1. Recursive Fibonacci Sequence Generator

Imagine you are developing a program to generate Fibonacci numbers using a recursive approach, catering to a range of mathematical applications. Each Fibonacci number is derived by adding the two preceding numbers, starting from 0 and 1. Your function must efficiently compute Fibonacci numbers up to a specified index requested by the user, adhering to recursive principles for elegant mathematical modeling.

As you embark on this task, users rely on your program to provide accurate Fibonacci sequences for various analytical and computational purposes. The recursive nature of your function allows for a clear and concise representation of Fibonacci calculations, showcasing the inherent beauty of recursive algorithms in generating sequences.

With a focus on clarity and mathematical integrity, your function seamlessly navigates through recursive calls to compute Fibonacci numbers, ensuring precision and reliability in sequence generation. This capability empowers users to explore complex patterns, growth rates, and numerical series with intuitive recursive methodologies.

Store the generated fibonacci Sequence in an array.

Test Case 1:
Input: n = 0
Expected Output: Fibonacci sequence: []
Explanation: For n = 0, the Fibonacci sequence is empty.

Test Case 2:
Input: n = 1
Expected Output: Fibonacci sequence: [0]

Test Case 3:
Input: n = 2
Expected Output: Fibonacci sequence: [0, 1]

Test Case 4:
Input: n = 5
Expected Output: Fibonacci sequence: [0, 1, 1, 2, 3]

Test Case 5:
Input: n = 10
Expected Output: Fibonacci sequence: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

Test Case 6:
Input: n = 50
Expected Output: Fibonacci sequence: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, 832040, 1346269, 2178309, 3524578, 5702887, 9227465, 14930352, 24157817, 39088169, 63245986, 102334155, 165580141, 267914296, 433494437, 701408733, 1134903170, 1836311903, 2971215073, 4807526976, 7778742049]

Test Case 7:
Input: n = -1

Test Case 8:
Input: n = 1000
Expected Output: Fibonacci sequence: [0, 1, ...] (up to the 1000th Fibonacci number)

Test Case 9:
Input: n = 10000
Expected Output: Fibonacci sequence: [0, 1, ...] (up to the 10000th Fibonacci number)

Test Case 10:
Input: n = 2.5 (floating point or non-integer input)

2. Palindrome Checker

You're developing a robust text processing tool designed to identify palindromes from user-input strings. This function needs to accurately detect sequences that read the same forward and backward, ignoring spaces, punctuation, and character case. Users rely on this tool for various applications, including linguistic analysis, content validation, and data processing tasks where identifying palindromes is essential.

As you implement the palindrome checker, your goal is to ensure it handles diverse inputs seamlessly, such as phrases, sentences, or alphanumeric combinations with special characters. The function's efficiency and accuracy are crucial, supporting quick validation of potential palindromes across different language sets and textual formats.

With a focus on reliability and performance, your function navigates through each input string, applying logical checks to determine palindrome properties without compromising computational efficiency. This capability enhances user experience by providing swift and precise identification of palindromic patterns, contributing to streamlined text analysis and quality assessment processes.

Test Case 1:
Input: "radar"
Expected Output: True
Explanation: "radar" reads the same forwards and backwards.

Test Case 2:
Input: "level"
Expected Output: True

Test Case 3:
Input: "madam"
Expected Output: True

Test Case 4:
Input: "A man, a plan, a canal, Panama!"
Expected Output: True

Test Case 5:
Input: "Was it a car or a cat I saw?"

Expected Output: True

Test Case 6:
Input: "hello"
Expected Output: False

Test Case 7:
Input: "12321"
Expected Output: True

Test Case 8:
Input: ""
Expected Output: True

Test Case 9:
Input: "a"
Expected Output: True

Test Case 10:
Input: "racecar!"
Expected Output: True

Test Case 11:
Input: "!@$%^&()(&^%$#@!"
Expected Output: True

Test Case 12:
Input: "123!321"
Expected Output: True

Test Case 13:
Input: "abcdefghijklmnopqrstuvwxyz" (long non-palindromic string)
Expected Output: False

Test Case 14:
Input: "a" repeated 10,000 times (long palindrome)
Expected Output: True

3. Prime Number Checker

You're developing a critical component for a cryptography application aimed at verifying the primality of numbers to ensure the security and integrity of generated cryptographic keys. The function you're tasked with must accurately determine whether a given number is prime, with a focus on efficiency even for large numbers crucial for cryptographic operations.

As you implement the prime number checker, users rely on this function to validate the primality of potential key components and ensure robust encryption protocols. The function's reliability in identifying prime numbers is paramount to maintaining the confidentiality and integrity of sensitive data in cryptographic communications.

