NATIONAL JUNIOR COLLEGE
Mathematics Department

General Certificate of Education Advanced Level
Higher 2

# COMPUTING 9569/2

Paper 2 (Lab-based)

**17 Aug 2021**
**3 hours**

| | |
|---|---|
| Additional Materials: | Electronic version of `COVID19.CSV` data file |
| | Electronic version of `PAID_EMPLOYEES.CSV` data file |
| | Electronic version of `ALL_EMPLOYEES.CSV` data file |
| | Electronic version of `NODES.TXT` data file |
| | Insert Quick Reference Guide |

**READ THESE INSTRUCTIONS FIRST**

Answer **all** questions**.**

All tasks must be done in the computer laboratory. You are not allowed to bring in or take out any pieces of work or materials on paper or electronic media or in any other form

Approved calculators are allowed.

Save each task as it is completed.

The use of built-in functions, where appropriate, is allowed for this paper unless stated otherwise.

Note that up to x marks out of x will be awarded for the use of common coding standards for programming style.

The number of marks is given in the brackets [ ] at the end of each question or part question.
The total number of marks for this paper is x.

This document consists of x printed pages and x blank pages.

**Instructions to candidates:**

Copy the folder from the thumb drive to the PC's desktop and rename the folder on the desktop to `<your name>_<NRIC number>`.(For example, TanKengHan_T0123456A).
All the resource files are found in the folder and you should work on the folder in the desktop.
Your program code and output for each of Task 1 to 3 should be saved in a single `.ipynb` file. For example, your program code and output for Task 1 should be saved as `Task_1_<your name>_<NRIC number>.ipynb`.
You should have a total of **three** `.ipynb` files to submit at the end of the paper.
At the end of the exam, copy the working folder on your desktop to the thumb drive.

---

**Task 1**

The file `COVID19.CSV` contains data of confirmed cases of COVID19 infections over a period of time.

Each row, except the header row, in the file records the cases for a particular province/state of a country **or** for the entire country.

The rows are not sorted in any order.

For each sub-task, add a comment statement, at the beginning of the code using the hash symbol "#", to indicate the sub-task the program code belongs to, for example:

```
In [1]:  #Task 1.1
         Program code
        Output:
```

The file provided is `COVID19.CSV`

**Task 1.1**

Write Python code to determine the following statistics from the file given:
- number of different countries in the data collection.
- number of days that data were collected.
- start date and end date of the data collection .

Output the 3 statistics above. [3]

**Task 1.2**

Write Python code to aggregate the data for each Country/Region, for each of the day and write the new data in a file named `countries.csv` in the following format:

```
Country, <start_date>,..., <end_date>
```

where `start_date` and `end_date` are the start and end dates of the data collected. The file is sorted in **ascending** order of the Country/Region name.

The file contents should look like this:

```
Country/Region,1/1/20,1/2/20,1/3/20,1/4/20,...
Australia,0,0,0,0,...
France,0,0,2,3,...
Mainland China,547,639,916,1399,...
:
```

[10]

**Task 1.3**

Write Python code to determine the top five country/region with the **largest** single day increase in confirmed cases and the date this occurred.

Example of output:

```
Mainland China    15133  2/13/20
South Korea       851    3/3/20
Iran              835    3/3/20
Italy             587    3/4/20
Others            99     2/17/20
```

[5]

The contents of the file, `countries.csv`, created in **Task 1.2** is to be stored as documents in a collection named **`cases`**, on a MongoDB database named **`covid`**.

Each document must be store in the following structure:

```
{
    "date": "1/22/20",
    "countries":
    [
             {
               "country": "Afghanistan",
               "cases": 0
             },
             {
               "country": "Algeria",
               "cases": 0
             },
             {
               "country": "Andorra",
               "cases": 0
             },
             {
               "country": "Argentina",
               "cases": 0
             }
      ]
}
```

**Task 1.4**

Write Python code to create the MongoDB database and collection and insert all the documents into the collection. [5]

**Task 1.5**

Write Python code to connect to the MongoDB database created in **Task 1.4** and print the number of cases reported on 1/28/20 in Mainland China. [2]

Name and save your Jupyter Notebook as
`Task1_<your name>_<NRIC number>.ipynb.`

**Task 2**

A human resource department of a Law firm needs to keep records of its employees and at the end of the month, it needs to process the pay due to the employees.

