COMP1010/COMP1002 Assignment 2

Submission deadline: 11:59 pm, 20 Nov 2025

Important instructions

- Built-in functions to achieve core functionalities are not allowed.
- You are not allowed to use any external libraries such as math or numpy for any part of this assignment.
- Do not use advanced algorithms in this assignment. Use a simple approach (from the topics covered in the labs or lectures) to write your algorithms. Code optimization is not required in this assignment.

Q1 (Fun with numbers): [30 Points]

You are given the following program specifications:

Input: A four-digit positive integer with at least two different digits (leading zeros are allowed). For example, 11 (it is considered 0011), 104 (it is considered 0104), 0011 (0 and 1 are two different digits), and 1234 (all four are different) are allowed, but 1111 is not allowed.

Program Body:

- If the input is not four digits, add the leading 0s
- Validate the four digit number by checking for at least two different digits in the number.
    - Prompt the user for another number if it is invalid.
- Set N to be the input
- **Main Loop:** Repeat the following until N no longer changes
    - Create two new four-digit numbers by arranging the digits of N in (a) ascending, and (b) descending order.
    - Set N to be the difference between the larger number and the smaller number.
- **Print out:** The end result **and** the number of iterations of the Main Loop.

Example:

- We input the number 0102, which is valid because it has at least two different digits.
- We create two new four-digit numbers by arranging the digits in ascending (0012) and descending (2100) order.
- We subtract the smaller number from the bigger number to get 2100-0012 = 2088
- Since 2088 is different from 0102, we repeat the process with 2088.
- …
- If the number at the end of the process is different from 2088, we repeat the process again.

Q1.1: [5 points] In Python, write a function `validate_input(n)`, which returns True if n is a valid input, and False if n is not a valid input.

Q1.2: [9 points] In Python, write a function `sort_digits(n,b)`, which is specified as follows.

Input: An integer n from 0 to 9999 (inclusive) and a Boolean variable b.

Output: An integer n_sorted, which consists of the digits of n sorted in ascending order if b is False (or 0), or in descending order if b is True (or 1).

The sorting must align with the sorting specified by the main program specification (e.g. sort_digits(12, 1) should output 2100). **You are not allowed to use sort(), sorted(), min(), max(), reversed() or any other in-built sorting related functions.**

You do **not** need to include input validation for this function.

Q1.3: [7 points] Write the main body of the program in a `main()` function. The body must involve a call to your validate_input and sort_digits functions. In a docstring, describe the output of the program (based on all the test cases that you have tried), and provide the first and third digits of the final number.

Test cases: 104, 0011, 1234, 1738, any four-digit number with at least two different digits.

Note: It is fine if numbers with leading zeros are represented without those leading zeros during the runtime of your program. E.g. it is fine if 0012 is stored as the integer 12 during the program execution.

Q1.4: [9 points] In a **separate** Python program, implement code which finds (and prints out) the maximum number of Main Loop iterations taken by any valid integer, by running the given program specification for **all** valid integers from 1 to 9998.

You may (and should) re-use and modify your existing code from Q1.1, Q1.2, and Q1.3.

Q2 (Fun with chess): [45 Points]

In chess, a queen is considered the most powerful piece. From its current position, a queen can attack horizontally, vertically, and diagonally for any number of places as shown below.

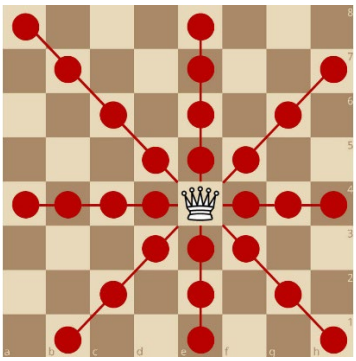


Figure 1: Queen's possible attack positions

In this question, we will deal with the problem of placing 4 queens on a size 4x4 chessboard such that no two queens threaten each other. This means no two queens can share the same row, column, or diagonal. You can play the game at: https://7owpn9cwpt2t.space.minimax.io/

Q2.1: [3 Points] Manually calculate the placement of four queens and show your final solution in the following format:

- <queen_number>-<row><col>
- E.g. Q1-00 (Queen 1 placed at column 0, row 0 from top left)
- Show these positions for all four queens such that no two queens threaten each other
- Assumption: Queen 1 is positioned in row 0, Queen 2 in row 1 and so on.

