CSCI 1133
Exercise Set 4

You should attempt as many of these problems as you can. Practice is *essential* to learning and reinforcing computational problem solving skills. We encourage you to do these without any outside assistance. Create a directory named `exercise4` and save each of your problem solutions in this directory. Name your individual Python source files using "`ex4`" followed by the letter of the exercise, e.g., "`ex4a.py`", "`ex4b.py`", "`ex4c.py`", etc.

Note that automatic testing of your solutions is performed using a GitHub agent, so it is necessary to name your source files precisely as shown and push them to your repository.

You need to include the following statement in the global namespace in order to automatically have the `main()` function executed when the module is loaded and executed in script mode:

```
if __name__ == '__main__':
    main()
```

A. **Binary to Decimal Conversion**

Write a Python program that does the following:

- Prompts the user and inputs a binary (base-2) value from the console as a string (e.g., "1110101")
- Calls a pure function, `binaryToInt` to compute the equivalent decimal value
- Displays the result on the terminal display
- Includes a loop that will continue this process as long as the user wishes.

Your program must include a pure function named `binaryToInt` that will take a binary value as a *string* argument and return the equivalent decimal value as an *integer*. The input argument consists of '0' and '1' digits only:

```
def binaryToInt(binarystring):
```

for example, the binary string '1110101' represents the value:

$$1x2^6 + 1x2^5 + 1x2^4 + 0x2^3 + 1x2^2 + 0x2^1 + 1x2^0$$

$$= 64 + 32 + 16 + 0 + 4 + 0 + 1$$

$$= 117$$

Example:

```
enter a binary value: 1110101
value is: 117
continue? (y/n): y

enter a binary value: 11
value is: 3
continue? (y/n): y

enter a binary value: 0
value is: 0
continue? (y/n): n
```

B. **Password Checking**

Many secure applications require that you use a "good" password. A "good" password is more difficult to guess and is therefore more secure. Write a well-structured Python program that does the following:

- Prompts the user and inputs a password (as a string) from the console
- Calls a function named `passwordCheck` to compute determine if it is a "good" password
- Displays the result on the terminal display
- Includes a loop that will continue this process until the user enters a "good" password.

Your program must include a pure function named `passwordCheck(password)` that will take a single string argument containing a proposed password and determine if it is "good" according to the following rules:

- must have at least 8 characters
- must have at least one uppercase letter
- must have at least 2 digits
- must have at least one non-alphabetic *and* non-digit character

Example:

```
enter a password: abCd.99
abCd.99 is not a valid password

enter a password: abcde099
abcde099 is not a valid password

enter a password: abcde09.@9
abcde09.@9 is not a valid password

enter a password: Abcde09.@9
Abcde09.@9 is valid
```

C. **Alternating Sum**

Write and test a well-structured Python program that will input and compute the *alternating sum* of a sequence of values. The *alternating sum* of the following sequence:

1, 4, 9, 16, 9, 7, 4, 9, 11

is computed by alternately adding and subtracting the elements in the sequence as follows:

1 - 4 + 9 - 16 + 9 - 7 + 4 - 9 + 11 --> -2

Your program must do the following:

- Include a pure function named `altSum(values)` that will take a Python `list` of floating-point values as its only argument and return the alternating sum of all the elements in the list
- Your `main` function should solicit values from the user until he/she enters a null string (Recall that pressing the `enter` key without any value will produce a null string)
- Convert each value (as its entered) into a floating-point number and dynamically add it to the end of a Python list
- After the list values have been entered, call the `altSum` function to determine the alternating sum and then print it

Example:

```
enter a floating point value: 1
enter a floating point value: 4
enter a floating point value: 9.2
enter a floating point value:

The alternating sum is: 6.2
```

D. **Roman Numerals**

Write a well-structured Python program that will input Roman Numerals as strings and convert them to an equivalent decimal value (integer). Include a pure function named `romanToDecimal(stringarg)` that takes a single string argument containing a Roman Numeral (e.g., 'IX', 'VIII', etc.) and returns its equivalent decimal value as an integer. Recall that individual Roman Numeral "digits" have the following values:

I = 1
V = 5
X = 10
L = 50
C = 100
D = 500
M = 1000

If the input contains a non-valid roman 'digit', return 0 (zero).

To convert a Roman Numeral to a decimal value, you need to consider the “rules” of Roman Numeral representation:

1. Reading left-to-right, higher valued symbols generally appear before lower valued symbols. The value is obtained by summing the individual symbol values.
2. However, if a *single* lower valued symbol *immediately* precedes a higher value, it is subtracted from the total rather than added.

3. The symbols I, X, C, and M cannot be repeated more than 3 times in succession.
4. 'I' can only be subtracted from 'V' and 'X'. 'X' can only be subtracted from 'L' and 'C'. 'C' can only be subtracted from 'D' and 'M'. 'V', 'L', and 'D' can never be subtracted.
5. Only a single lower valued symbol may be subtracted from a higher valued symbol.

Hint: You may find this easier if you first write a helper function that will convert individual Roman 'digits' to their equivalent decimal value.

Hint: Your function should sum up the symbol values from left-to-right, but *defer* adding/subtracting each value to the total until the subsequent symbol is determined to be larger/smaller.

Constraints:

- Use only simple indexing and/or iteration on the input string. Do not use any string class methods.
- Do not use lists in your solution.

Examples:

```
Enter a roman numeral: X
Decimal value is: 10

Enter a roman numeral: XIX
Decimal value is: 19

Enter a roman numeral: XII
Decimal value is: 12

Enter a roman numeral: MCCL
Decimal value is: 1250

Enter a roman numeral: XL
Decimal value is: 40

Enter a roman numeral: LX
Decimal value is: 60

Enter a roman numeral: MCDLVII
Decimal value is: 1457
```

E. **Sieve of Eratosthenes**

Although exhaustive enumeration is a simple way to identify all the prime numbers in some range, it is very inefficient. A much more efficient method is ascribed to the ancient Greek scholar Eratosthenes of Cyrene. The so-called *Sieve of Eratosthenes* is a prime filter that works like this: Starting with a list of all the integers from 2 through *n*, you begin by "crossing off" every value that is a multiple of the first prime value in the list, *p=2* (they're crossed off because multiples of a prime number are not prime).

$$2p=4, 3p=6, 4p=8, 5p=10, \ldots, x : x <= n$$

Repeat this process by setting the value of *p* to the *next* prime in the list, which happens to be the next element that hasn't already been "crossed off" (*p=3*) and proceed to cross off all the multiples of this prime:

$$2p=6, 3p=9, 4p=12, 5p=15, \ldots, x : x <= n$$

You continue until all the remaining elements > *p* are "crossed off". The primes will be those elements in the list that have not been eliminated.

Write a well-structured Python program that will input a positive integer $n > 1$ and implement *Eratosthenes' sieve* to output all of the prime numbers in the closed interval [2, *n*]. You should validate the input, but you do not need to provide a validation/correction loop (unless you want to). Do not use any imported functions or methods other than basic Python operations and built-in functions.

Hint: Start with a `list` of the integers 2 through *n* and "cross-off" the non-primes by overwriting them with some value such as zero.