The class `Employee` will store the data in the following attributes:

| **Attributes** | **Description** |
|---|---|
| `name: STRING` | `Name of employee` |
| `NRIC: STRING` | `NRIC of employee` |
| `contact: STRING` | `Telephone no. of employee` |

The class has 3 methods defined on it:

| **Methods** | **Description** |
|---|---|
| `constructor(name,NRIC,contact)` | Initialises the **private** attributes name, NRIC and contact. Only the contact can be modified after the object instance is created. |
| `__repr__()` | A string representation of the object in the following format: <NRIC>:<name> |
| `compare_with(object: Employee)` | The method takes in an Employee object and compares the instance with the object. The method<br>• returns -1 if the instance's name is lexicographically smaller than the object's name<br>• returns 1 if the instance's name is lexicographically larger than the object's name<br>• returns 0 if the names are the same |

The files provided are `PAID_EMPLOYEES.CSV,ALL_EMPLOYEES.CSV`

**Task 2.1**

Write Python code to implement the class `Employee.` [4]

There are 2 types of employee that need to be pay at the end of the month, full-time and part-time employee.

Full-time employees have monthly salary. Part-time employees are paid hourly and their monthly pay is based on the number of hours worked in a month. The number of hours work in a month is flexible. Their pay at the end of the month is days worked multiplied by their hourly rate.

Two classes **`FTEmployee`** and **`PTEmployee`** are to be implemented by inheritating from the **`Employee class`** implemented in **Task 2.1**. The `FTEmployee` must store the monthly salary and the `PTEmployee` must store the hourly rate and days worked in a month. You can assume that the days worked will be reset after the monthly paid out.

For **each** of the FTEmployee and PTEmployee classes, the following needs to be implemented.

- The `constructor()` method must be extended to reflect the extra attributes stored.
- The `__repr__()` method must be extended to reflect the extra attributes stored and the appropiate getters and setters.
- A method `calculate_pay()` must be implemented to return the monthly pay for the employee.

- The method `compare_with(Object)` is to be implemented to :
    - returns -1 if the instance's monthly pay is less than the object instance passed in.
    - returns 1 if the instance's monthly pay is more than the object instance passed in.
    - re-use the `Employee` class `compare_with()` algorithm.

**Task 2.2**

Write Python code to implement the classes `FTEmployee` and `PTEmployee`. You should make full use of Inheritance, Polymorphism and Encapsulation in your code. [5]

**Task 2.3**

The file `PAID_EMPLOYEES.CSV` contains a list of both full-time and part-time employee records. Full time employees have a value `"FT"` in the type attribute and part time employees have an a value of `"PT"`.

Read the data from the file and create a Python list named **`paid_employees`** of `FTEmployee` and `PTEmployee` objects. Print the contents of the `paid_employee` list. The output should look like this:

```
[Harley:F2879124I:40:128,Barris:G2162825P:40:134,Andre:G70286
12T:40:149,Dev:G8804899R:60:56,Florry:G5935054J:7642,Lauritz:
F3295152O:5657,Monroe:G9637986L:3936,Van:F4203985B:4198,Corre
na:F4954357C:40:72,...]
```

[2]

**Task 2.4**

Using an **in-place insertion sort** algorithm, write a Python function named **`insert_sort`**, to sort the **`paid_employee`** list created in **Task 2.3** by using the `compare_with` method in **ascending** order of their monthly pay. If they have the same pay then they are sorted in lexicographic ascending order of the names. You are not allow to use the Python built-in sort functions.

Print the sorted list. The sorted list should look like this:

```
[Gillan:F8422204Y:40:43,
 :
 Correna:F4954357C:40:72,
 Benedicto:S3238298D:60:52,
 Arlinda:F3036905T:3206,
 Gretal:G7505051N:3236,
 Dev:G8804899R:60:56,
 Joey:G83313890:3716,
:
]
```

[4]

Some employees in the law firm are pro-bono lawyers, they do not get paid every month.

The file `ALL_EMPLOYEES.CSV` contains a list of pro-bono, full-tme and part-time employees. Full time employees have a value "FT" in the type attribute, part time employees have an a value of "PT" and pro bono employees have a value of "EM".

**Task 2.5**

Read the data from the file `ALL_EMPLOYEES.CSV` and create a Python list named `all_employee` of `Employee, FTEmployee and PTEmployee` objects

Write Python code to **sort the list inplace**, such that the full-time and part-time employees are sorted according to the rules in **Task 2.4**. Pro bono employees wil be sorted in ascending lexicographic order of their names and they all must appear before the full-time and part-time employees in the Python list. You can re-use the `insert_sort` function in **Task 2.4**, but you are not allow to use any Python built-in sort functions.
Print the sorted list. The sorted order look like this:

```
[
 Aeriela:G8023668V,
 :
 Christye:G79735010,
 Corabel:G45585914Q,
 :
 Waylan:F7156030H,
 Wilhelm:G7229873Q,
 Joelie:S2743497J:50:50,
 :
 Benedicto:S3238298D:60:52,
 Arlinda:F3036905T:3206,
 :
]
```

[6]

**Task 3**

A hardware token is designed to store data on its flash memory. The flash memory has limited storage capacity. The flash drive memory is divided into `N` number of slots, each slot having the capacity to store one item of data. The device allowed data to be store and remove from the memory slots.
A singly-linked Linked List is used to implement the access to this flash memory.

