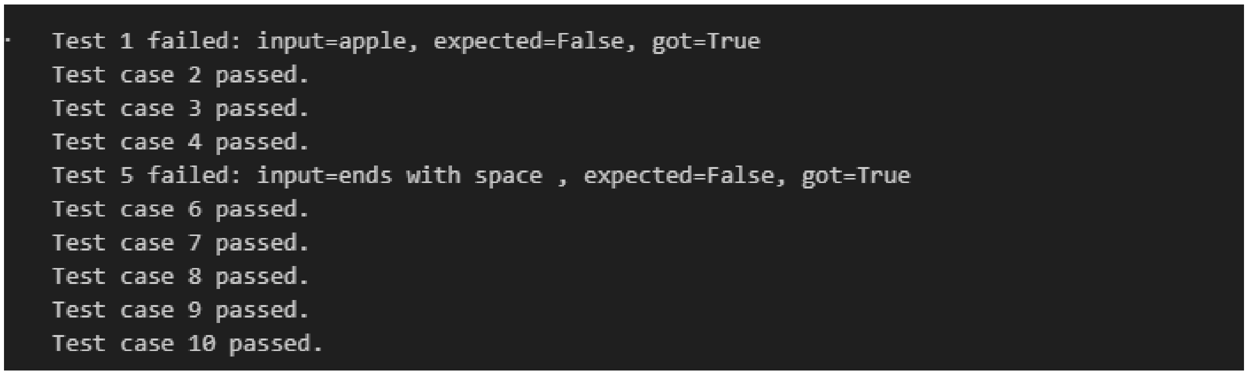
Group 83:

Alvin Yinglei Wu - 82929511
Dingxin Tao - 75864090

Option A

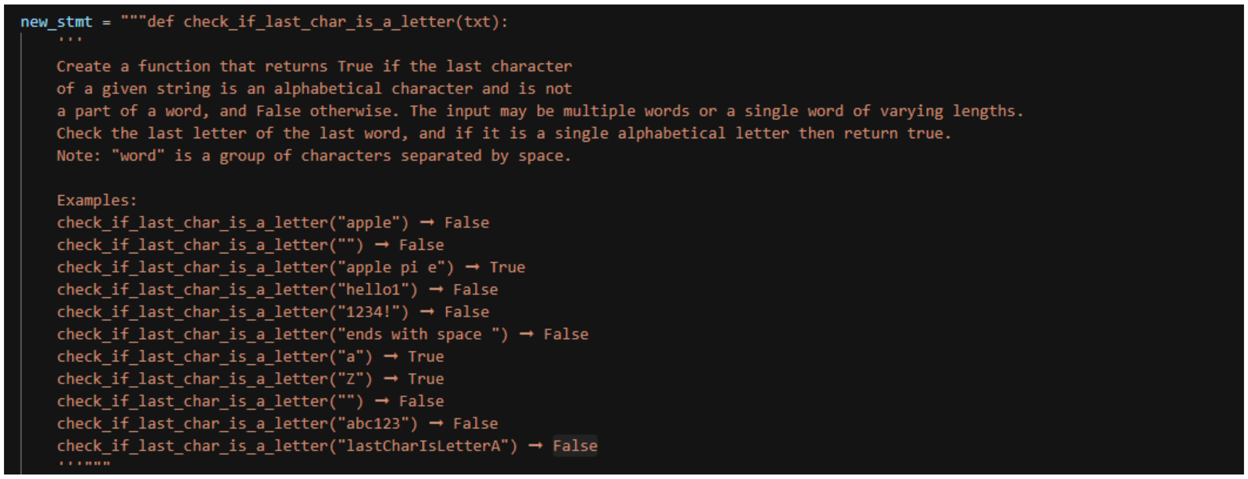
Observations: The more specific we were with our prompts, and especially with giving examples of expected output, the better our performance was overall. A big issue the AI was having was not properly accounting for edge cases and instead just writing the code EXACTLY within our specifications. By setting clearly defined rules (such as what constitutes a "word" or an "isolated letter"), and edge case scenarios, we gave the LLM a more accurate blueprint for generating correct code, This helped avoid overly narrow solutions like those seen in the original prompt, which failed by simply checking whether the last character was a letter without evaluating whether it was part of a word.