With a rigorous approach to computational efficiency, your function systematically evaluates each number, applying mathematical checks to confirm prime properties efficiently. This capability supports rapid validation of prime numbers, essential for timely key generation and secure cryptographic operations.
Basic Prime Numbers:

Test Case 1:
Input: 2
Expected Output: True

Test Case 2:
Input: 3
Expected Output: True

Test Case 3:
Input: 5
Expected Output: True
Explanation: 5 is a prime number
Composite Numbers:

Test Case 4:
Input: 1
Expected Output: False

Test Case 5:
Input: 4
Expected Output: False

Test Case 6:
Input: 15
Expected Output: False

Test Case 7:
Input: 997
Expected Output: True

Test Case 8:
Input: 1009
Expected Output: True

Test Case 9:
Input: 0
Expected Output: False

Test Case 10:
Input: -7
Expected Output: False

Test Case 11:
Input: 2147483647
Expected Output: True

Test Case 12:
Input: 97.0 (floating point input)
Expected Output: Non-integer input

Test Case 13:
Input: "23" (string input)
Expected Output: Non-numeric input

4. Anagram Detector

In your new word game venture, players are challenged to unravel anagrams swiftly and accurately. As the developer behind the scenes, your task is to create a function that can seamlessly detect whether two words are anagrams, disregarding differences in letter case and spaces. This capability is essential for ensuring a challenging and engaging gameplay experience where players must demonstrate their linguistic prowess.

Imagine players eagerly typing in their guesses, each word appearing on the screen awaiting confirmation. Your function plays a crucial role in instantly verifying if their guesses match the scrambled words provided. This real-time validation enhances the game's interactivity, allowing players to progress through levels confidently as they master the art of anagram solving.

With a focus on efficiency and accuracy, your function meticulously compares the letter compositions of the entered words, ensuring that even complex anagrams are swiftly identified. This functionality not only supports seamless gameplay but also fosters a competitive spirit among players striving to decode words within the shortest possible time.

Test Case 1:
Inputs: "listen", "silent"
Expected Output: True

Test Case 2:
Inputs: "evil", "vile"
Expected Output: True

Test Case 3:
Inputs: "restful", "fluster"
Expected Output: True
Explanation: "restful" and "fluster" are anagrams of each other.

Test Case 4:
Inputs: "Hello", "oLLeH"
Expected Output: True

Test Case 5:
Inputs: "Racecar", "racecar"
Expected Output: True

Test Case 6:
Inputs: "debit card", "bad credit"
Expected Output: True

Test Case 7:
Inputs: "astronomer", "moon starer"
Expected Output: True

Test Case 8:
Inputs: "hello", "world"
Expected Output: False

Test Case 9:
Inputs: "listen", "silentt"
Expected Output: False

Test Case 10:
Inputs: "", ""
Expected Output: True

Test Case 11:
Inputs: "a", "A"
Expected Output: True

Test Case 12:
Inputs: "apple!", "leppa"
Expected Output: False

Test Case 13:
Inputs: "night", "thing!"
Expected Output: False

5. Recursive Factorial Calculator

In a scientific computing project, you need to calculate factorial values for various mathematical computations. The function should handle both small and large integers efficiently.

Test Case 1:
Input: 5
Output: 120 (5! = 5 * 4 * 3 * 2 * 1)

Test Case 2:
Input: 3
Output: 6 (3! = 3 * 2 * 1)

Test Case 3:
Input: 10
Output: 3628800 (10! = 3628800)

Test Case 4:
Input: 1
Output: 1

Test Case 5:
Input: -5
Output:
Invalid Input

Test Case 6:
Input: 0
Output: 1

6. Power Function

You are developing a scientific calculator application that requires computing powers of numbers. The function should accurately handle both positive and negative exponents.

As part of the development team, you are tasked with implementing and testing the power function of the scientific calculator application. This function should accept a base and an exponent as inputs and compute the result accurately, supporting both positive and negative exponents.

Test Case 1:
Inputs: (2, 5)
Output: 32 (2^5 = 32)

Test Case 2:
Inputs: (2, -3)
Output: 0.125 (2^(-3) = 1 / (2^3) = 1 / 8 = 0.125)

Test Case 3:
Inputs: (5, 0)
Output: 1 (5^0 = 1)

Test Case 4:
Inputs: (-2, 4)
Output: 16 ((-2)^4 = 16)

Test Case 5:
Inputs: (1.5, 2)
Output: 2.25 (1.5^2 = 2.25)

Test Case 6:
Inputs: (2.5, 1.5)
Output: 3.95 (2.5^1.5 = sqrt(2.5) * 2.5 ≈ 3.95)

Test Case 7:
Inputs: (0,)
Output: Undefined

7. String Compression

Your data compression algorithm needs a function to compress repetitive sequences of characters in textual data. The function should reduce storage requirements without loss of information.

Test Case 1:
Input: "aabbbccc"
Output: "a2b3c3"

Test Case 2:
Input: "abcd"
Output: "abcd" (no compression needed)

Test Case 3:
Input: "aaaaa"
Output: "a5"

Test Case 4:
Input: "x"
Output: "x1"

Test Case 5:(Empty String)
Input: " "
Output:
Invalid Input

Test Case 6:
Input: "AAAABBBBCCCC "
Output:
A4B4C4

Test Case 7:
Input: "aAaBB1111 "
Output:
a2A1B2_14

8. Matrix Transposition

A computational geometry library is being developed for analyzing and manipulating geometric data represented by matrices. One critical functionality needed is a matrix transposition operation, which is essential for transforming and analyzing geometric transformations efficiently.

