100 Problems

AMK & SSC

The Coding Skills Program

August 7, 2024

Department of Information Technology

Faculty of Industrial Technology and Management

King Mongkut's University of Technology North Bangkok

Problem-01: Find Multiples of Three

Objective: Create a function that takes two integers as input, a start and an end, and returns a list of all numbers within this range (inclusive) that are divisible by 3.

Function Signature:

```
def find_multiples_of_three(start: int, end: int) -> list:
    pass
```

Instructions:

1. **Input Validation**
   - Ensure that the start value is less than or equal to the end value. If not, return an empty list.
2. **Find Multiples**
   - Find all numbers between start and end (inclusive) that are divisible by 3.
3. **Return Result**
   - Return the list of these numbers.

Example:

- **Input:** start = 10, end = 25
- **Output:** [12, 15, 18, 21, 24]

Constraints:

- Both start and end will be integers between 1 and 1000.
- The function should be optimized for efficiency.

Problem-02: Find Multiples of Both Three and Four

Objective: Create a function that takes two integers as input, a start and an end, and returns a list of all numbers within this range (inclusive) that are divisible by both 3 and 4.

Function Signature:

```
def find_multiples_of_three_and_four (start: int, end: int) -> list:
    pass
```

Instructions:

1. **Input Validation**
   - Ensure that the start value is less than or equal to the end value. If not, return an empty list.
2. **Find Multiples**
   - Find all numbers between start and end (inclusive) that are divisible by both 3 and 4. (Note: A number divisible by both 3 and 4 is also divisible by their least common multiple, which is 12.)
3. **Return Result**
   - Return the list of these numbers.

Example:

- **Input:** start = 10, end = 50
- **Output:** [12, 24, 36, 48]

Constraints:

- Both start and end will be integers between 1 and 1000.
- The function should be optimized for efficiency.

Problem-03: Find Non-Multiples of Three, Four, and Five

Objective: Create a function that takes two integers as input, a start and an end, and returns a list of all numbers within this range (inclusive) that are not divisible by 3, 4, and 5.

Function Signature:

```
def find_non_multiples(start: int, end: int) -> list:
    pass
```

Instructions:

1. **Input Validation**
   - Ensure that the start value is less than or equal to the end value. If not, return an empty list.
2. **Find Non-Multiples**
   - Find all numbers between start and end (inclusive) that are not divisible by 3, 4, or 5.
3. **Return Result**
   - Return the list of these numbers.

Example:

- **Input:** start = 10, end = 25
- **Output:** [11, 13, 17, 19, 23]

Constraints:

- Both start and end will be integers between 1 and 1000.
- The function should be optimized for efficiency.

Problem-04: Average Calculation

Objective: Create a function that prompts the user to input 5 real numbers and then calculates the sum and average of these inputs.

Function Signature:

```
def calculate_sum_and_average() -> None:
    pass
```

Instructions:

1. **Step 1: Input Collection**
    - Use a for loop to prompt the user to input 5 real numbers.
    - Store these inputs in a list.
2. **Step 2: Calculate the Sum**
    - Compute the sum of the numbers collected.
3. **Step 3: Calculate the Average**
    - Calculate the average of the numbers.
4. **Step 4: Display the Results**
    - Print the sum and the average of the input numbers.

Example:

- **Input:**
    - Enter number 1: 10.5
    - Enter number 2: 20.0
    - Enter number 3: 30.25
    - Enter number 4: 15.75
    - Enter number 5: 25.5
- **Output:**
    - Sum: 102.0
    - Average: 20.4

Constraints:

- The input numbers should be real numbers (i.e., integers or floats).
- The function should handle exactly 5 inputs.
- The result should be printed with a precision of 2 decimal places.

Problem-05: Divisor Finder

Objective: Create a function that takes an integer as input and finds all divisors of that integer. The function should return a list of divisors in ascending order.

Function Signature:

```
def find_divisors(n: int) -> List[int]:
    pass
```

Instructions:

1. **Step 1: Input Validation**
   - Ensure that the input is a positive integer greater than zero. If the input is invalid, return an empty list.
2. **Step 2: Find Divisors**
   - Calculate all divisors of the given number. A divisor is a number that divides the given number with no remainder.
   - For example, if the input is 20, the divisors are 1, 2, 4, 5, 10, 20.
3. **Step 3: Return Output**
   - Return a list of all divisors, sorted in ascending order.

Example:

- **Input:** 20
- **Output:** [1, 2, 4, 5, 10, 20]

Constraints:

- The input integer will be between 1 and 1000.
- The function should be optimized for efficiency.

Problem-06: Prime Number Checker

Objective: Create a function that takes an integer as input and checks whether it is a prime number. The function should return a string indicating whether the number is prime or not.

Function Signature:

```
def check_prime(n: int) -> str:
    pass
```

Instructions:

1. **Step 1: Input Validation**
    - Ensure that the input is a positive integer greater than 1. If the input is less than or equal to 1, return "is not prime".
2. **Step 2: Prime Check**
    - A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself.
    - If the number is prime, return the string "is prime".
    - If the number is not prime, return the string "is not prime".

Example:

- **Input:** 17
- **Step 1:** Validate Input: 17 is greater than 1, proceed to next step.
- **Step 2:** Prime Check: 17 is only divisible by 1 and 17.
- **Output:** "is prime"
- **Input:** 18
- **Step 1:** Validate Input: 18 is greater than 1, proceed to next step.
- **Step 2:** Prime Check: 18 is divisible by 1, 2, 3, 6, 9, 18.
- **Output:** "is not prime"

Constraints:

- The input integer will be between 1 and 10,000.
- The function should be optimized for efficiency, especially for larger numbers.

Problem-07: Prime Numbers in a Range

Objective: Write a function that takes a starting and ending number as input and identifies all prime numbers within that range. Additionally, calculate the sum of these prime numbers and return both the list of prime numbers and their sum.

Function Signature:

```
def prime_numbers_in_range(start: int, end: int) -> tuple:
    pass
```

Instructions:

1. **Input:**
    - Two integers, start and end, where 1 <= start <= end <= 10^6.
2. **Output:**
    - A tuple containing two elements:
        - A list of all prime numbers between start and end, inclusive.
        - An integer representing the sum of these prime numbers.

Steps:

1. **Find Prime Numbers:**
    - Identify all prime numbers in the range from start to end. A prime number is a number greater than 1 that has no positive divisors other than 1 and itself.
2. **Calculate Sum:**
    - Compute the sum of the prime numbers found in the range.

Example:

- **Input:**

  prime_numbers_in_range(10, 20)

- **Output:**

  ([11, 13, 17, 19], 60)

  Explanation:

    - The prime numbers between 10 and 20 are 11, 13, 17, and 19.
    - Their sum is 60.

Constraints:

- Both start and end will be integers between 1 and 1000.
- Consider using efficient algorithms for finding prime numbers, such as the Sieve of Eratosthenes or optimized trial division.

Problem-08: Extract Even/Odd Indexed Characters

Objective: Create a function that takes a string as input and outputs two separate strings. The first string should contain characters from the input string that are located at even indices, while the second string should contain characters from the input string that are located at odd indices.

Function Signature:

```
def separate_by_index(s: str) -> Tuple[str, str]:
    pass
```

Instructions:

1. **Extract Even Indexed Characters:**
    - Create a string containing characters from the input string where the index is an even number (0, 2, 4, ...).
2. **Extract Odd Indexed Characters:**
    - Create a string containing characters from the input string where the index is an odd number (1, 3, 5, ...).

Example:

- **Input:** "Hello World"
- **Even Indexed Characters:** "HloWrd"
- **Odd Indexed Characters:** "el ol"

Output:

- **Output:** ("HloWrd", "el ol")

Constraints:

- The input string will contain only alphabetic characters and spaces.
- The length of the string will be between 1 and 100 characters.
- The function should be optimized for efficiency.

Problem-09: Average Length of Strings

Objective: Create a function that takes five strings as input and outputs the average number of characters across all five strings.

Function Signature:

```
def average_length_of_strings(strings: List[str]) -> float:
    pass
```

Instructions:

1. **Input:**
    - The function should accept five strings as input.
    - Each string will contain alphabetic characters and spaces only.
2. **Calculate the Average Length:**
    - Calculate the length of each string.
    - Compute the average length by summing the lengths of all strings and dividing by 5.

Example:

- **Input:** ["apple", "banana", "cherry", "date", "elderberry"]
- **Lengths:** [5, 6, 6, 4, 10]
- **Average Length:** (5 + 6 + 6 + 4 + 10) / 5 = 6.2
- **Output:** 6.2

Constraints:

- Each string will have a length between 1 and 100 characters.
- The function should be efficient and handle the input within these constraints.

Problem-10: Character Frequency Count in Multiple Strings

Objective: Create a function that takes five input strings and returns the frequency count of each character across all strings combined.

Function Signature:

```
def character_frequency(*args: str) -> dict:
    pass
```

Instructions:

1. **Input**:
   - The function should accept exactly five strings as input.
2. **Character Frequency Count**:
   - Combine all five strings into one.
   - Calculate the frequency of each character in the combined string.
   - Return a dictionary where the keys are the characters and the values are their respective counts.
3. **Output**:
   - The output should be a dictionary containing each character as the key and the number of occurrences of that character across all five strings as the value.

Example:

Input: "hello", "world", "test", "case", "example"
Output: {'h': 1, 'e': 5, 'l': 4, 'o': 2, 'w': 1, 'r': 1, 'd': 1, 't': 3, 's': 2, 'c': 2, 'a': 2, 'x': 1, 'm': 1, 'p': 1}

Constraints:

- Each input string will contain only alphabetic characters and spaces.
- The length of each string will be between 1 and 50 characters.
- The function should handle the inputs efficiently.

Problem-11: Check for Vowel Presence in a String

Objective: Create a function that takes a string as input and checks whether the string contains any characters from the set of vowels: [a, e, i, o, u]. The function should return a boolean value indicating the presence of at least one vowel.

Function Signature:

```
def contains_vowel(s: str) -> bool:
    pass
```

Instructions:

1. **Vowel List:**
    - Use the list vowel = ['a', 'e', 'i', 'o', 'u'] to check against the characters in the input string.
2. **Check for Vowels:**
    - The function should iterate through the input string and check if any of the characters match those in the vowel list.
    - Return True if at least one vowel is found, otherwise return False.

Example:

- **Input:** "Hello World"
- **Vowel Presence:** True (because the string contains 'e' and 'o')

Output:

- **Output:** True

Constraints:

- The input string will contain only lowercase alphabetic characters and spaces.
- The length of the string will be between 1 and 100 characters.
- The function should be optimized for efficiency.

Problem-12: Replace Specific Characters in a String

Objective: Create a function that takes a string as input and returns a new string where certain characters are replaced according to specific rules.

Function Signature:

```
def replace_characters(s: str) -> str:
    pass
```

Instructions:

1. **Replace 'a' with '@'**:
    - In the input string, replace every occurrence of the character 'a' with '@'.
2. **Replace 'l' with '1'**:
    - In the input string, replace every occurrence of the character 'l' with '1'.
3. **Replace 'o' with '0'**:
    - In the input string, replace every occurrence of the character 'o' with '0'.

Example:

- **Input**: "Hello World"
- **Output**: "He110 W0r1d"

Constraints:

- The input string will contain only alphabetic characters and spaces.
- The length of the string will be between 1 and 100 characters.
- The function should handle both uppercase and lowercase characters.

Problem-13: Reverse Characters of a String

Objective:
Create a function that takes a single string as input and returns the string with its characters in reverse order.

Function Signature:

```
def reverse_string(s: str) -> str:
    pass
```

Instructions:

1. **Input a Single String:**
    - The function should accept a single string as input.
2. **Reverse the Characters:**
    - The function should reverse the order of characters in the input string.
3. **Output the Reversed String:**
    - Return the reversed string.

Example:

- **Input:** "Hello World"
- **Output:** "dlroW olleH"

Constraints:

- The input string will contain only alphabetic characters, spaces, and possibly punctuation marks.
- The length of the string will be between 1 and 100 characters.
- The function should be optimized for efficiency.

Problem-14: Unique Word Collector

Objective: Create a program that continuously accepts word inputs from the user and adds them to a list, but with a condition that no duplicate words are allowed. The process should stop when the list contains exactly 5 unique words.

Function Signature:

```
def collect_unique_words() -> List[str]:
    pass
```

Instructions:

1. **Initialize an Empty List:**
    - Start by creating an empty list to store the words.
2. **Input Words:**
    - Prompt the user to input words one by one.
3. **Add Words with Uniqueness Constraint:**
    - Add each input word to the list only if it is not already present in the list.
    - Continue to prompt the user for input until the list contains exactly 5 unique words.
4. **Stop When Condition is Met:**
    - The program should stop accepting input once 5 unique words have been collected.

Example:

- Input Sequence: "apple", "banana", "apple", "cherry", "date", "banana", "elderberry"
- Output List: ["apple", "banana", "cherry", "date", "elderberry"]

Constraints:

- The input words will contain only alphabetic characters and no spaces.
- The program should ensure that the list ends up with exactly 5 unique words, regardless of the number of inputs provided.

Problem-15: Count Word Occurrences

Objective: Create an empty list and accept input words. Then, add these words to the list along with the number of times each word appears.

Function Signature:

```
def count_word_occurrences(words: List[str]) -> Dict[str, int]:
    pass
```

Instructions:

1. **Create an Empty List:**
   - Initialize an empty list to store the words and their counts.
2. **Accept Input Words:**
   - Read a sequence of words from the input.
3. **Count Occurrences:**
   - For each word, update its count in the list.
4. **Return Results:**
   - Return a dictionary where the keys are the words and the values are their counts.

Example:

- **Input:** ["apple", "banana", "apple", "orange", "banana", "apple"]
- **Output:** {"apple": 3, "banana": 2, "orange": 1}

Constraints:

- The input list will contain only alphabetic words and spaces.
- The number of words in the list will be between 1 and 100 words.
- The function should be optimized for efficiency.

Problem-16: Insert New Words at the Front of a List

Objective: Create an empty list and accept input words. Add these words to the list such that each new word is inserted at the front of the list.

Function Signature:

```
def insert_at_front(words: List[str]) -> List[str]:
    pass
```

Instructions:

1. **Create an Empty List:**
    - Initialize an empty list.
2. **Accept Input Words:**
    - Read input words, one by one.
3. **Insert Words at the Front:**
    - For each new word received, insert it at the beginning of the list.

Example:

- **Input:** "apple", "banana", "cherry"
- **Operations:**
    - Add "apple" -> List becomes ["apple"]
    - Add "banana" -> List becomes ["banana", "apple"]
    - Add "cherry" -> List becomes ["cherry", "banana", "apple"]
- **Output:** ["cherry", "banana", "apple"]

Constraints:

- The function should handle a list of words and insert each new word at the front of the list.
- Words can contain alphabetic characters and may include spaces.

Problem-17: Word Search in List

Objective: Create a function that takes a list of 10 words and a search term as input. The function should check if the search term is present in the list or not.

Function Signature:

```
def is_word_in_list(word_list: List[str], search_term: str) -> bool:
    pass
```

Instructions:

1. **Input List:** The input list (word_list) will contain exactly 10 words, each of which is a non-empty string.
2. **Search Term:** The search term (search_term) will be a single word (a non-empty string).
3. **Output:** The function should return True if the search term is found in the list, and False otherwise.

Example:

- **Input:**

  word_list = ["apple", "banana", "cherry", "date", "elderberry", "fig", "grape", "honeydew", "kiwi", "lemon"]
  search_term = "cherry"

- **Output:**

  True

- **Input:**

  word_list = ["apple", "banana", "cherry", "date", "elderberry", "fig", "grape", "honeydew", "kiwi", "lemon"]
  search_term = "mango"

- **Output:**

  False

Constraints:

- The length of word_list is exactly 10.
- Each word in word_list and search_term consists only of alphabetic characters.
- The length of each word and search_term is between 1 and 100 characters.

Problem-18: Remove Word from List

Objective: Create a function that takes a list of 10 words and a word as input. The function should check if the given word exists in the list. If the word exists, it should be removed from the list.

Function Signature:

```
def remove_word_from_list(words: List[str], word: str) -> List[str]:
    pass
```

Instructions:

1. **Check for Word Existence:**
    - Determine if the input word exists in the provided list of 10 words.
2. **Remove Word:**
    - If the word is found in the list, remove the first occurrence of that word from the list.

Example:

- **Input:**
    - words = ["apple", "banana", "cherry", "date", "elderberry", "fig", "grape", "honeydew", "kiwi", "lemon"]
    - word = "cherry"
- **Output:**
    - ["apple", "banana", "date", "elderberry", "fig", "grape", "honeydew", "kiwi", "lemon"]

Constraints:

- The list will always contain exactly 10 words.
- The input word is guaranteed to be a single word (no spaces) and will be a valid string.
- If the word is not found in the list, the list should remain unchanged.