Original_stmt output:



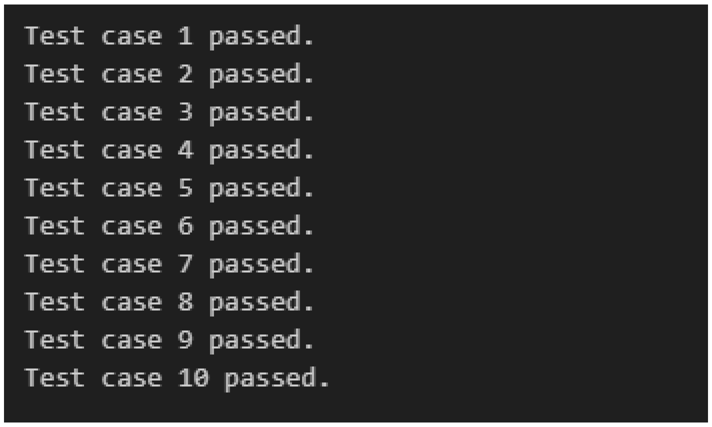
```
Test 1 failed: input=apple, expected=False, got=True
Test case 2 passed.
Test case 3 passed.
Test case 4 passed.
Test 5 failed: input=ends with space , expected=False, got=True
Test case 6 passed.
Test case 7 passed.
Test case 8 passed.
Test case 9 passed.
Test case 10 passed.
```

new_stmt prompt:



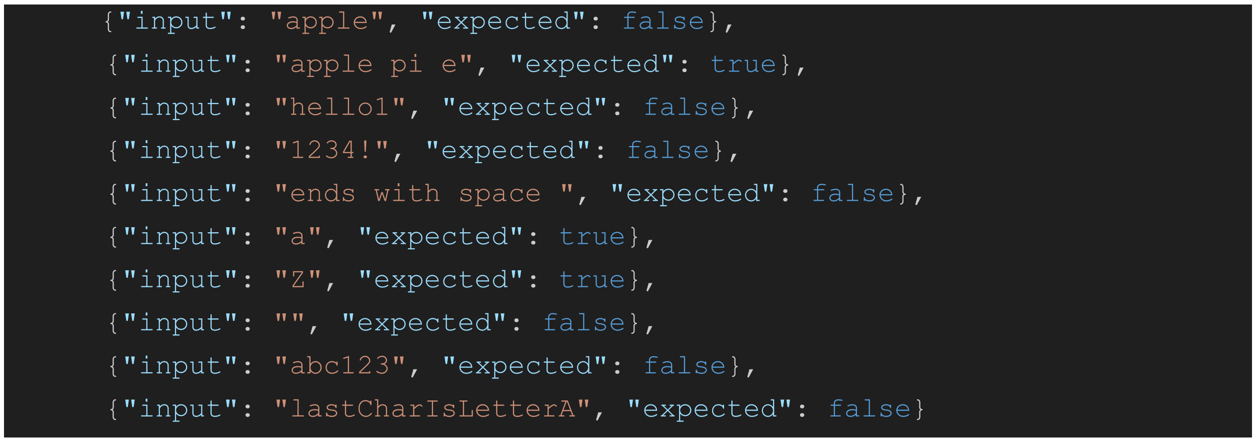
```
new_stmt = """def check_if_last_char_is_a_letter(txt):
    '''
    Create a function that returns True if the last character
    of a given string is an alphabetical character and is not
    a part of a word, and False otherwise. The input may be multiple words or a single word of varying lengths.
    Check the last letter of the last word, and if it is a single alphabetical letter then return true.
    Note: "word" is a group of characters separated by space.

    Examples:
    check_if_last_char_is_a_letter("apple") ➞ False
    check_if_last_char_is_a_letter("") ➞ False
    check_if_last_char_is_a_letter("apple pi e") ➞ True
    check_if_last_char_is_a_letter("hello1") ➞ False
    check_if_last_char_is_a_letter("1234!") ➞ False
    check_if_last_char_is_a_letter("ends with space ") ➞ False
    check_if_last_char_is_a_letter("a") ➞ True
    check_if_last_char_is_a_letter("Z") ➞ True
    check_if_last_char_is_a_letter("") ➞ False
    check_if_last_char_is_a_letter("abc123") ➞ False
    check_if_last_char_is_a_letter("lastCharIsLetterA") ➞ False
    '''"""
```

