



ANDERSON SERANGOON JUNIOR COLLEGE
JC2 Preliminary Examination 2025
Higher 2

COMPUTING

Paper 2 (Lab-based)

9569/02

20 August 2025 (Wednesday)

8 – 11 am

3 hours

Additional Materials: Electronic version of `encoding.txt` data file
 Electronic version of `sample_code.py` file
 Electronic version of `problems.txt` data file
 Electronic version of `users.txt` data file
 Electronic version of `credits.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.

Do NOT write or make any markings on the Insert Quick Reference Guide.

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 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 **8** printed pages and 1 insert.

1 Name your Jupyter Notebook as:

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

Token-based short run encoding is a compression technique that replaces consecutive repeated characters with a token followed by the character and the count of repetitions.

Unlike run-length encoding, token-based short run encoding only applies compression when there are 3 or more consecutive identical characters, leaving shorter sequences unchanged for better efficiency.

For example, the string

aaaaaaaaaaaaabbcccd

would be encoded as

%a14bb%c3d

where "%" is the token, followed by the character and its count.

The task is to write the encoding and decoding logic for this encoding method.

For each of the sub-tasks, add a comment statement at the beginning of the code using the 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 `encode_str(text, token)` to convert a long string into its shortened version using token-based short run encoding with a specified token character and return the encoded string.

If `text` contains the token character or digit characters, the function should instead return a meaningful error message.

Test your function with suitable test cases.

[5]

Task 1.2

Write a function `decode_str(encoded, token)` to convert an encoded string back to its original form and return this decoded string. You may assume that the encoded string is valid.

Test your function with the valid output strings you obtained in Task 1.1 to verify the original strings are recovered.

[5]

Task 1.3

Write program code to:

- read the file `encoding.txt`, which contains encoded strings
- decode each line and output the resulting ASCII art in the notebook, given that the token is "@".

This ASCII art should resemble something familiar.

[3]

Save your Jupyter Notebook for Task 1.

2 Name your Jupyter Notebook as:

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

A marathon organiser needs a system to manage registration data. Based on past years' participation data, the organiser is expecting 15,000 runners for this year's event. The system must handle runner registration data efficiently.

For each of the sub-tasks, add a comment statement at the beginning of the code using the 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 `generate_runners()` to:

- generate 15 000 **distinct** random 8-digit handphone numbers ranging from 8000 0000 to 9899 9999 (both inclusive)
- pair each phone number with a randomly selected race category from Men's Marathon, Women's Marathon, Men's Half-Marathon, Women's Half-Marathon to form a tuple
- store all tuples in a list and save the data in the list to a text file `runners.txt`
- return the list (containing 15 000 tuples)

For example, the first 2 lines of `runners.txt` might look like this:

```
87120982,Women's Marathon
91876543,Men's Half-Marathon
```

Test your function, storing the returned list in a variable named `runners`.
You do not need to output the list in the notebook.

[4]

Task 2.2

Write program code to:

- implement a **merge sort** algorithm to sort the data based on phone numbers in ascending order (you are **not** to use a built-in sorting function) and store the sorted list as `sorted_data`
- use the `timeit` module to measure and output the time taken for the merge sort operation (refer to `sample_code.py` for sample implementation)
- store the sorted data (phone number, category) in a text file `runners_sorted1.txt`.

Test the program and show the time taken for merge sort.

[7]

Task 2.3

Write program code to:

- create a `Node` class and an `OrderedLL` class to implement an **ordered** linked list
- define the following properties in the `Node` class: `phone`, `category`, `next`
- define the following methods in the `OrderedLL` class:
 - `insert(phone, category)`: insert a new runner while maintaining phone number order
 - `delete(phone)`: remove a runner by phone number, returns `True` if a runner is removed, and returns `False` if no runner is identified for removal
 - `update(phone, new_category)`: update the race category for a given phone number
 - `get_list()`: return a list of all tuples in traversal order
- use the `timeit` module to measure and output the overall time taken to populate the ordered linked list
- retrieve the sorted data using `get_list()` and write to a text file `runners_sorted2.txt`. [10]

Task 2.4

Write program code to implement a **recursive** binary search function to search for a runner by phone number.

