

2025 H2 Computing Prelim Exam Review

Name: _____

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 1.1

```
def encode_str(text, token):
    # validate input
    # check if token is detected
    if token in text:
        return f"Error: Text cannot contain the token {token}."

    # check for any existing digits
    for char in text:
        if char.isdigit():
            return "Error: Text cannot contain digits."

    # initialise variables
    i = 0
    result = []

    # loop through text
    while i < len(text):
        curr_char = text[i]
        count = 1

        # count consecutive characters
        while i + count < len(text) and text[i + count] == curr_char:
            count += 1

        # compress 3 or more consecutive characters
        if count >= 3:
            result.append(f"{token}{curr_char}{count}")
        else:
            result.append(curr_char * count)

        i += count

    return "".join(result)

# Test
print(encode_str("aaaaaaaaaaaabbcccd", "%")) # output: %a14bb%c3d
print(encode_str("HHEEEEEEEELLLPPPPPPPPPP", "$")) # output: HH$E7$L3$P9
print(encode_str("!!!@@@%%", "&")) # output: &!3&@4%%
print(encode_str("abcdef%", "%")) # contains token
print(encode_str("aaa123", "%")) # contains digits
```

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 1.2
def decode_str(encoded, token):
    i = 0
    result = []

    while i < len(encoded):
        if encoded[i] == token:
            # token found
            char = encoded[i + 1]
            count = "" # to extract count
            j = i + 2

            # extract count
            while j < len(encoded) and encoded[j].isdigit():
                count += encoded[j]
                j += 1

            result.append(char * int(count))
            i = j
        else:
            # no token
            result.append(encoded[i])
            i += 1

    return "".join(result)

# Test
print(decode_str("%a14bb%c3d", "%"))
print(decode_str("HH$E7$L3$P10", "$"))
print(decode_str("&!3&@4%", "&"))
```

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]

```
#Task 1.3
try:
    # file closes after with block
    # read lines from file
    with open("encoding.txt", "r") as file:
        for line in file:
            line = line.rstrip() # remove '\n'
            print(decode_str(line, "@"))
except FileNotFoundError:
    print("File not found.")
```


[illegible]

Write a function `generate_runners()` to:

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

Test your function, storing the returned list in a variable named `runners`. You do not need to output the list in the notebook. [4]

```
import random
```

3

```

while len(phone_nums) < 15_000:
    num = random.randint(8000_0000, 9899_9999)
    if num not in phone_nums:
        phone_nums.append(num)

# create tuples
for num in phone_nums:
    category = random.choice(categories)
    runners.append((num, category))

# write to file
# file closes after with block
with open("runners.txt", "w") as file:
    for num, category in runners:
        file.write(f"{num},{category}\n")

print("Generated", len(runners), "runners and saved file.")
return runners

# Test
runners = generate_runners()

```

Output:

Generated 15000 runners and saved file.

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 2.2

```

import timeit

def merge_sort(arr):
    # base case
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid]) # sorted left
    right = merge_sort(arr[mid:]) # sorted right

    return merge(left, right)

```



```

def merge(left, right):
    # helper function
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i][0] < right[j][0]: # compare only phone numbers
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result = result + left[i:] + right[j:] # store any remaining items at the rear
    return result

# Test
start_time = timeit.default_timer()
sorted_data = merge_sort(runners)
end_time = timeit.default_timer()

mergesort_time = end_time - start_time
print("Time taken:", mergesort_time, "seconds")

# Store data
with open("runners_sorted1.txt", "w") as file:
    for num, category in sorted_data:
        file.write(f"{num},{category}\n")
print("Data saved to file.")

```

Output:

Time taken: 0.029208699997980148 seconds
Data saved to file.

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 2.3

class Node:

```
def __init__(self, phone, category):
    self.phone = phone
    self.category = category
    self.next = None
```

class OrderedLL:

```
def __init__(self):
    self.head = None
```

```
def insert(self, phone, category):
    new_node = Node(phone, category)
```

```
    # Insert at the front if LL is empty / data smaller than head value
    if self.head is None or phone < self.head.phone:
```