new_stmt output:



```
Test case 1 passed.
Test case 2 passed.
Test case 3 passed.
Test case 4 passed.
Test case 5 passed.
Test case 6 passed.
Test case 7 passed.
Test case 8 passed.
Test case 9 passed.
Test case 10 passed.
```

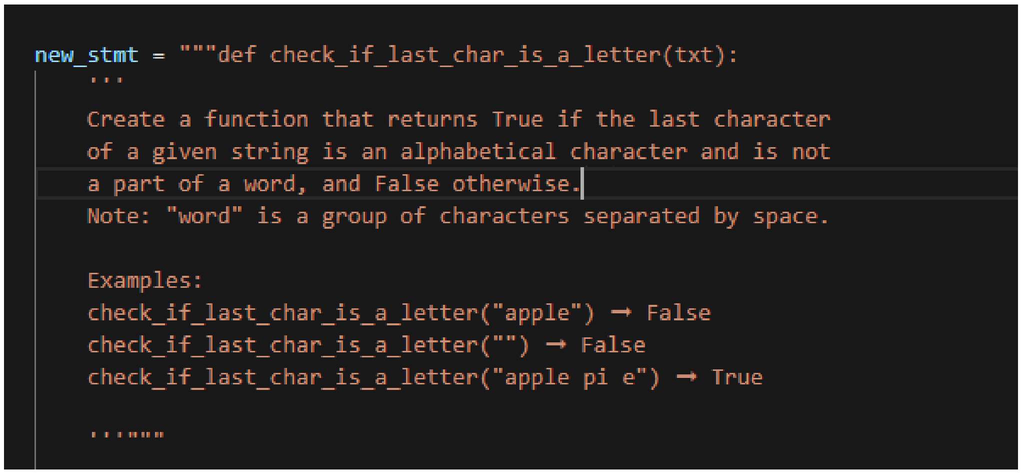
10 Test Cases:



```
        {"input": "apple", "expected": false},
        {"input": "apple pi e", "expected": true},
        {"input": "hello1", "expected": false},
        {"input": "1234!", "expected": false},
        {"input": "ends with space ", "expected": false},
        {"input": "a", "expected": true},
        {"input": "Z", "expected": true},
        {"input": "", "expected": false},
        {"input": "abc123", "expected": false},
        {"input": "lastCharIsLetterA", "expected": false}
```

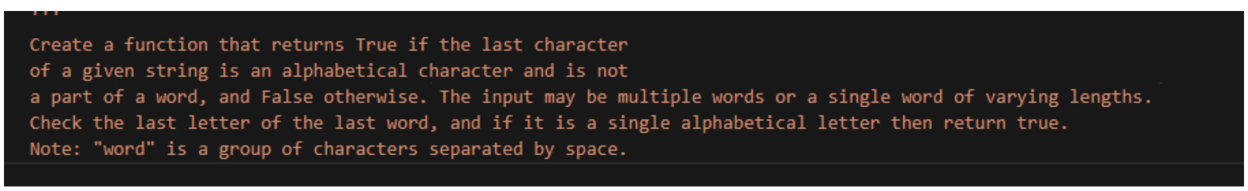
Prompts that improved performance:

1. Adding an edge case:



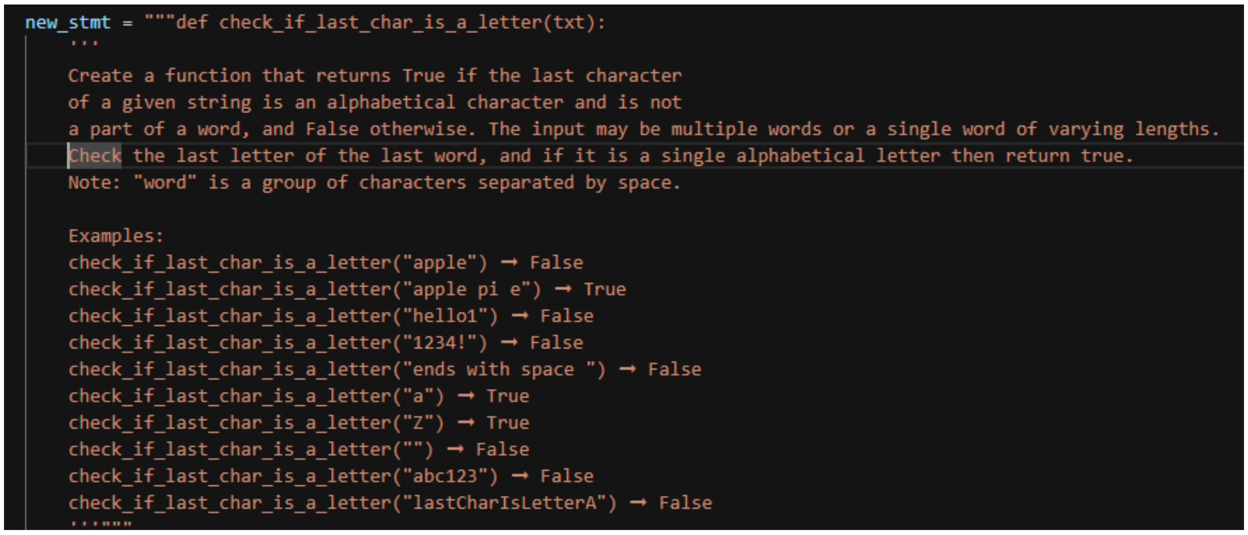
```
new_stmt = """def check_if_last_char_is_a_letter(txt):
    '''
    Create a function that returns True if the last character
    of a given string is an alphabetical character and is not
    a part of a word, and False otherwise.
    Note: "word" is a group of characters separated by space.

    Examples:
    check_if_last_char_is_a_letter("apple") → False
    check_if_last_char_is_a_letter("") → False
    check_if_last_char_is_a_letter("apple pi e") → True

    '''"""
```

2: Specifying input and output



```
    '''
    Create a function that returns True if the last character
    of a given string is an alphabetical character and is not
    a part of a word, and False otherwise. The input may be multiple words or a single word of varying lengths.
    Check the last letter of the last word, and if it is a single alphabetical letter then return true.
    Note: "word" is a group of characters separated by space.
```

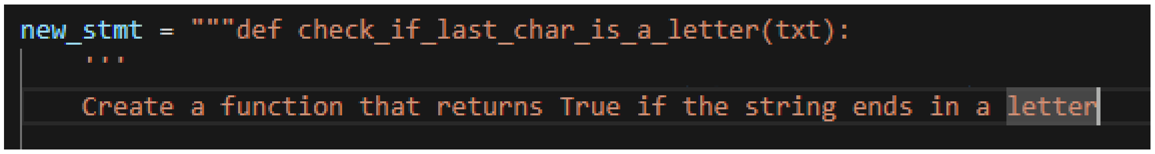
3: Specifying input and output and adding more test case examples



```
new_stmt = """def check_if_last_char_is_a_letter(txt):
    '''
    Create a function that returns True if the last character
    of a given string is an alphabetical character and is not
    a part of a word, and False otherwise. The input may be multiple words or a single word of varying lengths.
    Check the last letter of the last word, and if it is a single alphabetical letter then return true.
    Note: "word" is a group of characters separated by space.

    Examples:
    check_if_last_char_is_a_letter("apple") → False
    check_if_last_char_is_a_letter("apple pi e") → True
    check_if_last_char_is_a_letter("hello1") → False
    check_if_last_char_is_a_letter("1234!") → False
    check_if_last_char_is_a_letter("ends with space ") → False
    check_if_last_char_is_a_letter("a") → True
    check_if_last_char_is_a_letter("Z") → True
    check_if_last_char_is_a_letter("") → False
    check_if_last_char_is_a_letter("abc123") → False
    check_if_last_char_is_a_letter("lastCharIsLetterA") → False
    '''"""
```