Test Case 1:
Input: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Output: [[1, 4, 7], [2, 5, 8], [3, 6, 9]]

Test Case 2:
Input: [[1, 2], [3, 4]]
Output: [[1, 3], [2, 4]]

Test Case 3:
Input: [[1, 2], [3, 4], [5, 6]]
Output: [[1, 3, 5], [2, 4, 6]]

Test Case 4:
Input: [[1, 2, 3]]
Output: [[1], [2], [3]]

Test Case 5:
Input: [ ]
Output: [ ]

Test Case 6:
Input: [[1, 'a'], [2, 'b'], [3, 'c']]
Output: [[1, 2, 3], ['a', 'b', 'c']]

Test Case 7:
Input: [[1], [2], [3]]
Output: [[1, 2, 3]]

9.You are tasked with developing a text analysis tool that counts the frequency of each unique word in a given English document. The document may contain punctuation marks, which should be ignored while counting words. This tool is essential for various

applications, such as search engine optimization, content analysis, and data mining, where understanding word frequency is critical.

Formula:
Word Frequency with Text Preprocessing (f(document)):
f(document) = { <word, count(word)> | word ∈ Unique_Words(Preprocess(document)) }

Explanation:

- f(document): This represents the function that takes a document as input and returns a dictionary (or set of key-value pairs) where keys are unique words and values are their frequencies.
- {}: This curly brace notation defines a set or dictionary.
- <word, count(word)>: This represents a key-value pair within the dictionary.
    - word: This is the unique word.
    - count(word): This is the frequency count of the word within the document.
- ∈: This symbol represents "is an element of."
- Unique_Words(Preprocess(document)): This function call represents the combined logic of text preprocessing (removing punctuation, converting to lowercase, etc.) and then identifying unique words from the processed text.

Constraints:
The length of the input string text will be between 1 and 1000 characters.
Words are separated by spaces.
Punctuation marks (e.g., ,, ., !, ?) and other non-alphabetic characters should be removed or ignored during the counting process.
Case sensitivity is to be considered, meaning 'Word' and 'word' are treated as different words.

Test Case 1:
Input: "hello world hello universe world"
Output: {'hello': 2, 'world': 2, 'universe': 1}
Explanation: The word 'hello' appears twice, 'world' appears twice, and 'universe' appears once.

Test Case 2:
Input: "a a a b b c"
Output: {'a': 3, 'b': 2, 'c': 1}
Explanation: The word 'a' appears three times, 'b' appears twice, and 'c' appears once.

Test Case 3:
Input: "To be or not to be, that is the question."
Output: {'to': 2, 'be': 2, 'or': 1, 'not': 1, 'that': 1, 'is': 1, 'the': 1, 'question': 1}
Explanation: The word 'to' appears twice, 'be' appears twice, and each other word appears once.

Test Case 4:
Input: "Hello, world! Hello? World."
Output: {'hello': 2, 'world': 2}
Explanation: Punctuation is ignored; 'hello' and 'world' each appear twice.

Test Case 5:
Input: "The quick brown fox jumps over the lazy dog."
Output: {'the': 2, 'quick': 1, 'brown': 1, 'fox': 1, 'jumps': 1, 'over': 1, 'lazy': 1, 'dog': 1}
Explanation: 'the' appears twice, and each other word appears once.

Test Case 6:
Input: "Python is fun. Python is cool."
Output: {'python': 2, 'is': 2, 'fun': 1, 'cool': 1}
Explanation: 'python' appears twice, 'is' appears twice, and 'fun' and 'cool' each appear once.

Test Case 7:
Input: "Good morning! Good morning everyone."
Output: {'good': 2, 'morning': 2, 'everyone': 1}
Explanation: 'good' and 'morning' each appear twice, and 'everyone' appears once.

Test Case 8:
Input: "I am learning to code. Coding is fun."

Output: {'i': 1, 'am': 1, 'learning': 1, 'to': 1, 'code': 1, 'coding': 1, 'is': 1, 'fun': 1}
Explanation: Each word appears once.

Test Case 9:
Input: "This is a test. This test is only a test."
Output: {'this': 2, 'is': 2, 'a': 2, 'test': 3, 'only': 1}
Explanation: 'test' appears three times, and 'this', 'is', and 'a' each appear twice.

Test Case 10:
Input: "Data science, data analytics, data visualization."
Output: {'data': 3, 'science': 1, 'analytics': 1, 'visualization': 1}
Explanation: 'data' appears three times, and each of the other words appears once.

10. In your data analytics application, you need to implement a sorting algorithm capable of efficiently handling and sorting large datasets. This functionality is crucial for various analytical tasks, such as data cleaning, aggregation, and statistical analysis. Your function should be able to sort data of different types and sizes while ensuring correctness and efficiency.

Selection Function (Select-Min)
Select-Min(Data, i) = min(data[i], data[i+1], ..., data[len(Data)-1])

Explanation:

- Select-Min(Data, i): This function takes a data array Data and an index i as input.
- min: This function finds the minimum element within a specific range.
- data[i]: The element at index i.
- data[i+1], ..., data[len(Data)-1]: This represents the remaining elements in the data array starting from index i+1 (inclusive) to the last element.

Constraints:

- The length of the input array can be between 1 and 10^5
- The elements in the array can be integers, floating-point numbers, or strings.

