SFU SIMON FRASER UNIVERSITY

CMPT 120-D300 Assignment 2

Total points: 400, Weighting: 7.5%

Due: Sunday, February 19, 11:59 pm

Assignment 2 includes five 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.
- Assignments should be submitted on the due date in order to receive full credit. If you submit your assessment after the due date a penalty of 5% of the total mark allocated for the assignment is deducted per day for the first 3 days after which the submission is not accepted.
- 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 February 22, 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 not allowed to use *any* Python built-in methods or user-defined modules in your programs.
- We want you to practice modular programming in this assignment by writing functions and breaking your program down into pieces.
- For questions 1 to 5 you **must** name your code files **q1.py**, **q2.py**, **q3.py**, **q4.py** and **q5.py** respectively, and upload it to the *H2Q1: SubstringFinder1*, *H2Q2: SubstringFinder2*, *H2Q3: PaNNaP*, *H2Q4: NerdyGrocery* and *H2Q5: gE2kylcm* 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: SubstringFinder1 [70 marks]

DNA is a large molecule that can be abstractly defined as a sequence of nucleotides including Adenine (A), Cytosine (C), Guanine (G), and Thymine (T). A DNA sequence is a representation of a string of these four nucleotides, e.g. ATTCGTAACTAGTAAGTT. DNA sequencing is the process of identifying the sequence of nucleotides in a strand of DNA. This is a critical task for understanding human genomic variation and the genetic contributions to certain diseases. The human genome has about 3 billion nucleotides. DNA sequencing is a computationally intensive task that needs efficient string pattern and substring matching algorithms. And in the first two questions of the assignment, you are asked to write a program called SubstringFinder1 to find a substring with defined characteristics in a given string.

SubstringFinder1 works as follows:

- Takes a string called source as the first input.
- Takes a positive integer k as the second input. SubstringFinder1 assumes the user provides a number greater than zero.
- SubstringFinder1 does the processing and prints the k-length substring of source with the greatest number of *unique* characters.
- If there are multiple substrings that satisfy the condition in the previous step, SubstringFinder1 prints the one that appears first in the user-provided string, source.
- SubstringFinder1 continues to get inputs from the user until the user enters 'quit' for string source.

You are not allowed to use *any* Python built-in methods or user-defined modules in developing SubstringFinder1. You can see examples of SubstringFinder1 input and output in Table 1. 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: TheSubstringFinder1 Sample Input and Output

Example 1	**Example 2**	**Example 3**	**Example 4**
actxaxu 5 actxa abcafaghu 4 bcaf sfucmptsfu 8 sfucmpts quit	quit	hello hello 3 hel quit	yumghhs 6 yumghh quit

Question 2: SubstringFinder2 [60 marks]

The second problem of assignment 2 is also about finding substrings. This program, called SubstringFinder2, finds a *target* string in a *source* string and returns the *index* of the first character of *target* in *source*. SubstringFinder2 works as follows:

- SubstringFinder2 takes a string called source as the first input.
- Then, the program takes the string target as the second input.
- SubstringFinder2 is supposed to check if target is a substring of source, and if yes, it returns the index of the first character of target where it appears in source.
- If target does not appear in source, SubstringFinder2 prints -1.

In developing SubstringFinder2, you are not allowed to use *any* Python built-in methods or user-defined modules. Instead, you have to use the basic knowledge of string processing such as the index operator, the slice operator, and string comparisons. You can see examples of SubstringFinder2 input and output in Table 2. 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 SubstringFinder2 Sample Input and Output

Example 1	**Example 2**	**Example 3**	**Example 4**
abcabdab cab 2	sfuisauni sau 4	hhgywshxns wsx -1	yyyyyyy yyyy 0

Question 3: PaNNaP [60 marks]

A palindrome is a word that reads the same backwards as forwards such as radar, level, and kayak. An example of a palindrome sentence is "Was it a car or a cat I saw?". The concept of palindrome has a rich history but the first physical examples can be dated to the 1st century CE, the Sator square found on a wall in the medieval town of Oppède-le-Vieux, France. In this part of the assignment, you will write a program called PannaP to detect palindrome strings.

PaNNaP works as follows:

- Takes a string called source as the first input.
- PaNNaP checks if the user-provided string, source, is a palindrome.
- PaNNaP is case-insensitive, meaning that it does not discriminate between uppercase and lowercase letters.
- PaNNaP should ignore spaces and commas in the user-provided string.
- Finally, PaNNaP prints 'Yes' if the input is a palindrome and 'No' otherwise.

You are not allowed to use *any* Python built-in methods or user-defined modules in developing PaNNaP. Table 3 shows sample input and output of PaNNaP. Note that your program input and output format should be exactly the same as the format of the examples shown in Table 3.

Table 3: The PaNNaP Sample Inputs and Outputs

Example 1	**Example 2**	**Example 3**	**Example 4**
A xxs'sxxa Yes	hello No	Amore, Roma Yes	Yo, banana boy! No

Question 4: NerdyGrocery [120 marks]

`A programmer's wife tells him, "While you're at the grocery store, buy some eggs."`
`He never comes back.😎😄😎😄`

Now, imagine a situation in which such a programmer does shopping in a grocery owned by a nerdy! Things can even get more complicated, right? Joking aside, in this part of the assignment, you as a programmer are supposed to help a nerdy owner of a grocery by writing a program called NerdyGrocery as follows.

