_mail, mail_zone
Set row_index to 1
Set element_found to false
Set postage_charge to zero
Display postage_charge

...

ENDCalculate_postage_charge_by_zone

6 *The container with the most water

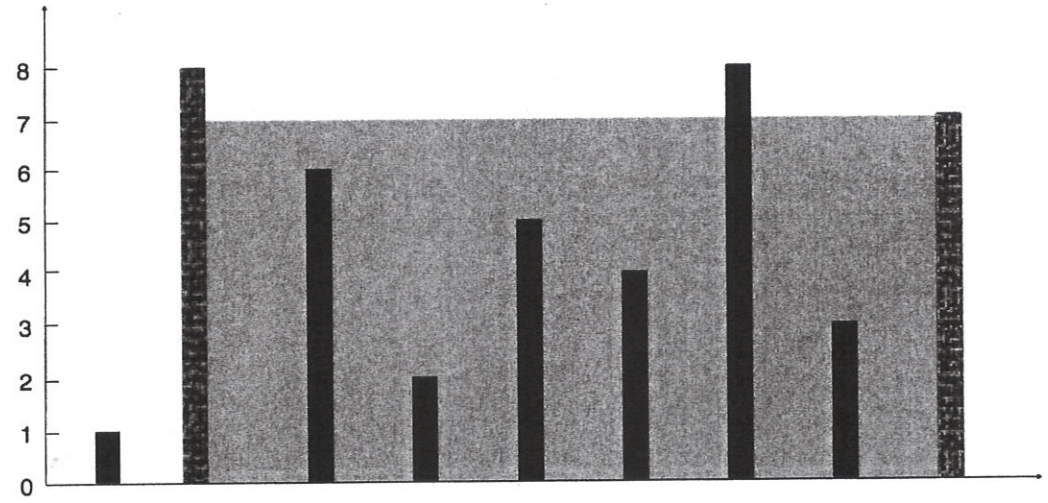
Given `n` non-negative integers a_1, a_2, ..., a_n, each of which represents a point `(i,`a_i`)` in the coordinate plane, draw `n` vertical lines in the coordinate plane.

The two end points of the vertical line `i` are `(i,`a_i`)` and `(i,0)` respectively.

Find two of the lines so that the container that they form with the x-axis can hold the most water.

Note: You cannot tilt the container.

An example is given below:



Input: `[1,8,6,2,5,4,8,3,7]`
Output: `49`

Explanation:
The vertical lines in the figure represent the elements of the input array `[1,8,6,2,5,4,8,3,7]`. In this case, the maximum volume of water that the container can (shown by the grey part) is `49`.

Write a pseudocode for the algorithm to find the maximum volume of water that the container can hold.

Dry run (desk check) your algorithm with the following cases:

Case	Input	Expected Output
1	[1, 1]	1
2	[4, 3, 2, 1, 4]	16
3	[1, 2, 1]	2

Document the process using a trace table.

1 Solution: Tutorial 6 Qn 7

```
DECLARE ROW, COL: INTEGER
CONSTANT ROWSIZE = 5, COLSIZE = 6
DECLARE SUBJRESULTS: ARRAY[1: ROWSIZE,1: COLSIZE] OF INTEGER
DECLARE ROWSUM: ARRAY[1: ROWSIZE] OF INTEGER
// Arrays Initialization
SUBJRESULTS ←
    [ [65,  68,  73,  85,  82,  87],
      [74,  87,  90,  88,  87,  88],
      [88,  97,  91,  92,  90,  89],
      [91,  83,  78,  89,  79,  87],
      [65,  76,  67,  50,  60,  66]   ]

ROWSUM ←   [0,  0,  0,  0,  0   ]
```

