Module 2 Graded Assignment 1 Editorial

Question : Money Management

```python
def solve(a, k):
    sum_val = sum(a)  # Python's built-in sum() function
    if sum_val < k:
        return "Save"
    elif sum_val == k:
        return "Neutral"
    else:
        return "Debt"

# Main program
n = int(input())  # Python's input() returns a string, so we convert to int
a = list(map(int, input().split()))  # Read space-separated integers into a list
k = int(input())
print(solve(a, k))
```

Explanation of Each Part

1. **Function Definition**:

   ```python
   def solve(a, k):
   ```

   - We define a function called `solve` which takes two parameters:
     - `a` : a list of integers (expenses).
     - `k` : an integer representing the monthly salary.
2. **Summing the Expenses**:

   ```python
   sum_val = sum(a)  # Python's built-in sum() function
   ```

   - We use Python's built-in `sum()` function to calculate the total of all expenses in the list `a` and store it in `sum_val` .
3. **Conditional Checks**:

```python
if sum_val < k:
    return "Save"
elif sum_val == k:
    return "Neutral"
else:
    return "Debt"
```

- **If `sum_val < k`:** We return `"Save"`, meaning the total expenses are less than the salary.
- **Else if `sum_val == k`:** We return `"Neutral"`, meaning the total expenses exactly match the salary.
- **Otherwise (`sum_val > k`):** We return `"Debt"`, meaning the total expenses exceed the salary.

4. **Reading Input**:

```python
n = int(input())  # Python's input() returns a string, so we convert to int
```

- We read a line from standard input (which will be a string) and convert it to an integer. This integer `n` represents the number of elements in the list of expenses (though in this solution, we don't directly use `n` beyond reading the list).

5. **Reading the Expenses List**:

```python
a = list(map(int, input().split()))  # Read space-separated integers into a list
```

- We read another line from input, split it by spaces (creating substrings), map each substring to an integer, and finally convert that map object to a list, stored in `a`.

6. **Reading the Salary**:

```python
k = int(input())
```

- We read one more line from input, convert it to an integer, and store it in `k` (the salary).

7. **Calling the Function and Printing**:

```python
print(solve(a, k))
```

- We call the `solve` function with the list of expenses `a` and the salary `k`, and immediately print the returned result (`"Save"`, `"Neutral"`, or `"Debt"`).

How It Works Overall:

1. The code reads the number of expenses (though not heavily used in the logic), the list of expenses, and the monthly salary.
2. It calculates the total of those expenses and compares it with the salary.
3. Based on the comparison, it prints whether you should **Save**, remain **Neutral**, or end up in **Debt**.

This straightforward approach ensures that the logic is clear and easy to maintain.

Question : Fibonacci-Recursion

```
def solve(n):

    if(n==0  or n<0):
        return
    if(n==1 or n==2):
        return 1

    return solve(n-1)+solve(n-2)

def inpt():

    n = int(input())
    print(solve(n))

inpt()
```

Explanation

1. **Function Definition (`solve`)**

   ```
   def solve(n):
   ```

   - We define a function named `solve` that calculates the *n*-th Fibonacci number using recursion.
2. **Base Case Checks**

```python
if(n==0  or n<0):
    return
if(n==1 or n==2):
    return 1
```

- If `n` is `0` or less than `0`, the function returns `None` (because there's no explicit return value).
- If `n` is `1` or `2`, the function returns `1`, which corresponds to the first two Fibonacci numbers in this particular definition (F(1) = 1, F(2) = 1).

3. **Recursive Case**

```python
return solve(n-1) + solve(n-2)
```

- For any `n` greater than 2, the function returns the sum of the two previous Fibonacci numbers, `solve(n-1)` and `solve(n-2)`.
- This directly implements the Fibonacci recurrence relation:

$$F(n) = F(n-1) + F(n-2)$$

4. **Reading Input and Printing**

```python
def inpt():
    n = int(input())
    print(solve(n))
```

- This function reads an integer `n` from standard input.
- It then calls the `solve` function with `n` and prints the result.

5. **Function Call**

```python
inpt()
```

- This line invokes the `inpt` function, triggering the entire process of reading input, computing the Fibonacci value, and printing it.