Test Case 1
Input: [5, 2, 9, 1, 5]
Output: [1, 2, 5, 5, 9]
Explanation: This test case involves a basic list of integers. The sorting algorithm correctly arranges the integers in ascending order. The repeated integer 5 is handled properly, ensuring all instances are included in the sorted output.

Test Case 2
Input: [3.1, 2.4, 5.6, 1.9]
Output: [1.9, 2.4, 3.1, 5.6]
Explanation: This test case contains floating-point numbers. The sorting algorithm arranges these numbers in ascending order, ensuring that each floating-point value is correctly positioned according to its magnitude.

Test Case 3
Input: ["apple", "banana", "Cherry", "date"]
Output: ["Cherry", "apple", "banana", "date"]
Explanation: The input consists of strings. The sorting is case-sensitive, meaning uppercase letters come before lowercase letters in lexicographical order. Thus, "Cherry" appears before "apple", and the rest are sorted accordingly.

Test Case 4
Input: [100, 200, 150, 120, 180, 190]
Output: [100, 120, 150, 180, 190, 200]
Explanation: This test case uses a larger dataset of integers. The algorithm sorts the integers in ascending order, correctly handling the wider range of values.

Test Case 5
Input: [3.14, 2.71, 1.41, 1.73, 0.577]
Output: [0.577, 1.41, 1.73, 2.71, 3.14]
Explanation: This test case involves a larger dataset of floating-point numbers. The numbers are sorted in ascending order based on their numeric value, demonstrating the algorithm’s ability to handle multiple decimal values accurately.

Test Case 6
Input: ["zebra", "apple", "Mango", "banana"]

Output: ["Mango", "apple", "banana", "zebra"]
Explanation: Here, the dataset includes strings with mixed case sensitivity. The sorting is case-sensitive, so "Mango" (uppercase M) comes before "apple" (lowercase a), and the rest are ordered accordingly.

Test Case 7
Input: [7]
Output: [7]
Explanation: This test case contains a single-element array. The algorithm should return the array unchanged, as a single element is trivially sorted.

Test Case 8
Input: []
Output: []
Explanation: The input is an empty array. The output is also an empty array, demonstrating that the sorting algorithm correctly handles the absence of elements.

Test Case 9
Input: [9, 8, 7, 6, 5, 4, 3, 2, 1]
Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Explanation: This input is an array sorted in descending order. The sorting algorithm reverses the order to produce an ascending sequence, demonstrating its ability to handle arrays in reverse order.

Test Case 10
Input: [1, 3.5, "apple", "banana", 2, 3.0]
Output: [1, 2, 3.0, 3.5, 'apple', 'banana']
Explanation: This test case features a mix of integers, floating-point numbers, and strings. The algorithm first sorts by type (integers, then floating-point numbers, and then strings), and within each type, it sorts the elements in ascending order.

11. You are developing a computational tool for handling large datasets in scientific simulations. The tool must efficiently perform matrix multiplication to solve equations that arise in simulations, such as systems of linear equations or transformations in multidimensional space. The matrices involved can vary in size and may require validation to ensure they meet the criteria for multiplication. The output must be computed efficiently to handle potentially large data volumes.

Constraints:

- The length of the input array can be between 1 and 10^5
- The elements in the array can be integers and float

Matrix Multiplication (C = A * B):
C[i, j] = Σ (A[i, k] * B[k, j])

Explanation:

- C[i, j]: This represents the element at row i and column j of the resulting product matrix C.
- Σ: This symbol represents summation.
- A[i, k]: This represents the element at row i and column k of the first matrix A.
- B[k, j]: This represents the element at row k and column j of the second matrix B.
- k: This is the loop index iterating from 1 to m, where m is the number of columns in A (which must also equal the number of rows in B) for valid multiplication.
- *: This symbol represents element-wise multiplication between corresponding elements from A and B.

Test Case 1
Input: [[1, 2], [3, 4]], [[5, 6], [7, 8]]
Output: [[19, 22], [43, 50]]
Explanation: Multiply a 2x2 matrix with another 2x2 matrix. The resulting matrix is also 2x2.

Test Case 2
Input: [[1, 2, 3], [4, 5, 6]], [[1, 4], [2, 5], [3, 6]]
Output: [[14, 32], [32, 77]]
Explanation: Multiply a 2x3 matrix with a 3x2 matrix. The resulting matrix is 2x2.

Test Case 3
Input: [[1, 2], [3, 4], [5, 6]], [[1, 1, 1], [1, 1, 1]]
Output: Error (Mismatched dimensions for matrix multiplication)
Explanation: Attempting to multiply a 3x2 matrix with a 2x3 matrix. This is a valid multiplication, so this test case should actually output [[3, 3, 3], [7, 7, 7], [11, 11, 11]]. The provided output reflects a misunderstanding of valid dimensions.

Test Case 4
Input: [[2, 0], [1, 3]], [[4, 1], [2, 5]]
Output: [[8, 2], [10, 16]]
Explanation: Multiply a 2x2 matrix with another 2x2 matrix. The resulting matrix is 2x2.