	SUBJRESULTS COL					
ROW	1	2	3	4	5	6
1	65	68	73	85	82	87
2	74	87	90	88	87	88
3	88	97	91	92	90	89
4	91	83	78	89	79	87
5	65	76	67	50	60	66

```
//Adding all elements of a row and store it in ROWSUM
//Using REPEAT-UNTIL, scan all rows of 2-D array SUBJRESULTS
ROW ← 1
REPEAT
    // Scan all elements (value of each column) of a row
    COL ← 1
    REPEAT
        // Add SUBJRESULTS[ROW, COL] to ROWSUM[ROW]
        ROWSUM[ROW] ← ROWSUM[ROW] + SUBJRESULTS[ROW, COL]
        COL ← COL + 1
    UNTIL COL > COLSIZE
    ROW ← ROW + 1
UNTIL ROW > ROWSIZE
```

Tracing table:		
ROW	COL	ROWSUM[ROW]
1	1	65
	2	133
	3	206
	4	291
	5	371
	6	458
2	1	74
	2	161
	3	
	4	
	5	
	6	

```
//Report generation
//Scan all rows of 2-D array SUBJRESULTS
ROW ← 1
REPEAT
//Scan all elements (values of each column) of a row
    COL ← 1
    REPEAT
        // Print the element
        PRINT SUBJRESULTS[ROW, COL]
        COL ← COL + 1
    UNTIL COL > COLSIZE
    //Print the average of the row
    PRINT "AVG = ", ROWSUM[ROW] / COLSIZE
    PRINT newline
    ROW ← ROW + 1
UNTIL ROW > ROWSIZE
```

Python solution:

```
#Tutorial 6 Arrays Question 3(a)
#TJC
#Programmer: Fong KK
#-----------------------------------------------------------
#CONSTANT ROWSIZE = 11, COLSIZE = 11
ROWSIZE = 12
COLSIZE = 12
#DECLARE ROW, COL: INTEGER
ROW = 0; COL = 0
# Using last column store sum of elements of row, and
#       last row store sum of elements of column

#DECLARE TRIValue: ARRAY [1: ROWSIZE,1: COLSIZE] OF INTEGER
TRIValue = [[0 for ROW in range(ROWSIZE)] for COL in range(COLSIZE)]

for ROW in range(1, ROWSIZE-1, 1):
    #print('ROW=', ROW,'==', end=' ')
    for COL in range(1, ROW+1, 1):
        #print('COL=', COL,'**', end=' ')
        TRIValue[ROW][COL] = COL
        #[Last column = Sum of all elements in the row horizontally]
        TRIValue[ROW][COLSIZE-1] = TRIValue[ROW][COLSIZE-1] + TRIValue[ROW][COL]
        #[Last row = Sum of all rows in the column vertically]
        TRIValue[ROWSIZE-1][COL] = TRIValue[ROWSIZE-1][COL] + TRIValue[ROW][COL]
        #print('[',ROW,']','[',COL,']','=',TRIValue[ROW][COL], end=' ')
    #print('\n')
print('aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\n')
for ROW in range(1, ROWSIZE, 1):
    for COL in range(1, COLSIZE, 1):
        if (TRIValue[ROW][COL] != 0):
            print("{0:3d}".format(TRIValue[ROW][COL]), end=' ')
        else:
            print("{0:3s}".format(""), end=' ')
    print('\n')
```

Output

```
File  Edit  Shell  Debug  Options  Window  Help
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:54:40) [MS(
 on win32
Type "copyright", "credits" or "license()" for more info:
>>>
 RESTART: D:\Computing\1. 9596 Computing\1.2 Algorithm De
ut_4_ArraysQ3_(a).py
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

  1                                                   1
  1   2                                               3
  1   2   3                                           6
  1   2   3   4                                      10
  1   2   3   4   5                                  15
  1   2   3   4   5   6                              21
  1   2   3   4   5   6   7                          28
  1   2   3   4   5   6   7   8                      36
  1   2   3   4   5   6   7   8   9                  45
  1   2   3   4   5   6   7   8   9  10              55
 10  18  24  28  30  30  28  24  18  10

>>>
```

