COMP 1405B
Fall 2019 – Practice Problems #4

Objectives

- Continue to practice using the programming concepts we have covered already
- Practice writing code with for loops, while loops, and nested loops
- Practice writing code that uses files for data storage
- Practice designing algorithms to work with structured file data
- Practice applying problem solving and computational thinking skills to programming problems

---

Remember to test your solutions to verify their correctness and discuss your solutions with both other students and TAs in the course. Verifying your solutions with TAs will be a good way of ensuring that you are on a path to success in the course.

Problem 1 (Divisors and Prime Numbers - **)

a) Write a program that uses a while loop to print all divisors of a number supplied by the user. The program should also print the sum of all the divisors and whether the number the user entered is a prime number or not. Note: The definition of a divisor is a number that divides another evenly (i.e., without a remainder) and the definition of a prime number is a number whose only divisors are 1 and itself. You can assume the user will enter a positive integer.

b) Implement the same program as above using a for loop instead of a while loop. Which implementation do you think is the better choice for this problem? Discuss with a friend and/or a TA if you are unsure.

Sample Outputs (user input highlighted)

Enter an integer: 20
The divisors are:
1
2
4
5
10
20
The sum of the divisors is 42
The number is not prime.

Enter an integer: 19
The divisors are:
1
19
The sum of the divisors is 20
The number is prime!

Problem 2 (Vegetable Garden – **)

You have a prized vegetable garden in your backyard but are worried about the water levels. You work a significant distance from your house and are unsure if it rains on your garden during the day, so you do not know if your garden needs watered or not. You install a rain sensor in your garden, which can answer the True/False question of whether it rained during the day. As you are a busy person, you cannot check the sensor each day, so you must write a program that will print out a warning message and stop sensing if it does not rain for 3 days in a row. Note: to keep tuition below $100,000/year, we do not have a real garden and sensor, so you will be responsible for typing in the sensor measurements (True or False) for the program. The nice thing is that with the use of abstraction, almost all of your code would be the exact same if you ever did get a real sensor. The only change required would be to change the user input line of code to a line that read a value from the sensor.

Sample Output

It rained today? True
It rained today? False
It rained today? True
It rained today? False
It rained today? False
It rained today? False
Quick! Water your garden before all the plants die and you starve to death!

Problem 3 (Number of Digits - ***)

Write a program that asks the user to enter an integer, then prints the number of digits in the integer. You should **not** use the len() command/function to get the length of the string. The code for this problem isn't too complex but thinking of a way to determine the number of digits may be more difficult. Hint: think about counting how many times you can move the decimal point (i.e., divide by 10).

Sample Output:

Enter an integer: 1234321
There are 7 digits in that number.

Problem 4 (Course Grade Analysis - ***)

For this problem, you will create a grade analysis program. When this program runs, it should prompt the user for a specific filename. You can assume the user input is valid. The program must then read the specified file and produce a summary of the grades, including: the number of people who passed the class, the number of people who did not pass the class, the average of all the final grades within the class and the name/grade of the students with the highest/lowest grades in the class. The final grade can be calculated with the following weights: assignment=25%, midterm=25%, exam=50%. To pass the course, a student must have received a final grade of 50% or greater and a final exam grade of 50% or higher. The studentinfoX.txt and studentinfoX-output.txt files included in the PP4-Resources.zip file on cuLearn contain test input files and their expected output respectively. You can use these files to verify the correctness of your solution. It is still advisable that you show your work to a TA or other students in addition to checking its correctness with the test files.

All files used for this problem will have the following line-by-line structure:

Student1_first_name
Student1_last_name
Student1_student_number
Student1_assignment_grade
Student1_midterm_grade
Student1_exam_grade
Student2_first_name
Student2_last_name
Student2_student_number
Student2_assignment_grade
Student2_midterm_grade
Student2_exam_grade
…etc…

Problem 5 (Tracking Numbers - **)

Write a program that repeatedly asks the user to enter an integer until they want to quit (e.g., by entering ‘q’). The program should then print the largest number entered, the

smallest number entered, the average of all numbers entered, the number of positive numbers entered, and the number of negative numbers entered (0 should not count as positive or negative).

Now, instead of having the user enter the set of numbers, ask the user to enter a filename. The program should then read all the numbers from the specified file and output the same data as before. You can assume the filename entered by the user specifies a file that has a single integer value on each line. The PP4-Resources.zip file on cuLearn contains numbertestX.txt files to test your program's output. The numbertest-resuts.txt file includes the correct values you should see for each file. Like the garden sensor problem, the ideas of abstraction and decomposition are important to identify within this problem. Until this point, we have been using the user to enter in any data that we need. But the logic for our program does not need to change significantly if we need to read the data from another source like a file. What we are really doing, is creating an algorithm to process a *stream* of data, regardless of its source.