Q2.2: [15 Points] Use a top-to-bottom approach to provide a solution for this problem. Follow the template provided in Appendix A and complete the pseudo code for each function and procedure. The following is an outline of the overall idea (top level).

1. Consider a 4x4 chess board of all zeros
2. Start with the top left location 0,0 (first row, first column) and place Q1 at 0,0.
3. Mark all conflicting locations with a 1. This includes all horizontal, vertical, and diagonal directions relative to the queen.
4. Find the next available location that is not threatened by the queen(s).
5. Continue placing queens until we reach a stopping point where there are no more places left to place a new queen. It is either a valid solution or a dead end.
6. In case of a dead end, restart the procedure but this time choose a new position for Q1.

Note: Do not modify the template provided in Appendix A and only provide the body of each of the functions in the template.

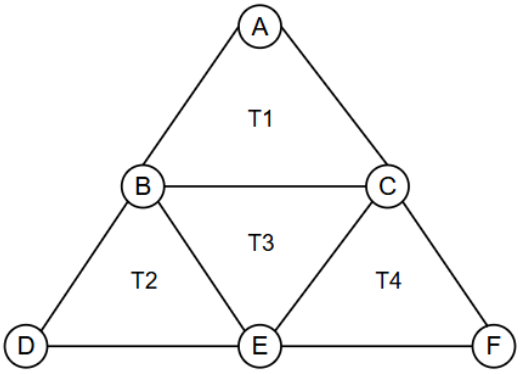
Q2.3: [25 Points] Provide Python code that implements the pseudo-code using the same template. Your Python code's logic must match your pseudo code. As good programming practice:

- Variable names should be meaningful and not generic
- Add 2 – 5 clarifying comments to complex sections of the code.
- In addition to those comments, provide docstring in functions about:
    - What the input and output represent
    - What the function does

Keep the code simple as defined in Q2.2. No advanced algorithms or techniques such as backtracking or recursion are needed.

Q2.4: [2 Points] Is it possible to scale this approach (from Q2.2) for more than 4 queens? Briefly explain. Mention two pros and cons of this current approach

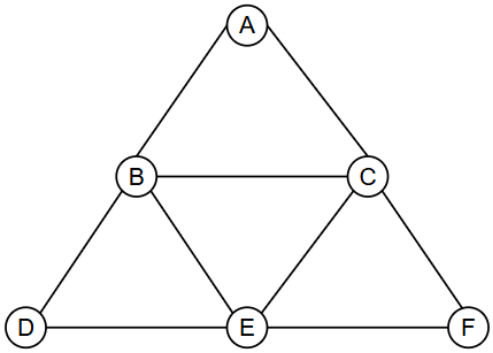
Q3 (Fun with graphs and matrices): [25 points] In an unweighted, undirected graph, a three-edge triangle is a cycle of length 3. A cycle is a route along edges which starts and ends at the same vertex. For instance, the graph below contains 4 three-edge triangles (T1 – T4).



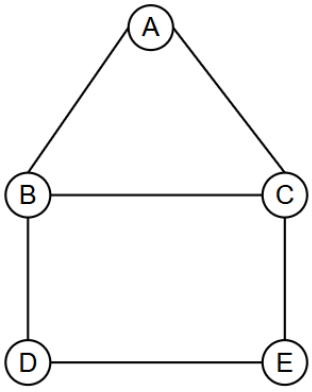
In this question, you will calculate the number of three-edge triangles in an unweighted, undirected graph by using its adjacency matrix.

Q3.1: [6 points] Write the adjacency matrix for the following graphs. In the adjacency matrices, order the rows and columns so that they are consistent with the alphabetical order of the vertices.

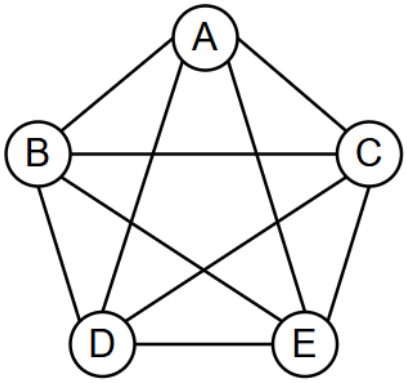
Q3.1(a)



Q3.1(b)



Q3.1(c) Note: this graph has 5 vertices.



The remainder of Question 3 is to be completed in Python code. Each matrix is to be represented as a list of lists.

Q3.2: [9 points] Write a function `matrix_multiplication(A, B)` which takes as input matrices A and B, and outputs/returns a matrix equal to AB.