How It Works

- The program starts by calling `inpt()`.
- `inpt()` reads an integer `n` and calls `solve(n)`.
- `solve(n)` checks the base cases:

- Returns `None` if `n` is `0` or negative.
- Returns `1` if `n` is `1` or `2`.
- Otherwise, it recursively calculates the Fibonacci number as `solve(n-1) + solve(n-2)`.

- Finally, the result from `solve(n)` is printed to the console.

Question : Highest Stock Value

```python
def highest_stock_value(n, arr):
    current_value = 0  # Start with stock value at 0
    max_value = 0      # Track the highest value seen

    # Iterate through the changes
    for change in arr:
        current_value += change  # Update the stock value with the net change
        max_value = max(max_value, current_value)  # Update max if current value is higher

    return max_value

# Input reading
n = int(input())                # Read the number of changes
arr = list(map(int, input().split()))  # Read the array of changes

# Output the result
print(highest_stock_value(n, arr))
```

Explanation of the Code

1. **Function Definition:**

```python
def highest_stock_value(n, arr):
```

- We define a function called `highest_stock_value` that takes two parameters:
  - `n` : the number of stock value changes.
  - `arr` : a list of integers representing the net changes in stock value at each point in time.

2. **Initialize Variables:**

```
current_value = 0  # Start with stock value at 0
max_value = 0      # Track the highest value seen
```

- `current_value` will hold the ongoing stock value as we iterate through each change.
- `max_value` will keep track of the maximum stock value encountered so far.

3. **Iterate Through the Changes:**

```
for change in arr:
    current_value += change  # Update the stock value with the net change
    max_value = max(max_value, current_value)  # Update max if current value is higher
```

- We loop through each element (`change`) in the list `arr` .
- For each `change` , we add it to `current_value` to get the new stock value.
- Then we compare `current_value` with `max_value` . If `current_value` is greater, we update `max_value` .

4. **Return the Highest Value:**

```
return max_value
```

- After processing all changes, the function returns the highest stock value (`max_value`) observed during the iteration.

5. **Reading Input:**

```
n = int(input())                # Read the number of changes
arr = list(map(int, input().split()))  # Read the array of changes
```

- The program reads an integer `n` , which indicates how many changes there are.
- It then reads a line of input, splits it by spaces, converts each piece to an integer, and stores the results in `arr` .

6. **Printing the Result:**

```
print(highest_stock_value(n, arr))
```

- The function `highest_stock_value(n, arr)` is called with the inputs `n` and `arr` .
- The returned value (the highest stock value) is printed to the console.

How It Works

- Initially, both `current_value` and `max_value` are set to 0.
- As the program reads each change in the list, it updates the current stock value and checks if this new value is greater than the previously recorded `max_value`.
- By the end, `max_value` holds the greatest stock value reached at any point in the sequence of changes.
- Finally, the code prints out `max_value` as the highest stock value.

Question : Number of ways problems

```
n = int(input())

def ways(n):
  if n < 0:
    return 0
  if n == 0:
    return 1
  return ways(n - 1) + ways(n - 2) + ways(n - 3)

ans = ways(n)
print(ans)
```

Explanation

1. **Reading Input**

   ```
   n = int(input())
   ```

   - Reads a single integer from standard input, which represents the total number of steps.
2. **Defining the `ways` Function**

   ```
   def ways(n):
   ```

   - The function `ways` takes an integer `n` (number of steps) and returns how many distinct ways there are to climb `n` steps when you can take 1, 2, or 3 steps at a time.
3. **Base Cases**

```python
if n < 0:
    return 0
if n == 0:
    return 1
```

- **If `n < 0`**: Return 0 because there is no valid way to climb a negative number of steps.
- **If `n == 0`**: Return 1 because there is exactly one way to stand still at the top if you have no steps to climb (you've already reached your goal).

4. **Recursive Case**

```python
return ways(n - 1) + ways(n - 2) + ways(n - 3)
```

- For any `n > 0`, the number of ways to reach step `n` is the sum of:
  - The number of ways to reach `n-1` steps and then take 1 step, plus
  - The number of ways to reach `n-2` steps and then take 2 steps, plus
  - The number of ways to reach `n-3` steps and then take 3 steps.

5. **Compute and Print the Result**

```python
ans = ways(n)
print(ans)
```

- We call `ways(n)` with the input value `n` and store the result in `ans`.
- Finally, we print `ans`, which is the total number of distinct ways to climb `n` steps.