Problem-19: Sum of Corresponding Elements in Two Matrices

Objective: Create a function that takes two matrices (2D lists) as input and outputs a new matrix where each element is the sum of the elements in the corresponding positions of the two input matrices.

Function Signature:

```
def sum_matrices(matrix1: List[List[int]], matrix2: List[List[int]]) -> List[List[int]]:
    pass
```

Instructions:

1. **Sum Corresponding Elements:**
    - For each position (i, j) in the matrices, compute the sum of the elements from matrix1 and matrix2 at that position and place the result in the corresponding position in the output matrix.
2. **Input:**
    - matrix1 and matrix2 are both 2D lists (matrices) with the same dimensions.

Example:

- **Input:**
    - matrix1 = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
    - matrix2 = [[4,3,2,1],[4,3,2,1],[4,3,2,1]]
- **Output:**
    - [[5,5,5,5],[9,9,9,9],[13,13,13,13]]

Constraints:

- The matrices matrix1 and matrix2 will have the same number of rows and columns.
- The number of rows and columns will be between 1 and 100.

Problem-20: Transpose a Matrix

Objective: Create a function that takes a 2D matrix as input and outputs its transpose.

Function Signature:

```
def transpose_matrix(matrix: List[List[int]]) -> List[List[int]]:
    pass
```

Instructions:

1. **Transpose the Matrix:**
    - Create a function that receives a 2D matrix (a list of lists) and returns its transpose.
    - The transpose of a matrix is obtained by swapping rows with columns. Specifically, the element at position (i, j) in the original matrix should be at position (j, i) in the transposed matrix.

Example:

- **Input:**

  matrix1 = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]

- **Output:**

  [[1, 5, 9],
  [2, 6, 10],
  [3, 7, 11],
  [4, 8, 12]]

Constraints:

- The matrix will contain only integers.
- The matrix will be rectangular (i.e., all rows will have the same number of columns).
- The function should be optimized for efficiency.

Problem-21: Calculate Squares and Statistics

Objective: Given a tuple of integers, write a program that calculates the square of each element in the tuple. Output the squared values as a tuple along with the maximum value, minimum value, sum, and average of the squared values.

Function Signature:

```
def calculate_statistics(t: Tuple[int, ...]) -> Tuple[Tuple[int, ...], int, int, int, float]:
    pass
```

Instructions:

1. **Calculate Squares:**
   - Create a tuple containing the squares of the integers from the input tuple.
2. **Compute Statistics:**
   - Find and output the maximum value, minimum value, sum, and average of the squared values.

Example:

- **Input:** (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
- **Squared Values:** (1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
- **Maximum Value:** 100
- **Minimum Value:** 1
- **Sum:** 285
- **Average:** 28.5

Output:

- **Output:** ((1, 4, 9, 16, 25, 36, 49, 64, 81, 100), 100, 1, 285, 28.5)

Constraints:

- The input tuple will contain integer values and can have between 1 and 100 integers.
- The function should be optimized for efficiency.

Problem-22: Create a Dictionary from Two Lists

Objective: Given two lists, list1 containing integers and list2 containing strings, write a program that creates a dictionary where the keys are elements from list1 and the corresponding values are elements from list2.

Function Signature:

```
def create_dictionary(list1: List[int], list2: List[str]) -> Dict[int, str]:
    pass
```

Instructions:

1. **Create a Dictionary:**
    - Each key in the dictionary should be an element from list1.
    - Each value in the dictionary should be the corresponding element from list2 at the same index.
2. **Assumptions:**
    - Both list1 and list2 will have the same length.
    - The elements in list1 are unique.

Example:

- **Input:**
    - list1 = [1, 2, 3, 4]
    - list2 = ["blue", "green", "pink", "yellow"]
- **Output:**
    - {1: "blue", 2: "green", 3: "pink", 4: "yellow"}

Constraints:

- The length of list1 and list2 will be between 1 and 100.

Problem-23: Create Dictionary from Tuples

Objective: Given two tuples, tuple1 and tuple2, write a program that creates a dictionary where the keys are the elements of tuple1 and the values are the elements of tuple2.

Function Signature:

```
def create_dictionary(tuple1: Tuple[int, ...], tuple2: Tuple[str, ...]) -> Dict[int, str]:
    pass
```

Instructions:

1. **Create Dictionary:**
   - Create a dictionary where each element in tuple1 is a key, and the corresponding element in tuple2 is the value.
   - Assume that both tuples have the same length.

Example:

- **Input:**
  tuple1 = (1, 2, 3, 4)
  tuple2 = ("ant", "cat", "dog", "cow")
- **Output:**
  {1: "ant", 2: "cat", 3: "dog", 4: "cow"}

Constraints:

- The input tuples will have the same length and can contain between 1 and 100 elements.
- The first tuple (tuple1) will contain integers, and the second tuple (tuple2) will contain strings.

Problem-24: Store Student Information in a Dictionary

Objective: Given the student IDs and nicknames of 3 students, write a program that stores this information in a dictionary where the student ID is the key and the nickname is the value.

Function Signature:

```
def store_student_info(student_data: List[Tuple[str, str]]) -> Dict[str, str]:
    pass
```

Instructions:

1. **Input Data:**
   - The function will receive a list of tuples, each containing a student ID (as a string) and a nickname (as a string).
2. **Store Data in a Dictionary:**
   - Create a dictionary where the student IDs serve as the keys, and the nicknames serve as the values.
3. **Return the Dictionary:**
   - The function should return the dictionary containing the student information.

Example:

- **Input:** [("123456", "Alice"), ("654321", "Bob"), ("112233", "Charlie")]
- **Output:** {"123456": "Alice", "654321": "Bob", "112233": "Charlie"}

Constraints:

- The input list will always contain exactly 3 tuples.
- Each tuple will contain a valid student ID (a string of digits) and a valid nickname (a string).

Problem-25: Store Student Scores in a Course

Objective: Given a list of student information, write a program that stores this information in a dictionary where each student's ID is the key, and the value is another dictionary containing the student's name and score.

Function Signature:

```
def store_student_scores(student_data: List[Tuple[str, str, float]]) -> Dict[str, Dict[str, float]]:
    pass
```

Instructions:

1. **Input Data:**
    - The function will receive a list of tuples, each containing a student ID (as a string), a name (as a string), and a score (as a float).
2. **Store Data in a Nested Dictionary:**
    - Create a dictionary where the student IDs serve as the keys. The value for each key should be another dictionary with the keys "name" and "score", containing the respective student's name and score.
3. **Return the Dictionary:**
    - The function should return the nested dictionary containing all the student information.

Example:

- **Input:**

```
[
    ("123456", "Alice", 85.5),
    ("654321", "Bob", 92.0),
    ("112233", "Charlie", 78.0)
]
```

- **Output:**

```
{
    "123456": {"name": "Alice", "score": 85.5},
    "654321": {"name": "Bob", "score": 92.0},
    "112233": {"name": "Charlie", "score": 78.0}
}
```

Constraints:

- The input list can contain any number of tuples.
- Each tuple will contain a valid student ID (a string of digits), a valid name (a string), and a valid score (a float).
- Scores can be any float value, typically between 0 and 100.

Problem-26: Search Country Names by Starting Letter

Objective: Given a dictionary of country codes and their corresponding country names, write a program that allows searching for country names that start with a specific letter. The country data is stored in a dictionary where the country code is the key, and the country name is the value.

Function Signature:

```
def search_countries_by_letter(country_data: Dict[str, str], letter: str) -> List[str]:
    pass
```

Instructions:

1. **Input Data:**
   - The function will receive a dictionary where the keys are country codes (as strings) and the values are country names (as strings).
   - The function will also receive a single letter (as a string) to search for country names that start with this letter.
2. **Search for Matching Country Names:**
   - Identify all country names in the dictionary that start with the given letter (case insensitive).
   - Return a list of matching country names.
3. **Return the Results:**
   - If no country names match the starting letter, return an empty list.
   - The list of country names should be sorted alphabetically.

Example:

- **Input:**

```
country_data = {
    "+1": "United States",
    "+44": "United Kingdom",
    "+91": "India",
    "+81": "Japan",
    "+49": "Germany",
    "+86": "China"
}
letter = "U"
```

- **Output:** ["United Kingdom", "United States"]

Constraints:

- The input dictionary can contain any number of country codes and names.
- The search letter will always be a single alphabetical character.
- The country names in the dictionary are unique.
- The search should be case-insensitive.

Problem-27: Build a Set with User Input

Objective: Write a program that creates an empty set and then continuously accepts integer inputs from the user to add to the set until the set contains exactly 5 unique elements.

Function Signature:

```
def build_set() -> Set[int]:
    pass
```

Instructions:

1. **Create an Empty Set:**
   - Initialize an empty set.
2. **Accept User Input:**
   - Continuously prompt the user to input an integer.
   - Add the input value to the set.
   - Stop accepting input once the set contains exactly 5 unique elements.
3. **Output:**
   - Return the set containing 5 unique integers.

Example:

- **Input:**
  - User enters: 10, 20, 30, 40, 50
- **Output:**
  - {10, 20, 30, 40, 50}

Constraints:

- The program should ensure that the set contains exactly 5 unique integers.
- If a duplicate value is entered, the program should prompt the user for another input until the set has 5 unique integers.

Problem-28: Check Membership in Set

Objective: Given a set containing integers, vowels, and names, write a program that checks if a given input value is present in the set.

Function Signature:

```
def check_membership(s: set, value: str) -> bool:
    pass
```

Instructions:

1. **Membership Check:**
    - Check if the given value is present in the set s.
2. **Output:**
    - Return True if the value is in the set; otherwise, return False.

Example:

- **Input:**
    - s = {1, 2, 3, 'a', 'e', 'i', 'o', 'u', "red", "green", "blue"}
    - value = 2
- **Output:**
    - True
- **Input:**
    - s = {1, 2, 3, 'a', 'e', 'i', 'o', 'u', "red", "green", "blue"}
    - value = 'a'
- **Output:**
    - True
- **Input:**
    - s = {1, 2, 3, 'a', 'e', 'i', 'o', 'u', "red", "green", "blue"}
    - value = 'yellow'
- **Output:**
    - False

Constraints:

- The input set s contains integers, vowels, and color names.
- The value to be checked is a string.

Problem-29: Set Operations

Objective: Given two sets of characters, set1 and set2, compute the union and intersection of these sets.

Function Signature:

```
def set_operations(set1: set, set2: set) -> Tuple[set, set]:
    pass
```

Instructions:

1. **Union**: Compute the union of set1 and set2. The union of two sets is a set containing all the unique elements from both sets.
2. **Intersection**: Compute the intersection of set1 and set2. The intersection of two sets is a set containing all the elements that are common to both sets.

Example:

- **Input:**
  - set1 = {'a', 'e', 'i', 'o', 'u'}
  - set2 = {'h', 'e', 'l', 'l', 'o'}
- **Union:** {'a', 'e', 'i', 'o', 'u', 'h', 'l'}
- **Intersection:** {'e', 'o'}

Output:

- **Output:** ({'a', 'e', 'i', 'o', 'u', 'h', 'l'}, {'e', 'o'})

Constraints:

- The input sets will contain character elements and can have between 1 and 100 elements.
- The function should be optimized for efficiency.

Problem-30: Set Difference Calculation

Objective: Given two sets of characters, each containing exactly 3 unique characters, compute the difference between the two sets in both directions.

Function Signature:

```
def calculate_set_differences(set1: set, set2: set) -> Tuple[set, set]:
    pass
```

Instructions:

1. **Input:**
   - set1: A set containing exactly 3 unique characters.
   - set2: A set containing exactly 3 unique characters.
2. **Calculate Differences:**
   - Compute set1 - set2 which is the set of elements in set1 that are not in set2.
   - Compute set2 - set1 which is the set of elements in set2 that are not in set1.

Example:

- **Input:**
  - set1 = {'a', 'b', 'c'}
  - set2 = {'b', 'c', 'd'}
- **Output:**
  - ({'a'}, {'d'})

Constraints:

- Both sets set1 and set2 will contain exactly 3 unique characters.
- The function should be optimized for efficiency.

Problem-31: Country Sales Analysis

Objective: Given a list of country names, write a program that receives sales figures for each country and then displays the name of the country with the highest sales along with the sales amount.

Input:

- A list of country names: countries = ["Thailand", "Laos", "Vietnam", "Japan", "China"].
- Sales figures for each country, where the country name is the key and the sales amount is the value.

Output:

- The name of the country with the highest sales and the corresponding sales amount.

Function Signature:

```
def highest_sales_country(sales: dict[str, int]) -> Tuple[str, int]:
    pass
```

Instructions:

1. **Receive Sales Data:**
    - Accept a dictionary where the keys are country names and the values are the sales figures.
2. **Compute the Maximum Sales:**
    - Identify the country with the highest sales.
3. **Output:**
    - Return a tuple containing the name of the country with the highest sales and the sales amount.

Example:

- **Input:**

```
sales_data = {
    "Thailand": 1500,
    "Laos": 1200,
    "Vietnam": 1800,
    "Japan": 1700,
    "China": 2000
}
```

- **Output:**

```
("China", 2000)
```

Constraints:

- The dictionary will contain between 1 and 100 countries.
- Sales figures are non-negative integers.

Problem-32: Calculate the Median

Objective: Given a list of integers, write a program that calculates the median value of the list.

Function Signature:

```
def calculate_median(lst: List[int]) -> float:
    pass
```

Instructions:

1. **Sort the List:**
   - Sort the list of integers in non-decreasing order.
2. **Calculate the Median:**
   - If the list has an odd number of elements, the median is the middle element.
   - If the list has an even number of elements, the median is the average of the two middle elements.

Example:

- **Input:** [8, 4, 7, 4, 6, 2, 10, 9, 3, 7, 1]
- **Sorted List:** [1, 2, 3, 4, 4, 6, 7, 7, 8, 9, 10]

Output:

- **Median:** 6

Constraints:

- The input list will contain integer values and can have between 1 and 1000 integers.
- The list may contain duplicate values.

Problem-33: Median Calculation

Objective: Given a dictionary of country names and the number of provinces in each country, write a program to find the median number of provinces and display the country names along with their province counts.

Function Signature:

```
def calculate_median(provinces: Dict[str, int]) -> List[Tuple[str, int]]:
    pass
```

Instructions:

1. **Compute Median**:
    - Extract the number of provinces from the dictionary and find the median value.
    - The median is the middle value in a sorted list of the number of provinces. If there is an even number of elements, the median is the average of the two middle values.
2. **Display Results**:
    - Output a list of tuples where each tuple contains the country name and the number of provinces, only for countries where the number of provinces matches the computed median.

Example:

- **Input**: {'Thailand':76, 'Laos':17, 'Vietnam':58, 'Japan':47, 'China':23}
- **Median**: 47 (since the sorted number of provinces is [17, 23, 47, 58, 76], and 47 is the middle value)
- **Output**: [('Japan', 47)]

Constraints:

- The dictionary will contain country names as keys and integer values representing the number of provinces.
- The dictionary will have between 1 and 100 entries.

Problem-34: Print a Rectangle Pattern

Objective: Write a program that prints a rectangular pattern of asterisks (*) based on the specified number of rows and columns.

Function Signature:

```
def print_rectangle_pattern(rows: int, columns: int) -> None:
    pass
```

Instructions:

1. **Print Pattern:**
   - Use nested loops to print the specified number of rows and columns of asterisks (*).
   - Each row should contain the specified number of asterisks.
   - After printing each row, move to the next line.

Example:

- **Input:** rows = 5, columns = 5
- **Output:**

```
*****
*****
*****
*****
*****
```

Constraints:

- The number of rows and columns will be positive integers between 1 and 100.
- The function should be optimized for efficiency and should not return any value.

Problem-35: Print a Diamond Pattern

Objective: Write a program that prints a diamond pattern of asterisks (*) based on the specified number of rows in the widest part of the diamond.

Function Signature:

```
def print_diamond_pattern(n: int) -> None:
    pass
```

Instructions:

1. **Print Pattern:**
    - Use loops to print a diamond shape with the specified number of rows in the widest part of the diamond.
    - The pattern should consist of increasing numbers of asterisks (*) up to n, and then decreasing back to 1.

Example:

- **Input:** n = 3
- **Output:**

```
*
**
***
**
*
```

Constraints:

- The number of rows n will be a positive integer between 1 and 100.
- The function should be optimized for efficiency and should not return any value.

Problem-36: Print a Diamond Pattern with Hyphens

Objective: Write a program that prints a diamond-shaped pattern using asterisks (*) and hyphens (-) based on the specified number of rows.