Problem 6 (Order/Inventory System Progressive Problem 1 - ***)

This problem is part of a set of progressive problems that will aim to build up a complex system by the end of the course. If you work through all of the Order/Inventory System Progressive Problems, by the end of the term you will have created an ordering and inventory system that will allow a user to search for manage products in a catalog, search for products in a catalog, process orders, track inventory levels, and calculate sales statistics. In this first part, you will develop a text-based ordering system for the Buy-nary Computing electronics store. Please be sure to read through the entire problem before beginning your design and be sure to design your approach before you begin to code it.

To start, you'll build a menu system that allows customers to purchase quantities of a single type of electronic product. The program should start by asking the customer for their name. Next it should display a menu that asks the user to select one of the following 4 kinds of products:

1. Desktop Computer ($850 each)
2. Laptop Computer ($1225 each)
3. Tablet ($600 each)
4. Toaster Oven ($85 each)

The user should make their selection by entering the corresponding number. You can assume the user will enter an integer, though they may enter an invalid integer (e.g., <=0, >4). Invalid selections should be handled with a suitable error message and the

menu should be repeated. Once the customer has selected a valid product, the program should ask how many of that product the customer would like to buy. Following this, the program should provide the user with their receipt (report the product, quantity, and total cost of the user's order). The program should also be personable and use the customer's name when printing the receipt. A sample run of the program that demonstrates how your output could look is included in the Ordering-System-Part-1-Demo.txt file within the PP4-Resources.zip from cuLearn.

Modify your code for Buy-nary Computing’s text-based purchasing system so that customers can purchase multiple types of products. To do this, the menu that you previously built should repeat to the customer until they select a new "Complete Order" option. Your program should keep track of the quantity of each kind of product that the customer orders. Note: if a customer selects the same product twice, their order amounts should be added together (i.e., they should be able to order 5 Desktop Computers and then order 5 more Desktop Computers for a total of 10). Once the customer selects the "Complete Order" option, their receipt should then display as before but now should include all of the products they have purchased. An example run of the program that demonstrates how your output could look is included in the Ordering-System-Part-2-Demo.txt file within the PP4-Resources.zip from cuLearn.

Modify your code by adding a new variable to keep track of the stock level of each product in the store. The initial stock quantity of each product should be 15. Update your menu display to include the stock levels of each product. Whenever a customer successfully selects a product and specifies an acceptable amount, the stock level should be decreased by the amount the customer has selected. If there is not enough stock to fulfill the request, the customer should receive a message indicating this and the menu should be printed again (similar to an incorrect input scenario). An example run of the program that demonstrates how your output could look is included in the Ordering-System-Part-3-Demo.txt file within the PP4-Resources.zip from cuLearn.

If you are looking for more of a challenge, you can extend this problem in several ways. Try modifying the code for this problem to store the inventory information in a file. Try loading the stock information from a file when you start the program and saving the updated information back to the file when the user is done. This way, changes in product stock will be saved across executions of your program. You can also add a management menu to allow the user to create, remove, or edit products (some of these will be difficult using only the concepts we have covered so far).

Problem 7 (Increasing Sequences - ****)

An increasing sequence of numbers is one in which each number is larger than the one before it. For example, [4, 8, 13, 14, 21] is an increasing sequence, but [4, 8, 13, **12**, 21] is not. Write a program that repeatedly reads positive integers from the user. Your program should track the length of the longest increasing sequence of numbers that the user has entered and display it when the user has finished entering numbers. The user is finished entering numbers when they input “q” or when they enter a decreasing sequence of numbers (i.e., each number is lower than the previous) of length 3. Alternatively, write a version of this program that works with a file-based approach. In this case, the program should just print out the longest increasing sequence present in the given file.

Sample Outputs (user input highlighted and longest sequence bolded)

Enter a positive integer or 'q' to quit: **1**
Enter a positive integer or 'q' to quit: **3**
Enter a positive integer or 'q' to quit: **4**
Enter a positive integer or 'q' to quit: **5**
Enter a positive integer or 'q' to quit: 4
Enter a positive integer or 'q' to quit: 6
Enter a positive integer or 'q' to quit: 5
Enter a positive integer or 'q' to quit: 4
Length of longest increasing sequence is 4

Enter a positive integer or 'q' to quit: 1
Enter a positive integer or 'q' to quit: 5
Enter a positive integer or 'q' to quit: **3**
Enter a positive integer or 'q' to quit: **6**
Enter a positive integer or 'q' to quit: **7**
Enter a positive integer or 'q' to quit: 5
Enter a positive integer or 'q' to quit: 8
Enter a positive integer or 'q' to quit: 6
Enter a positive integer or 'q' to quit: q
Length of longest increasing sequence is 3

Problem 8 (Trip Distance Calculator - ****)

Write a program that repeatedly reads x and y coordinates representing the location of cities. Each time a new city is entered, the program should calculate the Euclidean distance between the last city entered and the new city (i.e., how far it would be to travel between the two cities). The program should track the total distance and display it once

the user enters "q" for either the x or y coordinate. Your program can assume that the user will always enter integer values. To calculate the square root of a value, you can import the math library and use the square root function.