```
        new_node.next = self.head # link new node to existing head
        self.head = new_node # update head to this new node
```

```
    else:
```

```
        current = self.head
```

```
        # traverse to find the correct position to insert
```

```
        while current.next is not None and current.next.phone < phone:
```

```
            current = current.next # move to next node
```

```
        # insert node
```

```
        new_node.next = current.next
```

```
        current.next = new_node
```

```
def delete(self, phone):
```

```
    # check if empty
```

```
    if self.head is None: # nothing to delete
```

```
        return False
```

```
    # check if first node is to be deleted
```

```
    if self.head.phone == phone:
```

```
        self.head = self.head.next # skip first node
```

```
        return True
```

```
    # traverse linked list
```

```
    current = self.head
```

```
    while current.next is not None and current.next.phone != phone:
```

```
        current = current.next
```

```
    if current.next is not None: # not at the end of LL
```

```
        current.next = current.next.next # skip the node
```

```
        return True
```

```
    return False # did not find entry to delete
```

```
def update(self, phone, new_category):
```

```
    current = self.head
```

```
    while current:
```

```
        if current.phone == phone: # found match
```



```

        current.category = new_category # update category
        return True # exit while loop
    current = current.next # move to next node
return False

def get_list(self):
    result = []
    current = self.head
    while current:
        result.append((current.phone, current.category)) # store tuple
        current = current.next
    return result

# Main
runners_LL = OrderedLL() # instantiation

# Measure time for insertion
start_time = timeit.default_timer() # start timer
for phone, category in runners:
    runners_LL.insert(phone, category)
end_time = timeit.default_timer() # end timer

# Calculate time
OLL_time = end_time - start_time
print("Time taken:", OLL_time, "seconds")

# Extract list and store data
sorted_OLL = runners_LL.get_list()

with open("runners_sorted2.txt", "w") as file:
    for num, category in sorted_OLL:
        file.write(f"{num},{category}\n")
print("Data saved to file.")

```

Output:

Time taken: 1.8873272000055294 seconds
Data saved to file.

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 2.4

```
def bin_search(arr, target, left, right):
    if left > right: # invalid search range
        return "Runner not found."

    mid = (left + right) // 2
    mid_val = arr[mid][0] # extract phone number at middle index
    if target == mid_val:
        return arr[mid][1] # found match, return category
    elif target < mid_val:
        return bin_search(arr, target, left, mid - 1) # search left half
    else:
        return bin_search(arr, target, mid + 1, right) # search right half

# Test
test1 = sorted_data[-1][0] # last number
test2 = sorted_data[7495][0] # near middle
test3 = 99999999 # not in list

for test in [test1, test2, test3]:
    result = bin_search(sorted_data, test, 0, len(sorted_data) - 1)
    print(f"Search for {test}: {result}")
```

Output:

```
Search for 98999868: Men's Half-Marathon
Search for 89423990: Women's Half-Marathon
Search for 99999999: Runner not found.
```

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]

#Task 2.5

```
class CQueue:
```

```
    def __init__(self, capacity):
```

```
        self.data = [None] * capacity
```

```
        self.front = 0
```

```
        self.rear = 0 # points to the next available slot
```

```
        self.size = 0 # current number of items in queue
```

```
        self.capacity = capacity # max capacity
```

```
    def is_empty(self):
```

```
        return self.size == 0
```

```
    def is_full(self):
```

```
        return self.size == self.capacity
```

```
    def enqueue(self, phone, category):
```

```
        if self.is_full():
```

```
            print("Queue is full. Cannot enqueue.")
```

```
            return
```

```
        self.data[self.rear] = (phone, category) # store data
```

```
        self.rear = (self.rear + 1) % self.capacity # update rear
```

```
        self.size += 1
```

```
        print("Enqueued:", phone, category)
```

```
    def dequeue(self):
```

```
        if self.is_empty():
```

```
            print("Queue is empty. Cannot dequeue.")
```

```
            return
```

```
        data = self.data[self.front] # extra data from front
```

```
        self.data[self.front] = None # optional: clear data in slot
```

```
        self.front = (self.front + 1) % self.capacity # adjust front pointer
```