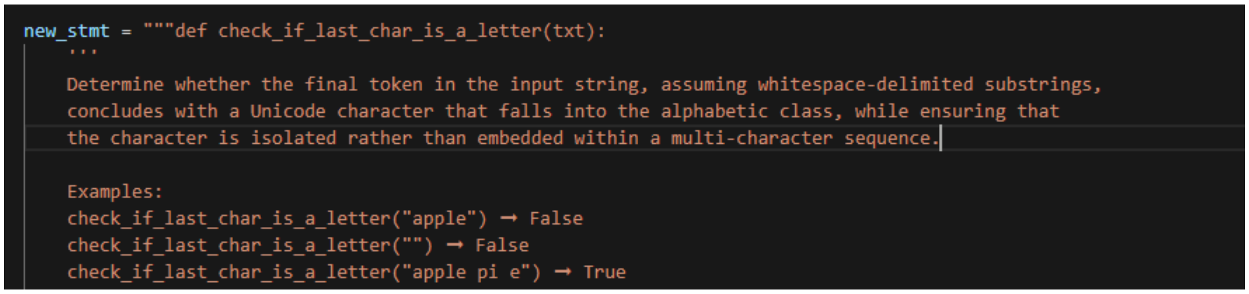
Prompts that decrease performance:

1: Vague prompt



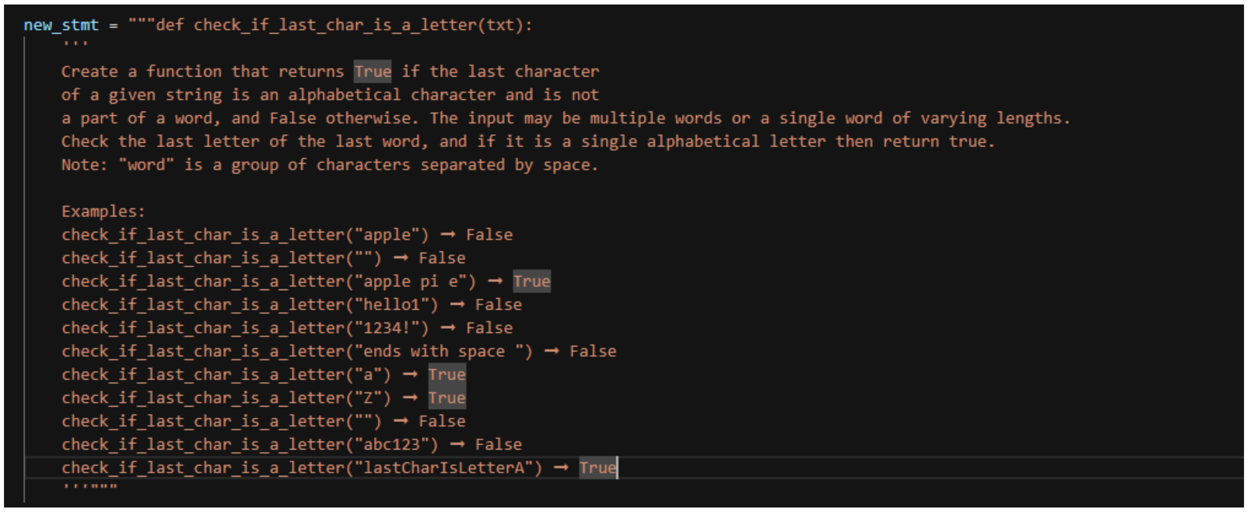
```
new_stmt = """def check_if_last_char_is_a_letter(txt):
    '''
    Create a function that returns True if the string ends in a letter
```

