NANYANG JUNIOR COLLEGE

JC2 PRELIMINARY EXAMINATIONS

Higher 2

COMPUTING 9569/02

Paper 2 (Lab-based) **23rd August 2024**

3 Hours

Additional Materials:	Removable storage device Electronic version of `Task1.db` data file Electronic version of `Task2_timing.py` 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 **6** marks out of 100 will be awarded for the use of common coding standards for programming style.

The number of marks is given in brackets [ ] at the end of each question or part question.
The total number of marks for this paper is 100.

This document consists of **9** printed pages, **1** blank page, and **1** insert.

Instruction to candidates:

Your program code and output for each of Task 1, 2 and 3 should be downloaded in a single `.ipynb` file. For example, your program code and output for Task 1 should be downloaded as

`TASK1_<your name>_<centre number>_<index number>.ipynb`

Make sure that each of your `.ipynb` files shows the required output in Jupyter Notebook.

1 Name your Jupyter Notebook as:

`TASK1_<your name>_<centre number>_<index number>.ipynb`

A credit card facility has a database of records, `Task1.db`, and is intending to implement the following:

- detect credit card fraud using the Luhn algorithm
- categorise transactions by customer.

The database has three tables to store information on:

- the user details
- the categories of items
- the transaction details.

The descriptions for the tables are as follows:

- Users (UserID, Name, Email, CreditCardNumber, CreditCardValidity)
- Transactions (TransactionID, UserID, Amount, Timestamp, CategoryID)
- Categories (CategoryID, CategoryName)

The Luhn Algorithm

The Luhn algorithm is used to validate credit card numbers. The algorithm is as follows:

1. Starting from the rightmost digit and moving left, double the value of every second digit. If doubling a digit results in a number greater than 9, subtract 9 from the result to get a single-digit number.
2. Sum all the resulting digits.
3. If the remainder is 0 when the total is divided by 10, the number is valid; otherwise, it is not valid.

For example, given the 16-digit credit card number 4539148803436467:

Step 1: 4, 5, 3, 9, 1, 4, 8, 8, 0, 3, 4, 3, 6, 4, 6, 7

becomes 8, 5, 6, 9, 2, 4, 7, 8, 0, 3, 8, 3, 3, 4, 3, 7

Step 2: 8 + 5 + 6 + 9 + 2 + 4 + 7 + 8 + 0 + 3 + 8 + 3 + 3 + 4 + 3 + 7 = 80

Step 3: 80 / 10 = 8 remainder 0.

Since the remainder is equal to 0, the number is valid.

For each of the sub-tasks, 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:
```

Task 1.1

Write a function `task1_1(input_value)` that returns a Boolean.

The function should:

- validate that the `input_value` is a 16-character string made up of digits
- return `True` if the above rule is met, otherwise return `False`. [3]

Task 1.2

Write a function `task1_2(input_value)` that checks the validity of a credit card number.

The function should:

- use `task1_1` to ensure the `input_value` is a 16-character string made up of digits
- return `False` if `input_value` is invalid for any reason
- otherwise, check the validity of the `input_value`, based on the Luhn algorithm
- return `True` if it is valid, otherwise return `False`. [6]

Task 1.3

Write a Python program to fetch all the credit card numbers from the `Users` table in `Task1.db` and check their validity using your function `task1_2`.

Write program code to update the result of the validity to the `CreditCardValidity` column in the `Users` table. [4]

Task 1.4

Write a Python program that uses SQL code to fetch the following information from the database:

- `user` – user name
- `category` – category name
- `total_transaction` – total transaction amount (per user, per category)
- sorted in ascending order of category name. [5]

Write program code to write the resulting records to a file labelled `transactions.txt`:

- one record per line
- with an appropriate heading
- in a comma-separated value format. [2]

Save your Jupyter notebook for Task 1.

2 Name your Jupyter Notebook as:

```
TASK2_<your name>_<centre number>_<index number>.ipynb
```

For each of the sub-tasks, 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 2.1
          Program code
          Output:
```

Task 2.1

Write a function `task2_1(n)` that:

- takes an integer `n`
- generates `n` random integers between 0 to 1000 inclusive
- returns a list of the generated integers. [3]

Task 2.2

One method of sorting is the merge sort.

Write a function `task2_2(list_of_integers)` that:

- takes a list of integers
- implements a merge sort algorithm
- returns the sorted list of integers. [7]

Task 2.3

One method of obtaining the index of a value in a list is the linear search.

Write a function `task2_3(list_of_integers, value)` that:

- takes a list of integers and a value
- implements a linear search algorithm
- returns `-1` if the value is not found
- otherwise returns the index of any occurrence of the value in the list. [2]

Task 2.4

Another method of obtaining the index of a value in a list is the binary search.

Write a function `task2_4(list_of_integers, value)` that:

- takes a sorted list of integers and a value
- implements a binary search algorithm
- returns `-1` if the value is not found
- otherwise returns the index of any occurrence of the value in the list. [5]

Task 2.5

The `time` library is built into Python and can be used to time simple function calls. Example code is shown in `Task2_timing.py`. (The sample code assumes that it has access to the `task2_2` and `task2_4` functions.)

For searching through unsorted data, two approaches are available. A linear search can be used, or the data can be sorted (using an algorithm such as merge sort) before using a binary search.

