Cambridge Assessment International Education

Cambridge International AS & A Level

COMPUTER SCIENCE **9618/43**

Paper 4 Practical **October/November 2024**

2 hours 30 minutes

You will need: Candidate source files (listed on page 2)
evidence.doc

6078340176

INSTRUCTIONS

- Carry out every instruction in each task.
- Save your work using the file names given in the task as and when instructed.
- You must **not** have access to either the internet or any email system during this examination.
- You must save your work in the evidence document as stated in the tasks. If work is **not** saved in the evidence document, you will **not** receive marks for that task.
- You must use a high-level programming language from this list:
  - Java (console mode)
  - Python (console mode)
  - Visual Basic (console mode)
- A mark of **zero** will be awarded if a programming language other than those listed here is used.

INFORMATION

- The total mark for this paper is 75.
- The number of marks for each question or part question is shown in brackets [ ].

This document has **16** pages. Any blank pages are indicated.

Open the evidence document, **evidence.doc**

Make sure that your name, centre number and candidate number appear on every page of this document. This document must contain your answers to each question.

Save this evidence document in your work area as:
evidence_ followed by your centre number_candidate number, for example: **evidence_zz999_9999**

A class declaration can be used to declare a record. If the programming language used does **not** support arrays, a list can be used instead.

One source file is used to answer **Question 1**. The file is called **`Data.txt`**

1 A program sorts string data using different sorting methods.

(a) The text file `Data.txt` stores string data items. Each data item is on a new line in the text file.

The function `ReadData()`:

- has a local array of strings that can store 45 items
- reads each line of data and stores it in the array
- returns the array.

Write program code for the function `ReadData()`.

Save your program as **Question1_N24**.

Copy and paste the program code into part **1(a)** in the evidence document.

[6]

(b) The function `FormatArray()` takes an array of strings as a parameter. It concatenates the contents of the array into one string with a space between each array element. The function returns the concatenated string.

(i) Write program code for `FormatArray()`.

Save your program. Copy and paste the program code into part **1(b)(i)** in the evidence document.

[2]

(ii) The main program:

- calls `ReadData()` and stores the returned array
- calls `FormatArray()` with the returned array and outputs the returned string.

Write program code for the main program.

Save your program. Copy and paste the program code into part **1(b)(ii)** in the evidence document.

[3]

(iii) Test your program.

Take a screenshot of the output.

Save your program. Copy and paste the screenshot into part **1(b)(iii)** in the evidence document.

[1]

(c) The function `CompareStrings()`:

- takes two strings as parameters
- compares each string, one character at a time, to identify which string comes first alphabetically. If the first two characters are the same, the second character of each string is compared. This continues until the two characters are different.

The function:

- returns `1` if the first parameter comes before the second alphabetically
- returns `2` if the second parameter comes before the first alphabetically.

Write program code for `CompareStrings()`.

Assume that all strings are in lower case.

Assume that a difference between two strings will always be identified before the end of one string is reached.

Do **not** use an in-built string comparison function.

The strings must be compared one character at a time.

Save your program. Copy and paste the program code into part **1(c)** in the evidence document.

[4]

(d) The function `Bubble()` takes an array of strings as a parameter and sorts the data into ascending alphabetical order, using a bubble sort. The bubble sort uses `CompareStrings()` to compare each string.

The function returns the sorted list.

(i) Write program code for `Bubble()`.

Save your program. Copy and paste the program code into part **1(d)(i)** in the evidence document.

[3]

(ii) Write program code to amend the main program to:

- call `Bubble()` with the unsorted array as a parameter
- call `FormatArray()` with the sorted array and output the returned string.

Save your program. Copy and paste the program code into part **1(d)(ii)** in the evidence document.

[2]

(iii) Test your program.

Take a screenshot of the output.

Save your program. Copy and paste the screenshot into part **1(d)(iii)** in the evidence document.

[1]

2 A computer program is designed to simulate horses doing show jumping. In show jumping, horses jump over obstacles called fences. A horse successfully jumps a fence if it does not knock the fence down.

The program is written using Object-Oriented Programming (OOP).

The class `Horse` stores data about the horses.

Horse	
`Name : STRING`	stores the name given to the horse
`MaxFenceHeight : INTEGER`	stores the maximum height in cm that the horse can jump, for example 132
`PercentageSuccess : INTEGER`	stores the percentage chance of a horse not knocking down a fence, for example 70 represents a 70% chance of jumping a fence successfully
`Constructor()`	initialises `Name`, `MaxFenceHeight` and `PercentageSuccess` to its parameter values
`GetName()`	returns the name of the horse
`GetMaxFenceHeight()`	returns the maximum height the horse can jump
`Success()`	calculates and returns the percentage chance of a horse successfully jumping a specific fence

(a) (i) Write program code to declare the class `Horse` and its constructor.