The grocery store owner designed a very strange pricing regulation for their products, in which the price of a product only depends on its *weight*.

- If the weight of a product is a prime number, the price of the product is equal to the number of prime numbers less than the weight of the product. For example, if the weight of a product is 7, the price would be $3, because there are three prime numbers less than 7, which are 2, 3, 5.
- If the weight of a product is not a prime number, the price is equal to the number of prime divisors of the weight. For example, if the weight of a product is 6, the price would be $2, because 6 has two prime divisors, which are 2 and 3.

Moreover, the grocery offers a discount based on the total price of a customer's shopping:

- If the total price of shopping is a prime number, the grocery offers a discount equal to the number of prime numbers less than the total price.
- If the total price of shopping is not a prime number, the grocery offers a discount equal to the number of prime divisors of the total price.

NerdyGrocery helps the grocery and its customers to facilitate the payment process as follows:

- NerdyGrocery first, receives the number of products in the customer's shopping basket. Let's call this number n. NerdyGrocery assumes the user only enters positive integers (greater than 0).
- Subsequently, NerdyGrocery receives the weight of each product in n separate lines. NerdyGrocery assumes the user only enters positive integers (greater than 0).
- NerdyGrocery calculates the total price of the customer's shopping considering the above-mentioned regulation and after applying the corresponding discount.

Two examples are shown in table 4. In example 1, the customer had 5 products in their basket. The weight of the first product is one. 1 is not a prime number and doesn't have any prime divisors, so its price is \$0. The weight of the next product is 7, and as explained earlier, its cost is \$3. The next product weighs 6, and it costs \$2, as explained above. The next item's weight is 12, which is not a prime number. The number of prime divisors of 12 is 2, so its price is \$2. Finally, the last product weight is 2, and it is a prime number. There isn't any prime number less than 2, so it costs \$0. Therefore, the total price is \$7.

Since the total price is a prime number, the total discount is equal to the number of prime numbers less than 7, which is 3. Consequently, the total price after applying the discount will be $\$7 - \$3 = \$4$ as printed in the last line as the expected output of NerdyGrocery for this example.

You are not allowed to use *any* Python built-in methods or user-defined modules in developing NerdyGrocery. The same logic is applied to the inputs provided in example 2. Note that your program input and output format should be exactly the same as the format of the examples shown in Table 4.

Table 4: The NerdyGrocery Sample Inputs and Outputs

Example 1	**Example 2**
5 1 7 6 12 2 4	4 17 12 1 20 8

Question 5: gE^{2}kylcm [90 marks]

A lazy student is attending a very basic math class! And sometimes laziness enables people to find the quickest ways to get things done! Our student is given an arbitrary number of fractions, and the task is to find out the total sum of the fractions. And students in the class are allowed to get help from others to solve the problem.

As you know, to add fractions with each other, one has to find the *least common multiple* of the denominators. The lazy student is not in the mood of calculating the least common multiple by hand! Your help is greatly needed by the lazy student to write a program, called gE^{2}kylcm, to calculate the least common multiple of a bunch of numbers! Let's explore the problem to know how to develop gE^{2}kylcm. The least common multiple (lcm) of two numbers can be calculated as:

$$lcm(a,b) = \frac{a.b}{gcd(a,b)} \tag{1}$$

where $gcd(a,b)$ is the *greatest common divisor*. This means, in order to calculate lcm of two numbers, you need to calculate their greatest common divisor (gcd) first. In order to find out the greatest common divisor of two numbers, you have to use the fact that:

$$gcd(a,b) = gcd(b,r) \tag{2}$$

where r is the remainder of the division of a/b. Therefore, for example, in order to find out $gcd(99,36)$, by using this fact iteratively, you would have: $gcd(99,36) = gcd(36,27) = gcd(27,9) = gcd(9,0)$. Hence, when the remainder gets zero, you would find out the gcd, which is 9 in this case.
Now, that you learn about the algorithm of computing lcm it is time to develop gE^{2}kylcm:

- gE^{2}kylcm first receives the number of denominators that you have to calculate their lcm. Let's call this number n which is always greater than 1.
- Subsequently, gE^{2}kylcm receives the denominators in n separate lines. gE^{2}kylcm assumes that the user only enters positive integers (greater than 0) in this step.
- Finally, gE^{2}kylcm will find out the least common multiple of all of the given numbers, and prints the result in a separate line.

Two examples are shown in Table 5. In example 1, the user asked the program to compute the lcm of 5 numbers (18,6,3,9,1) and the result is printed as 18. In example 2 the user asked the program to compute the lcm of 3 numbers (4,5,6) and the result is printed as 60. Note that your program input and output format should be exactly the same as the format of the examples shown in Table 5. You are not allowed to use *any* Python built-in methods or user-defined modules in developing gE^{2}kylcm.

Table 5: The gE^{2}kylcm Sample Inputs and Outputs

Example 1	**Example 2**
5 18 6 3 9 1 18	3 4 5 6 60