Function Signature:

```
def print_diamond_pattern(n: int) -> None:
    pass
```

Instructions:

1. **Print Pattern:**
    - Use loops to print the specified diamond pattern.
    - The pattern consists of rows where the first half of the pattern (including the middle row) has decreasing asterisks and increasing hyphens. The second half of the pattern mirrors the first half.
    - Each row should be formatted with a combination of asterisks and hyphens.

Example:

- **Input:** n = 10
- **Output:**

```
**********
****--****
***----***
**------**
*--------*
**------**
***----***
****--****
**********
```

Constraints:

- The number of rows n will be an even positive integer between 2 and 100.
- The function should be optimized for efficiency and should not return any value.

Problem-37: Print a Number Pattern

Objective: Write a program that prints a number pattern based on the specified number of rows.

Function Signature:

```
def print_number_pattern(rows: int) -> None:
    pass
```

Instructions:

1. **Print Pattern:**
    - Use nested loops to print the specified number of rows with the following pattern:
        - The first row should have rows - 1 dashes followed by the number 1.
        - The second row should have rows - 2 dashes followed by the numbers 2 and 1.
        - Continue this pattern until the last row, which should have no dashes and print numbers from rows down to 1.

Example:

- **Input:** rows = 5
- **Output:**

```
----1
---21
--321
-4321
54321
```

Constraints:

- The number of rows will be a positive integer between 1 and 100.
- The function should be optimized for efficiency and should not return any value.

Problem-38: Simple Calculator

Objective: Write a program that performs basic arithmetic operations (+, -, *, /) using separate functions for each operation. The program should take two numbers and an operator as input and return the result of the operation.

Function Signatures:

```
def add(a: float, b: float) -> float:
    pass
def subtract(a: float, b: float) -> float:
    pass
def multiply(a: float, b: float) -> float:
    pass
def divide(a: float, b: float) -> float:
    pass
```

Instructions:

1. **Addition Function:**
    - Implement a function add(a: float, b: float) -> float that returns the sum of a and b.
2. **Subtraction Function:**
    - Implement a function subtract(a: float, b: float) -> float that returns the difference between a and b.
3. **Multiplication Function:**
    - Implement a function multiply(a: float, b: float) -> float that returns the product of a and b.
4. **Division Function:**
    - Implement a function divide(a: float, b: float) -> float that returns the quotient of a divided by b. If b is 0, the function should handle the division by zero gracefully (e.g., raise an appropriate exception or return None).

Example:

- **Input:** add(5, 3)
    - **Output:** 8
- **Input:** subtract(5, 3)
    - **Output:** 2
- **Input:** multiply(5, 3)
    - **Output:** 15
- **Input:** divide(6, 3)
    - **Output:** 2.0
- **Input:** divide(6, 0)
    - **Output:** None or raise an exception

Constraints:

- The functions should handle floating-point numbers.
- The division function should handle division by zero gracefully.

Problem-39: Remove a Word from a Sentence

Objective: Write a function that removes a specified word from a given sentence.

Function Signature:

```
def remove_word(sentence: str, word_to_remove: str) -> str:
    pass
```

Instructions:

1. **Remove Word:**
    - The function should remove all occurrences of the specified word from the given sentence.
    - Ensure that the sentence remains properly formatted, with spaces appropriately adjusted after the word removal.

Example:

- **Input:** sentence = "Python is a popular programming language.", word_to_remove = "popular"
- **Output:** "Python is a programming language."

Constraints:

- The input sentence will be a non-empty string and will contain only alphabetic characters and spaces.
- The word to remove will be a non-empty string consisting of alphabetic characters.
- The function should handle cases where the word to remove is not present in the sentence by returning the original sentence.

Problem-40: Calculate Profit from Sales and Costs

Objective: Write a function that calculates the annual profit and total profit over the past 5 years given the sales and costs for each year.

Function Signature:

```
def calculate_profit(sales: Tuple[float, float, float, float, float], costs: Tuple[float, float, float, float, float]) -> Tuple[Tuple[float, float, float, float, float], float]:
    pass
```

Instructions:

1. **Calculate Annual Profit:**
    - For each year, compute the profit as the difference between sales and costs for that year.
    - Output the annual profits as a tuple.
2. **Compute Total Profit:**
    - Calculate the total profit over the 5 years by summing up the annual profits.
    - Return the total profit.

Example:

- **Input:** sales = (10000.0, 15000.0, 20000.0, 25000.0, 30000.0), costs = (7000.0, 8000.0, 9000.0, 11000.0, 12000.0)
- **Output:** ((3000.0, 7000.0, 11000.0, 14000.0, 18000.0), 75000.0)

Constraints:

- The input tuples will each contain exactly 5 elements, representing sales and costs for each of the 5 years.
- Sales and costs values will be positive floating-point numbers.

Problem-41: Calculate Discounted Prices

Objective: Write a program that calculates the discounted prices for a list of products given their original prices and a discount percentage.

Function Signature:

```
def calculate_discounted_prices(prices: List[float], discount_percentage: float) -> List[float]:
    pass
```

Instructions:

1. **Calculate Discounted Prices:**
    - For each product in the list, calculate the discounted price based on the given discount percentage.
    - The discounted price is computed using the formula:
      Discounted Price = Original Price × (1 - Discount Percentage / 100)
    - Return a list of discounted prices rounded to two decimal places.

Example:

- **Input:** prices = [100.0, 250.0, 75.0], discount_percentage = 20.0
- **Output:** [80.0, 200.0, 60.0]

Constraints:

- The list of prices will contain between 1 and 100 prices.
- The discount percentage will be a float between 0 and 100.
- The function should handle rounding to two decimal places correctly.

Problem-42: Frog Jump Calculation

Objective: Write a program that calculates the number of jumps a frog needs to cover a specified distance, given the distance it can jump in each attempt.

Function Signature:

```
def calculate_jumps(d: int, s: int) -> int:
    pass
```

Instructions:

1. **Calculate Jumps:**
    - Determine the number of jumps required for the frog to cover a distance of d, given that each jump covers a distance of s.
    - If the distance d is less than or equal to the distance s, the frog only needs one jump.
    - Otherwise, calculate the total number of jumps needed to cover the distance.

Example:

- **Input:** d = 20, s = 7
- **Output:** 3

Explanation: The frog can jump 7 units per jump. To cover a distance of 20 units, the frog will need 3 jumps:

- Jump 1: Covers 7 units, remaining distance = 13
- Jump 2: Covers another 7 units, remaining distance = 6
- Jump 3: Covers the remaining 6 units

Constraints:

- Both d and s will be positive integers.
- The distance s will be greater than 0 and less than or equal to the distance d.

Problem-43: Automatic Coin Exchange

Objective: Write a program that determines the number of each type of coin needed to make up a specified amount of money using coins of denominations 10, 5, 2, and 1.

Function Signature:

```
def calculate_coins(amount: int) -> Tuple[int, int, int, int]:
    pass
```

Instructions:

1. **Determine Coin Counts:**
    - Given an amount of money, determine how many coins of each denomination (10, 5, 2, and 1) are needed to make up that amount.
    - Use the largest denomination coins first to minimize the total number of coins.
2. **Output Format:**
    - Return a tuple containing four integers representing the count of 10-unit coins, 5-unit coins, 2-unit coins, and 1-unit coins respectively.

Example:

- **Input:** amount = 28
- **Output:** (2, 1, 1, 1)

Explanation: To make up 28 units:

- Use 2 coins of 10 units each (20 units total)
- Use 1 coin of 5 units (5 units total)
- Use 1 coin of 2 units (2 units total)
- Use 1 coin of 1 unit (1 unit total)

This makes a total of 28 units.

Constraints:

- The amount will be a positive integer.
- The function should handle amounts up to the limits of typical integer values.

Problem-44: Accumulated Investment Calculation

Objective: Write a program that calculates the accumulated amount of an investment over a period of 5 years, given the initial investment amount, annual interest rate, and number of years.

Function Signature:

```
def calculate_investment_growth(principal: float, annual_rate: float, years: int) -> List[float]:
    pass
```

Instructions:

1. **Calculate Investment Growth:**
   - Compute the accumulated amount for each year up to 5 years using the formula for compound interest:

$$A = P \times \left(1 + \frac{r}{100}\right)^n$$

     Where:

     - A is the accumulated amount.
     - P is the principal amount (initial investment).
     - r is the annual interest rate (as a percentage).
     - n is the number of years.
   - Return the accumulated amounts for each year from 1 to 5.

Example:

- **Input:** principal = 1000, annual_rate = 5, years = 5
- **Output:** [1050.0, 1102.5, 1157.63, 1215.51, 1276.28]

Explanation:

- For the first year: $1000 \times (1 + \frac{5}{100})^1 = 1050.0$
- For the second year: $1000 \times (1 + \frac{5}{100})^2 = 1102.5$
- For the third year: $1000 \times (1 + \frac{5}{100})^3 = 1157.63$
- For the fourth year: $1000 \times (1 + \frac{5}{100})^4 = 1215.51$
- For the fifth year: $1000 \times (1 + \frac{5}{100})^5 = 1276.28$

Constraints:

- The principal will be a positive floating-point number.
- The annual_rate will be a positive floating-point number representing the percentage interest rate.
- The years will be a positive integer, but only the first 5 years are considered for output.

Problem-45: Calculate Annual Rate of Return

Objective: Write a program that calculates the annual rate of return (in percentage) based on the initial investment, the investment value after n years, and the number of years of investment.

Function Signature:

```
def calculate_annual_return(initial_investment: float, final_investment: float, years: int) -> float:
    pass
```

Instructions:

1. **Calculate Annual Rate of Return:**
   - The annual rate of return is calculated using the formula:

$$\text{Annual Return} = \left(\frac{\text{Final Investment}}{\text{Initial Investment}}\right)^{\frac{1}{n}} - 1$$

   - Multiply the result by 100 to convert it into a percentage.
2. **Input Parameters:**
   - initial_investment: The amount of money initially invested.
   - final_investment: The amount of money after n years of investment.
   - years: The number of years the money was invested.
3. **Output:**
   - Return the annual rate of return as a percentage (rounded to two decimal places).

Example:

- **Input:** initial_investment = 1000, final_investment = 1500, years = 5
- **Output:** 8.45

Explanation:

- The formula calculates the compound annual growth rate (CAGR), which in this case is 8.45%.

Constraints:

- initial_investment and final_investment will be positive numbers.
- years will be a positive integer greater than 0.

Problem-46: Calculate the Perimeter of a Triangle

Objective: Write a program that calculates the perimeter of a triangle given the lengths of its three sides.

Function Signature:

```
def calculate_perimeter(a: float, b: float, c: float) -> float:
    pass
```

Instructions:

1. **Calculate Perimeter:**
   - The perimeter of a triangle is the sum of the lengths of its three sides.
   - Formula: Perimeter=a+b+c
2. **Input Parameters:**
   - a: Length of the first side of the triangle.
   - b: Length of the second side of the triangle.
   - c: Length of the third side of the triangle.
3. **Output:**
   - Return the perimeter of the triangle.

Example:

- **Input:** a = 3, b = 4, c = 5
- **Output:** 12

Explanation:

- The perimeter is calculated as the sum of the sides: 3+4+5=123 + 4 + 5 = 123+4+5=12.

Constraints:

- The values of a, b, and c will be positive numbers.
- The input values will form a valid triangle (i.e., the sum of any two sides will be greater than the third side).

Problem-47: Calculate the Area of a Rectangle

Objective: Write a program that calculates the area of a rectangle given its width and length.

Function Signature:

```
def calculate_rectangle_area(width: float, length: float) -> float:
    pass
```

Instructions:

1. **Calculate Area:**
    - The area of a rectangle is calculated using the formula:

      $$\text{Area} = \text{Width} \times \text{Length}$$

    - The function should return the area of the rectangle.
2. **Input Parameters:**
    - width: The width of the rectangle.
    - length: The length of the rectangle.
3. **Output:**
    - Return the area of the rectangle as a floating-point number.

Example:

- **Input:** width = 5, length = 10
- **Output:** 50.0

Explanation:

- The area is calculated as 5×10=505 \times 10 = 505×10=50 square units.

Constraints:

- Both width and length will be positive numbers greater than 0.
- The function should handle floating-point numbers to allow for non-integer dimensions.

Problem-48: Calculate Cumulative Sum

Objective: Write a program that calculates the cumulative sum of all integers from 1 up to a given integer n.

Function Signature:

```
def cumulative_sum(n: int) -> int:
    pass
```

Instructions:

1. **Calculate Cumulative Sum:**
   - The cumulative sum is calculated by summing all integers from 1 to n.
   - The formula for the cumulative sum of the first n natural numbers is:

$$\text{Cumulative Sum} = \frac{n \times (n+1)}{2}$$

2. **Input Parameter:**
   - n: A positive integer representing the upper limit of the sum.
3. **Output:**
   - Return the cumulative sum as an integer.

Example:

- **Input:** n = 5
- **Output:** 15

Explanation:

- The sum of the integers from 1 to 5 is 1+2+3+4+5=15

Constraints:

- The input n will be a positive integer greater than 0.

Problem-49: Toggle Case of Characters in a String

Objective: Write a program that takes an input string and toggles the case of each character. If a character is in lowercase, convert it to uppercase. If it is in uppercase, convert it to lowercase.

Function Signature:

```
def toggle_case(s: str) -> str:
    pass
```

Instructions:

1. **Toggle Case:**
   - Iterate through each character in the input string.
   - If the character is lowercase, convert it to uppercase.
   - If the character is uppercase, convert it to lowercase.
   - Leave any non-alphabetic characters unchanged.
2. **Input Parameters:**
   - s: A string that may contain both uppercase and lowercase letters, as well as other characters.
3. **Output:**
   - Return the modified string with the case of each alphabetic character toggled.

Example:

- **Input:** "Hello World!"
- **Output:** "hELLO wORLD!"

Explanation:

- The function converts 'H' to 'h', 'e' to 'E', 'l' to 'L', and so on. Non-alphabetic characters like spaces and exclamation marks remain unchanged.

Constraints:

- The input string can be of any length, including an empty string.
- The string may contain letters, numbers, spaces, and special characters.

Problem-50: Find Words of Specified Length

Objective: Write a program that takes a list of words and an integer representing the desired word length. The program should return a list of words from the input list that have the specified length.

Function Signature:

```
def find_words_of_length(words: List[str], length: int) -> List[str]:
    pass
```

Instructions:

1. **Filter Words:**
    - Iterate through the list of words and check the length of each word.
    - If a word matches the specified length, include it in the result.
2. **Input Parameters:**
    - words: A list of strings where each string is a word.
    - length: An integer specifying the desired length of the words to be found.
3. **Output:**
    - Return a list of words that match the specified length.

Example:

- **Input:** words = ["apple", "banana", "cherry", "date", "fig", "grape"], length = 5
- **Output:** ["apple", "grape"]

Explanation:

- The words "apple" and "grape" have exactly 5 characters, so they are included in the output list.

Constraints:

- The list of words can contain any number of words, including an empty list.
- All words in the list will consist of alphabetic characters only.
- The length parameter will be a positive integer.

Problem-51: Separate Even and Odd Numbers

Objective: Write a program that takes a list of 10 integers and separates them into even and odd numbers. Implement a function that checks whether each number is even or odd. If a number is even, it should be added to an even_list. If a number is odd, it should be added to an odd_list.

Function Signature:

```
def separate_even_odd(numbers: List[int]) -> Tuple[List[int], List[int]]:
    pass
```

Instructions:

1. **Check Even or Odd:**
   - Iterate through the list of numbers and determine if each number is even or odd.
   - If a number is even, append it to the even_list.
   - If a number is odd, append it to the odd_list.
2. **Input Parameters:**
   - numbers: A list of 10 integers.
3. **Output:**
   - Return a tuple containing two lists: one for even numbers (even_list) and one for odd numbers (odd_list).

Example:

- **Input:** numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
- **Output:** ([2, 4, 6, 8, 10], [1, 3, 5, 7, 9])

Explanation:

- The even numbers [2, 4, 6, 8, 10] are stored in the even_list.
- The odd numbers [1, 3, 5, 7, 9] are stored in the odd_list.

Constraints:

- The input list will always contain exactly 10 integers.
- All numbers will be non-negative integers.

Problem-52: Group Numbers by Unit Digit

Objective: Write a program that takes 10 integers as input. Create a function that checks the unit digit of each number and groups the numbers into separate lists based on their unit digits.

Function Signature:

```
def group_by_unit_digit(numbers: List[int]) -> List[List[int]]:
    pass
```

Instructions:

1. **Group Numbers:**
   - Iterate through the list of numbers.
   - For each number, check its unit digit (the last digit of the number).
   - Group numbers with the same unit digit together in a list.
2. **Input Parameters:**
   - numbers: A list of 10 integers.
3. **Output:**
   - Return a list of 10 lists. Each sublist should contain the numbers that share the same unit digit. The first sublist corresponds to the unit digit 0, the second sublist corresponds to the unit digit 1, and so on.

