SFU SIMON FRASER UNIVERSITY

CMPT 120-D300 Assignment 3

Total points: 400, Weighting: 7.5%

Due: Sunday, March 12, 11:59 pm

Assignment 3 includes four programming problems. Read the following instructions before you start working on it:

- Assignments should be submitted to an electronic marking tool called *Gradescope*. If you have not registered in Gradescope yet follow the instructions in this document. You can submit your assignment as many times as you like until the deadline. If necessary, we will review and grade submissions individually to ensure that you receive due credit for your work.
- Follow the requested format for the file name, program inputs, and program outputs. Otherwise, your submission will not be graded. Use Python to write all the programs in this assignment.
- You are encouraged to ask your conceptual questions regarding the assignment on the course *Piazza*.
- You should NOT post any code segment about the assignment problems to Piazza or share your code with other students until March 12, 11:59 PM, the assignment extended deadline. Direct questions regarding your code and solution should be asked privately through Piazza.
- Gradescope is sensitive to user input prompts, as expected. Use the EXACT user input prompts as specified in the assignment questions, otherwise, some test cases may fail.
- You are allowed to use Python modules introduced during the semester to solve problems.
- We recommend you practice modular programming by writing functions and breaking your program down into pieces.
- For questions 1 to 4 you **must** name your code files **q1.py**, **q2.py**, **q3.py**, and **q4.py** respectively, and upload it to the, *H3Q1: AnagramsSolver*, *H3Q2: M1C2S3*, *H3Q3: GoDuckDuck* and *H3Q4: iGrade* sections on Gradescope.
- Commonalities between students' codes will be identified using a Code Similarity detection tool which might be considered plagiarism.
- Course assignments are individual tasks. Please respect Academic Integrity and follow the university's code of conduct.

Question 1: AnagramsSolver [100 marks]

Anagrams are words made by rearranging the letters of other words. For example, "cheater" is an anagram of "teacher", or "dictionary" is an anagram of "indicatory". Anagrams have been around since the ancient Greeks and are used in different ways from showing mystical meaning in names to anagrammatic poetry. An interesting example of using anagrams is how scientists announce new discoveries in the 17th century. In 1610, the Italian astronomer Galileo Galilei used the anagram "smaismrmilmepoetaleumibunenugttauiras" for "Altissimum planetam tergeminum observavi" ("I have observed the most distant planet to have a triple form") for discovering the rings of Saturn.

You are asked to write a program called AnagramSolver to find the number of all lists of words that are anagrams of each other in a given list of strings. To define more exactly, two words are considered anagrams if 1) There is the same number of characters in both words, and 2) You can form one word by rearranging the characters of another word.

AnagramSolver works as follows:

- It takes a positive integer n as the first input. Integer n determines the number of words that AnagramSolver is supposed to receive from the user. AnagramSolver assumes the user provides a number greater than zero.
- Subsequently, AnagramSolver receives the words in n separate lines.
- AnagramSolver is *case-insensitive*, meaning that it does not discriminate between uppercase and lowercase letters.
- AnagramSolver should ignore *space* and *comma* characters in the user-provided words. For example, Eleven plus two should be considered as Elevenplustwo.
- AnagramSolver should find all sets of words that are anagrams of each other and print out the size of the *largest* set in a separate line.

Three examples are shown in table 1. In example 1, first, the user entered 6 for the number of words. This means AnagramSolver should take 6 words in separate lines from the user. Next, the user entered the words eat, tea, tan, ate, nat and bat. Now, AnagramSolver processes the given words to find the set of words that are anagrams of each other. There are two such sets: $\{eat, tea, ate\}$ and $\{tan, nat\}$. The largest set size is 3 and AnagramSolver prints out 3. The same logic is applied to other examples.

You are free to use the modules introduced during the semester (and nothing beyond) to solve the problem. Note that your program input and output format should be *exactly* the same as the format of the examples shown in Table 1.

Table 1: The AnagramSolver Sample Input and Output

Example 1	**Example 2**	**Example 3**
6	2	6
eat	Eleven plus two	dormitory
tea	Twelve plus one	dirty room
tan	2	Angel
ate		arc
nat		glean
bat		car
3		2

Question 2: M1C2S3 [100 marks]

Numbers and counting are an integral part of everyone's daily life. The earliest known number system is the Mesopotamian base 60 system (3400 BC), and the Egyptians' system of hieroglyphs is the first known base 10 system (3100 BC). Nowadays, numbers are part of every discipline in science, from quantum mechanics to biology to music. Numbers are one of the simplest yet most impactful inventions of human history. To appreciate the role of numbers, in this part of the assignment, you write a program, called M1C2S3, to work with numbers.

M1C2S3 task is finding the contiguous subsequence that has the largest sum in a sequence of numbers. Given k numbers $n_1, n_2, n_3, \ldots, n_{k-1}, n_k$, M1C2S3 should detect consecutive numbers $n_i, n_{i+1}, \ldots, n_{j-1}, n_j$ which their summation is the greatest compared to any other consecutive numbers in the list. For example in the following list of numbers, the red box shows the contiguous subsequence producing the largest sum:

[4,-3, 4, 1,-7, 1, 5, 3]

L1S2C3 works as follows:

- M1C2S3 takes a positive integer n as the first input which determines the number of integers that the program is supposed to receive from the user.
- Subsequently, M1C2S3 receives the n integers in separate lines. Note that the user-provided numbers can be *positive* or *negative*.
- M1C2S3 does the processing and prints out the largest sum that can be obtained from the contiguous subsequence of the given list of integers in a new line.

