



**ANGLO-CHINESE JUNIOR COLLEGE
JC2 PRELIMINARY EXAMINATION**

Higher 2

COMPUTING

9569/02

Paper 2 (Lab-based)

6 August 2024

3 hours

Additional Materials: Electronic version of `words.txt` data file
 Electronic version of `names.txt` data file
 Electronic version of `ITEM.txt` data file
 Electronic version of `CUSTOMER.txt` data file
 Electronic version of `PURCHASE.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 **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 and **1** blank page.



Anglo-Chinese Junior College

[Turn Over

Instruction to candidates:

Your program code and output for each of Task 1 to 4.3 should be saved in a single `.ipynb` file using Jupyter Notebook. For example, your program code and output for Task 1 should be saved 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`

In this task, you are required to implement a sorting algorithm known as Stalin Sort.

The algorithm runs as follows:

1. Starting from the beginning of the list, compare each item with the one following it in the list.
2. If the i^{th} item in the list is **not** in the correct order relative to the $(i + 1)^{\text{th}}$ item, pop the $(i + 1)^{\text{th}}$ item from the list and insert it at the beginning of the list.
3. Move on to compare the $(i + 1)^{\text{th}}$ item with the $(i + 2)^{\text{th}}$ item.
4. Continue this process until you reach the end of the list.
5. Steps 1 to 4 are considered a pass. Repeat steps 1 to 4 until the list is sorted. The list is determined to be sorted when no items are inserted at the front of the list during a complete pass.

The following example shows how the algorithm would sort a list of integers in ascending order.

Pass 0: [2, 0, 1, 4, 3]
 Pass 1: [3, 1, 0, 2, 4]
 Pass 2: [2, 0, 1, 3, 4]
 Pass 3: [1, 0, 2, 3, 4]
 Pass 4: [0, 1, 2, 3, 4]

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

The value of a string is defined as the number of characters in the string multiplied by the number of vowels in the string.

Write a function `string_value(s1)` that returns the value of a string. Note that the string might contain a mixture of upper- and lowercase characters. [3]

Task 1.2

Write a function `StalinSort(lst)` that would sort a list of strings in ascending order of value, as determined in Task 1.1. If two strings have the same value, sort them in lexicographic order.

Test your function with the following test cases:

```
print(StalinSort(["Peter", "Josh", "See", "Hi" ]))
print(StalinSort(["PetEr", "JOsh", "Hi", "See" ]))
print(StalinSort(["Peter", "Hello", "BYE", "Hi", "ah", "bbbb" ]))
```

[8]

Task 1.3

Write program code to:

- read the text file `words.txt` and store the content in a list;
- sort the list using `StalinSort(lst)`;
- display the sorted list.

[3]

Task 1.4

Write a function `QuickSort(lst)` that would sort a list of strings in ascending order of value, as determined in Task 1.1, using a quicksort algorithm. For strings of the same value, sort them in alphabetical order.

[7]

Task 1.5

Write a function `BinarySearch(lst, value)` that will perform a binary search through the sorted list of words from Task 1.3 for any word whose value, as determined in Task 1.1, is `value`.

- If such a word can be found, return its index in the list.
- If such a word cannot be found, return `False`.

[4]

Save your Jupyter Notebook for Task 1.

2 Name your Jupyter Notebook as:

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

The task is to create a game where players will try to locate "mines" which are hidden at random locations in field which is a rectangular grid. The player's task is to determine where the mines are.

On each turn, the player selects one square. If the player selects a square with a mine, it blows up and the game is over. Otherwise, the player is told how many mines there are in the eight squares surrounding the selected square, and the player's next turn begins. If the player manages to select all the squares without mines before blowing up any mines, the player wins.

For example, the diagram below shows a 3 x 4 grid. There are two mines hidden at (0, 0) (the 0th row and the 0th column, i.e. the top left corner) and (1, 2) (the 1st row and the 2nd column), as indicated by the x's.

```
X . . .
. . X .
. . . .
```

Since the player does not know where the mines are, the field is displayed to the player as follows.

```
. . . .
. . . .
. . . .
```

If the player selects (1, 1) on the first turn, the player is told that there are 2 mines in the eight squares surrounding the (1, 1). The field is then displayed to the player as follows.

```
. . . .
. 2 . .
. . . .
```

If the player selects (0, 2) on the second turn, the player is told that there is 1 mine in the five squares surrounding (0, 2) .

```
. . 1 .
. 2 . .
. . . .
```

If the player selects (0, 0) on the third turn, the mine at (0, 0) blows up and the game ends.

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

The player sets up the game by giving the dimensions of the field and the number of mines.

Write code to:

- ask the player for the height and width of the field;
- ask the player for the number of mines;
- validate that the number of mines is larger than zero and smaller than the size of the field;
- create the field by putting the mines at random locations in the field.

There should only be at most one mine at each location in the field.