Question : Flower Management

```
tc = int(input())
for _ in range(tc):
    m, n = list(map(int, input().split()))
    arr = list(map(int, input().split()))

    count = 0
    i = 0

    while i < m:
        if arr[i] == 0:
            # Check if the left plot is empty or doesn't exist (i.e., i == 0)
            left = (i == 0 or arr[i - 1] == 0)

            # Check if the right plot is empty or doesn't exist (i.e., i == m - 1)
            right = (i == m - 1 or arr[i + 1] == 0)

            # If both left and right plots are empty, place a flower here
            if left and right:
                count += 1
                # Move 2 steps ahead to avoid placing a flower in the adjacent plot
                i += 2
            else:
                i += 1
        else:
            i += 1

    if count >= n:
        print("Yes")
    else:
        print("No")
```

Explanation of the Code

1. **Reading the Number of Test Cases**

```
tc = int(input())
```

- This reads an integer `tc` representing how many times we need to run the flower-placing logic.

2. **Looping Over Each Test Case**

```
for _ in range(tc):
    m, n = list(map(int, input().split()))
    arr = list(map(int, input().split()))
```

- For each test case:
  - `m` is the size of the flowerbed array (the number of plots).
  - `n` is the number of new flowers we want to plant.
  - `arr` is a list of length `m`, where each element can be:
    - `0` (an empty plot),
    - `1` (a plot where a flower already exists).

3. **Initializing Counters**

```
count = 0
i = 0
```

- `count` will track how many flowers we can successfully place.
- `i` will be used to iterate through the `arr` list.

4. **While Loop to Check Plots**

```
while i < m:
    if arr[i] == 0:
        ...
    else:
        i += 1
```

- We iterate through each plot in `arr` using `i` as the current index.

5. **Checking Empty Plots**

```
if arr[i] == 0:
    left = (i == 0 or arr[i - 1] == 0)
    right = (i == m - 1 or arr[i + 1] == 0)

    if left and right:
        count += 1
        i += 2
    else:
        i += 1
```

- When we encounter an empty plot (`arr[i] == 0`), we check the **left** and **right** neighbors:
    - **Left** is empty if `i == 0` (no left neighbor) or `arr[i - 1] == 0` .
    - **Right** is empty if `i == m - 1` (no right neighbor) or `arr[i + 1] == 0` .
- If both sides are empty, we can plant a flower at `i` , so we:
    - Increment `count` (we've planted one flower).
    - Jump `i` by 2 to skip the next index, ensuring we don't violate the "no adjacent flowers" rule.
- If the spot is not suitable, we simply move `i` by 1 to check the next plot.

6. **If the Plot Already Has a Flower**

```
else:
    i += 1
```

- If `arr[i] == 1` , we just move to the next index, since we cannot place a flower here.

7. **Check Final Flower Count**

```
if count >= n:
    print("Yes")
else:
    print("No")
```

- After processing the entire flowerbed, if the number of flowers we managed to place (`count`) is **at least** `n` , we print `"Yes"` .
- Otherwise, we print `"No"` .

How It Works Overall

- For each test case, we read the flowerbed configuration and how many new flowers we need to plant.
- We scan through the array:
    - Whenever we find an empty spot, we check if it's valid to place a flower (i.e., its neighbors are also empty).
    - If valid, we place a flower and skip the next spot to avoid adjacency conflicts.
- By the end, if we have placed **at least** as many flowers as required, we print `"Yes"` ; otherwise, `"No"` .

Question : Make Leaderboard

```python
def bubbleSortOnName(names, scores):
    N = len(names)
    for i in range(N-1):
        for j in range(N-i-1):
            if names[j] > names[j+1]:
                names[j], names[j+1] = names[j+1], names[j]
                scores[j], scores[j+1] = scores[j+1], scores[j]

def bubbleSortOnScore(names, scores):
    N = len(names)
    for i in range(N-1):
        for j in range(N-i-1):
            if scores[j] < scores[j+1]:
                names[j], names[j+1] = names[j+1], names[j]
                scores[j], scores[j+1] = scores[j+1], scores[j]

def printLeaderBoard(names, scores):
    N = len(names)
    rank = 1
    for i in range(N):
        print(rank, names[i])
        if i != N-1 and scores[i] > scores[i+1]:
            rank = i + 2

def solve(names, scores):
    bubbleSortOnName(names, scores)
    bubbleSortOnScore(names, scores)
    printLeaderBoard(names, scores)

def inp():
    N = int(input())
    names = []
    scores = []
    for i in range(N):
        name, score = input().split()
        names.append(name)
        scores.append(int(score))
    solve(names, scores)

inp()
```