2: Overly Complicated and Ambiguous prompt



```
new_stmt = """def check_if_last_char_is_a_letter(txt):
    '''
    Determine whether the final token in the input string, assuming whitespace-delimited substrings,
    concludes with a Unicode character that falls into the alphabetic class, while ensuring that
    the character is isolated rather than embedded within a multi-character sequence.

    Examples:
    check_if_last_char_is_a_letter("apple") → False
    check_if_last_char_is_a_letter("") → False
    check_if_last_char_is_a_letter("apple pi e") → True
```

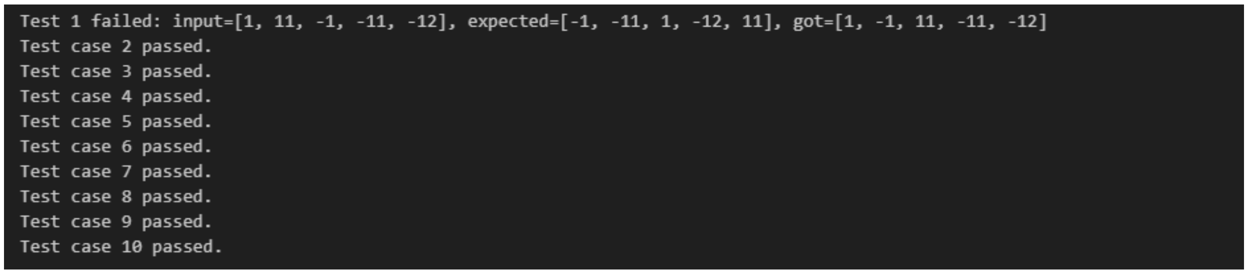
3: incorrect test case


```
new_stmt = """def check_if_last_char_is_a_letter(txt):
    '''
    Create a function that returns True if the last character
    of a given string is an alphabetical character and is not
    a part of a word, and False otherwise. The input may be multiple words or a single word of varying lengths.
    Check the last letter of the last word, and if it is a single alphabetical letter then return true.
    Note: "word" is a group of characters separated by space.

    Examples:
    check_if_last_char_is_a_letter("apple") ➞ False
    check_if_last_char_is_a_letter("") ➞ False
    check_if_last_char_is_a_letter("apple pi e") ➞ True
    check_if_last_char_is_a_letter("hello1") ➞ False
    check_if_last_char_is_a_letter("1234!") ➞ False
    check_if_last_char_is_a_letter("ends with space ") ➞ False
    check_if_last_char_is_a_letter("a") ➞ True
    check_if_last_char_is_a_letter("Z") ➞ True
    check_if_last_char_is_a_letter("") ➞ False
    check_if_last_char_is_a_letter("abc123") ➞ False
    check_if_last_char_is_a_letter("lastCharIsLetterA") ➞ True
    '''"""
```