Four examples are shown in table 2. As seen in example 1, the user first entered 8 which means M1C2S3 should receive 8 numbers from the users in separate lines. Next, the user entered the integers $-2, -3, 4, -1, -2, 1, 5, -3$ in separate lines. Finally, M1C2S3 identifies the contiguous subsequences with the largest sum and prints it out. For this example, the largest sum is 7 and M1C2S3 prints 7 in a separate line.
You are free to use the modules introduced during the semester to solve the problem. Note that your program input and output format should be exactly the same as the format of the examples shown in Table 2.

Table 2: The M1C2S3 Sample Input and Output

Example 1	**Example 2**	**Example 3**	**Example 4**
8	7	6	6
-2	2	-4	4
-3	-5	-3	4
4	4	4	12
-1	8	7	-9
-2	-9	1	6
1	1	5	25
5	6	17	42
-3	12		
7			

Question 3: GoDuckDuck [100 marks]

Search engines have a central role in information retrieval from the web. More than 90% of online experiences begin with a search engine. Search engines have a very complicated architecture but it has three major components: crawling, indexing and ranking. Crawling is about exploring the web to find new pages and indexing the process of storing information in a way for faster information retrieval. In this assignment, your focus will be on the search engine ranking component. The estimated number of web pages indexed in Google is about 50 billion. Returning pages most relevant to a query is one of the most important measures to evaluate a search engine's performance. And this is all about the ranking algorithm that a search engine uses. Two important sources that search engines use in their ranking algorithms include the content of web pages and links between them. In this part of the assignment, you will develop a content-based search engine called, GoDuckDuck. Given a query, GoDuckDuck task is to explore a set of text files, and employ a *ranking* algorithm to return the most relevant file with regard to the given query considering the files' content.

To implement GoDuckDuck first, we need to define the ranking algorithm we use. For a given query, the ranking algorithm computes a relevance score for each document; a greater relevance score ranks a document higher. The main input of the ranking algorithm is the frequency of query words in different documents:

$$frequency(word,\ document) = number\ of\ occurrences\ of word\ in\ document$$

And the score of a document with respect to a query is computed as follows:

$$score(query,\ document) = \sum_{word\ in\ query} frequency(word,\ document)$$

meaning that the score is the total sum of the frequency of each word in the document. The documents are ranked based on their scores. Imagine your document content is: "they ran into many problems in the last year. They are trying to solve them.". And the user query is "they went". First, we parse the query to get the single words in the query including "they" and "went". The frequency of these words are

$$frequency(they,\ document) = 2$$
$$frequency(went,\ document) = 0$$

and consequently the score of this document is equal to 2. GoDuckDuck works as follows:

- First, GoDuckDuck receives the total number of documents n (greater than 0) that it does the search over. GoDuckDuck assumes there are n text files in the same directory as your program with the names $1.txt$, $2.txt$, $\cdots$, $n.txt$. Each file contains words separated by *space*.
- Next, GoDuckDuck receives a query from the user in a new line. This is a string containing words separated by *space*. Then, GoDuckDuck does the processing and computes the score of each document with respect to the given query using the ranking algorithm described above.
- Finally, GoDuckDuck prints out the *name* of the file with the *maximum* score, which is a number *between 1 and n*, in a new line. For example, if "5.txt" has the highest score, your program simply prints out 5. If there are two or more documents having the same largest score, your program should print the document with the smaller name. For example, if files '4.txt' and '7.txt" gains the same score as the highest score, GoDuckDuck prints 4 (as a smaller file name) in the output.

- In counting words, GoDuckDuck only considers the *exact* ones. Meaning that the program in the counting process ignores the derivatives of a given word, or if the word is surrounded by characters such as comma, quote, hyphen, parenthesis and apostrophe. Moreover, GoDuckDuck is case-sensitive, meaning that it differentiates between lowercase and uppercase letters.

Three examples in Table 3 are related to the result of GoDuckDuck on the corpus of three files shown in Figures 1, 2 and 3. In all three examples, the first line is equal to three which shows the number of text files GoDuckDuck should process. The second line shows the submitted query and the third line is the name of the file with the highest score relevant to the given query. In example 1, the query contains two words "SFU" and "Vancouver". Only the word "SFU" occurred one time in "1.txt" resulting in score of 1 for this document. Each of the words "SFU" and "Vancouver" occurred in the file "2.txt" one time. This means the score of this file is 2. And "3.txt" does not contain these words and this assigns a score of 0 to this file. Accordingly "2.txt" gains the highest score and GoDuckDuck prints 2 in the output. Note that GoDuckDuck ignored "SFU's" and "(SFU)" in counting the number of "SFU". In example 2 the query consists of only one word "Italy" and only "3.txt" includes this word. So, GoDuckDuck prints 3 as result. For example 3, the query includes three words each has appeared only once in "1.txt" and "2.txt" This means these two files get the same score and GoDuckDuck prints 1 as the name of the file which has a smaller value.