Example:

- **Input:** numbers = [21, 34, 51, 23, 37, 44, 60, 11, 91, 99]
- **Output:** [[60], [21, 51, 11, 91], [], [23], [34, 44], [], [], [37], [], [99]]

Explanation:

- The numbers 21, 51, 11, and 91 all have a unit digit of 1, so they are grouped together in the second list.
- The number 60 has a unit digit of 0, so it is in the first list.
- The output consists of 10 lists corresponding to unit digits from 0 to 9.

Constraints:

- The input list will always contain exactly 10 integers.
- The integers can be positive or negative.

Problem-53: Convert Buddhist Era (B.E.) to Gregorian Year (A.D.)

Objective: Write a program that takes a year in the Buddhist Era (B.E.) and converts it to the corresponding Gregorian year (A.D.).

Function Signature:

```
def convert_be_to_ad(be_year: int) -> int:
    pass
```

Instructions:

1. **Conversion Logic:**
   - The Buddhist Era (B.E.) is 543 years ahead of the Gregorian calendar (A.D.).
   - To convert a B.E. year to an A.D. year, subtract 543 from the B.E. year.
2. **Input Parameters:**
   - be_year: An integer representing a year in the Buddhist Era (B.E.).
3. **Output:**
   - Return an integer representing the corresponding year in the Gregorian calendar (A.D.).

Example:

- **Input:** be_year = 2567
- **Output:** 2024

Explanation:

- The year 2567 in the Buddhist Era (B.E.) corresponds to the year 2024 in the Gregorian calendar (A.D.) because 2567 - 543 = 2024.

Constraints:

- The input year will be a positive integer representing a valid year in the Buddhist Era (B.E.).

Problem-54: Convert Temperature Between Fahrenheit and Celsius

Objective: Write two functions that convert temperatures between Fahrenheit and Celsius. The first function should convert a temperature from Fahrenheit to Celsius, and the second function should convert a temperature from Celsius to Fahrenheit.

Function Signatures:

```
def fahrenheit_to_celsius(fahrenheit: float) -> float:
    pass
def celsius_to_fahrenheit(celsius: float) -> float:
    pass
```

Instructions:

1. **Conversion Logic:**
    - **Fahrenheit to Celsius:**
        - Use the formula:

$$\text{Celsius} = (\text{Fahrenheit} - 32) \times \frac{5}{9}$$

    - **Celsius to Fahrenheit:**
        - Use the formula:

$$\text{Fahrenheit} = \text{Celsius} \times \frac{9}{5} + 32$$

2. **Input Parameters:**
    - **fahrenheit_to_celsius:**
        - fahrenheit: A floating-point number representing a temperature in Fahrenheit.
    - **celsius_to_fahrenheit:**
        - celsius: A floating-point number representing a temperature in Celsius.
3. **Output:**
    - **fahrenheit_to_celsius:** Return a floating-point number representing the corresponding temperature in Celsius.
    - **celsius_to_fahrenheit:** Return a floating-point number representing the corresponding temperature in Fahrenheit.

Examples:

- **Example 1:**
  - **Input:** fahrenheit = 32.0
  - **Output:** celsius = 0.0
  - **Explanation:** 32°F is equivalent to 0°C.
- **Example 2:**
  - **Input:** celsius = 100.0
  - **Output:** fahrenheit = 212.0
  - **Explanation:** 100°C is equivalent to 212°F.

Constraints:

- The input temperatures will be valid floating-point numbers, and the conversion should be accurate to at least one decimal place.

Problem-55: Convert Thai Baht (THB) to Multiple Currencies

Objective: Write a program that converts an amount in Thai Baht (THB) to one of five different currencies using predefined exchange rates.

Function Signature:

```
def convert_thb_to_currency(amount: float, to_currency: str) -> float:
    pass
```

Instructions:

1. **Supported Currencies:**
   - USD (United States Dollar)
   - EUR (Euro)
   - GBP (British Pound Sterling)
   - JPY (Japanese Yen)
   - AUD (Australian Dollar)
2. **Conversion Logic:**
   - Use predefined exchange rates for conversions. For example:
     - 1 THB = 0.030 USD
     - 1 THB = 0.027 EUR
     - 1 THB = 0.023 GBP
     - 1 THB = 3.4 JPY
     - 1 THB = 0.045 AUD
   - The function should convert the given amount from Thai Baht to the specified target currency.

3. **Input Parameters:**
    - amount: A floating-point number representing the amount of money in Thai Baht (THB) to convert.
    - to_currency: A string representing the currency code of the target currency (e.g., "USD").
4. **Output:**
    - Return a floating-point number representing the converted amount in the target currency.

Example:

- **Example 1:**
    - **Input:** amount = 1000.0, to_currency = "USD"
    - **Output:** 30.0
    - **Explanation:** 1000 THB is equivalent to 30 USD using the exchange rate 1 THB = 0.030 USD.
- **Example 2:**
    - **Input:** amount = 1000.0, to_currency = "JPY"
    - **Output:** 3400.0
    - **Explanation:** 1000 THB is equivalent to 3400 JPY using the exchange rate 1 THB = 3.4 JPY.

Constraints:

- The input amount will be a positive floating-point number.
- The to_currency will always be a valid currency code among the supported ones.

Problem-56: Currency Conversion Between 5 Currencies

Objective: Write a program that converts an amount from one currency to another using predefined exchange rates. The program should support conversion between 5 different currencies.

Function Signature:

```
def convert_currency(amount: float, from_currency: str, to_currency: str) -> float:
    pass
```

Instructions:

1. **Supported Currencies:**
    - USD (United States Dollar)
    - EUR (Euro)
    - GBP (British Pound Sterling)
    - JPY (Japanese Yen)
    - THB (Thai Baht)
2. **Conversion Logic:**
    - Use predefined exchange rates for conversions. For simplicity, here are some example rates:
        - 1 USD = 0.85 EUR
        - 1 USD = 0.75 GBP
        - 1 USD = 110.0 JPY
        - 1 USD = 32.0 THB
    - The function should handle conversions between any pair of the supported currencies.
3. **Input Parameters:**
    - amount: A floating-point number representing the amount of money to convert.
    - from_currency: A string representing the currency code of the amount to convert from (e.g., "USD").
    - to_currency: A string representing the currency code of the target currency (e.g., "EUR").
4. **Output:**
    - Return a floating-point number representing the converted amount in the target currency.

Example:

- **Example 1:**
    - **Input:** amount = 100.0, from_currency = "USD", to_currency = "EUR"
    - **Output:** 85.0
    - **Explanation:** 100 USD is equivalent to 85 EUR using the exchange rate 1 USD = 0.85 EUR.
- **Example 2:**
    - **Input:** amount = 1000.0, from_currency = "JPY", to_currency = "THB"
    - **Output:** 290.91
    - **Explanation:** 1000 JPY is equivalent to 290.91 THB using the exchange rates provided.

Constraints:

- The input amount will be a positive floating-point number.
- The from_currency and to_currency will always be valid currency codes among the supported ones.

Problem-57: Compare the Length of Two Strings

Objective: Write a function that takes two strings and compares their lengths. The function should return which string is longer and by how many characters.

Function Signature:

```
def compare_string_lengths(str1: str, str2: str) -> str:
    pass
```

Instructions:

1. **Comparison Logic**:
    - The function should calculate the length of both input strings.
    - Determine which string is longer.
    - Calculate the difference in length between the two strings.
2. **Input Parameters**:
    - str1: A string representing the first text.
    - str2: A string representing the second text.
3. **Output**:
    - Return a string indicating which text is longer and by how many characters.

Example:

- **Input**: str1 = "apple", str2 = "banana"
- **Output**: "The second string is longer by 1 character(s)."

Explanation:

- The length of "apple" is 5 characters.
- The length of "banana" is 6 characters.
- Since "banana" is longer by 1 character, the output is "The second string is longer by 1 character(s)."

Constraints:

- The input strings will not be empty and will contain only printable characters.

Problem-58: Find All Divisors of a Given Number

Objective: Write a function that takes an integer as input and returns a list of all the divisors of that number.

Function Signature:

```
def find_divisors(num: int) -> list:
    pass
```

Instructions:

1. **Divisor Definition**:
    - A divisor of a number is an integer that divides the number completely without leaving a remainder.
    - For example, for the number 12, the divisors are 1, 2, 3, 4, 6, and 12.
2. **Input Parameters**:
    - num: An integer representing the number for which divisors need to be found.
3. **Output**:
    - Return a list of integers, where each integer is a divisor of the input number.
4. **Constraints**:
    - The input number will be a positive integer.

Example:

- **Input**: num = 12
- **Output**: [1, 2, 3, 4, 6, 12]

Explanation:

- The divisors of 12 are 1, 2, 3, 4, 6, and 12 because each of these numbers divides 12 without leaving a remainder.

Problem-59: Calculate the Number of Days Between Two Gregorian Dates

Objective: Write a program that calculates the number of days between two dates, both provided in the Gregorian calendar (A.D.).

Function Signature:

```
def days_between_dates(date1: str, date2: str) -> int:
    pass
```

Instructions:

1. **Input Parameters**:
    - date1: A string representing the first date in the format "YYYY-MM-DD".
    - date2: A string representing the second date in the format "YYYY-MM-DD".
2. **Output**:
    - Return an integer representing the number of days between date1 and date2.
    - The function should return a positive number regardless of the order of the dates.
3. **Example**:
    - **Input**:
        - date1 = "2024-08-01"
        - date2 = "2024-08-10"
    - **Output**: 9

Explanation:

- The number of days between August 1, 2024, and August 10, 2024, is 9 days.

Constraints:

- The input dates will be valid Gregorian calendar dates.
- The dates can be in any order, and the function should handle both cases.

Problem-60: Calculate Parking Fee

Objective: Write a function that calculates the parking fee based on the number of hours and minutes a car is parked.

Function Signature:

```
def calculate_parking_fee(hours: int, minutes: int) -> int:
    pass
```

Instructions:

1. **Fee Calculation Logic:**
    - The first hour of parking is free.
    - For each subsequent hour, the fee is 50 Thai Baht.
    - Any fraction of an hour is considered as a full hour.
2. **Input Parameters:**
    - hours: An integer representing the number of full hours the car is parked.
    - minutes: An integer representing the number of additional minutes the car is parked beyond the full hours.
3. **Output:**
    - Return an integer representing the total parking fee in Thai Baht.

Example:

- **Input:** hours = 2, minutes = 30
- **Output:** 50

Explanation:

- The first hour is free.
- The remaining time is 1 hour and 30 minutes.
- Since any fraction of an hour is rounded up, 1 hour and 30 minutes is counted as 2 hours.
- The fee for 2 hours is 2 * 50 = 100 Thai Baht.

Constraints:

- The input values for hours and minutes will be non-negative integers.

Problem-61: Calculate the Total Payment After Applying Discounts

Objective: Write a program that calculates the total payment amount after applying discounts based on the total sum of multiple bills. The program will first take an input for the number of bills and then the amount paid in each bill. The discount conditions are as follows:

- If the total sum of all bills is 10,000 Baht or more, apply a 20% discount.
- If the total sum is 5,000 Baht or more, apply a 10% discount.
- If the total sum is 1,000 Baht or more, apply a 5% discount.
- No discount applies if the total sum is below 1,000 Baht.

Function Signature:

```
def calculate_total_payment(num_bills: int, bills: List[float]) -> float:
    pass
```

Instructions:

1. **Input Parameters:**
   - num_bills: An integer representing the number of bills.
   - bills: A list of floats, where each float represents the amount paid in a single bill.
2. **Output:**
   - Return a float representing the total payment amount after applying the applicable discount.
3. **Discount Logic:**
   - Calculate the sum of all bills.
   - Apply the discount based on the total sum according to the conditions provided.

Example:

- **Input:**
  num_bills = 3
  bills = [3000, 4000, 3500]
- **Output:**
  8400.0

Explanation:

- The total sum of the bills is 3000+4000+3500=10500Baht.
- Since the total is 10,500 Baht, a 20% discount is applied.
- The discounted total is 10500−(0.2×10500)=8400 Baht.

Constraints:

- The number of bills will be a positive integer.
- Each bill amount will be a positive float or integer value.

Problem-62: Speeding Violation Detection and Fine Issuance

Objective: Write a program that detects speeding violations based on the speed of a vehicle and issues a fine according to the speed limit regulations.

Function Signature:

```
def calculate_speeding_fine(speed: float, speed_limit: float) -> str:
    pass
```

Instructions:

1. **Input Parameters:**
   - speed: A float representing the speed of the vehicle (in km/h).
2. **Output:**
   - Return a string indicating whether the driver is fined, and if so, the amount of the fine.
3. **Fine Calculation:**
   - No fine if the speed is within the speed limit or below.
   - Fine structure based on the amount the speed exceeds the limit:
     - 90-120 km/h over the limit: Fine of 500 Baht.
     - 121-140 km/h over the limit: Fine of 1,000 Baht.
     - 141-160 km/h over the limit: Fine of 1,500 Baht.
     - More than 160 km/h over the limit: Fine of 2,000 Baht.
4. **Conditions:**
   - If the speed is equal to or less than the speed limit, return "No fine."
   - If the speed exceeds the limit, return the amount of the fine in the format: "Fine: X Baht."

Example:

- **Input:**
  speed = 141
- **Output:**
  "Fine: 1,500 Baht"

Explanation:

- The vehicle is driving at 80 km/h in a 60 km/h zone, exceeding the limit by 20 km/h.
- According to the fine structure, a 20 km/h over-speeding results in a 1,000 Baht fine.

Constraints:

- The speed and speed limit will be positive float values.

Problem-63: Plan How Many Songs to Listen To

Objective: Write a program that takes the input of how many hours you want to listen to music and calculates how many songs you can listen to, based on an average song length.

Function Signature:

```
def calculate_number_of_songs(hours: float, avg_song_length: float = 3.5) -> int:
    pass
```

Instructions:

1. **Input Parameters:**
   - hours: A float representing the number of hours you plan to listen to music.
   - avg_song_length: A float representing the average length of a song in minutes (default is 3.5 minutes).
2. **Output:**
   - Return an integer representing the number of songs you can listen to within the given hours.
3. **Calculation Logic:**
   - Convert the total listening time from hours to minutes.
   - Divide the total minutes by the average song length to determine the number of songs.

Example:

- **Input:**
  hours = 2 (2 hours)
  avg_song_length = 4 (4 minutes per song)
- **Output:**
  30

Explanation:

- Total listening time is 2×60=120 minutes.
- With an average song length of 4 minutes, you can listen to 120/4=30 songs.

Constraints:

- The number of hours will be a positive float.
- The average song length will be a positive float.

Problem-64: Grade Planning for Semester Courses

Objective: Write a program that helps a student plan the grades they need to achieve in 5 courses they are enrolled in this semester. The program should calculate the minimum letter grades required in each course to achieve a target GPA, using the specified grade points for each letter grade.

Function Signature:

```
def calculate_required_grades(current_gpa: float, target_gpa: float, credits: List[int]) -> List[str]:
    pass
```

Instructions:

1. **Input Parameters:**
   - current_gpa: A float representing the student's current GPA.
   - target_gpa: A float representing the target GPA that the student wants to achieve by the end of the semester.
   - credits: A list of integers representing the credit hours for each of the 5 courses.
2. **Output:**
   - Return a list of 5 strings representing the minimum letter grades required in each course to reach the target GPA.
3. **Grade Calculation Logic:**
   - The GPA is typically calculated as the sum of (grade points ×\times× credit hours) divided by the total credit hours.
   - The program should compute the minimum letter grades needed in the 5 courses based on the credit hours and the target GPA.
   - Use the following grade points:
     - A = 4
     - B+ = 3.5
     - B = 3
     - C+ = 2.5
     - C = 2
     - D+ = 1.5
     - D = 1

Example:

- **Input:**
  current_gpa = 2.8
  target_gpa = 3.2
  credits = [3, 4, 3, 2, 3]
- **Output:**
  ['B+', 'B+', 'B+', 'B+', 'B+']

Explanation:

- The student wants to achieve a target GPA of 3.2.
- Based on the program's calculations, the student needs to aim for a grade of B+ (3.5 points) in each course to reach the target GPA.

Constraints:

- The GPA is usually on a 4.0 scale.
- The list of credits will always have 5 integers, each representing the credit hours for a course.