Explanation of Each Part

1. `bubbleSortOnName(names, scores)`

```
def bubbleSortOnName(names, scores):
    N = len(names)
    for i in range(N-1):
        for j in range(N-i-1):
            if names[j] > names[j+1]:
                names[j], names[j+1] = names[j+1], names[j]
                scores[j], scores[j+1] = scores[j+1], scores[j]
```

- This function sorts the `names` list in **ascending** alphabetical order using a **bubble sort** approach.
- We perform `(N-1)` passes, and in each pass, we compare adjacent elements `names[j]` and `names[j+1]`.
- If `names[j]` is lexicographically **greater** than `names[j+1]`, we swap both `names` and their corresponding `scores`.
- By the end, `names` will be sorted alphabetically, and `scores` will be reordered accordingly so that each student's score remains aligned with their name.

2. `bubbleSortOnScore(names, scores)`

```
def bubbleSortOnScore(names, scores):
    N = len(names)
    for i in range(N-1):
        for j in range(N-i-1):
            if scores[j] < scores[j+1]:
                names[j], names[j+1] = names[j+1], names[j]
                scores[j], scores[j+1] = scores[j+1], scores[j]
```

- This function sorts the `scores` in **descending** order using another bubble sort pass.
- We again perform `(N-1)` passes, but this time we check if `scores[j] < scores[j+1]`.
- If so, we swap both the scores and the corresponding names to keep them aligned.
- After completion, the list will be ordered such that the highest score is at the front, moving down to the lowest score.

3. `printLeaderBoard(names, scores)`

```
def printLeaderBoard(names, scores):
    N = len(names)
    rank = 1
    for i in range(N):
        print(rank, names[i])
        if i != N-1 and scores[i] > scores[i+1]:
            rank = i + 2
```

- After sorting, this function prints out each student's **rank** and **name**.
- We start with `rank = 1`.
- For each student in the list:
  - We print the current `rank` and the student's `name`.
  - Then we check if the current student's score is greater than the **next** student's score:
    - If `scores[i] > scores[i+1]`, it means the next student has a strictly lower score, so the next student should get a new rank (i.e., `i+2`).
    - Otherwise, if the scores are the same, the next student shares the same rank.
- This logic effectively handles the case where multiple students have the same score (they share the same rank, and the next student with a lower score gets the next rank).

4. `solve(names, scores)`

```
def solve(names, scores):
    bubbleSortOnName(names, scores)
    bubbleSortOnScore(names, scores)
    printLeaderBoard(names, scores)
```

- The `solve` function orchestrates the process:
  i. Sort the list by name (alphabetically).
  ii. Sort by score (descending).
  iii. Print the leaderboard with the correct rank ordering.

5. `inp()` and Main Execution

```
def inp():
    N = int(input())
    names = []
    scores = []
    for i in range(N):
        name, score = input().split()
        names.append(name)
        scores.append(int(score))
    solve(names, scores)

inp()
```

- The `inp()` function:
  - Reads an integer `N` from input (number of students).
  - Initializes empty lists `names` and `scores` .
  - Repeats `N` times:
    - Reads a line, splits it into `name` (string) and `score` (integer).
    - Appends `name` to `names` and `score` to `scores` .
  - Calls the `solve(names, scores)` function to perform the sorting and ranking.
- Finally, we call `inp()` to run the entire process.

How It All Fits Together

1. The user inputs the number of students `N` , followed by `N` lines of name-score pairs.
2. `bubbleSortOnName` arranges students alphabetically, keeping each name-score pair intact.
3. `bubbleSortOnScore` then sorts these pairs by scores in descending order.
4. `printLeaderBoard` prints the rank and name for each student, adjusting the rank properly when scores differ.
5. As a result, you get a final sorted leaderboard with the highest scoring student(s) at the top, ties handled gracefully, and ranks assigned appropriately.

