1. Administrivia

- **Course Updates:**
    - **Question of the Day:**
        - Question Day 21 was due today.
        - Question Day 22 will be released later today (by noon).
    - **Homework:**
        - Homework 1 grading delayed; Homework 2 will be released Friday.
    - Grading scale update based on "true-up" calculations:
        - Grades reflect work completed so far (55% of the total).
        - Weights:
            - Questions of the Day: 5%.
            - Labs: 14% (lowest two scores dropped for now).
            - Midterms: 36% (18% each).
        - Encouragement to struggling students:
            - Seek help during office hours or from TAs.
            - Emphasis on effort and participation in determining passing grades.

---

2. Question of the Day 21: Recursive Roman Numeral Conversion

- **Problem:** Convert a Roman numeral string `R` into an integer recursively.
- **Framework for Recursive Solution:**
    - Decompose the string `R` into manageable parts.
    - Use a dictionary `values` for Roman numeral to integer mappings.
    - **Recursive Design:**
        - Base case:
            - When `R` is empty, return 0 (Roman numeral "null").
        - Recursive case:
            - Process the first character (or two, for special cases like `IV`).
            - Call the function recursively on the rest of the string.
    - **Key Considerations:**
        - Avoid modifying the global `values` dictionary during recursion.
        - Use the "head-tail" recursion pattern:
            - Process the head of the string and recurse on the tail.
    - **Implementation Highlights:**
        - Test for the base case (`len(R) == 0`).
        - Extract 1-2 characters from `R` as needed.
        - Lookup their value in the dictionary and add it to the recursive result.

---

3. Lab 10: The `Vector` Class

- **Purpose:** Explore object-oriented programming (OOP) concepts in Python.
- **Features of `Vector`:**
    - **Inheritance:**
        1. `Vector` extends the functionality of Python's `list` class.
        2. Inherits all `list` behaviors while adding specialized methods.
    - **Custom Methods:**
        1. **`__repr__`:**
            - Overrides the default representation to display vectors with angle brackets (`< >`) instead of square brackets (`[ ]`).
        2. **Scalar Magnitude:**
            - Computes the Euclidean distance of the vector from the origin: magnitude= sqrt($\sum$ element^2)
            - Rounds the result to 10 decimal places.
        3. **Unit Vector Normalization:**
            - Scales the vector to a magnitude of 1.
            - Handles division-by-zero errors (e.g., if the vector is `[0, 0, ...]`).
        4. **Dot Product:**
            - Computes the dot product of two vectors using Python's `zip()`: dot product= $\sum$element$_1$ * element$_2$
            - Demonstrates the use of `try-except` for error handling.

---

4. The One Address Machine (OAM)

- **Introduction:**
    - A simplified model of a von Neumann machine.
    - Simulates low-level machine operations to understand how computers execute instructions.
- **Von Neumann Architecture Overview:**
    - **Key Components:**
        - **Processor:** Executes instructions and performs calculations.
        - **Memory:** Stores data and instructions.
        - **Bus:** Transfers data between the processor and memory.
    - **Key Features:**
        - Programs and data share the same memory space.
        - Programs can modify themselves or other programs (treating instructions as data).

        - Input/Output interactions handled through memory-mapped addresses.
- **OAM Design Simplifications:**
    - **Infinite Memory:** No storage limitations.
    - **Memory-Mapped I/O:** Uses special memory locations for input (e.g., keyboard) and output (e.g., screen).
- **Processor Components:**
    - **Arithmetic Logic Unit (ALU):** Performs arithmetic and logical operations.
    - **Registers:**
        - **Program Counter (PC):** Tracks the address of the next instruction.
        - **Instruction Register (IR):** Stores the current instruction.
        - **Address Register (AR):** Handles memory address lookups.
        - **Accumulator (ACC):** Holds intermediate results.
        - **B Register:** Stores the second operand for operations.
    - **Instruction Cycle:**
        - **Fetch Phase:** Retrieve the next instruction from memory.
        - **Increment Phase:** Update the PC to point to the next instruction.
        - **Execute Phase:** Perform the operation specified by the instruction.

Next Steps:

- Continue exploring the OAM architecture in detail.
- Write OAM programs in machine language.
- Simulate the OAM in Python to observe instruction execution.
-