[7]

Task 2.2

The player can now play the game as described above.

Write code to:

- display the field to the player as described above;
- ask the player for the row and column number of the location to try;
- end the game if the player's location has a mine, or, otherwise, reveal the number of mines in the squares surrounding it;
- repeat the above steps until the player selects a mine, or has revealed all squares without mines.

Your program should print appropriate messages to the player when the game ends because the player selects a mine, or because the player wins. [11]

Save your Jupyter Notebook for Task 2.

3 Name your Jupyter Notebook as:

TASK3_<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 3.1
        Program code
        Output:
```

Task 3.1

A linked list is implemented using Object-Oriented Programming (OOP) to store data about students' names and scores.

Write the `Node` class, including the following attributes:

- `data` stores a tuple of the form `(name, score)`, e.g. `("Sam", 95)`.
- `pointer` points to the following `Node` in the linked list if one exists, or `None` otherwise.

Write the `LinkedList` class, including the following attribute:

- `headpointer` points to the head node of the linked list;

and the following methods:

- `insert_head(data)` inserts a node with the given data at the head of the linked list;
- `display()` displays the data in all the nodes of the linked list in order of the list;
- `search(name)` returns `True` if `name` is present in the data of one of the nodes in the linked list, and `False` otherwise;
- `remove(name)` attempts to delete the node with `name` from the linked list, and returns `True` if the node was successfully deleted, and `False` if `name` does not exist in any node in the linked list;
- `reverse_list()` that reverses the order of the nodes in the linked list. [10]

Task 3.2

Write program code to:

- create an instance of `LinkedList`;
- store the data found in the text file `names.txt` in the `LinkedList`, where the data in each line in the file should be stored in a `Node`;
- call the `reverse_list()` and `display()` methods on the `LinkedList` object. [3]

Task 3.3

Linked lists can be used to resolve collisions in a hash table.

For this question, the data stored would be tuples of the form `(name, score)`, e.g. `("Sam", 95)`. Each tuple is stored inside a `Node` which is inside a `LinkedList`. The hash table consists of an array of `LinkedLists`.

Write the `HashTable` class, including the following attributes:

- `max_size` is the maximum number of items stored in the hash table, initialized to 97;
- `table` is a 1-dimensional array of 97 instances of a `LinkedList`;

and the following methods:

- `hash_name(name)` is a hash function that adds up the ASCII values of the characters in `name` and returns the sum modulo 97 (i.e., the remainder of the sum after dividing by 97);
- `add(tup)` inserts a key-value tuple to the hash table;
- `retrieve(name)` returns the `score` associated with `name` if it is in the hash table, and `False` otherwise;
- `show()` displays all the data stored in the hash table.

In the event of a collision, the `Node` containing the data would be inserted at the head of the `LinkedList` found at the appropriate bucket of the hash table. [8]

Task 3.4

Write program code to:

- create an instance of `HashTable`;
- store the data found in the text file `names.txt` in the `HashTable` object;
- call the `show()` method on the `HashTable` object. [3]

Save your Jupyter Notebook for Task 3.

4 Name your Jupyter Notebook as:

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

A supermarket keeps records about its customers, items and purchases. The supermarket wants to create a suitable database to store the data and allow them to run searches for specific data. The database will have three tables: a table to store data about the items, a table about the customers, and a table about the purchases. The fields in each table are:

Item:

- ItemID – unique item number, for example, 1023
- Name – the name of the item
- Price – the price of the item in cents, for example, 350 for \$3.50

Customer:

- CustomerNumber – customer's unique number, for example, 456
- Name – customer's name

Purchase:

- PurchaseID – the purchase's unique ID, for example, 12
- CustomerNumber – the customer's unique number
- ItemID – the unique item number
- Quantity – the number of that item purchased
- DateTime – the date and time of the purchase in YYYYMMDD format.

For each of the sub-tasks 4.1 to 4.3, 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 uses SQL code to create the database `SUPERMARKET` with the three tables given. Define the primary and foreign keys for each table. [6]

Task 4.2

The text files `ITEM.txt`, `CUSTOMER.txt` and `PURCHASE.txt` contains the comma-separated values for each of the tables in the database.

Write a Python program to read the data from each file and then store each item of data in the correct place in the database. [5]

Task 4.3

Write a Python program to input a customer's number and date and return the names and quantities of all the items that they have purchased on or after that date. [6]

Test your program by running the application with the member number 200 and date 20240102. [2]

Save your Jupyter Notebook.

Task 4.4

Write a Python program and the necessary files to create a web application that displays the following data for all purchases made in the year 2024.

- Customer name
- Item name
- Quantity purchased

The program should return an HTML document that enables the web browser to display a table with the required data.

Save your program code as

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

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

TASK4_4_<your name>_<centre number>_ <index number> [6]

Run the web application.

Save the webpage output as:

TASK4_4_<your name>_<centre number>_ <index number>.html [2]