Note, if city #1 is at location (x1,y1) and city #2 is at location (x2,y2), the Euclidean distance between them can be calculated as:

$$dist = \sqrt{(x1 - x2)^2 + (y1 - y2)^2}$$

Problem 9 (Trip Planner - *****)

The goal of this problem will be to produce a program that is easier for the user to use than the Trip Distance Calculator problem. To start, design a file format to store city information. Each city should be represented by a name, an X coordinate, and a Y coordinate. Write a program that prompts a user to decide if they want to:

1) Enter a new city
2) Remove a city
3) List cities
4) Plan a trip
5) Quit

If the user decides to enter a new city, your program should get the appropriate information from them and add the entry to your city file. **If the user wants to remove a city**, the program should list all of the city names and give the user a method for selecting which city they want to remove (e.g., by selecting a menu number or a city name). Your program should then update your city file. Note that we do not currently have a way to easily remove information from a file. To solve this problem, you may have to be creative with how you manipulate files. Think about some potential solutions and discuss with others. Ultimately, if the removal operation is unsupported in your program, that is fine. You can come back to this problem after we have covered lists later in the course and it will be easy to accomplish. **If the user decides to list the cities**, your program should print out a list of the cities and possibly include their location information. You should format this information nicely.

If the user chooses to plan a trip, your program should ask them for the name of a starting city. You should check that any city name the user enters is valid (i.e., exists within the file) before accepting it. The program should allow the user to repeatedly enter more city names that they need to visit until they signify that they are finished. When they are finished, your program should print out the total distance of their trip if they were to go to all cities in the order specified. When the user is finished, they should return to the main menu.

Problem 10 (Trivia Game Progressive Problem 2 - ***)

If you completed the trivia problem from Practice Problems #3, you should currently have a trivia 'game' that involves only a single question, or possibly several questions. Now that we can loop and use files, we can expand this into an actual game. First, create a question/answer file that will store all the questions for your game. You can decide on the format, but question/answer/repeat would be a logical line-by-line breakdown. Now modify your original game so that it loops through all the questions and has the user guess the answer to each one. Add the idea of score to your game by giving the user X points for getting a question correct and -Y points for getting it wrong. Finally, add a high score file to your game. Record the name and score of the player with the highest score (or top 3, 5, 10, etc. if you are looking for an extra challenge).

Problem 11 (Search Engine Progressive Problem 1 - ****)

This problem is part of a set of progressive problems that will aim to build up a complex system by the end of the course. If you work through the "Search Engine" problems throughout the course, by the end of the term you will have created the basic components of a search engine. The search engine will be able to read pages from the internet, summarize/model the content of pages, accept search requests from users, and provide suggested pages. Note: this will be an offline search engine with commands specified through the console, but the algorithms used will still be applicable to an online version. If you looked up a way to create an actual web-connected program in Python, you could easily use the code you create as part of these problems to make it a real search engine.

We will start with a more basic search engine that will search through files. In an abstract sense, though, these files can be thought of as representing pages from the internet. Later, we can just add in functionality to read information from the web and save it to files before processing those files. To start this problem, look in the search-engine folder within the PP4-Resources.zip. Within the folder, there is a pages.txt file, along with some N-X.txt files and a search-results.txt file. The pages.txt file contains a list of the other files included, while the N-X.txt files represent web pages with various words on them (each line contains a word) The search-results.txt file can be used for testing. You can use the pages.txt file to read the names of the other files that you will have to search.

The main goal of this problem will be to have a user enter a search word and then have the program tell the user the page (in this case, file) that has the most occurrences of the search word. To do this, you will have to read through each file's contents and count how many times the search word occurs, remembering the highest page/count in a variable. Remember to break the problem down! For each page/file there is, you will

need to count the search word frequency. This will likely involve a nested loop. First, try writing a program that prints the file names from the pages.txt file. Once you have this down, you can replace the logic for printing the file name with logic for reading the file and counting the search term frequency. You can then add logic to remember the page with the highest search term frequency (similar to problems 4-6).

Once you have implemented your solution, you can use the search-results.txt file to see if it is working correctly. This file contains some sample search words, along with the page with the highest count of that search word (ignore the parts about ratio for now).

While counting the word frequency works, it does have some drawbacks. For one, large pages will have more words in general. Some pages may then seem more important because they have a higher count, even if the word only makes up a small portion of the page's total contents. In some other cases, pages may repeat the same words repeatedly on purpose to seem like better matches (this was a real problem for early search engines, with frequently searched words being hidden many times on pages unrelated to those words). So, you may want to modify your logic to find the ratio of words that match the search word. That is, the percentage of all words on the page that match, instead of just the count. If you do this, you can use the ratio results from the search-results.txt to test your implementation.

Later in the course, we will begin to focus on program efficiency in a formal way, using some basic mathematical analysis. For now, we can just acknowledge that reading through the entire contents of a file would be slower than reading through a summary of that file. The current solution you have involves reading through the contents of a file and counting the frequency of a word. Can you think of a way to increase the efficiency of this? If we aren't interested in the context of the words (e.g., the order and location of all words within the document), is there a way you can store the same information (i.e., words and their frequencies) using less text? Can you read the original pages once, produce a structured summary file, and use this summary file to perform your search?