Problem-65: Calculate Term GPA Based on Grades

Objective: Write a program that calculates the Grade Point Average (GPA) for a term based on the grades for 5 subjects. Each subject has a numeric grade, and you need to determine the GPA based on the following grading scale:

- 0-49: F
- 50-54: D
- 55-59: D+
- 60-64: C
- 65-69: C+
- 70-74: B
- 75-79: B+
- 80-100: A

The GPA is calculated as the average of the corresponding grade points, where the grade points are assigned as follows:

- F: 0
- D: 1
- D+: 1.5
- C: 2
- C+: 2.5
- B: 3
- B+: 3.5
- A: 4

Function Signature:

```
def calculate_gpa(grades: List[int]) -> float:
    pass
```

Instructions:

1. **Conversion Logic:**
    - Convert each numeric grade to the corresponding grade point.
    - Compute the average of these grade points to get the GPA.
2. **Input Parameters:**
    - grades: A list of 5 integers representing the numeric grades for the subjects.
3. **Output:**
    - Return a float representing the GPA for the term.

Example:

- **Input:** grades = [85, 72, 63, 58, 49]
- **Output:** 2.5

Explanation:

- Grades converted to grade points:
  - 85 → A → 4
  - 72 → B → 3
  - 63 → C → 2
  - 58 → D+ → 1.5
  - 49 → F → 0
- GPA Calculation:
  - Average of grade points: (4+3+2+1.5+0)/5=2.5(4 + 3 + 2 + 1.5 + 0) / 5 = 2.5(4+3+2+1.5+0)/5=2.5

Constraints:

- The input list grades will always contain exactly 5 integers, each between 0 and 100.

Problem-66: Restaurant Menu for Egg Dishes

Objective: Write a program for a restaurant that sells only egg dishes. The program should allow users to select items from the menu, specify quantities, and then calculate and display the total cost.

Menu:

- Fried Egg: 7 Baht
- Omelet: 10 Baht
- Boiled Egg: 5 Baht

Function Signature:

```
def calculate_total_cost(fried_eggs: int, omelets: int, boiled_eggs: int) -> int:
    pass
```

Instructions:

1. **Menu Pricing:**
    - Fried Egg: 7 Baht each
    - Omelet: 10 Baht each
    - Boiled Egg: 5 Baht each
2. **Input Parameters:**
    - fried_eggs: An integer representing the quantity of Fried Eggs ordered.
    - omelets: An integer representing the quantity of Omelets ordered.
    - boiled_eggs: An integer representing the quantity of Boiled Eggs ordered.
3. **Output:**
    - Return an integer representing the total cost for the selected items.

Example:

- **Input:** fried_eggs = 2, omelets = 3, boiled_eggs = 1
- **Output:** 37

Explanation:

- Cost for 2 Fried Eggs: 2×7=14 Baht
- Cost for 3 Omelets: 3×10=30 Baht
- Cost for 1 Boiled Egg: 1×5=5 Baht
- Total Cost: 14+30+5=49 Baht

Constraints:

- The input parameters will be non-negative integers.

Problem-67: Calculate Age from Date of Birth

Objective: Write a program that calculates the age of a person based on their date of birth. The program should determine how many years, months, and days have passed since the date of birth up to the current date.

Function Signature:

```
def calculate_age(day_of_birth: int, month_of_birth: int, year_of_birth: int) -> Tuple[int, int, int]:
    pass
```

Instructions:

1. **Input Parameters:**
    - day_of_birth: An integer representing the day of birth (1-31).
    - month_of_birth: An integer representing the month of birth (1-12).
    - year_of_birth: An integer representing the year of birth (e.g., 1990).
2. **Output:**
    - Return a tuple (years, months, days) representing the age in years, months, and days.

Example:

- **Input:** day_of_birth = 15, month_of_birth = 8, year_of_birth = 1990
- **Output:** (33, 0, 21)

Explanation:

- The person was born on August 15, 1990.
- As of today's date (assuming it's August 6, 2024), the person is 33 years, 0 months, and 21 days old.

Constraints:

- The input date of birth will be a valid date.
- The current date is assumed to be today's date.

Problem-68: Number Guessing Game

Objective: Write a program that provides a number guessing game with a menu-driven interface.

Function Signature:

```
def number_guessing_game() -> None:
    pass
```

Instructions:

1. **Menu Options:**
    - The program should display a menu with the following options:
        1. **Start New Game**: Start a new guessing game.
        2. **Show Instructions**: Display the rules and instructions for the game.
        3. **Exit**: Exit the program.
2. **Game Logic:**
    - When the user selects **Start New Game**, the program should:
        - Generate a random target number between 1 and 100.
        - Prompt the user to guess the number.
        - Inform the user if their guess is too high, too low, or correct.
        - Allow the user to continue guessing until they guess correctly.
        - After a correct guess, display the number of attempts taken and return to the main menu.
    - When the user selects **Show Instructions**, display the following instructions:

      Welcome to the Number Guessing Game!
      1. The system will randomly select a number between 1 and 100.
      2. Your task is to guess the number.
      3. After each guess, you will be informed if your guess is too high, too low, or correct.
      4. Keep guessing until you find the correct number.
      5. The game will show you the number of attempts you took to guess the number.
    - When the user selects **Exit**, terminate the program.

Input:

- The program should interact with the user through input prompts. The user provides input via the menu and guesses.

Output:

- Display appropriate messages based on the user's menu selection and game status.

Example:

Menu:
1. Start New Game
2. Show Instructions
3. Exit

Please select an option: 1

Guess a number between 1 and 100: 50
Your guess is too low. Try again!

Guess a number between 1 and 100: 75
Your guess is too high. Try again!

Guess a number between 1 and 100: 62
Congratulations! You guessed the number correctly in 3 attempts.

Returning to main menu...

Menu:
1. Start New Game
2. Show Instructions
3. Exit

Please select an option: 3

Thank you for playing! Goodbye!

Constraints:

- The menu options should be limited to the given choices.
- The number of attempts should be counted accurately.
- The game should handle invalid inputs gracefully.

Problem-69: Personal Finance Tracker

Objective:
Create a program that manages personal finances by tracking income, expenses, and providing a balance summary.

Functionality Requirements:

1. **Income Menu:**
   - Add a new income entry.
   - Display a list of all income entries.
2. **Expenses Menu:**
   - Add a new expense entry.
   - Display a list of all expense entries.
3. **Balance Menu:**
   - Calculate and display the current balance based on the total income and total expenses.

Details:

1. **Add Income Entry:**
   - Input: Amount (positive number), Description (string).
   - Output: Confirmation of income entry added.
2. **Add Expense Entry:**
   - Input: Amount (positive number), Description (string).
   - Output: Confirmation of expense entry added.
3. **Display Income Entries:**
   - Output: List all income entries with their amounts and descriptions.
4. **Display Expense Entries:**
   - Output: List all expense entries with their amounts and descriptions.
5. **Calculate Balance:**
   - Output: Current balance, which is the total income minus the total expenses.

Function Signature:

```
def add_income(amount: float, description: str) -> None:
    pass

def add_expense(amount: float, description: str) -> None:
    pass

def display_income_entries() -> None:
    pass

def display_expense_entries() -> None:
    pass

def calculate_balance() -> float:
    pass
```

Constraints:

- Amounts for income and expenses are positive numbers.
- Descriptions are non-empty strings.
- The program should handle multiple entries and perform calculations accurately.

Example Usage:

1. Adding income:

   ```
   add_income(1000.0, "Salary")
   ```

2. Adding expense:

   ```
   add_expense(150.0, "Groceries")
   ```

3. Displaying income entries:

   ```
   display_income_entries()
   # Output: ["Salary: 1000.0"]
   ```

4. Displaying expense entries:

   ```
   display_expense_entries()
   # Output: ["Groceries: 150.0"]
   ```

5. Calculating balance:

   ```
   balance = calculate_balance()
   # Output: 850.0
   ```

Problem-70: Bank Account Management System

Objective: Write a program to simulate a bank account management system with a menu-driven interface. The program should allow the user to perform the following operations:

1. **Deposit Money**: When this menu option is selected, the program should prompt the user to enter the amount of money to deposit. It should then add this amount to the account balance and display the current balance.
2. **Withdraw Money**: When this menu option is selected, the program should prompt the user to enter the amount of money to withdraw. If the account balance is insufficient, the program should display a warning message and should not allow the withdrawal.
3. **Check Balance**: When this menu option is selected, the program should display the current account balance.
4. **Exit Program**: When this menu option is selected, the program should terminate.

Function Signature:

```
def bank_account_management():
    pass
```

Instructions:

1. **Initialize**: Start with an initial balance of 0.
2. **Menu Options**: The program should continuously display a menu with the following options until the user chooses to exit:
    1. Deposit Money
    2. Withdraw Money
    3. Check Balance
    4. Exit Program
3. **Input Handling**:
    - For Deposit Money: Prompt the user to enter the amount to deposit.
    - For Withdraw Money: Prompt the user to enter the amount to withdraw.
    - For Check Balance: Display the current balance.
    - For Exit Program: Terminate the program.

Example: If the user selects:

- **Deposit Money** and enters 1000, the balance should be updated to 1000.
- **Withdraw Money** and enters 500, the balance should be updated to 500.
- **Check Balance** should display 500.

Constraints:

- The deposit and withdrawal amounts should be positive integers.
- The withdrawal amount should not exceed the current balance.

Problem-71: Person Class

Objective: Write a program to define a Person class with attributes for a person's name and age.

Class Definition:

```
class Person:
    def __init__(self, name: str, age: int):
        """
        Initialize a new Person instance with name and age.

        Parameters:
        name (str): The name of the person.
        age (int): The age of the person.
        """
        pass

    def get_name(self) -> str:
        """
        Return the name of the person.

        Returns:
        str: The name of the person.
        """
        pass

    def get_age(self) -> int:
        """
        Return the age of the person.

        Returns:
        int: The age of the person.
        """
        pass

    def set_name(self, name: str) -> None:
        """
        Set the name of the person.

        Parameters:
        name (str): The new name of the person.
        """
        pass

    def set_age(self, age: int) -> None:
        """
        Set the age of the person.

        Parameters:
```

```
    age (int): The new age of the person.
    """
    pass
```

Instructions:

1. **Class Attributes:**
    - name (str): The name of the person.
    - age (int): The age of the person.
2. **Methods:**
    - __init__(self, name: str, age: int): Constructor to initialize the name and age attributes.
    - get_name(self) -> str: Returns the name of the person.
    - get_age(self) -> int: Returns the age of the person.
    - set_name(self, name: str) -> None: Updates the name of the person.
    - set_age(self, age: int) -> None: Updates the age of the person.
3. **Constraints:**
    - name should be a non-empty string.
    - age should be a non-negative integer.

Example:

```
# Create a Person instance
person = Person("Alice", 30)

# Get person's details
print(person.get_name())  # Output: Alice
print(person.get_age())   # Output: 30

# Update person's details
person.set_name("Bob")
person.set_age(35)

# Get updated details
print(person.get_name())  # Output: Bob
print(person.get_age())   # Output: 35
```

Problem-72: Circle Class

Objective: Write a Python class named Circle that models a circle with a given radius. The class should include methods for calculating the area and the circumference of the circle.

Class Definition:

```
class Circle:
    def __init__(self, radius: float):
        """
        Initialize a Circle object with the given radius.

        Parameters:
        radius (float): The radius of the circle.
        """
        pass

    def area(self) -> float:
        """
        Calculate and return the area of the circle.

        Returns:
        float: The area of the circle.
        """
        pass

    def circumference(self) -> float:
        """
        Calculate and return the circumference of the circle.

        Returns:
        float: The circumference of the circle.
        """
        pass
```

Instructions:

1. **Initialization:**
   - The __init__ method should initialize the Circle object with a given radius. The radius is a floating-point number.
2. **Area Calculation:**
   - Implement the area method to compute the area of the circle. The area of a circle is given by the formula:

$$\text{Area} = \pi \times \text{radius}^2$$

   - Use the value of π\piπ from the math module.

3. **Circumference Calculation:**
    - Implement the circumference method to compute the circumference of the circle. The circumference of a circle is given by the formula:

$$\text{Circumference} = 2 \times \pi \times \text{radius}$$

    - Use the value of π\piπ from the math module.

Constraints:

- The radius will be a positive floating-point number.

Example:

```
# Example Usage
circle = Circle(5.0)
print(circle.area())       # Output: 78.53981633974483 (approximately)
print(circle.circumference()) # Output: 31.41592653589793 (approximately)
```

Problem-73: Book Class

Objective: Write a Python program that defines a Book class and provides a menu-driven interface to interact with book objects.

Class Signature:

```
class Book:
    def __init__(self, name: str, status: str):
        pass

    def __str__(self) -> str:
        pass
```

Instructions:

1. **Attributes:**
    - name (str): The name of the book.
    - status (str): The status of the book (e.g., "available", "checked out").
2. **Constructor:**
    - The constructor __init__ should initialize the name and status attributes with the provided values.
3. **Method:**
    - Implement the __str__ method to return a string representation of the book in the format:

      "Book Name: [name], Status: [status]"

4. **Menu Interface:**
    - Provide a menu-driven interface to interact with book objects. The menu should offer options to:
        1. Create a new book
        2. Display the book's information
        3. Update the book's status
        4. Exit the program
5. **Input Handling:**
    - The user should be able to enter book details and choose options from the menu.

Example Menu:

1. Create a new book
2. Display book information
3. Update book status
4. Exit

Constraints:

- The name and status attributes will be strings.
- The status attribute should be descriptive of the book's availability.

Problem-74: Car Class

Objective: Write a Python program that defines a Car class and provides a menu-driven interface to interact with car objects.

Class Signature:

```
class Car:
    def __init__(self, brand: str, model: str, year: int, color: str):
        pass

    def display_details(self) -> None:
        pass
```

Instructions:

1. **Attributes:**
    - brand (str): The brand of the car (e.g., "Toyota").
    - model (str): The model of the car (e.g., "Camry").
    - year (int): The year of manufacture of the car (e.g., 2022).
    - color (str): The color of the car (e.g., "Blue").
2. **Constructor:**
    - The constructor __init__ should initialize the brand, model, year, and color attributes with the provided values.

3. **Method:**
   - Implement the display_details method to print the details of the car in the following format:

     Brand: [brand]
     Model: [model]
     Year: [year]
     Color: [color]

4. **Menu Interface:**
   - Provide a menu-driven interface to interact with car objects. The menu should offer options to:

     1. Create a new car
     2. Display car details
     3. Exit

5. **Input Handling:**
   - The user should be able to enter car details and choose options from the menu.

Example Menu:

1. Create a new car
2. Display car details
3. Exit

Constraints:

- The brand, model, and color attributes are non-empty strings.
- The year attribute is a positive integer representing a valid year.
- Ensure that the menu displays properly and that the user can interact with the options.

Problem-75: Pet Class for Different Animals

Objective: Write a program that defines a class for pets, specifically for dogs, cats, and birds. Each type of pet should have attributes such as breed, color, name, height, and weight. The program should also include a menu interface that allows the user to select which type of pet they want to store information about.

Class Signature

```
class Pet:
    def __init__(self, species: str, breed: str, color: str, name: str, height: float, weight: float):
        pass

    def display_info(self):
        pass
```

Instructions:

1. **Attributes:**
    - **species**: A string representing the type of animal (e.g., 'Dog', 'Cat', 'Bird').
    - **breed**: A string representing the breed of the pet.
    - **color**: A string representing the color of the pet.
    - **name**: A string representing the name of the pet.
    - **height**: A float representing the height of the pet in centimeters.
    - **weight**: A float representing the weight of the pet in kilograms.

2. **Constructor (__init__ method):**
    - The constructor should initialize the above attributes for a pet instance.

3. **Methods:**
    - display_info(): This method should print out all the details of the pet.

4. **Menu Interface:**
    - The program should present a menu where the user can select the type of pet they want to store information about (Dog, Cat, Bird).
    - After selecting the pet type, prompt the user to input the details for the pet and then store the information using the Pet class.
    - After storing the information, display the pet's information using the display_info() method.

Example:

```
# Example Input (Assume user selects 'Dog')
species = "Dog"
breed = "Labrador"
color = "Yellow"
name = "Buddy"
height = 60.5
weight = 25.3

# Example Output
# Pet Information:
# Species: Dog
# Breed: Labrador
# Color: Yellow
# Name: Buddy
# Height: 60.5 cm
# Weight: 25.3 kg
```

Constraints:

1. **Species Selection:**
   - The user must select from the predefined options: "Dog", "Cat", or "Bird". Any other input should be considered invalid, and the program should prompt the user to re-enter a valid choice.
2. **Breed, Color, and Name:**
   - These inputs should be non-empty strings. The program should validate that the user has entered a value for these fields and prompt again if any field is left blank.
3. **Height and Weight:**
   - Height and weight should be positive numerical values.
   - Height must be greater than 0 and measured in centimeters.
   - Weight must be greater than 0 and measured in kilograms.
   - If an invalid number (e.g., a negative value or a non-numeric input) is provided, the program should prompt the user to re-enter a valid value.
4. **Input Validation:**
   - The program should implement input validation to handle incorrect data types or empty inputs for all fields.
   - If the user fails to meet the constraints for any field, they should be prompted to enter the information again until the input is valid.