The function should take the sorted list and a phone number as parameters, and return the runner's category if found, or "Runner not found." if the phone number does not exist in the list.

Test the function by searching for at least 3 different phone numbers in `sorted_data`: one at the end of the sorted list, one near the middle of the sorted list, and one that does not exist in the list. [4]

Task 2.5

On race pack collection day, the marathon organiser has support crew members responsible for packing race bibs and t-shirts into race packs. For efficient operations, when runners arrive at the entrance of the collection venue, they will scan their personalised QR codes. Their runner data will be added to a queue for the crew members to process. These crew members then prepare the race packs according to the queue sequence and deliver them to the collection counter, allowing runners to receive their race packs promptly without extended waiting times.

Write program code to implement a **circular queue** class `CQueue` that uses a 1D array of length 5 to store the data, as well as a front pointer and a rear pointer to track the array indices.

Include the following methods:

- `is_full()`: check if the queue is fully occupied
- `is_empty()`: check if the queue is empty
- `enqueue(phone, category)`: add runner data to the queue
- `dequeue()`: remove and return the runner data
- `display()`: print the current state of the queue, from the front to the rear

Create a queue `rq` and test the key methods.

[7]

Save your Jupyter Notebook for Task 2.

3 Name your Jupyter Notebook as:

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

A H2 Computing teacher has a collection of fill-in-the-blanks problems that needs to be managed using a MongoDB database.

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

In [1]:

```
#Task 3.1
Program code
```

Output:

Task 3.1

The file `problems.txt` contains problems separated by an empty line.

Write program code to extract the data from the file and store them in a MongoDB collection named `problems` within a database named `computing`.

Note that the answers to the blanks in a problem should be stored as a list of strings, and the attempt count stored as an integer value. [5]

Task 3.2

Write MongoDB query logic to extract problems that meet both of the following criteria:

- the topic is either 'Data Structures' or 'Algorithms'
- the number of attempts for the problems is between 5 and 10 (both inclusive).

Display the question texts of these matching problems. [3]

Task 3.3

Since the topic 'Socket Programming' will no longer be there in the revised H2 Computing syllabus, the teacher wants to update the collection by removing all problems tagged with that topic.

Write MongoDB logic to perform this update and execute it. [2]

Task 3.4

Write program code that:

- randomly selects a problem from the MongoDB collection
- displays the topic and the question text of the selected problem
- prompts the student to enter an answer for each blank in the question, reminding him to include the first letter in his response
- checks the student's answers against the expected answers (case-insensitive)
- outputs summative results (e.g. "You got 3 out of 5 answers correct!"), without revealing what the correct answers are
- increments the attempt count for that problem by 1 in the database.

If the student's answers to the problem are not all correct, prompt the student to enter all the answers again for the same problem. This continues until the student answers the entire problem correctly.

Test your code appropriately. [7]

Save your Jupyter Notebook for Task 3.

4 Name your Jupyter Notebook as:

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

ASRTech Solutions is a technology company that provides subscription-based services to its users. The company currently has user information in a text file `users.txt` containing user IDs, names, emails and salted hashed credit card numbers.

While actual credit card processing is handled securely through external payment services, ASRTech maintains hashed versions of credit card numbers within its own system. The hashing process uses a fixed salt string which prevents rainbow table attacks by hackers. Storing these salted hashed credit card numbers allows the company to perform secure lookups and comparisons. For example, the company can check if a card has been used before without risking exposure of sensitive data.

The company currently also maintains a separate text file `credits.txt` that tracks all credit transactions. Each transaction record contains the user ID, date and time of transaction, transaction type, and amount. These credits can be topped up or deducted.

For each of the sub-tasks 4.1 to 4.4, add a comment statement at the beginning of the code using the 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 Python program code (with the use of SQL commands) to:

- create a database `USER_DATA` with the following tables:
 - `User`: storing user information, using field names that match those in the text file and using a suitable field as the primary key,
 - `Credit`: storing credit transaction information, using field names that match those in the text file, with a composite key consisting of the user's ID and the datetime of transaction, ensuring referential integrity between the two tables.
 (You should ensure that the fields are of appropriate data types.)
- read the data from the text files and insert them into the database.