You can use the modules introduced during the semester to solve the problem. Your program input and output format should be exactly the same as the format of the examples shown in Table 3.

Table 3: GoDuckDuck Sample Input and Output

Example 1	**Example 2**	**Example 3**
3 SFU Vancouver 2	3 Italy 3	3 Simon Fraser University 1

Figure 1: First document called *1.txt*

Simon Fraser University is recognized as one of the top employers in both the Province of British Columbia and in Canada. SFU's strategic vision of being Canada's most engaged university is supported by our ability to attract and retain the best people. SFU staff play a major role in helping the university maintain its unwavering commitment to educational and research excellence.

Figure 2: Second document called *2.txt*

Simon Fraser University (SFU) is one of most community-engaged comprehensive universities in Canada. With a wide range of activities and programs at both undergraduate and graduate levels, SFU is one of the top choice for both domestic and international students in Vancouver.

Figure 3: Third document called *3.txt*

The sea surrounds Italy and mountains crisscross the interior, dividing it into regions. The Alps cut across the top of the country and are streaked with long, thin glacial lakes. From the western end of the Alps, the Apennines mountains stretch south down the entire peninsula.

Question 4: iGrade [100 marks]

A database management system (DBMS) is a software application for creating and managing databases. A DBMS makes it feasible for end users to create, protect, read, update and delete data in a database. The three most widely used DBMSs are Oracle, MySQL and Microsoft SQL server. In this part of the assignment, you as a programmer are supposed to help a university student services to organize courses and students' grades. Imagine the DBMS that handles the university's course management system has crashed, and you are asked to develop a program called, iGrade, to help the school temporarily manage the students' grades.

Program Input. You will be given a text file, called grades.txt, containing the grades of different students who took different courses from different departments. Each line of the given file includes four pieces of information: department, course number, student number and grade, which are separated by a *space*.

- *Department* is a string.
- *Course number* is a 3-digit integer.
- *Student number* is a 9-digit integer.
- *Grade* is an integer between 0 and 100.

Below is an example content of the file "grades.txt":

CMPT 120 926456789 100
ECON 103 423453389 82
BUS 217 534755481 79

Program Tasks. iGrade is capable of running three tasks: 1) computing the average grade of students in a specific course, 2) the GPA of a specific student in their courses, and 3) the number of failed students in a specific course. iGrade receives four types of commands as follows:

1. **avg.** The avg command asks for the average grades of a specific course. For this command, the user command's format is as follows:

   avg department Course_number

   - iGrade should process the given text file, compute the average grade of all students who took the course defined in the command, and print it out.
   - You should print the result with exactly two decimal points. To do so, you should use the statement: `print("{:.2f}".format(average))`, where 'average' is the result of your computation. For example, if the average is 67.346, it will be rounded to 67.35. If it's 67.344, it will be rounded to 67.34.
   - The program assumes the user doesn't ask for a course that doesn't exist in the text file.

2. **gpa.** The gpa command asks to compute the GPA of a specific student given by the user. For this task, the user command's format is as follows:

   gpa Student_number

- Your program should retrieve the grades of the given student and print the average of the student's grades.
- You should print the number with exactly two decimal points. To do so, you should use: `print("{:.2f}".format(qpas))`, where 'qpas' is the result of your computation.
- The program assumes the user doesn't ask for a student that doesn't exist in the text file.

3. **fails.** The fails command asks for the number of failed students of a specific course. Grades strictly less than 75 counts as fails. For this command, the user command's format is as follows:

$$fails\ department\ Course\_number$$

- Your program should process the given text file, and count the number of failed students of the course defined in the command, and print it.
- The program assumes the user doesn't ask for a course that doesn't exist in the text file.

4. **quit.** The program continues to get new commands from the user until the user enters the command 'quit'.

You should assume that the text file *grades.txt* is in the *same directory* of your code. You will be provided an example test file (Canvas —> Assignments) but your program should be able to handle other files having the same format and running commands defined here. Table 4 shows an example output of iGrade on the provided test file. First asks for the average grades of students who took course 065 in the IAT department. Based on the data in the grades file, the answer for this command is 52.6 but print it as 52.60 to have exactly two decimal points. The next command is to find out the gpa of a student with student number 009204304. Their gpa was 49.92 as printed in the output. The last command is to find out the number of failed students enrolled in course 246 in the GERO department. As the results show only one student had a score strictly less than 75 in this course. Example two follows the same process but the program ends when it receives the command 'quit'. Note that iGrade is case-sensitive, meaning that it differentiates between lowercase and uppercase letters.

You can use the modules introduced during the semester to solve the problem. Your program input and output format should be exactly the same as the format of the examples shown in Table 4.

Table 4: iGrade Sample Input and Output

Example 1	**Example 2**
avg IAT 065 52.60 gpa 009204304 49.92 fails GERO 246 1 quit	avg IAT 065 52.60 quit