```
        self.size -= 1
```

```
        print("Dequeued:", data)
```

```
        return data
```

```
    def display(self):
```

```
        if self.is_empty():
```

```
            print("Queue is empty.")
```

```
            return
```

```
        print("Contents (front to rear):")
```

```
        curr = self.front
```

```
        for i in range(self.size):
```

```
            print(self.data[curr])
```

```
            curr = (curr + 1) % self.capacity
```

```
# Test
```

```
rq = CQueue(5)
```

```
print("Queue is empty:", rq.is_empty())
```

```
rq.enqueue(80000000, "Men's Marathon")
```

```
rq.enqueue(80000001, "Men's Marathon")
```

```
rq.enqueue(80000002, "Men's Marathon")
```

```

rq.enqueue(80000003, "Men's Marathon")
rq.enqueue(80000004, "Women's Marathon")
rq.enqueue(80000005, "Women's Marathon")
print("Queue is full:", rq.is_full())
rq.dequeue()
rq.dequeue()
rq.display()

```

Output:

```

Queue is empty: True
Enqueued: 80000000 Men's Marathon
Enqueued: 80000001 Men's Marathon
Enqueued: 80000002 Men's Marathon
Enqueued: 80000003 Men's Marathon
Enqueued: 80000004 Women's Marathon
Queue is full. Cannot enqueue.
Queue is full: True
Dequeued: (80000000, "Men's Marathon")
Dequeued: (80000001, "Men's Marathon")
Contents (front to rear):
(80000002, "Men's Marathon")
(80000003, "Men's Marathon")
(80000004, "Women's Marathon")

```

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 3.1
import pymongo

# connect to MongoDB
client = pymongo.MongoClient("127.0.0.1", 27017)
db = client.get_database("computing")
coll = db.get_collection("problems")

problems = []

# extract data from file
with open("problems.txt", "r") as file:
    content = file.read()

# split into problem blocks, by empty lines
problem_list = content.strip().split("\n\n")

for block in problem_list:

```



```

# extract relevant data from each block
lines = block.strip().split("\n") # split the block into 4 lines
topic = lines[0].replace("Topic: ", "")
question = lines[1].replace("Question: ", "")
ans_str = lines[2].replace("Answers: ", "")
answers = ans_str.split(",") # split to form a list of strings
attempts = int(lines[3].replace("Attempts: ", "")) # no. of attempts as integer

# form a dictionary with the data
problem = {
    "topic": topic,
    "question": question,
    "answers": answers,
    "attempts": attempts
}
problems.append(problem)

coll.drop() # drop collection if exists
result = coll.insert_many(problems) # insert all problems
print(f"Inserted {len(result.inserted_ids)} problems.")

```

Output:

Inserted 29 problems.

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 3.2

```

query = {
    "topic": {"$in": ["Data Structures", "Algorithms"]},
    "attempts": {"$gte": 5, "$lte": 10}
}

# alternatively
query = {"$and": [{"topic": {"$in": ["Data Structures", "Algorithms"]}}, {"attempts": {"$gte": 5, "$lte": 10}}]}

result = coll.find(query)

for item in result:
    print(item["question"])

```

Output:

A memory leak occurs when dynamically allocated memory is not properly freed. This causes programs to consume increasing amounts of memory.
 In a binary search tree, all values in the left subtree are smaller than the root, and all values in the right subtree are larger.
 Bubble sort compares n elements and swaps them if they are in wrong order.

Merge sort uses d___ and c___ strategy, recursively splitting arrays then m___ sorted subarrays back together.
A stack follows the L___ I___ F___ O___ principle where the last element added is the first one r___.