Test Case 5
Input: [[1, 2, 3], [4, 5, 6]], [[7, 8], [9, 10], [11, 12]]
Output: [[58, 64], [139, 154]]
Explanation: Multiply a 2x3 matrix with a 3x2 matrix. The resulting matrix is 2x2.

Test Case 6
Input: [[2, 3], [5, 7]], [[1, 2], [4, 6]]
Output: [[14, 22], [35, 53]]
Explanation: Multiply a 2x2 matrix with another 2x2 matrix. The resulting matrix is 2x2.

Test Case 7
Input: [[1, 0], [0, 1]], [[5, 7], [6, 8]]
Output: [[5, 7], [6, 8]]
Explanation: Multiply an identity matrix with another 2x2 matrix. The result is the same as the second matrix.

Test Case 8
Input: [[3, 2, 1], [6, 5, 4]], [[7, 8], [9, 10], [11, 12]]
Output: [[40, 46], [114, 134]]
Explanation: Multiply a 2x3 matrix with a 3x2 matrix. The resulting matrix is 2x2.

Test Case 9
Input: [[1]], [[1]]

Output: [[1]]
Explanation: Multiply a 1x1 matrix with another 1x1 matrix. The resulting matrix is 1x1.

Test Case 10
Input: [[1, 2], [3, 4], [5, 6]], [[7, 10, 11], [8, 12, 13]]
Output: [[23, 34, 39], [53, 78, 87], [83, 122, 135]]
Explanation: Multiply a 3x2 matrix with a 2x3 matrix. The resulting matrix is 3x3.

12. You are developing a text manipulation tool that needs to reverse the order of characters in a string provided by the user. The tool should handle various types of input, including sentences with punctuation, numeric strings, and strings with special Unicode characters. Your task is to implement a function that takes an input string and returns the string with its characters in reverse order. Ensure that the function can handle edge cases such as empty strings, single-character strings, and strings containing spaces or special characters**.**

Constraints:

1. The input string can contain alphanumeric characters, punctuation, and whitespace.
2. The maximum length of the input string is 1000 characters.
3. The input string can be empty.
4. The function should maintain the same case for each character.
5. The function should handle and correctly reverse Unicode characters.

f(string) = Σ (string[(len(string) - 1 - i) % len(string)])

Explanation:
Σ: This symbol represents summation.
string[(len(string) - 1 - i) % len(string)]: This expression retrieves the character at a specific index calculated as follows:

len(string) - 1: Represents the index of the last character.
- i: Subtracts the current loop index i to move from the end of the string towards the beginning.

% len(string): This modulo operation ensures the resulting index stays within the valid range (0 to len(string)-1) even when i becomes larger than the string length. This handles wrapping around to the beginning of the string.

i: This is the loop index iterating from 0 to len(string) - 1.

Test Case 1:
Input: "hello"
Expected Output: "olleh"
Explanation: Simple reversal of a single word.

Test Case 2:
Input: "hello world"
Expected Output: "dlrow olleh"
Explanation: Reversal of a sentence with spaces.

Test Case 3:
Input: "12345"
Expected Output: "54321"
Explanation: Reversal of a string with numeric characters.

Test Case 4:
Input: "a"
Expected Output: "a"
Explanation: Reversal of a single character string.

Test Case 5:
Input: "A man, a plan, a canal, Panama!"
Expected Output: "!amanaP ,lanac a ,nalp a ,nam A"
Explanation: Reversal of a string with punctuation and mixed case.

Test Case 6:
Input: "The quick brown fox jumps over the lazy dog"

Expected Output: "god yzal eht revo spmuj xof nworb kciuq ehT"
Explanation: Reversal of a longer sentence with multiple words.

Test Case 7:
Input: " "
Expected Output: " "
Explanation: Reversal of a single space.

Test Case 8:
Input: ""
Expected Output: ""
Explanation: Reversal of an empty string.

Test Case 9:
Input: "racecar"
Expected Output: "racecar"
Explanation: Reversal of a palindrome, which remains the same.

Test Case 10:
Input: "Python3.8!"
Expected Output: "!8.3nohtyP"
Explanation: Reversal of a string with alphanumeric characters and punctuation.

13. Calculating Building Dimensions

You are an architect working on the design of a new office building. As part of your design process, you need to calculate the dimensions of various structural elements, such as the size of the foundation, the height of the floors, and the thickness of the walls.
One of the key calculations you need to perform is finding the square root of the total floor area of the building. This will help you determine the appropriate size and layout of the foundation.

Write a program that includes a function **"findSquareRoot()"** that takes a float value representing the total floor area and returns the square root of that value as a float. Use

this function in your program to calculate the square root of the floor area and display the result.
Assume that the user will input the total floor area of the building when prompted.

Test Case 1:
Input: 100
Output: The square root of 100.00 is 10.00

Test Case 2:
Input: 225.0
Output: The square root of 225.00 is 15.00

Test Case 3:
Input: -16.0
Output: Error: Cannot calculate the square root of a negative number.

14. Analyzing Text Documents

You are working on a project that involves analyzing text documents to extract various statistics, such as the number of words, the number of unique words, and the length of the longest word. As a first step, you need to write a program that can determine the length of a given string.