The Node object is use to implement the data structure for a memory slot. The `Node` class is described as follows:

```
                  Node
+data: OBJECT
+next: INTEGER
+constructor(data:OBJECT)
-__repr__() : STRING
```

where

- `data,` stores the data object.
- `next,` a index reference to the next node. A `None` value indicates no more next node .
- `constructor(data:OBJECT)` initialises the `data` and `next` attributes.
- `_repr__()` returns a string representation of the `Node` object as follows: `<data>` where data is the value of its data attribute.

The Linked List class is used to implement the operations and data structure for the flash memory as described below:

```
                     LinkedList
+start: INTEGER
+free: INTEGER
-memory: ARRAY[0:N] OF Node
+constructor(N: INTEGER)
-__repr__() : STRING
+insert(position:INTEGER, node: Node):BOOLEAN
+delete(position:INTEGER): BOOLEAN
+insert_seq(position:INTEGER,
             nodes: ARRAY[0:N] OF Node): BOOLEAN
```

where

- `start`, an index in the memory array where the Linked List starts.
- `free`, an index in the memory array where the free memory slots starts.
- `constructor(N)` performs the following:
  - initialises the `memory` array to a size of `N` elements of `Node` objects. The `Node` object in the array should have their `next` attribute set to the index of its adjacent `Node`, except for the last `Node` which will set to `None`. The `data` attribute should be set to `None`.
  - initialise the `start` and `free` attributes.
- `__repr__()` returns a string representation of the Linked List object as follows:
  `[<Node>, <Node>,… ]` where `<Node>` is the string representation of a `Node` object. This is the logical data linked list and **NOT** the contents of the `memory` array. See illustration below.

- `insert(position:INTEGER, node: Node)` returns `TRUE` if a node is inserted and `FALSE` if there are no more free slots in the `memory` array.
    - The `position` parameter specifies the position in the Linked List to insert the `node,` starting at 1. For example, position 1, inserts at the beginning of the Linked List, position 3, inserts it after the first 2 nodes. If `position` is greater than the size of the Linked List, it will be appended at the end.
- `delete(position:INTEGER)` returns `True` if the node at `position` is deleted, `False` if the position is not a valid position in the Linked List. The deleted node's memory slot must be returned to the free slots list
- `insert_seq(position:INTEGER,nodes: ARRAY[0:N] OF Node)` returns `True` if all the nodes in the `nodes` array are inserted into the Linked List staring at `position`, `False` if there are no more slots to store the the entire `nodes` array.

There are 2 logical Linked List that needs to be maintained in the specification described above as illustrated by the following diagram. The Linked List that stores data and a free node list where free memory slots can be obtained to store data. All the nodes are stored in the `memory` attribute of the `LinkedList` class

$$\begin{tikzcd}
\text{start} \arrow[r] & \boxed{\text{data}} \arrow[r] & \boxed{\text{data}} \arrow[r] & \boxed{\text{data}} \\
\text{free} \arrow[r] & \boxed{\text{None}} \arrow[r] & \boxed{\text{None}} &
\end{tikzcd}$$

The file provided is `NODES.TXT`

**Task 3.1**

Write Python code to implement the classes `Node` and `LinkedList`, except for the `insert` , `insert_seq` and `delete` methods. [6]

**Task 3.2**

The `insert` and `delete` methods are to be implemented using recursive algorithm. Write the Python code for the `insert` and `delete` methods.

[15]

**Task 3.3**

Write the Python code for the `insert_seq` method. [3]

**Task 3.4**

The file `NODES.TXT` contains the test cases to be used for inserting nodes to the Linked List. The first field on each row is the position to insert the node in the Linked List, the second field is the constructor to create the node. The nodes must be inserted in the same order as they appear in the file.

Write Python code to create the Linked List that was implemented and insert the nodes as decribed above.

Print the contents of the linked list.

The output should like this:

```
[<1>,<2>,<2.5>,<3>,<4>,<4.5>,<4.8>,<5>,<8>,<10>]
```

Using the `delete` method, remove the nodes at position 6 and 9 in that order.

Print the contents of the Linked List after the two delete operations. [3]

**END OF PAPER**