Run your program to transfer the data to the database and display the number of inserted users and transaction records. [6]

Task 4.2

The Luhn algorithm is a checksum formula used to validate credit card numbers and detect errors. How it works is as follows:

1. Starting from the rightmost digit (excluding the final check digit), double the value of every second digit.
2. If doubling results in a 2-digit number, add the digits together instead.
3. Sum all the digits.
4. If the total sum is divisible by 10, it passes the Luhn validation check, and the card number is considered valid.

Consider the credit card number **4417123456789113**.

Step 1: **8, 4, 2, 7, 2, 2, 6, 4, 10, 6, 14, 8, 18, 1, 2, 3**

Step 3: **$8 + 4 + 2 + 7 + 2 + 2 + 6 + 4 + (1 + 0) + 6 + (1 + 4) + 8 + (1 + 8) + 1 + 2 + 3 = 70$**

Step 4: 70 divided by 10 produces a remainder of 0, so card number is considered **valid**.

The function `validate_card(card_number)` takes `card_number` as a parameter, which is a string of digits, and implements the Luhn algorithm to return a boolean indicating whether the card number is valid.

Test your function with the following test cases:

- 4532840913491054 (16 digits)
- 5555734352999785 (16 digits)
- 378282904122908 (15 digits)
- 4111025640404402 (16 digits)
- 5105207030955616 (16 digits)

[6]

Task 4.3

Write a function `check_user(user_id)` that runs an SQL query to determine whether the given user ID is present in the `User` table of the database. The function should return `True` if the user ID is found and `False` otherwise.

Test with suitable test cases.

[3]

Task 4.4

The user management system should allow users to update their personal credit card information.

Note that the database should only store salted hashed credit card numbers.

Use the following fixed salt: `"@SR_SUP3R_S3CURE_S4LT!"`.

To create the salted hash, concatenate the salt with the card number (salt + card number), then apply SHA-3 (a strong cryptographic hash function) to the combined string.

Refer to the sample code provided in `sample_code.py` for SHA-3 implementation.

Write program code to declare a function `update_card(user_id, new_num)` that:

- checks whether the user ID exists in the database
- checks whether the new credit card number is valid using the Luhn algorithm
- generates a salted hash of the new credit card number using SHA-3
- updates the credit card hash in the database for the specified user ID.

The function should return an appropriate success or error message.

Display the output of the following:

```
update_card("swimmerz", "30569309025904")
update_card("koh-world", "1122334455667788")
```

[6]

Save your Jupyter Notebook for Task 4.

Task 4.5

[Turn over

ASRTech Solutions requires a web application that allows staff to search for users and view their transaction history.

Write a Python program and the necessary files to create a web application that:

- provides a search interface where the admin can enter a user's name
- performs name searches in the database
- shows a table of all credit transactions made by the user within the past 90 days (with reference to the server's current system date)

Below are some additional requirements:

- the table should have 3 columns: Date and Time, Transaction Type, Transaction Amount
- transactions within the timeframe should be displayed from the latest to the oldest
- the background colour of the row in the table should be #58D68D if the transaction amount is positive, and #FFC300 if negative.

Save your Python program with any additional files/subfolders in a folder named:

TASK_4_5_<your_name>_<centre_number>_<index_number>

Below is a mock-up of how the web application interface might look:

ASRTech Solutions

View Transaction Records

Search by name:

90-day transaction history for Emmanuel

Date and Time	Transaction Type	Transaction Amount
2025-08-12 09:31:27	top-up	925.0
2025-08-10 14:47:29	top-up	650.0
2025-06-27 16:45:31	top-up	950.0
2025-06-21 13:19:38	deduction	-188.8

Run the web application and test with a search of the user "Caleb Tay".

Save the web page output as:

TASK_4_5_<your_name>_<centre_number>_<index_number>.html

[11]