Your program should include a function called **"findStringLength()"** that takes a string as input and returns the length of the string as an integer. You will then use this function in your main program to prompt the user to enter a string, call the findStringLength() function, and display the length of the string.
This functionality will be a crucial component of your larger text analysis project, as it will allow you to quickly and accurately determine the length of words, sentences, and entire documents.

Test Case 1:
Input: ""
Output: The length of the string is: 0

Test Case 2:
Input: " Hello, world! "

Output: The length of the string is: 17

Test Case 3:
Input: "Hello, world!"
Output: The length of the string is: 13

15. Calculating Sums for Budgeting

You are working on a budgeting application that helps users keep track of their expenses. One of the features you want to implement is the ability to calculate the sum of all even or odd expenses within a given range.
Your program should include a recursive function called **"sumEvenOdd()"** that takes the starting number, ending number, and a flag to indicate whether to sum even or odd numbers. The function should return the sum of all even or odd numbers in the given range.

In the main program, you will prompt the user to enter the starting and ending numbers of the range, as well as a choice to sum either even or odd numbers. You will then call the "sumEvenOdd()" function with the user-provided inputs and display the result.
This functionality will be useful for users who want to analyze their spending patterns and identify areas where they can cut back on expenses.

Test Case 1:
Input:
Start = 2, End = 10, Choice = 1
Output:
The sum of even numbers in the range 2 to 10 is: 30

Test Case 2:
Input:
Start = 3, End = 11, Choice = 0
Output:
The sum of odd numbers in the range 3 to 11 is: 36

Test Case 3:
Input:
Start = -4, End = -2, Choice = 1
Output:
The sum of even numbers in the range -4 to -2 is: -6

16. GCD (Greatest Common Divisor) Calculator

You are developing a utility for a mathematics app aimed at students and educators. This app includes various mathematical tools, one of which is a function to compute the greatest common divisor (GCD) of two integers. The GCD is essential for simplifying fractions, finding common factors, and solving problems related to number theory.

Your task is to implement a function that calculates the GCD of two numbers using recursion. This function will help users simplify fractions and understand the relationship between numbers more effectively.

The app will provide a user-friendly interface where users can input two integers, and the app will return their GCD. The recursive approach will demonstrate the elegance of mathematical algorithms and provide an opportunity for users to learn about recursion.

Constraints:
Input Type: The program expects integer values. Non-numeric inputs would lead to undefined behavior.

Input Range: The program doesn't explicitly check for valid input ranges. Extremely large or small values might cause overflow or underflow errors depending on the data type used (e.g., integer limitations of a 32-bit system).
Output Range: The output represents the result of the specified operation (assumed to be addition based on the test cases). It should be within the representable range of the chosen integer data type (same as point 2).
Operation: The prompt doesn't explicitly specify the operation. If it's addition as suggested by the test cases, no specific constraints apply beyond the input and output range considerations.

1. **Input Constraints:**
   - Let a and b be the two integers for which the GCD is to be computed.
   - The inputs a and b are integers:
   $$a, b \in \mathbb{Z}$$
   - The function expects valid integer inputs. Non-numeric inputs lead to undefined behavior.
2. **Input Range:**
   - There are no explicit checks for valid input ranges.
   - The input values a and b can be any integers, including very large or very small values.
   - For practical implementation, assume a and b are within the limits of the integer data type used. For example, for a 32-bit signed integer:
   $$-2,147,483,648 \leq a, b \leq 2,147,483,647$$
   - Negative values are acceptable but are usually converted to their absolute values during GCD calculation.
3. **Output Constraints:**
   - The output, which is the GCD of a and b, is always a non-negative integer.
   - The GCD of any two integers a and b is defined as:
   $$\text{GCD}(a, b) \geq 0$$
   - If both a and b are zero, the GCD is typically defined as 0.
4. **Operation Constraints:**
   - The function calculates the GCD using recursion, which is a standard mathematical approach.
   - For non-negative integers, the GCD satisfies:
   $$\text{GCD}(a, b) = \text{GCD}(|a|, |b|)$$
   - This is because the GCD function is symmetric with respect to the absolute values of its arguments.

Test Case 1:
Input: 48, 18
Output: 6

Test Case 2:

Input: 101, 103
Output: 1

Test Case 3:
Input: 56, 98
Output: 14

Test Case 4:
Input: 0, 15
Output: 15

Test Case 5:
Input: 0, 0
Output: 0

Test Case 6:
Input: -48, -18
Output: 6

Test Case 7:
Input: 48, -18
Output: 6

Test Case 8:
Input: -56, 98
Output: 14

Test Case 9:
Input: 123456, 789012
Output: 12

Test Case 10:
Input: 42, 42
Output: 42

17. Binary to Decimal Conversion

You are developing a digital logic simulation tool for an electronics engineering course. This tool is designed to assist students in learning about binary arithmetic, digital circuits, and number systems. One of the key features of this tool is the ability to convert binary numbers, represented as strings, into their decimal equivalents. This conversion is essential for students to verify their calculations and understand the relationship between binary and decimal representations of numbers.

Your task is to implement a function that takes a binary number as a string input and returns its decimal equivalent. This will help students check their work and reinforce their understanding of binary numbers in digital logic design.

