1. **Total Points:**

   This assignment carries 150 points (50 points each for the first three questions). You are encouraged to attempt question 4, which holds 20 bonus points.

2. **Submission Policy:**

   For each group, one student should submit the assignment files on Canvas. Ensure your submission includes a README file detailing the names of all team members and their contributions to the project. Simpilar to your homework submission policy, you need to provide your main submission files (Haskell code) in one folder 'main' and all team members should have their works included in separate folders named according to their name 'Fi stNameLastName". Only the content of the 'main' folder will be graded. Other folders will be used to verify that all team members contributed as per the work distribution mentioned in the README file. You are required to submit a single zip file containing these folders, alongside the README file. Follow this link for guidance on creating a README file.

3. **Note to students:**

   As you delve into this Haskell assignment, be mindful to craft your solutions independently, without resorting to direct copying from online resources or using automated tools like GPTs. The unique nature of functional programming and Haskell encourages creative problem-solving and a deep understanding of the language's capabilities. Your original work will reflect your actual grasp of the material and adherence to academic integrity.

4. **Haskell instructions:**

   You will need to have Haskell installed on your computer to complete this assignment. Follow this link for Haskell installation instructions. While any text editor can be used to write Haskell code, we recommend using an IDE such as Visual Studio Code with the Haskell Syntax Highlighting extension for a more integrated development experience.

   You will be held accountable for any empty, incomplete, corrupted, or incorrect files submitted.

5. **Prohibition on AI Tools:**

   The use of AI tools, including GPT, for this assignment is strictly prohibited. Submissions will undergo a review process to ensure compliance with this rule.

1 Capitalize and Reverse a String

1.1 Problem Statement: (50pts)

Write a Haskell program with the following specifications:

1. Initialize a variable with the message "Hello World" and print this message to the console.**(5pts)**
2. Using the **same variable**, reverse all characters and convert them to uppercase (without utilizing any of Haskell's built-in string manipulation functions – this is crucial). The final output should be "DLROW OLLEH", printed to the console.**(35pts)**
3. Include comments explaining the functionality of each segment or significant line. This is crucial for demonstrating your understanding and thought process.**(10pts)**

Constraint: You must not use Haskell's built-in functions for reversing strings or converting characters to uppercase. This exercise aims to challenge your ability to manually implement these functionalities, thereby enhancing your grasp of Haskell's functional programming capabilities.

1.2 Expected Output

Your program should produce the following output in the console:

```
Hello World
DLROW OLLEH
```

Ensure that the transformation from "Hello World" to "DLROW OLLEH" is achieved using the same variable, demonstrating effective variable manipulation and string processing in Haskell.

1.3 Submission:

Place all functions and test examples in a single Haskell file named '`q1.hs`'.
Include comments within your code to explain each part clearly.
Create a separate text file named '`q1.txt`' to save the output of all test examples.

2 Recursive Function for Exponentiation

2.1 Problem Statement: (50pts)

You are tasked with implementing a recursive function for calculating exponentiation in Haskell. The function, named `exmp`, takes two arguments: a base (`x`) of type `Floating` and an exponent (`n`) of type `Integer`. The function computes the result of raising the base to the power of the exponent.

- Function Name: `exmp`
- Arguments:
  - `x`: Base of the exponentiation, a floating-point number (`Floating a`).
  - `n`: Exponent, an integer (`Integer`).Return Type: Floating-point number (`Floating a`).
- Base Case: When the exponent `n` is 0, the function returns 1.
- Set guard for error handling:
  - If the exponent `n` is negative, the function raises an error with the message "Exponent must be non-negative".
  - If the base `x` is 0, the function raises an error with the message "Base cannot be 0".
- Recursive Case: For any other positive exponent `n`, the function recursively computes the result by multiplying the base `x` with the result of raising the base to the power of (`n` - 1).

2.2 Test Example:

Test your implementation with the following examples to demonstrate correctness:

- `(exmp 2 3)`
- `(exmp 0.5 3)`
- `(exmp 5 0)`
- `(exmp (-1) 3)`
- Error test case: `(exmp 2 (-2))`
- Error test case: `(exmp 0 5)`

2.3 Output Format:

Print out the result for exponentiation calculation for the given base `x` and exponent `n`.

2.4 Submission:

Place all functions and test examples in a single Haskell file named '`q2.hs`'.
Include comments within your code to explain each part clearly.
Create a separate text file named '`q2.txt`' to save the output of all test examples.

3 Caesar Cipher

3.1 Problem Statement: (50pts)

You are required to implement a Haskell program that encodes and decodes text using a Caesar cipher, a type of substitution cipher where each letter in the plaintext is shifted a certain number of places down or up the alphabet.

3.2 Requirements:

- `caesarEncode` **Function**: Implement a function named `caesarEncode` that takes an integer (shift) and a string (plaintext) and returns the encoded text using the Caesar cipher.
- `caesarDecode` **Function**: Implement a function named `caesarDecode` that takes an integer (shift) and a string (encoded text) and returns the decoded text, reversing the Caesar cipher.

3.3 Main Function Tasks:

In the `main` function, execute the following without user input:

1. Use predefined integer shift values and plaintext strings for encoding.
2. Encode the plaintext using `caesarEncode`.
3. Print the encoded text.
4. Decode the text using `caesarDecode` to verify it matches the original plaintext.
5. Print the decoded text.

3.4 Test Example:

Provide a test within the `main` function using the shift of `3` and the plaintext "Haskell is fun!" and "Hello, World!" to demonstrate encoding and decoding. Ensure non-alphabetic characters remain unchanged.

3.5 Output Format:

For the provided test example, print:

- The original plaintext.
- The encoded text.
- The decoded text, verifying it matches the original plaintext.

3.6 Submission:

Provide all functions and test examples in a single Haskell file named '`q3.hs`'.
Include comments within your code to explain each part clearly.
Create a separate text file named '`q3.txt`' to save the output of all test examples.

4 Merge and Sort Lists (Bonus)

4.1 Problem Statement: (20pts)

You are required to implement a Haskell program that performs the following tasks:

- Implement a function `mergeSorted` that takes two lists of elements and merges them into a single sorted list in ascending order. The function should merge the lists while maintaining the sorted order. You must use recursion to implement this function.
- Implement a function `occurrenceNum` that counts the occurrences of a specified element in a merged and sorted list. The function should return the number of times the element appears in the list.
- In the main function `main`, you will perform the following steps:
    1. Provide two sets of unsorted lists as predefined input.

2. Sort and merge the given two unsorted lists using the `mergeSorted` function.
3. Print the original two unsorted lists, their lengths, the merged and sorted list, its length, and the number of occurrences of a specified element.
4. Test the program with at least two different sets of unsorted lists and count the occurrence of a specified element in the merged and sorted list.

4.2 Test Example:

Test your program with the following sets of unsorted lists:

- Lists: `[1, 1, 0.5, 7, 9]` and `[2, 14, 6, -8, 10]`. Count the occurrence of the number 0.
- Lists: `[0, 10, 0.6, 0, 59]` and `[-1, 2.4, 5, 0, -11, 1/4, 0.1]`. Count the occurrence of the number 1.

4.3 Output Format:

Print out the following information for each test example:

- Original unsorted lists and their lengths.
- Merged and sorted list along with its length.
- Number of occurrences of the specified element.

4.4 Submission:

Provide all functions and test examples in a single Haskell file named '`q4.hs`'.
Include comments within your code to explain each part clearly.
Create a separate text file named '`q4.txt`' to save the output of all test examples.