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 3.3

```
result = coll.delete_many({"topic": "Socket Programming"})
```

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]

#Task 3.4

```
import random
```

```
# get all problems, storing as list
all_problems = list(coll.find())
```

```
# select random problem
selected = random.choice(all_problems)
expected_ans = selected["answers"]
```

```
# display problem
print("Topic:", selected["topic"])
print("Question:", selected["question"])
print("\nWhen typing your answers, include the first letter of the word.")
```

```
passed = False
while not passed:
    stu_ans = [] # to store student's responses
    # get answers from student
    for i in range(len(expected_ans)):
        ans = input(f"Answer to blank {i + 1}: ")
        stu_ans.append(ans)
```



```

# check against expected answers
correct_count = 0
for i in range(len(expected_ans)):
    if expected_ans[i].lower() == stu_ans[i].lower():
        correct_count += 1

# calc score
print(f"You got {correct_count} out of {len(expected_ans)} answers correct.")

# Use $inc to increment attempts directly in the database
coll.update_one(
    {"_id": selected["_id"]},
    {"$inc": {"attempts": 1}}
)
if correct_count == len(expected_ans):
    print("Excellent!")
    passed = True # update flag to exit loop
else:
    print("Let's try again.")
    print()

client.close() # close MongoDB connection (IMPORTANT!)

```

Output:

Topic: Computer Networking

Question: The modern TCP/IP model is an updated version of the original model. It has 5 layers: A___, T___, N___, Data link, and P___ layers. Data is e___ at each layer.

When typing your answers, include the first letter of the word.

Answer to blank 1: Adam
 Answer to blank 2: Terry
 Answer to blank 3: Natalie
 Answer to blank 4: Perry
 Answer to blank 5: extracted
 You got 0 out of 5 answers correct.
 Let's try again.

Answer to blank 1: Application
 Answer to blank 2: Transport
 Answer to blank 3: Network
 Answer to blank 4: Physical
 Answer to blank 5: extracted
 You got 4 out of 5 answers correct.
 Let's try again.

Answer to blank 1: Application
 Answer to blank 2: Transport
 Answer to blank 3: Network
 Answer to blank 4: Physical
 Answer to blank 5: encapsulated
 You got 5 out of 5 answers correct.
 Excellent!

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 4.1
import csv
import sqlite3

# connect to db
conn = sqlite3.connect("USER_DATA.db")
cursor = conn.cursor()

# create User table
cursor.execute("""
CREATE TABLE IF NOT EXISTS User (
    user_id TEXT PRIMARY KEY,
    name TEXT,
    email TEXT,
    hash TEXT
)
""")

# create Credit table
cursor.execute("""
CREATE TABLE IF NOT EXISTS Credit (
    user_id TEXT,
    dt_transaction TEXT,
    type TEXT,
    amt REAL,
    PRIMARY KEY (user_id, dt_transaction),
    FOREIGN KEY (user_id) REFERENCES Users(user_id)
)
""")

# read data from files
with open("users.txt", "r") as file:
    reader = csv.reader(file)
    next(reader) # skip header
    user_count = 0
    for row in reader:
        cursor.execute("""
```



```

INSERT INTO User (user_id, name, email, hash)
VALUES (?, ?, ?, ?)
"""", (row[0], row[1], row[2], row[3]))
user_count += 1
print(f"Inserted {user_count} users.")

```

```

with open("credits.txt", "r") as file:
    reader = csv.reader(file)
    next(reader) # skip header
    transact_count = 0
    for row in reader:
        cursor.execute("""
INSERT INTO Credit (user_id, dt_transaction, type, amt)
VALUES (?, ?, ?, ?)
"""", (row[0], row[1], row[2], float(row[3])))
        transact_count += 1
    print(f"Inserted {transact_count} transactions.")

conn.commit() # commit changes

```

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 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 4.2

```

def validate_card(card_number):
    digits = [int(digit) for digit in card_number]

```



```

for i in range(len(digits) - 2, -1, -2):
    digits[i] *= 2
    if digits[i] > 9: # 2-digit number after doubling
        digits[i] = digits[i] // 10 + digits[i] % 10

total = sum(digits)
return total % 10 == 0 # check divisibility by 10

# Test
cases = [
    4532840913491054,
    5555734352999785,
    378282904122908,
    4111025640404402,
    5105207030955616
]

for card in cases:
    if validate_card(str(card)):
        print(f"{card} is valid.")
    else:
        print(f"{card} is invalid.")

```

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 4.3