Constraints:

Binary Input Only: The program expects strings containing only "0" and "1" characters. Other characters will lead to undefined behavior (e.g., Test Case 8).

Valid Length: The program doesn't explicitly check for input length. Extremely long strings (like Test Case 6) might overflow integer limitations depending on the implementation (e.g., 32-bit integer overflow at 2^32 - 1).

Leading Zeros: Leading zeros are allowed (e.g., Test Cases 1, 5, 9) and contribute to the final value.

Output Range: The output represents the decimal equivalent of the binary string. It should be within the representable range of the chosen integer data type (e.g., -2,147,483,648 to 2,147,483,647 for a 32-bit signed integer). Excessively large inputs (like Test Case 6) might cause overflow errors.

1. **Input Constraints:**
   - Let b be the binary string.
   - The string b consists of only '0' and '1' characters:

$$b \in \{'0', '1'\}^*$$

   - Any other characters in the string (such as '2') result in undefined behavior. Thus, if b contains any characters not in { '0', '1' }, the output should be -1 or an error.

2. **Length Constraints:**
   - Let $|b|$ denote the length of the binary string b.
   - The length of b is not explicitly limited but should be within the constraints of the integer data type used for output. Extremely long binary strings might exceed the limits of standard integer representations.

3. **Output Constraints:**
   - The decimal equivalent of the binary string should be within the range of the integer data type used:
     - For a 32-bit unsigned integer, the maximum value is $2^{32} - 1$.
     - For a 32-bit signed integer, the range is from $-2,147,483,648$ to $2,147,483,647$.
   - The result d (decimal equivalent) should satisfy:

$$0 \leq d \leq 2^{|b|} - 1$$

     if b is considered as a non-negative binary string.
   - If the length of b exceeds the maximum representable value for the chosen integer type, it might cause overflow errors.

Test Case 1:
Input: "1010"
Output: 10

Test Case 2:
Input: "1111"
Output: 15

Test Case 3:

Input: "100101"
Output: 37

Test Case 4:
Input: "00001010"
Output: 10

Test Case 5:
Input: "0"
Output: 0

Test Case 6:
Input: "11111111111111111111111111111111"
Output: 4294967295 (which is 2^32 - 1)

Test Case 7:
Input: "1"
Output: 1

Test Case 8:
Input: "102"
Output: -1 (or an error message)

Test Case 9:
Input: "0000"
Output: 0

Test Case 10:
Input: "110110101011"
Output: 3515

18. Convert Lowercase into Uppercase using Function

You are developing a text processing application for a publishing company that needs to standardize the formatting of titles and headings in their documents. The company has a

large dataset of text entries, and they want to ensure that all titles are consistently formatted in uppercase letters for better visibility and uniformity.
Your task is to write a C program that takes user input and converts any lowercase letters entered into uppercase. This conversion will help maintain consistency across all titles and headings, making it easier for editors and designers to work with the text.
To achieve this, you will implement a function that specifically handles the conversion of lowercase letters to uppercase. This function will be reusable throughout the application, ensuring clarity and maintainability in the code.

Constraints:
Input length: Limited by chosen string array size (e.g., 100 chars).
Characters: Handles lowercase (converted), uppercase (unchanged), numbers/symbols (unchanged), whitespace (unchanged).
Newlines: Trailing newlines removed (affects output formatting).
Special chars: Not handled specifically (may require additional logic).

1. **Input Constraints:**
   - Let s be the input string.
   - The length of the input string $|s|$ is constrained by the size of the string array, which is denoted as N.

     $$|s| \leq N$$

     where N is the maximum allowable length of the string, e.g., $N = 100$ characters.
   - The string s consists of:
     - Lowercase English letters (a to z)
     - Uppercase English letters (A to Z)
     - Digits (0 to 9)
     - Special symbols and whitespace characters
2. **Character Constraints:**
   - Lowercase letters a to z in s should be converted to uppercase letters A to Z.
   - Uppercase letters, digits, symbols, and whitespace should remain unchanged.
3. **Output Constraints:**
   - The output string should have the same length as the input string:

     $$|s| = |s_{\text{output}}|$$
   - The output string s_{output} will have all lowercase letters converted to uppercase, while other characters remain unchanged.
4. **Additional Constraints:**
   - Newlines at the end of the input string should be removed before processing (affects output formatting).
   - Special characters and whitespace should not be altered but preserved in their original positions.

Test Case1:
Input: hello
Output: HELLO

Test Case 2:

Input: bit
Output: BIT

Test Case 3:
Input: bannari
Output: BANNARI

Test Case 4:
Input: HelloWorld
Output: HELLOWORLD

Test Case 5:
Input: abc123!@#
Output: ABC123!@#

Test Case 6:
Input: UPPERCASE
Output: UPPERCASE

Test Case 7:
Input: hello world
Output: HELLO WORLD

Test Case 8:
Input: test input
Output: TEST INPUT

Test Case 9:
Input:
Output:

Test Case 10:
Input: cOnVeRt ThiS
Output: CONVERT THIS

19. Leap Year Check using Function

You are working on a data analysis project for a research institute that studies climate change. The institute has a large dataset of historical weather data, and they want to analyze the relationship between weather patterns and the occurrence of leap years.
Your task is to write a function that takes a year as input and determines whether it is a leap year or not based on the following criteria:
A year is a leap year if it is divisible by 4.
However, if the year is divisible by 100, it is not a leap year unless it is also divisible by 400.