Do **not** declare the other methods.

Use your programming language's appropriate constructor.

All attributes must be private. If you are writing in Python, include attribute declarations using comments.

Save your program as **Question2_N24**.

Copy and paste the program code into part **2(a)(i)** in the evidence document.

[4]

(ii) The get methods `GetName()` and `GetMaxFenceHeight()` each return the relevant attribute.

Write program code for the get methods.

Save your program.

Copy and paste the program code into part **2(a)(ii)** in the evidence document.

[3]

(b) The array `Horses` stores objects of type `Horse`.

(i) The program has two horses:

- The horse named 'Beauty' can jump a maximum height of 150 cm and has a success percentage rate of 72%.
- The horse named 'Jet' can jump a maximum height of 160 cm and has a success percentage rate of 65%.

Write program code to:

- declare the array, `Horses`, local to the main program with space for **two** `Horse` objects
- store the **two** horses described in the array
- output the name of both `Horse` objects from the array.

Save your program. Copy and paste the program code into part **2(b)(i)** in the evidence document.

[5]

(ii) Test your program.

Take a screenshot of the output.

Save your program. Copy and paste the screenshot into part **2(b)(ii)** in the evidence document.

[1]

(c) The class `Fence` stores data about the fences. Each fence has a height in cm and a risk number.

The risk is a whole number between 1 and 5 inclusive. A risk of `1` means the fence is the easiest type to jump. A risk of `5` means the fence is the hardest type to jump.

Fence	
`Height : INTEGER`	stores the height of the fence in cm the height is between 70 and 180 inclusive
`Risk : INTEGER`	stores the risk as a whole number between 1 and 5 inclusive
`Constructor()`	initialises `Height` and `Risk` to its parameter values
`GetHeight()`	returns the height of the fence
`GetRisk()`	returns the risk of the fence

(i) Write program code to declare the class `Fence`, its constructor and get methods.

Use your programming language's appropriate constructor.

All attributes must be private.

If you are writing in Python, include attribute declarations using comments.

Save your program.

Copy and paste the program code into part **2(c)(i)** in the evidence document.

[4]

(ii) The array `Course` stores **four** `Fence` objects. The user inputs the height and risk for each fence, and these are validated before each fence is created.

Amend the main program to:

- declare the local array `Course`
- take as input the data for **four** fences from the user
- loop the input until both the height and risk are valid for each fence
- create an instance of `Fence` for each of the **four** valid fences and store each instance in the array.

Save your program.

Copy and paste the program code into part **2(c)(ii)** in the evidence document.

[5]

(d) The chance of a horse jumping a fence without knocking it down is calculated as follows.

If the height of the fence is more than the maximum height a horse can jump, the success percentage is 20% of the horse's `PercentageSuccess`. The risk does not affect this value.

If the height of the fence is less than or equal to the maximum height a horse can jump, the risk gives a modifier value to multiply with the horse's `PercentageSuccess`.

The risk values and their modifiers are given in this table:

Risk	Modifier
5	0.6
4	0.7
3	0.8
2	0.9
1	1.0

For example:

- The horse Jet has `PercentageSuccess` of 65 and `MaxFenceHeight` of 160.
- A fence has a height of 140 and a risk of 3.
- The height of the fence is less than the horse's `MaxFenceHeight`, therefore the risk is used.
- The risk of 3 gives the modifier 0.8.
- The modifier 0.8 is multiplied by the horse's `PercentageSuccess` of 65, which gives 52.
- The chance of the horse successfully jumping this fence is 52%.

The method `Success()` in the `Horse` class:

- takes the height and risk of a fence as parameters
- calculates the percentage chance of success for that horse jumping the fence without knocking it down
- returns the calculated percentage chance of success as a real number.

Write program code for `Success()`.

Save your program.

Copy and paste the program code into part **2(d)** in the evidence document.

[5]

(e) (i) Write program code to amend the main program to:

- calculate and output the chance of the first horse jumping each of the **four** fences without knocking each fence down
- calculate and output the chance of the second horse jumping each of the **four** fences without knocking each fence down.

All outputs must have appropriate messages including the name of the horse and the fence number.

An example output for one horse jumping two fences is:

```
"The horse Fox at fence 1 has a 68% chance of success
The horse Fox at fence 2 has a 72% chance of success"
```

Save your program.