```

def check_user(user_id):
    cursor.execute("""
        SELECT COUNT(*) FROM User
        WHERE user_id = ?
        """, (user_id,))
    count = cursor.fetchone()[0]
    return count > 0

# Test
users = ["among-us", "among-them", "i-like-fortnite"]

for user in users:
    print(f"{user} exists: {check_user(user)}")

```

Task 4.4

The user management system must allow users to update their personal credit card information in accordance with the Access and Correction Obligation under the PDPA.

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]
```

#Task 4.4

```
import hashlib
```

```
def update_card(user_id, new_num):
    # validation
    if not check_user(user_id):
        return f"User {user_id} does not exist."

    if not validate_card(new_num):
        return f"Card number {new_num} is invalid."

    # create salted hash
    salt = "@SR_SUP3R_S3CURE_S4LT!"
    salted_card = salt + new_num
    card_hash = hashlib.sha3_256(salted_card.encode()).hexdigest()

    # update database
    cursor.execute("""
        UPDATE User
        SET hash = ?
        WHERE user_id = ?
    """, (card_hash, user_id))
    conn.commit()
    conn.close() # close connection

    return f"Card number hash updated for {user_id}."

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

Task 4.5

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>

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]

templates/index.html

```
<!DOCTYPE html>
<html>

<head>
    <title>Transactions</title>
    <style>
        body {
            font-family: Arial;
        }
        .positive {
            background-color: #58D68D;
        }
        .negative {
            background-color: #FFC300;
        }
    </style>
</head>

<body>
    <h1>ASRTech Solutions</h1>
    <h2>View Transaction Records</h2>
```



```

<form method="POST">
    <label>Search by name:</label>
    <input type="text" name="user_name" value="{{ search_name }}">
    <button type="submit">Search</button>
</form>

{% if search_name %}
<h3>90-day transaction history for {{ search_name }}</h3>

    {% if transactions %}
    <table border="3">
        <tr>
            <th>Date and Time</th>
            <th>Transaction Type</th>
            <th>Transaction Amount</th>
        </tr>
        {% for record in transactions %}
        <tr class="{% if record[3] > 0 %}positive
            {% elif record[3] < 0 %}negative{% endif %}">
            <td>{{ record[1] }}</td>
            <td>{{ record[2] }}</td>
            <td>{{ record[3] }}</td>
        </tr>
        {% endfor %}
    </table>
    {% else %}
    <p>No transactions found in the past 90 days.</p>
    {% endif %}

{% endif %}
</body>
</html>

```

app.py

```

from flask import Flask, render_template, request
import sqlite3
from datetime import datetime, timedelta

app = Flask(__name__)

@app.route("/", methods=["GET", "POST"])
def home():
    transactions = []
    search_name = ""

    if request.method == "POST":
        # get data from inputbox
        search_name = request.form.get("user_name", "").strip()

        if len(search_name) > 0:
            # connect to DB to extract data
            conn = sqlite3.connect("USER_DATA.db")
            cursor = conn.cursor()

```

```
# calculate 90 days before
# expected to choose the first option below to get current date
curr_date = datetime.now() # using system date
curr_date = datetime(2025, 8, 20) # OR: using hardcoded date 20/8/2025
```

```
start_date = curr_date - timedelta(days=90)
start_date_str = start_date.strftime("%Y-%m-%d")
```

```
query = """
SELECT User.name, Credit.dt_transaction, Credit.type, Credit.amt
FROM User
INNER JOIN Credit ON User.user_id = Credit.user_id
WHERE User.name = ?
AND Credit.dt_transaction >= ?
ORDER BY Credit.dt_transaction DESC;
"""
```

```
cursor.execute(query, (search_name, start_date_str))
transactions = cursor.fetchall()
conn.close()
```

```
return render_template("index.html", transactions=transactions,
                        search_name=search_name)
```

```
if __name__ == "__main__":
    app.run(port=5678)
```