Constraints:
The function should efficiently determine whether a given year is a leap year or not.
The function should handle a wide range of input years, including years in the past, present, and future.
The function should be part of a larger data analysis pipeline that processes the weather data and identifies patterns related to leap years.

1. **Input Constraints:**
   - Let y be the input year.
   - The value of y can be any integer, representing the year to be checked.
   - There are no explicit bounds on the year y in terms of minimum or maximum value, so it can include both historical and future years.
2. **Leap Year Conditions:**
   - A year y is a leap year if:

     $$(y \mod 4 = 0) \land (y \mod 100 \neq 0) \lor (y \mod 400 = 0)$$

     where:
     - \mod represents the modulo operation.
     - $\land$ denotes logical AND.
     - $\lor$ denotes logical OR.
   - Specifically:
     - **Condition 1:** The year must be divisible by 4.
     - **Condition 2:** If the year is divisible by 100, it must also be divisible by 400 to be a leap year.
3. **Output Constraints:**
   - The function should output one of the following:
     - "Leap Year" if the year meets the leap year conditions.
     - "Non-Leap Year" if the year does not meet the leap year conditions.
4. **Efficiency Constraints:**
   - The function should be able to determine whether the year is a leap year or not in constant time, i.e., $O(1)$, since the operation involves simple arithmetic and logical checks.

By solving this problem, you will be able to provide the institute with a valuable tool for their data analysis efforts, which could ultimately help them better understand the relationship between weather patterns and the occurrence of leap years.

Test Case 1:
Input: 2016
Output: Leap Year

Test Case 2:
Input: 1900
Output: Non-Leap Year

Test Case 3:
Input: 2000
Output: Leap Year

Test Case 4:
Input: 2023
Output: Non-Leap Year

Test Case 5:
Input: 1997
Output: Non-Leap Year

Test Case 6:
Input: 2100
Output: Non-Leap Year

Test Case 7:
Input: 2001
Output: Non-Leap Year

Test Case 8:
Input: 2099
Output: Non-Leap Year

Test Case 9:
Input: 2001
Output: Non-Leap Year

Test Case 10:
Input: 2000
Output: Leap Year

20. Remove Duplicates in a given array using Function

You are working on a data processing project for a software company that needs to analyze large datasets. The company has a dataset that contains arrays of integers, and they want to remove any duplicate elements from these arrays while maintaining the order of the remaining elements.
Your task is to write a function that takes an array of integers as input, removes any duplicate elements, and returns the new size of the array after removing the duplicates. This function will be used as part of a larger data processing pipeline to clean and prepare the data for further analysis.

Constraints:
The function should efficiently remove duplicates from the input array.
The function should maintain the order of the remaining elements in the array.
The function should return the new size of the array after removing duplicates.

1. **Input Constraints:**
   - Let A be the input array with elements $A[i]$ where $i \in \{0, 1, 2, \ldots, n-1\}$, and n is the length of A.
   - The length of the input array $|A|$ is constrained as follows:

$$0 \leq |A| \leq 100{,}000$$

   - Each element $A[i]$ in the array is an integer. The constraints on the value of the integers are not specified, so they can be any integer value.
2. **Output Constraints:**
   - The function should return a new array A' with duplicates removed, maintaining the original order of the elements.
   - The length of the resulting array $|A'|$ should be returned as the new size of the array.
3. **Efficiency Constraints:**
   - The function should handle the maximum input size efficiently.
   - The expected time complexity for the function should be approximately $O(n)$ where n is the length of the input array A, to ensure that the function can process large datasets within reasonable time limits.

By solving this problem, you will be able to provide the company with a valuable tool for their data processing efforts, which could ultimately help them gain insights from their datasets and make better business decisions.

Test Case1:
Input: {1, 2, 2, 3, 4, 4, 5}
Output: {1, 2, 3, 4, 5}
Size: 5

Test Case 2:
Input: {5, 5, 5}
Output: {5}
Size: 1

Test Case 3:
Input: {6, 3, 8, 2, 6, 3, 8, 3}

Output: {6, 3, 8, 2}
Size: 4

Test Case 4:
Input: {1, 2, 3, 4, 5}
Output: {1, 2, 3, 4, 5}
Size: 5

Test Case 5:
Input: {1, 2, 3, ..., 100000}
Output: {1, 2, 3, ..., 100000}
Size: 100000

Test Case 6:
Input: {}
Output: {}
Size: 0

Test Case 7:
Input: {1, 2, 2, 3, 3, 3, 4}
Output: {1, 2, 3, 4}
Size: 4

Test Case 8:
Input: {1, 2, 3, 4, 5, 1, 2, 3}
Output: {1, 2, 3, 4, 5}
Size: 5

Test Case 9:
Input: {1, 1, 2, 3, 4, 5}
Output: {1, 2, 3, 4, 5}
Size: 5

Test Case 10:
Input: {1, 2, 3, 4, 5, 5}
Output: {1, 2, 3, 4, 5}
Size: 5