Question : Bubble Sort Problem

```
n = int(input())
l = list(map(int, input().split()))

for i in range(n):
    for j in range(0, n - i - 1):
        if l[j] > l[j + 1]:
            l[j], l[j + 1] = l[j + 1], l[j]

print(*l)
```

Explanation

1. **Reading Input**

    ```
    n = int(input())
    l = list(map(int, input().split()))
    ```

    - `n` is the number of elements in the array.
    - `l` is the list of unsorted numbers, obtained by splitting the input string into substrings and mapping each to an integer.

2. **Outer Loop**

    ```
    for i in range(n):
    ```

    - This loop runs `n` times. Each pass places the next-largest element in its correct position at the end of the list.

3. **Inner Loop**

    ```
    for j in range(0, n - i - 1):
        if l[j] > l[j + 1]:
            l[j], l[j + 1] = l[j + 1], l[j]
    ```

    - In each pass of the outer loop, the inner loop compares adjacent elements (`l[j]` and `l[j + 1]`) and swaps them if they are in the wrong order (`l[j] > l[j + 1]`).
    - The expression `range(0, n - i - 1)` ensures that with each outer loop pass, we do not re-check the already sorted elements at the end of the list.

- By the end of the first pass, the largest element “bubbles up” to the last position. By the end of the second pass, the second-largest is in its place, and so on.

4. **Printing the Result**

```
print(*l)
```

- The asterisk (`*`) unpacks the list, printing its elements separated by spaces in ascending order.

Bubble Sort works by repeatedly swapping adjacent elements if they are out of order, ensuring that with each pass, the largest remaining element is moved to its correct position at the end of the list.

Question : Again a classical problem

```
def solve(s):
    dic = {
        ')': '(',
        ']': '[',
        '}': '{'
    }
    stk = []
    for char in s:
        if char == "(" or char == "[" or char == "{":
            stk.append(char)
        else:
            if len(stk):
                # Notice this condition checks the same thing thrice; it's redundant but kept a:
                if dic[char] == stk[-1] or dic[char] == stk[-1] or dic[char] == stk[-1]:
                    stk.pop()
            else:
                return "not balanced"
    if len(stk):
        return "not balanced"
    return "balanced"


def inp():
    N = int(input())
    for i in range(N):
        s = input()
        ans = solve(s)
        print(ans)

inp()
```

Explanation

1. **Dictionary of Matching Brackets**

```python
dic = {
    ')': '(',
    ']': '[',
    '}': '{'
}
```

- This dictionary maps each **closing** bracket to its corresponding **opening** bracket.

2. **Stack Initialization**

```python
stk = []
```

- We use a list called `stk` as a stack to keep track of opening brackets.

3. **Iterating Through Each Character**

```python
for char in s:
    if char == "(" or char == "[" or char == "{":
        stk.append(char)
    else:
        ...
```

- For each character in the string `s`, we check if it is an **opening** bracket (`(`, `[`, `{`). If it is, we push it onto the stack.

4. **Handling Closing Brackets**

```python
else:
    if len(stk):
        if dic[char] == stk[-1] or dic[char] == stk[-1] or dic[char] == stk[-1]:
            stk.pop()
    else:
        return "not balanced"
```

- If the character is a **closing** bracket:
    - We first check if the stack is **not empty** (i.e., `len(stk) != 0`).
    - Then we compare the top of the stack (`stk[-1]`) with the expected matching bracket (`dic[char]`).
    - If they match, we pop from the stack (removing the matched opening bracket).
    - If they do **not** match (or the stack was empty), we return `"not balanced"`.

Note: The condition `if dic[char] == stk[-1] or dic[char] == stk[-1] or dic[char] == stk[-1]:` is effectively the same check repeated three times. It is redundant, but we are keeping it exactly as in the provided code.

5. **Final Stack Check**

```
if len(stk):
    return "not balanced"
return "balanced"
```

   - After processing all characters, if the stack is **empty**, it means every opening bracket had a matching closing bracket in the correct order, so the string is `"balanced"`.
   - If the stack is **not empty**, it means there are unmatched opening brackets left, so the string is `"not balanced"`.

6. `inp()` **Function**

```
def inp():
    N = int(input())
    for i in range(N):
        s = input()
        ans = solve(s)
        print(ans)
```

   - Reads the number of test cases `N`.
   - For each test case, reads a string `s`, calls `solve(s)`, and prints the result (`"balanced"` or `"not balanced"`).