Problem-76: Stats Class for Statistical Operations

Objective: Write a program that defines a class named Stats with an attribute data, which is a list that stores numerical values. The class should include methods to calculate the sum, mean, minimum, and maximum of the data.

Class Signature

```
class Stats:
    def __init__(self, data: list):
        pass

    def sum(self) -> float:
        pass

    def mean(self) -> float:
        pass

    def min(self) -> float:
        pass

    def max(self) -> float:
        pass
```

Instructions:

1. **Attributes:**
   - **data:** A list of numerical values (floats or integers).
2. **Constructor (__init__ method):**
   - The constructor should initialize the data attribute with the list of numbers provided.
3. **Methods:**
   - sum(): Returns the sum of all the numbers in the data list.
   - mean(): Returns the mean (average) of the numbers in the data list.
   - min(): Returns the minimum value in the data list.
   - max(): Returns the maximum value in the data list.
4. **Input Validation:**
   - The data list should only contain numerical values (integers or floats).
   - If the data list is empty, methods should handle this appropriately (e.g., return None or raise an error).

Example:

```
# Example Input
data = [10, 20, 30, 40, 50]

# Example Usage
stats = Stats(data)
print(stats.sum())   # Output: 150
print(stats.mean())  # Output: 30.0
print(stats.min())   # Output: 10
print(stats.max())   # Output: 50
```

Constraints:

1. **Data Type Check:**
    - The data list should only contain integers or floats. If any non-numeric value is included, the constructor should raise a ValueError.
2. **Empty List Handling:**
    - If the data list is empty, methods like sum(), mean(), min(), and max() should return None or handle the case appropriately.
3. **Numeric Operations:**
    - The methods should only operate on valid numeric data and ensure the correct handling of edge cases, such as when all numbers are the same or the list contains only one element.

Problem-77: Currency Conversion Class

Objective: Write a program that defines a class for converting currencies among 5 different types. The class should have attributes to store the amount and currency type, and methods to convert the currency into the other 4 types.

Class Signature

```
class CurrencyConverter:
    def __init__(self, amount: float, currency: str):
        pass

    def convert_to(self, target_currency: str) -> float:
        pass
```

Instructions:

1. **Attributes:**
    - **amount**: A float representing the amount of money.
    - **currency**: A string representing the current currency type (e.g., 'USD', 'EUR', 'GBP', 'JPY', 'THB').
2. **Constructor (__init__ method):**
    - The constructor should initialize the amount and currency attributes for an instance of the CurrencyConverter class.
3. **Methods:**
    - convert_to(target_currency: str) -> float: This method should take a target currency type as input and return the converted amount. Implement conversion rates within the method for the following currencies:
        - **USD**: United States Dollar
        - **EUR**: Euro
        - **GBP**: British Pound
        - **JPY**: Japanese Yen
        - **THB**: Thai Baht
    - The conversion rates can be hardcoded for simplicity. For example:
        - 1 USD = 0.85 EUR
        - 1 USD = 0.75 GBP
        - 1 USD = 110 JPY
        - 1 USD = 32 THB
        - (Similarly define conversion rates between other currencies)
4. **Menu Interface:**
    - The program should present a menu where the user can input an amount, select a currency, and choose a target currency for conversion.
    - The program should then display the converted amount.

Example:

```
# Example Input
amount = 100.0
currency = "USD"
target_currency = "EUR"

# Example Output
# 100.0 USD is equal to 85.0 EUR
```

Constraints:

1. **Amount:**
   - The amount must be a positive float.
   - If the user enters a non-numeric value or a negative number, prompt them to re-enter a valid amount.
2. **Currency and Target Currency:**
   - The input for both the source and target currency must be one of the predefined currency codes: 'USD', 'EUR', 'GBP', 'JPY', 'THB'.
   - If an invalid currency code is entered, prompt the user to re-enter a valid currency.
3. **Conversion Rates:**
   - The conversion rates are hardcoded and must be used as provided.
   - The conversion rates should be accurate up to two decimal places.

Problem-78: Cashier Class for Managing Products

Objective: Write a program that defines a class called Cashier to manage a collection of products. Each product has attributes like name, price, and quantity. The class should include methods to add products, remove products, calculate the total cost of all products, and display the product list.

Class Signature

```
class Cashier:
    def __init__(self):
        pass

    def add_product(self, name: str, price: float, quantity: int):
        pass

    def remove_product(self, name: str):
        pass

    def calculate_total(self) -> float:
        pass

    def display_products(self):
        pass
```

Instructions:

1. **Attributes:**
    - **products**: A dictionary to store product information. Each key is the product name, and the value is another dictionary containing price and quantity.
2. **Constructor (__init__ method):**
    - The constructor should initialize an empty dictionary products to store the product details.
3. **Methods:**
    - **add_product(name: str, price: float, quantity: int):**
        - Adds a product to the products dictionary.
        - If the product already exists, update the quantity and price.
    - **remove_product(name: str):**
        - Removes a product from the products dictionary based on the product name.
        - If the product does not exist, display an appropriate message.
    - **calculate_total() -> float:**
        - Calculates and returns the total cost of all products (sum of price * quantity for each product).
    - **display_products():**
        - Displays a list of all products with their names, prices, and quantities.

4. **Menu Interface:**
    - The program should present a menu to allow the user to:
        1. Add a product.
        2. Remove a product.
        3. Calculate the total cost.
        4. Display all products.
        5. Exit the program.

Example:

```
# Example Input
# Add products
name = "Apple"
price = 0.5
quantity = 10

name = "Banana"
price = 0.3
quantity = 5

# Example Output
# Product List:
# Apple - Price: $0.5, Quantity: 10
# Banana - Price: $0.3, Quantity: 5
# Total Cost: $6.5
```

Constraints:

1. **Product Name:**
    - The product name should be a non-empty string. If the user enters an empty name, prompt them to re-enter a valid name.
2. **Price:**
    - The price must be a positive float. If the user enters a non-numeric value or a negative number, prompt them to re-enter a valid price.
3. **Quantity:**
    - The quantity must be a positive integer. If the user enters a non-numeric value or a negative number, prompt them to re-enter a valid quantity.
4. **Input Validation:**
    - The program should validate inputs for all fields and prompt the user to re-enter data if the inputs do not meet the constraints.

Problem-79: Tree Class

Objective: Write a program that defines a class called Tree to represent various types of trees. Each tree has attributes like species, height, age, and location. The class should include methods to grow the tree (increase its height), change its location, and display information about the tree.

Class Signature

```
class Tree:
    def __init__(self, species: str, height: float, age: int, location: str):
        pass

    def grow(self, growth_amount: float):
        pass

    def change_location(self, new_location: str):
        pass

    def display_info(self):
        pass
```

Instructions:

1. **Attributes:**
    - **species**: A string representing the species of the tree (e.g., "Oak", "Maple").
    - **height**: A float representing the height of the tree in meters.
    - **age**: An integer representing the age of the tree in years.
    - **location**: A string representing the current location of the tree.
2. **Constructor (__init__ method):**
    - The constructor should initialize the species, height, age, and location attributes.
3. **Methods:**
    - **grow(growth_amount: float):**
        - Increases the height of the tree by the specified growth_amount in meters.
    - **change_location(new_location: str):**
        - Changes the location of the tree to the specified new_location.
    - **display_info():**
        - Displays the tree's species, height, age, and location in a readable format.

Example:

```
# Example Input
species = "Oak"
height = 5.0
age = 10
location = "Central Park"

# Example Usage
tree = Tree(species, height, age, location)
tree.grow(0.5)
tree.change_location("Botanical Garden")
tree.display_info()
```

Example Output:

```
plaintext
Copy code
Species: Oak
Height: 5.5 meters
Age: 10 years
Location: Botanical Garden
```

Constraints:

1. **Species:**
   - The species name should be a non-empty string. If the user tries to set an empty string as the species, the program should prompt for a valid name.
2. **Height:**
   - The height must be a positive float. If an attempt is made to set the height to a negative value or zero, the program should reject the input and prompt for a valid height.
3. **Age:**
   - The age must be a positive integer. The program should not allow negative ages or non-numeric values.
4. **Location:**
   - The location should be a non-empty string. If the user tries to set an empty string as the location, the program should prompt for a valid location.
5. **Input Validation:**
   - The program should validate inputs for all attributes and handle incorrect data by prompting the user to re-enter valid data where necessary.

Problem-80: LibraryBook Class with a Menu Interface

Objective: Write a program that defines a class called LibraryBook to manage books in a library. Each book has attributes like title, author, year_published, isbn, and status (whether the book is available or checked out). The class should include methods to check out the book, return the book, and display the book's information. Additionally, implement a menu interface to interact with the library.

Class Signature

```
class LibraryBook:
    def __init__(self, title: str, author: str, year_published: int, isbn: str):
        pass

    def check_out(self):
        pass

    def return_book(self):
        pass

    def display_info(self):
        pass
```

Instructions:

1. **Attributes:**
    - **title**: A string representing the title of the book.
    - **author**: A string representing the author of the book.
    - **year_published**: An integer representing the year the book was published.
    - **isbn**: A string representing the ISBN (International Standard Book Number) of the book.
    - **status**: A string representing the status of the book, either "available" or "checked out".
2. **Constructor (__init__ method):**
    - The constructor should initialize the title, author, year_published, and isbn attributes.
    - The status should be set to "available" by default.
3. **Methods:**
    - **check_out():**
        - Changes the status of the book to "checked out" if the book is currently available. Otherwise, display a message that the book is already checked out.
    - **return_book():**
        - Changes the status of the book to "available" if the book is currently checked out. Otherwise, display a message that the book is already available.

- **display_info():**
  - Displays the book's title, author, year published, ISBN, and current status in a readable format.

4. **Menu Interface:**
   - The program should present a menu to allow the user to:
     1. Add a new book to the library.
     2. Check out a book.
     3. Return a book.
     4. Display information about a specific book.
     5. Display all books in the library.
     6. Exit the program.

Example:

```
# Example Input
title = "The Great Gatsby"
author = "F. Scott Fitzgerald"
year_published = 1925
isbn = "9780743273565"

# Example Usage (Menu)
# 1. Add a new book
# 2. Check out a book
# 3. Return a book
# 4. Display information about a specific book
# 5. Display all books in the library
# 6. Exit

# Example Output (Option 4: Display Information)
Title: The Great Gatsby
Author: F. Scott Fitzgerald
Year Published: 1925
ISBN: 9780743273565
Status: available
```

Constraints:

1. **Title and Author:**
   - **Non-Empty:** The title and author must be non-empty strings. If an empty string is provided, the program should prompt the user to enter a valid value.
   - **String Type:** The title and author should be of string type. Any non-string input should be rejected with a prompt to enter a valid string.
2. **Year Published:**
   - **Positive Integer:** The year published must be a positive integer. If a negative number, zero, or non-integer is provided, the program should prompt the user to enter a valid year.
   - **Past or Present Year:** The year should not be a future year. The program should ensure that the input year is less than or equal to the current year.

3. **ISBN:**
    - **Length:** The ISBN should be either 10 or 13 digits long. If the length does not match these criteria, the program should prompt the user to enter a valid ISBN.
    - **Numeric:** The ISBN should consist of digits only. Any non-numeric input should be rejected with a prompt to enter a valid ISBN.
4. **Status:**
    - **Controlled by Methods:** The status attribute should be controlled strictly by the check_out and return_book methods. Direct modification of this attribute outside of these methods should not be allowed.
5. **Input Validation:**
    - **Error Handling:** The program should implement input validation to handle incorrect data types, empty inputs, and invalid values. If the user enters incorrect data, the program should prompt for a re-entry of valid data.
    - **Menu Selection:** The program should ensure that the menu selection is valid (i.e., a number between 1 and 6). Any invalid selection should result in an error message and a prompt to enter a valid choice.

Problem-81: Advanced String Transformation

Objective:

Create a function that takes a string as input and performs multiple transformations based on specific rules. The function should return the transformed string.

Function Signature:

```
def advanced_string_transformation(s: str) -> str:
    pass
```

Instructions:

1. **Step 1: Reverse Words**
   - Reverse the order of characters in each word of the string. A word is defined as a sequence of non-space characters.
2. **Step 2: Swap Case**
   - Swap the case of each character in the string. Convert all lowercase letters to uppercase and all uppercase letters to lowercase.
3. **Step 3: Replace Vowels**
   - Replace all vowels in the string with the next vowel in the sequence:
     - 'a' -> 'e'
     - 'e' -> 'i'
     - 'i' -> 'o'
     - 'o' -> 'u'
     - 'u' -> 'a'
     - The same rule applies to uppercase vowels.
4. **Step 4: Add Special Characters**
   - Insert a special character ('#') after every third character in the string.

Example:

- **Input:** "Hello World"
- **Step 1 - Reverse Words:** "olleH dlroW"
- **Step 2 - Swap Case:** "OLLEh DLRoW"
- **Step 3 - Replace Vowels:** "ULLih DLRoW"
- **Step 4 - Add Special Characters:** "ULL#ih #DL#RoW"
- **Output:** "ULL#ih #DL#RoW"

Constraints:

- The input string will contain only alphabetic characters and spaces.
- The length of the string will be between 1 and 1000 characters.
- The function should be optimized for efficiency.

Problem-82: Recursive Permutation Finder

Objective:

Create a function that generates all possible permutations of a given string and returns them in a sorted list.

Function Signature:

```
def find_permutations(s: str) -> list:
    pass
```

Instructions:

1. **Generate Permutations:**
   - The function should generate all possible permutations of the input string. A permutation is defined as any possible rearrangement of the characters in the string.
2. **Use Recursion:**
   - Implement the solution using a recursive approach. Avoid using built-in Python libraries like itertools.permutations to solve this problem.
3. **Sort the Results:**
   - Return the permutations as a sorted list. The sorting should be in lexicographical (alphabetical) order.

Example:

Input: "abc"

Output: ["abc", "acb", "bac", "bca", "cab", "cba"]

Input: "dog"

Output: ["dgo", "dog", "gdo", "god", "odg", "ogd"]

Constraints:

- The input string will contain only lowercase alphabetic characters.
- The length of the string will be between 1 and 8 characters.
- The function should be optimized for efficiency, particularly for larger strings.

Problem-83: Optimal Path in a Weighted Grid

Objective:

Create a function that finds the minimum cost path from the top-left corner to the bottom-right corner of a grid. The grid consists of positive integers representing weights, and the path can only move right or down.

Function Signature:

```
def min_cost_path(grid: list) -> int:
    pass
```

Instructions:

1. **Grid Input:**
   - The function receives a grid (2D list) of integers. Each integer represents the cost of entering that cell.
   - You start at the top-left corner (0, 0) and must reach the bottom-right corner (n-1, m-1).
2. **Allowed Moves:**
   - You can only move right (to the cell on the right) or down (to the cell below).
3. **Objective:**
   - The function should calculate the minimum cost required to reach the bottom-right corner from the top-left corner.
4. **Dynamic Programming Approach:**
   - Implement the solution using a dynamic programming approach to optimize the pathfinding process.
5. **Return the Minimum Cost:**
   - The function should return the minimum cost of the path.

Example:

Input:

```
grid = [ [1, 3, 1],
         [1, 5, 1],
         [4, 2, 1] ]
```

Output: 7

Explanation: The path with the minimum cost is 1 → 3 → 1 → 1 → 1, with a total cost of 7.

Input:

grid = [[1, 2, 3],
[4, 8, 2],
[1, 5, 3]]

Output: 8

Explanation:

- The path with the minimum cost is 1 → 2 → 3 → 2 → 3, with a total cost of 8.

Constraints:

- The grid will have at least 2 rows and 2 columns.
- The grid size n x m can be up to 100 x 100.
- Each cell will contain a positive integer cost, ranging from 1 to 100.

Additional Challenge:

- For an additional challenge, modify the function to return not only the minimum cost but also the actual path taken as a list of tuples representing the coordinates of each cell in the path.

Problem-84: K-Partition Subset Sum

Objective:

Create a function determining if a given set of integers can be partitioned into k non-empty subsets with equal sums.

Function Signature:

```
def can_partition_k_subsets(nums: list, k: int) -> bool:
    pass
```

Instructions:

1. **Set Input:**
   - The function receives a list of integers nums and an integer k.
2. **Equal Subset Sums:**
   - The function should determine if it's possible to partition the list nums into k subsets where each subset has an equal sum.
3. **Recursive Backtracking Approach:**
   - Implement the solution using a recursive backtracking approach to explore all possible partitionings. Avoid brute-force approaches that do not scale well with larger inputs.
4. **Efficiency Considerations:**
   - The solution must be optimized to handle larger inputs, considering both time and space complexity.
5. **Return a Boolean:**
   - The function should return True if the partition is possible and False otherwise.