Using the `time` module, determine the time taken to search for a value in a list of 10, 100, 1000, and 10000 integers for each approach. Display the results with the following headers:

- approach
- item_count
- time_in_seconds.

For example:

```
Approach          item_count  time_in_seconds
Linear            10          0.23245
sort and binary   10          0.34345
```

Using your results or other relevant evidence, compare and contrast the time complexities of the two approaches for searching through unsorted data. In your answer, include the orders of growth for each method. [5]

Save your Jupyter notebook for Task 2.

3 Name your Jupyter Notebook as:

`TASK3_<your name>_<centre number>_<index number>.ipynb`

A programmer is writing a class, `LinkedList`, to represent a linked list of jobs to be done. A linked list is a collection of data elements, whose order is not given by their physical placement in memory. Instead, each element points to the next.

For each of the sub-tasks, 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 3.1
          Program code
          Output:
```

Task 3.1

Write the `LinkedList` class in Python. Use of a simple Python list is not sufficient.

Each node in the linked list stores the following data:

- `priority` – an integer representing the job's priority (larger value means higher priority)
- `job` – a string description of the job to be done
- `nextNode` – a pointer to the next node, or None if the node is the last in the linked list.

The linked list should include the following methods:

- `is_empty()` returns `True` if the list is empty, otherwise it returns `False`
- `insert(priority, job)` inserts the data at the beginning (head) of the list
- `count()` returns the number of elements in the list
- `displayJobs()` should display each job's description. The number of jobs should be displayed at the end of the job listing.

  For example:
  ```
  Delegate tasks to members
  Schedule date for first meeting
  Plan project schedule
  Total jobs: 3
  ```
  [9]

Test `LinkedList` by inserting the following items in order:

Priority	Job Description
2	Schedule date for first meeting
1	Delegate tasks to members
3	Plan project schedule

Display the contents of the linked list using the `displayJobs()` method. [1]

Task 3.2

Write a Python subclass `Stack` using `LinkedList` as its superclass.

Additional push and pop methods are to be defined on the Stack class:

- `push(priority, job)` will insert the data at the top of the stack
- `pop()` will remove and return the first job in the stack, or `None` if the stack is empty. Only the job description needs to be returned. [4]

Test `Stack` by pushing the same jobs from Task 3.1 in the same order. Display the jobs popped from the stack until it is empty. [3]

Task 3.3

Write a Python subclass `PriorityList` using `Stack` as its superclass.

The insert method in the `PriorityList` subclass should ensure that elements are stored in descending order of priority (higher priority items are stored nearer the head). [5]

Test `PriorityList` by inserting the same jobs from Task 3.1 in the same order. Display the jobs popped from the stack until it is empty. [2]

Save your Jupyter notebook for Task 3.

4 Name your Jupyter Notebook as:

`TASK4_<your name>_<centre number>_<index number>.ipynb`

A simple game allows users to guess the coordinates of a hidden mine on a grid.

The game has a 3×3 grid where a single mine is hidden at a random location. Players input their guesses by specifying the x- and y-coordinates, and the game updates the grid to show whether the guess was a hit or a miss. The game ends when the user correctly guesses the mine's location.

For each of the sub-tasks, 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 4.1
           Program code
           Output:
```

Task 4.1

Write a Python program that creates the 3×3 grid by:

- initialising a global game board as a 2-D list in the format:
  `[['--', '--', '--'], ['--', '--', '--'], ['--', '--', '--']]`
- initialising a global counter `tries` with the initial value set to 0
- initialising global variables `mine_x` and `mine_y`, representing the x- and y-coordinates of the mine.

Run your program. [3]

Task 4.2

Write program code to:

- declare a function that sets a random location for the mine
- declare a function that asks the player to input x- and y-coordinates, displaying an appropriate error message until the user inputs valid coordinates (between 0 and 2)
- declare a function that takes x- and y-coordinates as parameters, replaces the contents of the board at the given coordinates with an `"X"` if the mine is at the coordinates, with an `"O"` if the mine is not at the coordinates, and updates the `tries` global variable. [7]

Test your program with a suitable set of values. [2]

Task 4.3

Write program code to:

- declare a function that displays the game board as a 3×3 grid, with each row displayed as a string on a new line
- declare a function that returns a Boolean representing whether the game has ended. The game has ended if there is an `"X"` on any of the grid coordinates
- display the game board, prompt the user to input valid coordinates, and update the game board appropriately until the game ends, by calling the functions from Tasks 4.1, 4.2, and 4.3. Display the number of tries needed by the player after the game ends. [3]

Task 4.4

The above game is to be played in a web browser.

Write a Python program and the necessary files to create a web application for the above game.

The web application should:

- allow the player to enter the x- and y-coordinates of the player's guess
- display the game board at the start of the game and after each guess
- display an appropriate error message if the player enters an invalid set of coordinates
- display the number of tries needed for the player to guess the location of the mine at the end of the game.

Save your program code as

`TASK4_4_<your name>_<centre number>_<index number>.py`

with any additional files/subfolders as needed in a folder name

`TASK4_4_<your name>_<centre number>_<index number>` [10]

Run the web application and save the output of the program as

`TASK4_4_<your name>_<centre number>_<index number>.html` [3]

-- END OF PAPER --

BLANK PAGE