Problem

Write a python program to find all 10-key sequences that can be keyed into the keypad in the following manner:

The initial keypress can be any of the keys.
Each subsequent keypress must be a knight move from the previous keypress.
There can be at most 2 vowels in the sequence.

A knight move is made in one of the following ways:

Move two steps horizontally and one step vertically
Move two steps vertically and one step horizontally

There is no wrapping allowed on a knight move.

Your program should write the number of valid 10-key sequences on a single line to standard out.
Both top-down and bottom-up solutions are possible, so please rationalise your choice.

Implementation

Knight Moves

A knight in chess moves in an "L" shape:
- Two steps in one direction (either horizontally or vertically) and one step in the other direction.
- Therefore, the possible moves from a given position `(r, c)` on the keypad can be:
  - `(r + 2, c + 1)`
  - `(r + 2, c - 1)`
  - `(r - 2, c + 1)`
  - `(r - 2, c - 1)`
  - `(r + 1, c + 2)`
  - `(r + 1, c - 2)`
  - `(r - 1, c + 2)`
  - `(r - 1, c - 2)`

Vowels

We will treat vowels specially to enforce the rule that there can be at most 2 vowels in the sequence. The vowels on the keypad are:
- 'A', 'E', 'I', 'O'

Methodology

1. Representation of the Keypad: We'll represent the keypad as a 2D grid.
2. Recursion: The core idea is to use recursion to explore all possible sequences of length 10 starting from any key.
3. Memory optimisation: Since many sequences will overlap (i.e., you can reach the same key multiple times with the same remaining moves and vowel count), a dictionary is used to avoid recalculating results for the same state.

The methodology chosen to solve this problem is a top-down approach. A bottom-up approach is also possible to use, however, the reasons for this choice are:

- Ease of Implementation: The top-bottom approach is intuitive and straightforward to implement in this case.
- Memory Efficiency: Although both approaches are theoretically efficient, the top-down approach with the memory-saving dictionary efficiently handles the constraint of at most 2 vowels, pruning invalid sequences early without needing to track every possible state and storing only necessary intermediate states.

Because of the recursive nature, deep recursion could potentially hit stack limits, although Python's default recursion depth is usually sufficient for this problem.

Key Definitions

State:

- Current position on the keypad (row, column).
- Length of the sequence (how many keys have been pressed).
- The number of vowels used so far.

Transitions:

- From any key, you can move to any other key that can be reached by a knight move.
- For each valid move, increment the length and update the vowel count if the next key is a vowel.

Base Case:

When the sequence length reaches 10, the only valid sequence is the key itself.

Code walkthrough / summary

The main code is stored in main.py.

1. **Keypad Setup:** The keypad is set up as a 2D array, with None used to represent invalid positions.
2. **Knight Moves:** The possible knight moves are defined as pairs of (row_move, column_move).

3. **Memory Dictionary:** A dictionary memo stores already computed results for specific states (row_move, column_move, length, vowel_count) to avoid redundant computations.
4. **Recursive Function count_sequences:**
    - **Parameters:**
        - r and c are the current row and column on the keypad.
        - length is how many keypresses are have already taken place in the sequence.
        - vowel_count keeps track of how many vowels have been used so far.
    - **Base Case:** If length is 10, return 1 because this is the final move for this sequence of length 10.
    - **Dictionary Check:** Before computing, check if the result for the current state is already cached.
    - **Transition:** For each valid knight move, recurse to the next position, increase the length by 1, and update the vowel count if necessary.
    - **Store Result:** After calculating the number of sequences for the current state, store it in memo.
5. **Main Function total_valid_sequences:**
    - This function iterates over all starting positions on the keypad and sums up the number of valid sequences of length 10 starting from each key.

Unit testing

The tests.py code covers tests the main functionalities of the code.
Here's a breakdown of the tests:

1. test_is_valid: Checks if the is_valid function correctly identifies valid and invalid positions on the keypad.
2. test_count_sequences: Tests the count_sequences function with a few different scenarios.
3. test_total_valid_sequences: Verifies that total_valid_sequences returns a positive integer.
4. test_keypad_structure: Ensures that the keypad is structured correctly.
5. test_vowels: Checks that the vowels set contains the correct letters.
6. test_knight_moves: Verifies that the knight_moves list contains the correct moves.