Example:

Input: nums = [4, 3, 2, 3, 5, 2, 1], k = 4

Output: True

Explanation: The array can be partitioned into 4 subsets of equal sum: [5], [1, 4], [2, 3], and [2, 3].

Input: nums = [1, 2, 3, 4], k = 3

Output: False

Explanation: The array cannot be partitioned into 3 subsets with equal sum.

Constraints:

- The list nums will have at least k elements and contain positive integers.
- The sum of elements in nums is divisible by k.
- The value of k will be between 2 and len(nums).
- The size of nums will be between 1 and 16 elements.

Additional Challenge:

- For an additional challenge, modify the function to return the actual partitions as a list of lists, each representing a subset.

Problem-85: Network Delay Time

Objective:

Create a function that calculates the time it will take for all nodes in a network to receive a signal sent from a starting node, given the transmission times between nodes. The network is represented as a directed graph.

Function Signature:

```
def network_delay_time(times: list, N: int, K: int) -> int:
    pass
```

Instructions:

1. **Input:**
    - The function receives a list of times where each element is a tuple (u, v, w) representing a directed edge from node u to node v with a transmission time of w units.
    - N is the total number of nodes in the network, labeled from 1 to N.
    - K is the starting node from which the signal is sent.
2. **Calculating Network Delay:**
    - The function should calculate the time it will take for all nodes to receive the signal sent from node K.
    - If it's impossible for all nodes to receive the signal, return -1.
3. **Return the Result:**
    - The function should return the time it takes for the last node to receive the signal. This is the maximum time among the shortest paths from the starting node K to all other nodes.

Example:

Input: times = [(2, 1, 1), (2, 3, 1), (3, 4, 1)], N = 4, K = 2

Output: 2

Explanation: The network can be visualized as:

- 2 -> 1 with time 1
- 2 -> 3 with time 1
- 3 -> 4 with time 1

- The signal starts at node 2, reaches nodes 1 and 3 in 1 unit of time, and finally reaches node 4 in 2 units of time.

Input: times = [(1, 2, 1), (2, 3, 2), (1, 3, 4)], N = 3, K = 1

Output: 3

Explanation: The signal starts at node 1, reaches node 2 in 1 unit of time, and node 3 in 3 units via the fastest route 1 -> 2 -> 3.

Input: times = [(1, 2, 1),(1, 3, 2),(2, 4, 1),(3, 4, 2), (4, 5, 1), (5, 6, 2), (1, 6, 4)] N = 6 K = 1

Output: 4

Explanation: The signal starts at node 1 and spreads to all other nodes.

- The signal reaches node 6 directly with time 4 or via the path 1 -> 2 -> 4 -> 5 -> 6 with time 1 + 1 + 1 + 2 = 5.
- Since 4 is less than 5, the shortest time to reach node 6 is 4 units.

Thus, the network delay time is 4, which is the time it takes for the last node to receive the signal.

Constraints:

- The number of nodes N will be between 1 and 100.
- The number of edges in times will be between 1 and 6000.
- The transmission time w will be a positive integer.

Additional Challenge:

- For an additional challenge, modify the function to return the full shortest path to the node that takes the longest to receive the signal.

Problem-86: Maximum Subarray Sum with One Deletion

Objective:

Create a function that finds the maximum sum of a subarray in a given array with the condition that you can optionally delete one element to maximize the sum.

Function Signature:

```
def max_sum_with_one_deletion(arr: list) -> int:
    pass
```

Instructions:

1. **Array Input:**
    - The function receives a list of integers arr. The list can contain both positive and negative integers.
2. **Subarray Definition:**
    - A subarray is a contiguous portion of the array.
3. **Delete One Element:**
    - You may delete only one element from the array to maximize the sum of the resulting subarray, or you may choose not to delete any elements.
4. **Optimal Sum Calculation:**
    - The function should calculate and return the maximum possible sum of a subarray, considering the option to delete one element.
5. **Dynamic Programming Approach:**
    - Implement the solution using a dynamic programming approach to calculate the maximum sum efficiently.

Example:

Input: arr = [1, -2, 0, 3]

Output: 4

Explanation: The optimal subarray is [1, 0, 3] with a sum of 4 (deleting -2).

Input: arr = [1, -2, -2, 3]

Output: 3

Explanation: The optimal subarray is [3] with a sum of 3 (deleting the first -2 or the second -2).

Input: arr = [-1, -1, -1, -1]

Output: -1

Explanation: The optimal subarray is [-1] (with no deletion), since deleting any element would not increase the sum.

Constraints:

- The length of the array arr will be between 1 and 10^5.
- The elements in the array will be integers ranging from -10^4 to 10^4.

Additional Challenge:

- For an additional challenge, extend the function to return the indices of the subarray that produces the maximum sum.

Problem-87: Traveling Salesman Problem (TSP) with Dynamic Programming

Objective:

Create a function to solve the Traveling Salesman Problem (TSP) using dynamic programming with memoization. The goal is to find the shortest route that visits each city exactly once and returns to the origin city.

Function Signature:

```
def tsp(graph: list) -> int:
    pass
```

Instructions:

1. **Graph Input:**
   - The function receives a 2D list graph, where graph[i][j] represents the distance between city i and city j.
   - The graph is complete, meaning a direct path exists between any two cities.
2. **Start and End at the Same City:**
   - The traveling salesman must start and end in the same city (e.g., city 0).
3. **Visiting All Cities:**
   - The function should find the shortest path that visits each city exactly once and returns to the starting city.
4. **Dynamic Programming with Memoization:**
   - Implement the solution using dynamic programming with memoization to solve the problem efficiently.
   - Use a bitmask to represent the cities visited so far, and use recursion to explore all possible routes.
5. **Return the Minimum Cost:**
   - The function should return the minimum possible cost of the route.

Example:

Input:

```
graph = [
        [0, 10, 15, 20],
        [10, 0, 35, 25],
        [15, 35, 0, 30],
        [20, 25, 30, 0]
]
```

Output: 80

Explanation: The optimal route is 0 → 1 → 3 → 2 → 0 with a total cost of 80.

Input:

```
graph = [
        [0, 29, 20, 21],
        [29, 0, 15, 17],
        [20, 15, 0, 28],
        [21, 17, 28, 0]
]
```

Output: 76

Explanation: The optimal route is 0 → 2 → 1 → 3 → 0 with a total cost of 76.

Constraints:

- The number of cities (length of graph) will be between 2 and 20.
- The distances will be non-negative integers.

Additional Challenge:

- For an additional challenge, modify the function to return the actual sequence of cities in the optimal route, not just the minimum cost.

Problem-88: Word Break Problem with Memoization Using Set

Objective:

Create a function determining if a given string can be segmented into a sequence of one or more words from a set.

Function Signature:

```
def word_break(s: str, word_set: set) -> bool:
    pass
```

Instructions:

1. **String and Set Input:**
   - The function receives a string s and a set word_set, where word_set contains valid words.
2. **Segmenting the String:**
   - Determine if the string s can be segmented into a sequence of one or more words found in word_set.
3. **Memoization:**
   - Implement the solution using dynamic programming with memoization to determine if the segmentation is possible efficiently.
4. **Return a Boolean:**
   - The function should return True if the string can be segmented according to the words in the set and False otherwise.

Example:

Input: s = "leetcode", word_set = {"leet", "code"}

Output: True

Explanation: The string "leetcode" can be segmented as "leet" and "code".

Input: s = "applepenapple", word_set = {"apple", "pen"}

Output: True

Explanation: The string "applepenapple" can be segmented as "apple", "pen", and "apple".

Input: s = "catsandog", word_set = {"cats", "dog", "sand", "and", "cat"}

Output: False

Explanation: The string "catsandog" cannot be fully segmented into words from the set.

Constraints:

- The length of the string s will be between 1 and 300.
- The size of the set word_set will be between 1 and 1000.
- All words in word_set are non-empty and consist of lowercase English letters.

Additional Challenge:

- For an additional challenge, modify the function to return all possible segmentations of the string as a list of lists, each containing a valid sequence of words from the set.

Problem-89: Longest Path in a Directed Acyclic Graph (DAG)

Objective:

Create a function that finds the longest path in a Directed Acyclic Graph (DAG). The path should be measured by the sum of the weights of the edges on the path.

Function Signature:

```
def longest_path_dag(graph: dict, start_node: int) -> int:
    pass
```

Instructions:

1. **Graph Input:**
   - The function receives a graph represented as an adjacency list. The graph is a dictionary where the keys are nodes, and the values are lists of tuples. Each tuple represents a directed edge with a target node and a weight.
   - The function also receives the start_node from which the longest path calculation should begin.
2. **Directed Acyclic Graph (DAG):**
   - The graph is guaranteed to be acyclic, meaning there are no cycles.
3. **Longest Path Calculation:**
   - The function should find the longest path starting from the start_node and ending at any other node in the graph.
   - The path length is defined as the sum of the weights of the edges in the path.
4. **Dynamic Programming Approach:**
   - Implement the solution using a dynamic programming approach. Use topological sorting to ensure that each node is processed before any of its successors.
   - Maintain a memoization table to store the longest path length for each node.
5. **Return the Maximum Path Length:**
   - The function should return the length of the longest path found.

Example:

Input:

```
graph = {
        1: [(2, 3), (3, 6)],
        2: [(3, 4), (4, 11)],
        3: [(4, 8)],
        4: []
}
start_node = 1
```

Output: 15

Explanation: The longest path from node 1 is 1 -> 2 -> 3 -> 4, with a total path length of 15 (3 + 4 + 8).

Input:

```
graph = {
        0: [(1, 5), (2, 3)],
        1: [(3, 6)],
        2: [(3, 7)],
        3: [(4, 4)],
        4: []
}
start_node = 0
```

Output: 15

Explanation: The longest path from node 0 is 0 -> 2 -> 3 -> 4, with a total path length of 15 (3 + 7 + 4).

Constraints:

- The graph will contain between 2 and 1000 nodes.
- Each node in the graph has a unique identifier (an integer).
- The weights of the edges will be non-negative integers.

Additional Challenge:

- For an additional challenge, modify the function to return not only the length of the longest path but also the sequence of nodes on that path.

Problem-90: K-Shortest Paths in a Weighted Graph

Objective:

Create a function that finds the k shortest paths between two nodes in a weighted graph. The paths should be ranked by their total weight, from the shortest to the longest.

Function Signature:

```
def k_shortest_paths(graph: dict, start_node: int, end_node: int, k: int) -> list:
    pass
```

Instructions:

1. **Graph Input:**
   - The function receives a graph represented as an adjacency list. The graph is a dictionary where the keys are nodes, and the values are lists of tuples. Each tuple represents a directed edge with a target node and a weight.
   - The function also receives start_node and end_node as the nodes between which the k shortest paths need to be found.
2. **Finding K-Shortest Paths:**
   - The function should return the k shortest paths from start_node to end_node ranked by their total weight. The paths should be distinct (i.e., no path should be repeated).
   - If fewer than k paths exist between the nodes, return all available paths.
3. **Path Representation:**
   - Each path should be represented as a list of nodes, and the function should return a list of these paths.
4. **Optimization Approach:**
   - To efficiently find the k shortest paths, use an algorithm such as Yen's K-Shortest Paths algorithm or an appropriate graph search technique.
5. **Edge Weights:**
   - The graph's edges may have positive or negative weights, but no negative cycles exist.

Example:

Input:

```
graph = {
        1: [(2, 1), (3, 5)],
        2: [(3, 1), (4, 2)],
        3: [(4, 1)],
        4: []
}
start_node = 1
end_node = 4
k = 2
```

Output: [[1, 2, 3, 4], [1, 2, 4]]

Explanation:

- The shortest path from node 1 to node 4 is 1 -> 2 -> 3 -> 4 with a total weight of 3 (1 + 1 + 1).
- The second shortest path is 1 -> 2 -> 4 with a total weight of 3 (1 + 2).

Input:

```
graph = {
        0: [(1, 2), (2, 4)],
        1: [(2, 1), (3, 7)],
        2: [(3, 3)],
        3: []
}
start_node = 0
end_node = 3
k = 3
```

Output: [[0, 1, 2, 3], [0, 2, 3], [0, 1, 3]]

Explanation:

- The shortest path from node 0 to node 3 is 0 -> 1 -> 2 -> 3, weighing 6.
- The second shortest path is 0 -> 2 -> 3, with a total weight of 7.
- The third shortest path is 0 -> 1 -> 3, with a total weight of 9.

Constraints:

- The graph will contain between 2 and 1000 nodes.
- Each node in the graph has a unique identifier (an integer).
- The weights of the edges can be positive or negative, but no negative cycles exist.
- The value of k will be between 1 and 100.

Additional Challenge:

- For an additional challenge, modify the function to handle graphs with negative weight cycles by detecting such cycles and returning an appropriate message.

Problem-91: Graph Centroid: Node with the Lowest Average Shortest Path

Objective:

Create a function that finds the centroid of a graph. The centroid is defined as the node with the lowest average shortest path length to all other nodes in the graph.

Function Signature:

```
def find_graph_centroid(graph: dict) -> tuple:
    pass
```

Instructions:

1. **Graph Input:**
   - The function receives a graph represented as an adjacency list. The graph is a dictionary whose keys are nodes and values are lists of tuples. Each tuple represents a neighboring node and the weight of the edge to that neighbor.
   - The graph may be directed or undirected.
2. **Finding the Centroid:**
   - The function should calculate the average shortest path length from each node to all other nodes in the graph.
   - The node with the lowest average shortest path length is considered the centroid of the graph.
3. **Handling Disconnected Graphs:**
   - If the graph is disconnected, only consider the most significant connected component when determining the centroid.
4. **Shortest Path Calculation:**
   - Calculate the shortest paths from each node to every other node using Dijkstra's algorithm or another appropriate shortest path algorithm.
5. **Return the Centroid and Average Path Length:**
   - The function should return a tuple containing the node that is the centroid of the graph and the average shortest path length from that node to all other nodes.

Example:

Input:

```
graph = {
        0: [(1, 1), (2, 2)],
        1: [(0, 1), (3, 1)],
        2: [(0, 2), (3, 3)],
        3: [(1, 1), (2, 3)]
}
```

Output: (1, 2.0)

Explanation: The average shortest path length from node 1 to all other nodes is 2.0, the smallest in the graph.

Input:

```
graph = {
        0: [(1, 2), (2, 4)],
        1: [(0, 2), (2, 1)],
        2: [(0, 4), (1, 1), (3, 1)],
        3: [(2, 1)]
}
```

Output: (2, 2.0)

Explanation: The average shortest path length from node 2 to all other nodes is 2.0, which is the smallest in the graph.

Constraints:

- The graph will contain between 2 and 1000 nodes.
- The weights of the edges are non-negative integers.
- The graph may be directed or undirected.

Problem-92: Minimum Edit Distance with Three Operations

Objective:

Create a function that calculates the minimum edit distance between two strings. The edit distance is the minimum number of operations required to transform one string into the other. The allowed operations are insertion, deletion, and substitution.

Function Signature:

```
def min_edit_distance(str1: str, str2: str) -> int:
    pass
```

Instructions:

1. **String Input:**
   - The function receives two strings, str1 and str2.
2. **Edit Operations:**
   - **Insertion:** Insert a character into one of the strings.
   - **Deletion:** Delete a character from one of the strings.
   - **Substitution:** Replace a character in one of the strings with another character.
3. **Calculating Minimum Edit Distance:**
   - The function should calculate the minimum number of edit operations required to transform str1 into str2.
4. **Dynamic Programming Approach:**
   - Use a dynamic programming approach to build a table that keeps track of the minimum edit distances between all prefixes of the two strings.
5. **Return the Edit Distance:**
   - The function should return the minimum edit distance between str1 and str2.

Example:

Input: str1 = "kitten", str2 = "sitting"

Output: 3

Explanation: The minimum edit distance between "kitten" and "sitting" is 3:

- Substitute 'k' with 's': "kitten" -> "sitten"
- Substitute 'e' with 'i': "sitten" -> "sittin"
- Insert 'g' at the end: "sittin" -> "sitting"

Input: str1 = "flaw", str2 = "lawn"

Output: 2

Explanation: The minimum edit distance between "flaw" and "lawn" is 2:

- Delete 'f': "flaw" -> "law"
- Insert 'n' at the end: "law" -> "lawn"

Constraints:

- The strings str1 and str2 lengths will be between 1 and 1000.
- The strings will contain only lowercase English letters.

Additional Challenge:

- For an additional challenge, modify the function to return the minimum edit distance and the sequence of operations that achieve this distance.

Problem-93: Grouping Anagrams by Length and Frequency

Objective:

Create a function that groups anagrams together based on both their length and the frequency of their characters. The function should return a dictionary where the keys are the lengths of the anagrams, and the values are lists of groups of anagrams.