7. **Main Call**

```
inp()
```

   - This line starts the process by calling `inp()`, which handles input and output for multiple test cases.

How It Works Overall:

1. For each test case (string of brackets), the program uses a stack to match every closing bracket with its corresponding opening bracket.
2. If at any point there is a mismatch or the stack is empty when expecting a match, the string is considered `"not balanced"`.
3. After processing the entire string, if the stack is empty, the string is `"balanced"`; otherwise, `"not balanced"`.

Question : Characters & 2D Array & Sum

```
n, m = map(int, input().split())
s = 0
for i in range(n):
    arr = input()
    for j in range(len(arr)):
        if arr[j] == "*":
            s += 0
        elif arr[j] == "/":
            s += 2
        else:
            s += 1
print(s)
```

Explanation of Each Part

1. **Reading the Dimensions**

   ```
   n, m = map(int, input().split())
   ```

   - The first line of input contains two integers, `n` and `m`.
   - `n` represents the number of rows in the 2D array.
   - `m` represents the number of columns in the 2D array.
   - We use `map(int, input().split())` to split the input string by spaces and convert each part to an integer.

2. **Initialize a Sum Variable**

   ```
   s = 0
   ```

   - We initialize a variable `s` to `0`. This will accumulate the sum based on the symbols we encounter in the 2D array.

3. **Reading Each Row**

   ```
   for i in range(n):
       arr = input()
       ...
   ```

   - We loop `n` times to read each row of the 2D array.

- For each iteration, we read a string `arr` from input. This string represents one row of the array.

4. **Iterating Over Each Character**

```
for j in range(len(arr)):
    if arr[j] == "*":
        s += 0
    elif arr[j] == "/":
        s += 2
    else:
        s += 1
```

- We loop over each character in the current row `arr`.
- Based on the character, we update the sum `s`:
    - If the character is `'*'`, we add `0`.
    - If the character is `'/'`, we add `2`.
    - Otherwise (for any other character), we add `1`.

5. **Printing the Final Sum**

```
print(s)
```

- After processing all rows and all characters, we print the accumulated sum `s`.

How It Works Overall

- The program first reads `n` and `m` from input, though it primarily uses `n` to know how many rows to read.
- For each row, it reads a string of length `m` (or possibly shorter/longer, depending on the input).
- Each character in the row contributes a certain value to the total sum:
    - `'*'` contributes `0`.
    - `'/'` contributes `2`.
    - Any other character contributes `1`.
- After reading and processing all rows, it prints the final sum `s`.

Question : The Peak Point

```
def solve(n, arr):
    for i in range(n - 1):
        if arr[i] > arr[i + 1]:
            return i

tc = int(input())
for i in range(tc):
    n = int(input())
    arr = list(map(int, input().split()))
    val = solve(n, arr)
    print(val)
```

Explanation of the Code

1. **Function Definition:** `solve(n, arr)`

   ```
   def solve(n, arr):
       for i in range(n - 1):
           if arr[i] > arr[i + 1]:
               return i
   ```

   - This function scans the array from index `0` to `n-2`.
   - It checks each element `arr[i]` against the next element `arr[i + 1]`.
   - As soon as it finds a position `i` where `arr[i] > arr[i + 1]`, it returns `i`.
   - Since the problem states the array first strictly increases and then strictly decreases, `i` is effectively the **peak index** (the point at which the sequence stops increasing and starts decreasing).
2. **Reading the Number of Test Cases**

   ```
   tc = int(input())
   ```

   - `tc` is the number of test cases we need to handle.
3. **Iterating Through Each Test Case**

```python
for i in range(tc):
    n = int(input())
    arr = list(map(int, input().split()))
    val = solve(n, arr)
    print(val)
```

- For each test case:
  - Read `n` , the size of the array.
  - Read the array `arr` of `n` integers.
  - Call `solve(n, arr)` to find the index of the peak.
  - Print the returned index.

How It Works Overall

- For each test case, we read the array.
- We then look for the first instance where the sequence goes from increasing to decreasing (i.e., where `arr[i] > arr[i+1]`).
- We return that `i` , which represents the 0-based index of the peak.
- Finally, we print this peak index for each test case.