Q3.3: [5 points] Write a function `trace(A)` which takes as input matrix A, and outputs/returns the sum of the main diagonal elements of A.

Q3.4: [5 points] In the main body of your program (the `main()` function), initialize matrices Aa, Ab, and Ac, which correspond to the graphs in Q3.1(a), Q3.1(b), and Q3.1(c) respectively. For each matrix A, print out the value of trace(A^3)/6. Compare the number of three-edge triangles in the graph with the respective value of trace(A^3)/6.

Submission Instructions

Follow the steps below:

1. Create a folder and name it as <student no>_<your name>,
   e.g., **12345678d_CHANTaiMan**
2. For Q2.1, Q2.2, Q2.4, and Q3.1, type your answers in a word document and save it as a **.pdf** file. Name the single **.pdf** file as A2_<student no>_<your name>**.pdf**,
   e.g., **A2_12345678d_CHANTaiMan.pdf**
3. For Q1Part1, Q1Part2, Q2, and Q3, submit the source files (**.py**). Name the **.py** files as A2_Q<question no>_<student no>_<your name>**.py**,
   e.g., **A2_Q1Part2_12345678d_CHANTaiMan.py**
4. Put all the **.pdf** and **.py** files into the folder created in Step 1.
5. Compress the folder (**.zip**, **.7z**).
6. Submit the file to Blackboard.

A maximum of **3 attempts** for submission are allowed. **Only the last attempt will be graded**. A late penalty of 20% per day will be imposed.

A corrupted file will be given ZERO marks. It is your obligation to carefully check the files in your submission.

If you are using Windows, the file extension may be hidden by the operating system. Follow the steps of below links to make sure the file extension is not hidden:

https://www.howtohaven.com/system/show-file-extensions-in-windows-explorer.shtml

If your program cannot be run successfully (i.e., having any syntax error(s)) when it is triggered, ZERO marks will be awarded for that Python program, regardless of how much you have coded.

This is an individual assignment. All work must be done on your own. Plagiarism is a serious offence. You are not allowed to consult any external channels, e.g., discussion forums, or copy code from any web resources, to assist your completion of your assignments. The Moss (https://theory.stanford.edu/~aiken/moss/) system will be adopted for plagiarism checking for program code. Submissions with high similarity, in terms of code patterns and structures, in addition to direct-copy-and-paste, will be extracted and reviewed. Any plagiarism cases (both copier and copiee) will be given ZERO mark**s** plus a deduction of the maximum mark of this assignment. Serious cases will be submitted to the disciplinary committee of the department for further actions.

The use of GenAI tools such as chatGPT for this assignment is strictly forbidden

Appendix A - Pseudo code template for Q2

`declare Board_2D` # sets a 2D matrix of size 4x4 as a list of lists

`procedure board_reset( )` # sets all values of the 2D matrix to zero

`function find_next_location(starting_position)` # return the first available position i.e. value 0, starting from starting_position. In case no position is available, return -1, representing a dead end

`function place_queen(queen_number, starting_position=None)` # sets the queen on the board in an available position starting from starting_position. Starting position is useful when we resume from dead end

`procedure set_conflict_locations(queen_number, queen_row, queen_col)` # update conflict positions for queen placed at location queen_row and queen_col

`procedure dead_end( )` # There are no more places available. Reset the board and restart the process with a different starting position for Q1

`procedure display_board():` # Presents the chess board in a nice format using string formatting. *This procedure is optional*

`function main()` # Control overall operations, control queen number and places queens on board, check and control dead ends and resumption. Continues until all queens are placed properly.

Note: Board_2D represents the chess board of size 4x4. Each cell can have one of the following values:

- 0 indicates that nothing is placed at that position.
- Q1 – Q4 indicates that a queen is placed at that position.
- 1 – 4 indicates that position is threatened by a queen (with the given queen number 1-4). A single position threatened by multiple queens uses the lowest queen number

Q2 output samples

Initial board setting

```
0   0   0   0
0   0   0   0
0   0   0   0
0   0   0   0
```

After placing first Queen

```
Q1  1   1   1
1   1   0   0
1   0   1   0
1   0   0   1
```

After first dead end

```
Q1  1   1   1
1   1   Q2  2
1   2   1   2
1   Q3  2   1
```

Final Solution

```
1   Q1  1   1
1   1   1   Q2
Q3  1   2   1
3   1   Q4  2
```