Function Signature:

```
def group_anagrams(words: list) -> dict:
    pass
```

Instructions:

1. **Input:**
   - The function receives a list of strings words. Each string consists of lowercase English letters.
2. **Grouping by Anagram:**
   - An anagram is a word formed by rearranging the letters of another word using all the original letters exactly once.
   - Group words that are anagrams of each other into lists.
3. **Grouping by Length:**
   - The anagrams should also be grouped by the length of the words.
   - The output dictionary should have keys corresponding to the lengths of the words, and the values should be lists of lists, where each sublist contains a group of anagrams.
4. **Sorting the Groups:**
   - The list of anagram groups should be sorted by the number of words in each length group in descending order.
5. **Return the Dictionary:**
   - The function should return the dictionary with length keys and grouped anagrams as described.

Example:

Input: words = ["bat", "tab", "cat", "act", "tac", "rat", "tar", "art", "star", "rats"]

Output:

```
{
    3: [["bat", "tab"], ["cat", "act", "tac"], ["rat", "tar", "art"]],
    4: [["star", "rats"]]
}
```

Explanation:

- For words of length 3, we have three groups of anagrams: ["bat", "tab"], ["cat", "act", "tac"], and ["rat", "tar", "art"].
- For words of length 4, we have one group of anagrams: ["star", "rats"].

Input: words = ["listen", "silent", "enlist", "inlets", "google", "gogole", "elgoog", "cat", "tac", "act"]

Output:

{
    3: [["cat", "tac", "act"]],
    6: [["listen", "silent", "enlist", "inlets"], ["google", "gogole", "elgoog"]]
}

Explanation:

- For words of length 3, there is one group of anagrams: ["cat", "tac", "act"].
- For words of length 6, there are two groups of anagrams: ["listen", "silent", "enlist", "inlets"] and ["google", "gogole", "elgoog"].

Constraints:

- The input list words will contain between 1 and 10^4 strings.
- Each string in words will have a length between 1 and 100.

Additional Challenge:

- For an additional challenge, modify the function to return the groups of anagrams in the order of their frequency of occurrence in the input list, with the most frequent groups appearing first.

Problem-94: Nested Dictionary Flattening

Objective:

Create a function that takes a deeply nested dictionary and flattens it. The keys in the resulting dictionary should be the path to the original value, concatenated by a separator.

Function Signature:

```
def flatten_dict(d: dict, separator: str = '.') -> dict:
    pass
```

Instructions:

1. **Input:**
   - The function receives a nested dictionary d. The dictionary may have multiple levels of nesting.
   - The separator parameter is a string that defines how the keys should be concatenated when flattening the dictionary.
2. **Flattening the Dictionary:**
   - The function should recursively flatten the nested dictionary.
   - The keys in the resulting dictionary should represent the path from the root to each value, with each level of nesting separated by the separator.
3. **Handling Edge Cases:**
   - If a value in the dictionary is another dictionary, continue flattening.
   - If a value is not a dictionary, include it in the resulting flattened dictionary with the appropriate key path.
4. **Return the Flattened Dictionary:**
   - The function should return the flattened dictionary with keys representing the paths to the original values.

Example:

Input:

```
d = {
        "a": 1,
        "b": {
                "c": 2,
                "d": {
                "e": 3,
                "f": 4
                }
        }
}
separator = "."
```

Output:

```
{
        "a": 1,
        "b.c": 2,
        "b.d.e": 3,
        "b.d.f": 4
}
```

Explanation: The nested dictionary is flattened, with keys representing the full path to each value, separated by ".".

Input:

```
d = {
        "user": {
                "name": "Alice",
                "address": {
                "city": "Wonderland",
                "zip": "12345"
                },
                "email": "alice@example.com"
        }
}
separator = "_"
```

Output:

```
{
        "user_name": "Alice",
        "user_address_city": "Wonderland",
        "user_address_zip": "12345",
        "user_email": "alice@example.com"
}
```

Explanation: The nested dictionary is flattened using "_" as the separator.

Constraints:

- The dictionary d will contain between 1 and 10^4 key-value pairs.
- The keys in the dictionary will be strings and values may be strings, integers, lists, or other dictionaries.

Additional Challenge:

- For an additional challenge, modify the function to handle dictionaries with lists of dictionaries, flattening the lists by using the index of the list element in the key path.

Problem-95: JSON Diff Tool

Objective:

Create a function that takes two JSON-like dictionaries and returns the differences between them. The differences should include added, removed, and modified keys.

Function Signature:

```
def json_diff(dict1: dict, dict2: dict) -> dict:
    pass
```

Instructions:

1. **Input:**
   - The function receives two dictionaries, dict1 and dict2, which represent JSON objects.
2. **Calculating Differences:**
   - The function should identify the following types of differences:
     - **Added Keys:** Keys that are in dict2 but not in dict1.
     - **Removed Keys:** Keys that are in dict1 but not in dict2.
     - **Modified Keys:** Keys that exist in both dictionaries but have different values.
   - If a key's value is a nested dictionary, the function should recursively check for differences within that nested dictionary.
3. **Returning the Differences:**
   - The function should return a dictionary with three keys: "added", "removed", and "modified".
   - Each key should map to a dictionary that lists the corresponding differences.
4. **Handling Nested Structures:**
   - The function must handle nested dictionaries and reflect the differences at all levels.

Example:

Input:

```
dict1 = {
        "name": "Alice",
        "age": 30,
        "address": {
        "city": "Wonderland",
        "zip": "12345"
        }
}
```

```
dict2 = {
        "name": "Alice",
        "age": 31,
        "address": {
        "city": "Wonderland",
        "zip": "54321"
        },
        "email": "alice@example.com"
}
```

Output:

```
{
        "added": {
        "email": "alice@example.com"
        },
        "removed": {},
        "modified": {
        "age": {"from": 30, "to": 31},
        "address": {
                "zip": {"from": "12345", "to": "54321"}
        }
        }
}
```

Explanation:

- **Added:** The "email" key is in dict2 but not in dict1.
- **Removed:** There are no removed keys.
- **Modified:** The "age" key has a different value in dict2 compared to dict1, and the "zip" key inside the "address" dictionary also has different values.

Constraints:

- The dictionaries dict1 and dict2 will have a depth between 1 and 10.
- The dictionaries can contain nested dictionaries, lists, strings, integers, and other JSON-compatible data types.
- The total number of keys in each dictionary will not exceed 10^3.

Additional Challenge:

- For an additional challenge, modify the function to also identify changes in lists within the dictionaries.

Problem-96: Expression Evaluator with Variables

Objective:

Create a function that evaluates a mathematical expression containing variables and returns the result. The function should also support variable assignment within the expression.

Function Signature:

```
def evaluate_expression(expression: str, variables: dict = None) -> int:
    pass
```

Instructions:

1. **Input:**
   - The function receives a string expression containing a mathematical expression. The expression may include integers, variables, and operators: +, -, *, /.
   - The expression may also include variable assignments as variable_name = expression.
   - The variables parameter is an optional dictionary that holds initial values for variables that may be used in the expression.
2. **Evaluating the Expression:**
   - The function should evaluate the expression and return the final result.
   - If the expression includes a variable assignment (e.g., x = 3 + 2), the function should update the variable in the variables dictionary and return the result of the assignment.
3. **Handling Precedence:**
   - The function should correctly handle operator precedence (*, / before +, -).
   - Parentheses may be used in the expression to define precedence explicitly.
4. **Return the Result:**
   - The function should return the evaluated result of the expression.
5. **Edge Cases:**
   - If the expression refers to an undefined variable, raise an appropriate error.
   - Handle division by zero by raising an error.

Example:

Input: expression = "x = 3 + 5 * (2 - 1)", variables = {}

Output: 8

Explanation: The expression is evaluated as x = 3 + 5 * 1, resulting in x = 8. The function should return 8.

Input: expression = "y = x + 4", variables = {"x": 8}

Output: 12

Explanation: The expression is evaluated as y = 8 + 4, resulting in y = 12. The function should return 12.

Input: expression = "z = (x + y) * 2", variables = {"x": 3, "y": 5}

Output: 16

Explanation: The expression is evaluated as z = (3 + 5) * 2, resulting in z = 16. The function should return 16.

Constraints:

- The expression will have a maximum length of 1000 characters.
- Variable names will consist of lowercase English letters and will not exceed 20 characters.
- The variables dictionary may contain up to 100 variables.

Additional Challenge:

- Extend the function to handle floating-point arithmetic with precision up to two decimal places for an additional challenge.

Problem-97: Itinerary Reconstruction

Objective:

Create a function that reconstructs an itinerary from a list of flight tickets. The itinerary should start from a specific airport and visit all airports according to the tickets, using each ticket exactly once.

Function Signature:

```
def find_itinerary(tickets: list) -> list:
    pass
```

Instructions:

1. **Input:**
   - The function receives a list of flight tickets, where each ticket is represented as a tuple (origin, destination).
2. **Reconstructing the Itinerary:**
   - If the itinerary exists, the function should reconstruct it starting from "JFK" (John F. Kennedy International Airport).
   - The itinerary must visit all airports using all tickets exactly once.
   - If there are multiple valid itineraries, return the one that is lexicographically smallest.
3. **Lexicographical Order:**
   - Lexicographical order means that "JFK" should come before "LAX" if they were compared as strings, just like in dictionary order.
4. **Handling Multiple Flights:**
   - If there are multiple flights from the same origin, choose the destination that comes first lexicographically.
5. **Return the Itinerary:**
   - The function should return the reconstructed itinerary as a list of airport codes.

Example:

Input: tickets = [("MUC", "LHR"), ("JFK", "MUC"), ("SFO", "SJC"), ("LHR", "SFO")]

Output: ["JFK", "MUC", "LHR", "SFO", "SJC"]

Explanation: The itinerary starts at "JFK" and follows the tickets in the order "JFK" -> "MUC" -> "LHR" -> "SFO" -> "SJC."

Input: tickets = [("JFK", "KUL"), ("JFK", "NRT"), ("NRT", "JFK")]

Output: ["JFK", "NRT", "JFK", "KUL"]

Explanation: There are two options: "JFK" -> "KUL" and "JFK" -> "NRT" -> "JFK". The latter is chosen because it is lexicographically smaller.

Constraints:

- The number of tickets will be between 1 and 300.
- The airport codes will be three uppercase English letters (e.g., "JFK", "LAX").
- The itinerary must use all the tickets exactly once.

Additional Challenge:

- For an additional challenge, modify the function to handle cases without valid itineraries and return an appropriate message.

Problem-98: Palindrome Partitioning with Minimum Cuts

Objective:

Create a function that partitions a string such that every substring is a palindrome and returns the minimum number of cuts needed to achieve this.

Function Signature:

```
def min_cut_palindrome_partition(s: str) -> int:
    pass
```

Instructions:

1. **Input:**
   - The function receives a string s consisting of lowercase English letters.
2. **Palindrome Partitioning:**
   - A palindrome is a string that reads the same forward and backward (e.g., "madam").
   - The goal is to partition the string into the minimum number of substrings such that each substring is a palindrome.
3. **Calculating Minimum Cuts:**
   - The function should calculate the minimum number of cuts needed to partition the string such that every substring is a palindrome.
   - A cut is a split between two characters in the string. For example, the string "abac" can be partitioned into "a|ba|c" with two cuts.
4. **Return the Minimum Cuts:**
   - The function should return the minimum number of cuts needed.

Example:

Input: s = "aab"

Output: 1

Explanation: The string "aab" can be partitioned into "aa" and "b". Since "aa" is a palindrome and "b" is a palindrome, only 1 cut is needed.

Input: s = "abccba"

Output: 0

Explanation: The string "abccba" is already a palindrome, so no cuts are needed.

Input: s = "aabbc"

Output: 2

Explanation: The string can be partitioned as "aa|bb|c". Two cuts are needed.

Constraints:

- The length of the string s will be between 1 and 2000.
- The string will contain only lowercase English letters.

Additional Challenge:

- For an additional challenge, modify the function to return the minimum number of cuts and the actual partitions.

Problem-99: Meeting Rooms II

Objective:

Create a function determining the minimum number of meeting rooms required to accommodate all the meetings without overlap.

Function Signature:

```
def min_meeting_rooms(intervals: list) -> int:
    pass
```

Instructions:

1. **Input:**
   - The function receives a list of intervals, where each interval is a tuple (start, end) representing the start and end times of a meeting.
2. **Calculating Minimum Meeting Rooms:**
   - The function should calculate the minimum number of meeting rooms required so that all meetings can occur without overlap.
3. **Handling Overlaps:**
   - If two meetings overlap, they must be held in different rooms. For example, if one meeting ends at 10:00 AM and another starts at 10:00 AM, they can be scheduled in the same room.
4. **Return the Result:**
   - The function should return the minimum number of rooms required.

Example:

Input: intervals = [(0, 30), (5, 10), (15, 20)]

Output: 2

Explanation:

- The first meeting starts at 0 and ends at 30.
- The second meeting starts at 5 and ends at 10.
- The third meeting starts at 15 and ends at 20.
- The second meeting overlaps with the first, and the third overlaps with the first, but not with the second. Therefore, we need at least 2 rooms.

Input: intervals = [(7, 10), (2, 4)]

Output: 1

Explanation: The two meetings do not overlap, so only one room is required.

Input: intervals = [(1, 5), (2, 6), (3, 8), (5, 7), (8, 9)]

Output: 3

Explanation:

Time 1: Meeting 1 starts.

- Rooms in use: 1 (Meeting 1)

Time 2: Meeting 2 starts and overlaps with Meeting 1.

- Rooms in use: 2 (Meeting 1, Meeting 2)

Time 3: Meeting 3 starts and overlaps with both Meeting 1 and Meeting 2.

- Rooms in use: 3 (Meeting 1, Meeting 2, Meeting 3)

Time 5: Meeting 1 ends, freeing up 1 room. Meeting 4 starts at 5 and uses this room.

- Rooms in use: 3 (Meeting 4, Meeting 2, Meeting 3)

Time 6: Meeting 2 ends, freeing up 1 room.

- Rooms in use: 2 (Meeting 4, Meeting 3)

Time 7: Meeting 4 ends, freeing up 1 room.

- Rooms in use: 1 (Meeting 3)

Time 8: Meeting 3 ends, freeing up 1 room. Meeting 5 starts and uses this room.

- Rooms in use: 1 (Meeting 5)

Time 9: Meeting 5 ends.

- Rooms in use: 0

Constraints:

- The number of meetings n will be between 1 and 10^4.
- The start and end times of meetings will be non-negative integers.

Additional Challenge:

- For an additional challenge, modify the function to return the schedule of which meeting goes into which room.

Problem-100: Word Ladder II: Finding All Shortest Transformation Sequences

Objective:

Create a function that finds all the shortest transformation sequences from a start word to an end word. The transformation must follow specific rules: each transformation changes exactly one letter, and each intermediate word must be valid in the given dictionary.

Function Signature:

```
def find_ladders(begin_word: str, end_word: str, word_list: list) -> list:
    pass
```

Instructions:

1. **Input:**
   - The function receives three inputs:
     - begin_word: The starting word.
     - end_word: The target word to transform into.
     - word_list: A list of valid words (the dictionary).
2. **Transformation Rules:**
   - Each word in a transformation sequence must differ by exactly one letter from the previous word.
   - Each intermediate word in the transformation sequence must exist in the word_list.
   - The goal is to find all shortest sequences that transform begin_word to end_word.
3. **Return All Shortest Sequences:**
   - The function should return a list of all the shortest transformation sequences from begin_word to end_word, each containing a list of words.
4. **Handling Edge Cases:**
   - If there is no possible transformation sequence, return an empty list.

Example:

Input:

```
begin_word = "hit"
end_word = "cog"
word_list = ["hot", "dot", "dog", "lot", "log", "cog"]
```

Output:

```
[
  ["hit", "hot", "dot", "dog", "cog"],
  ["hit", "hot", "lot", "log", "cog"]
]
```

Explanation: The shortest sequences from "hit" to "cog" are:

- "hit" -> "hot" -> "dot" -> "dog" -> "cog"
- "hit" -> "hot" -> "lot" -> "log" -> "cog"

Input:

```
begin_word = "hit"
end_word = "cog"
word_list = ["hot", "dot", "dog", "lot", "log"]
```

Output: []

Explanation: The word "cog" is not in the word list, so no valid transformation sequence exists.

Input:

```
begin_word = "hit"
end_word = "cog"
word_list = ["hot", "dot", "tod", "fog", "log", "cog"]
```

Output: []

Explanation: Transformation Path Attempt:

- "hit" -> "hot" -> "dot" -> (no valid continuation to reach "cog")
- **Stuck:** Even though "cog" is in the word list, there's no valid sequence of transformations that can reach "cog" from "dot" or "tod".

Constraints:

- All words have the same length.
- All words consist of lowercase English letters.
- The word_list contains at most 5000 words.

Additional Challenge:

- For an additional challenge, optimize the solution to run efficiently even when the word list is large.