Copy and paste the program code into part **2(e)(i)** in the evidence document.

[3]

(ii) Write program code to amend the main program to:

- calculate and output the average chance of success for each horse jumping over all **four** fences without knocking each fence down (the average is the total of values divided by the quantity of values). An example output for one horse jumping all of the fences is:

```
"The horse Fox has an average 70% chance of jumping over all four
fences"
```

- output the name of the horse that has the highest average chance of success.

You can assume that each average will be different.

All outputs must have appropriate messages.

Save your program.

Copy and paste the program code into part **2(e)(ii)** in the evidence document.

[2]

(iii) Test your program with the following input data for **four** fences:

Height	Risk
152	5
121	1
130	3
145	4

Take a screenshot of the output.

Save your program.

Copy and paste the screenshot into part **2(e)(iii)** in the evidence document.

[2]

3 A linked list stores positive integer data in a 2D array. The first dimension of the array stores the integer data. The second dimension of the array stores the pointer to the next node in the linked list.

A linked list node with no data is initialised with the integer `-1`. These nodes are linked together as an empty list. A pointer of `-1` identifies that node as the last node.

The linked list can store 20 nodes.

The global 2D array `LinkedList` stores the linked list.

`LinkedList` is initialised as an empty list. The data in each node is initialised to `-1`. Each node's pointer stores the index of the next node. The last node stores the pointer value `-1`, which indicates it is the last node.

The global variable `FirstEmpty` stores the index of the first element in the empty list. This is the first node in the empty linked list when it is initialised, which is index `0`.

The global variable `FirstNode` stores the index of the first element in the linked list. There is no data in the linked list when it is initialised, so `FirstNode` is initialised to `-1`.

This diagram shows the content of the initialised array.

```
FirstEmpty = 0

FirstNode = -1
```

Index	Data	Pointer
0	-1	1
1	-1	2
2	-1	3
3	-1	4
4	-1	5
19	-1	-1

(a) Write program code for the main program to declare and initialise `LinkedList`, `FirstNode` and `FirstEmpty`.

Save your program as **Question3_N24**.

Copy and paste the program code into part **3(a)** in the evidence document.

[2]

(b) The procedure `InsertData()` takes **five** positive integers as input from the user and inserts these into the linked list.

Each data item is inserted at the front of the linked list.

The table shows the steps to follow depending on the state of the linked list:

Linked list state	Steps
not full	insert the data in the index pointed to by `FirstEmpty` change the pointer to the index pointed to by `FirstNode` change the values of `FirstNode` and `FirstEmpty`
full	end the procedure

Any node that is at the end of the linked list has a pointer of `-1`.

Write program code for `InsertData()`.

Save your program.

Copy and paste the program code into part **3(b)** in the evidence document.

[6]

(c) The procedure `OutputLinkedList()` outputs the data in the linked list in order by following the pointers from `FirstNode`.

(i) Write program code for `OutputLinkedList()`.

Save your program.

Copy and paste the program code into part **3(c)(i)** in the evidence document.

[2]

(ii) Amend the main program to call `InsertData()` and then `OutputLinkedList()`.

Save your program.

Copy and paste the program code into part **3(c)(ii)** in the evidence document.

[1]

(iii) Test your program with the test data:

5 1 2 3 8

Take a screenshot of the output.

Save your program.

Copy and paste the screenshot into part **3(c)(iii)** in the evidence document.

[1]

(d) The procedure `RemoveData()` removes a node from the linked list.

The procedure takes the data item to be removed from the linked list as a parameter.

The procedure checks each node in the linked list, starting with the node `FirstNode`, until it finds the node to be removed. This node is added to the empty list, and pointers are changed as appropriate. The procedure only removes the first occurrence of the parameter.

Assume that the data item being removed is in the linked list.

(i) Write program code for `RemoveData()`.

Save your program. Copy and paste the program code into part **3(d)(i)** in the evidence document.

[5]

(ii) Amend the main program to:

- call `RemoveData()` with the parameter 5
- output the word `"After"`
- call `OutputLinkedList()`.

Save your program. Copy and paste the program code into part **3(d)(ii)** in the evidence document.

[1]

(iii) Test your program with both sets of given test data:

Test data set 1: 5 6 8 9 5

Test data set 2: 10 7 8 5 6

Take a screenshot of each output.

Save your program. Copy and paste the screenshot(s) into part **3(d)(iii)** in the evidence document.

[1]

BLANK PAGE