3(a) Solution:

```
CONSTANT ROWSIZE = 11, COLSIZE = 11
DECLARE ROW, COL: INTEGER
// Note: Using last column store sum of elements of row, and
//       last row store sum of elements of column
DECLARE TRIValue: ARRAY [1: ROWSIZE, 1: COLSIZE] OF INTEGER        Column 11
// Initialization
TRIValue ←  [  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
               [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
               [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
               [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
               [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
               [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
               [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
               [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
               [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
               [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Row 11         [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]  ]

//Assign values into elements of array column by column for all rows
FOR ROW = 1 TO ROWSIZE-1    //Skip last row
     FOR COL = 1 TO ROW
     // Assign given value to relevant elements in the row
          TRIValue[ROW, COL] = COL

          //Add relevant value of column to last column
          //[Sum of all elements in the row horizontally]
          TRIValue[ROW, COLSIZE] =
                    TRIValue[ROW, COLSIZE] + TRIValue[ROW, COL]

          //Add relevant value of column to last row
          //[Sum of all rows in the column vertically]

          TRIValue[ROWSIZE, COL] =
                    TRIValue[ROWSIZE, COL] + TRIValue[ROW, COL]
     ENDFOR
ENDFOR
```

Data generated and stored in `TRIValue`

Col ↓

Row	1	2	3	4	5	6	7	8	9	10	11
→1	1	0	0	0	0	0	0	0	0	0	1
2	1	2	0	0	0	0	0	0	0	0	3
3	1	2	3	0	0	0	0	0	0	0	6
4	1	2	3	4	0	0	0	0	0	0	10
5	1	2	3	4	5	0	0	0	0	0	15
6	1	2	3	4	5	6	0	0	0	0	21
7	1	2	3	4	5	6	7	0	0	0	28
8	1	2	3	4	5	6	7	8	0	0	36
9	1	2	3	4	5	6	7	8	9	0	45
10	1	2	3	4	5	6	7	8	9	10	55
11	10	18	24	28	30	30	28	24	18	10	

Note: Row 11 and column 11 will be updated after the change of each element of TRIValue

Timeline of flow of control

		Tracing Table										
ROW	COL	TRIValue[ROW, COL]										COL 11↓
		1	2	3	4	5	6	7	8	9	10	11
ROW **1**	1	1	0	0	0	0	0	0	0	0	0	1
ROW 11		1										
ROW **2**	1	1										1
	2		2									3
ROW 11		2	2									
ROW **3**	1	1										1
	2		2									3
	3			3								6
ROW 11		3	4	3								
ROW **4**	1	1										1
	2		2									3
	3			3								6
	4				4							10
ROW 11		4	6	9	4							
ROW **5**	1	1										1
	2		2									3
	3			3								6
	4				4							10
	5					5						15
ROW 11		5	8	12	8	5						
ROW **6**												

// Printing report

```
FOR ROW = 1 TO ROWSIZE
      FOR COL = 1 TO COLSIZE
      //
            IF TRIValue[ROW, COL] <> 0
                  THEN
                        OUTPUT TRIValue[ROW, COL]
            ENDIF
      ENDFOR
      OUTPUT newline
ENDFOR
```

COL	1	2	3	4	5	6	7	8	9	10	11
ROW 1	1	0	0	0	0	0	0	0	0	0	1
2	1	2	0	0	0	0	0	0	0	0	3
3	1	2	3	0	0	0	0	0	0	0	6
4	1	2	3	4	0	0	0	0	0	0	10
5	1	2	3	4	5	0	0	0	0	0	15
6	1	2	3	4	5	6	0	0	0	0	21
7	1	2	3	4	5	6	7	0	0	0	28
8	1	2	3	4	5	6	7	8	0	0	36
9	1	2	3	4	5	6	7	8	9	0	45
10	1	2	3	4	5	6	7	8	9	10	55
11	10	18	24	28	30	30	28	24	18	10	