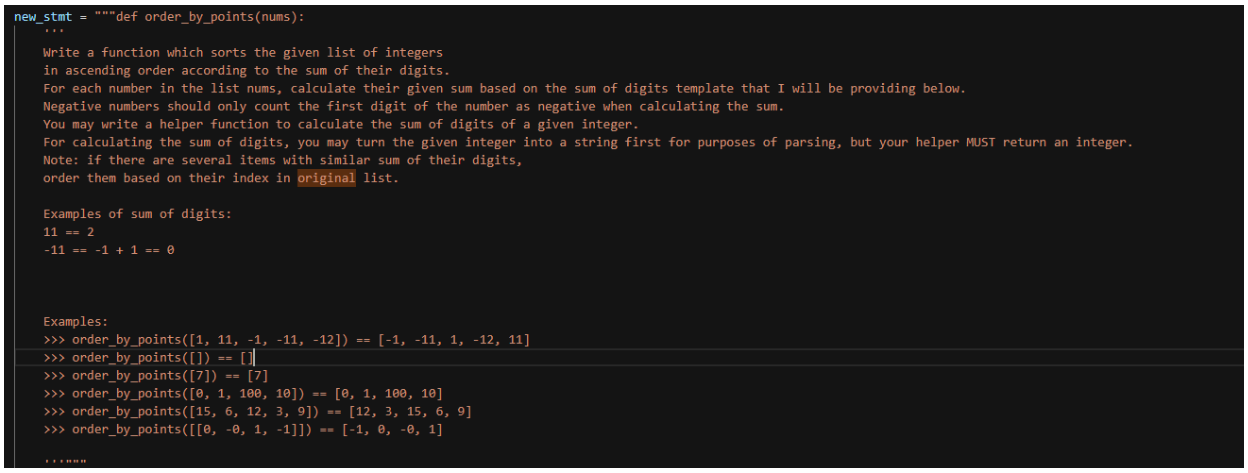
Option B

Observations: Similarly to Option A, the performance of the prompt in Option B improved significantly when we provided more specific instructions. The most notable improvement came from explicitly defining how the sum of digits should be calculated, which helped eliminate ambiguity around handling negative numbers or multi-digit values. Additionally, clearly stating the expected output format for the order_by_points() function made it easier for the LLM to generate correct and consistent results. One major factor that was reducing our LLM's performance was overloading it with excessive or contradictory information. One such example would be telling it to do things such as using loops/recursion without specifying why or how, overall confusing the LLM which often resulted in incorrect or dysfunctional code.

original _stmt output:

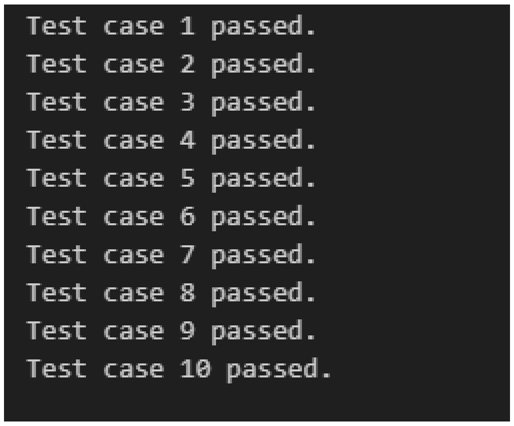

```
Test 1 failed: input=[1, 11, -1, -11, -12], expected=[-1, -11, 1, -12, 11], got=[1, -1, 11, -11, -12]
Test case 2 passed.
Test case 3 passed.
Test case 4 passed.
Test case 5 passed.
Test case 6 passed.
Test case 7 passed.
Test case 8 passed.
Test case 9 passed.
Test case 10 passed.
```

new_stmt Prompt:



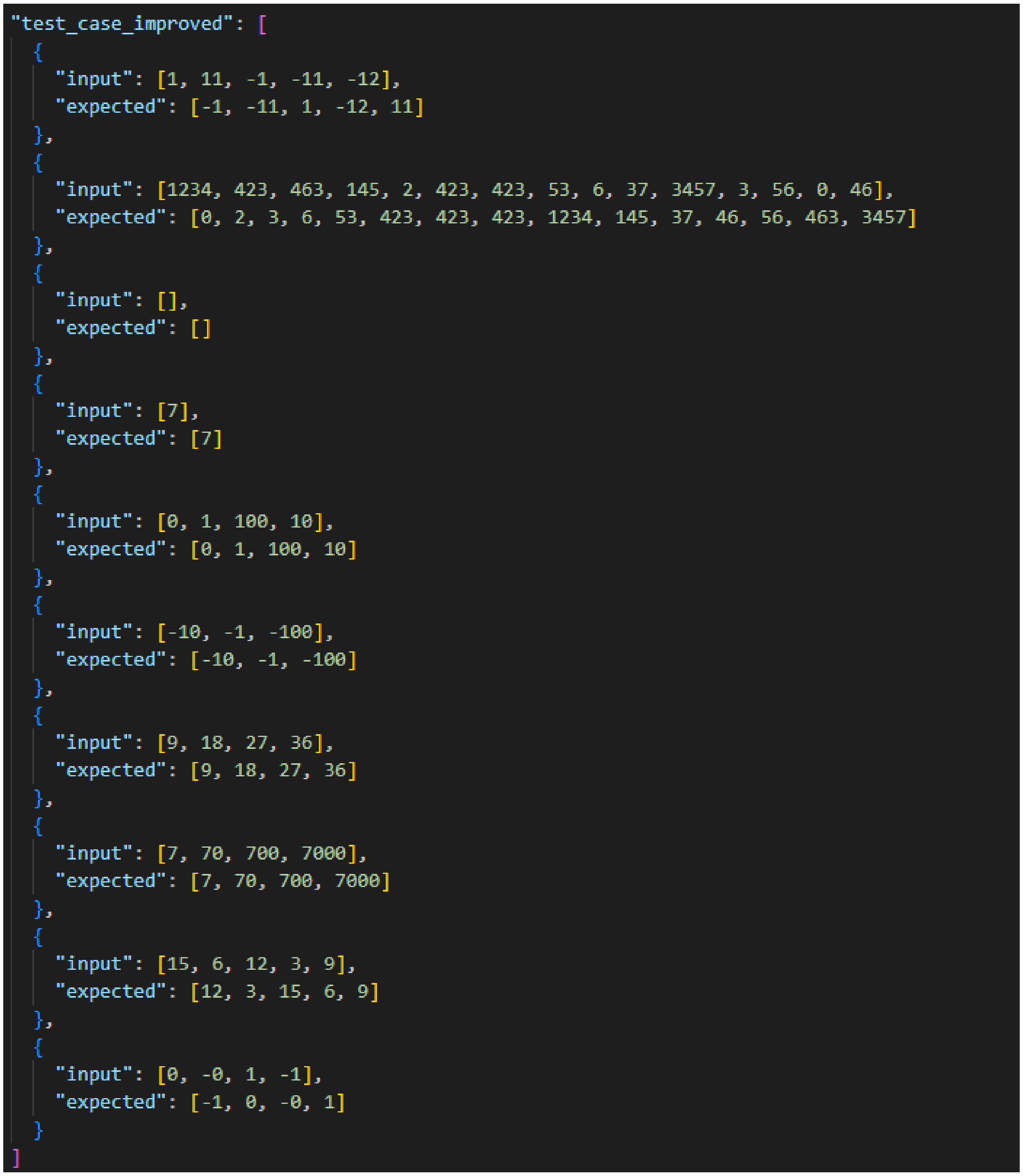
```
new_stmt = """def order_by_points(nums):
    '''
    Write a function which sorts the given list of integers
    in ascending order according to the sum of their digits.
    For each number in the list nums, calculate their given sum based on the sum of digits template that I will be providing below.
    Negative numbers should only count the first digit of the number as negative when calculating the sum.
    You may write a helper function to calculate the sum of digits of a given integer.
    For calculating the sum of digits, you may turn the given integer into a string first for purposes of parsing, but your helper MUST return an integer.
    Note: if there are several items with similar sum of their digits,
    order them based on their index in original list.

    Examples of sum of digits:
    11 == 2
    -11 == -1 + 1 == 0



    Examples:
    >>> order_by_points([1, 11, -1, -11, -12]) == [-1, -11, 1, -12, 11]
    >>> order_by_points([]) == []
    >>> order_by_points([7]) == [7]
    >>> order_by_points([0, 1, 100, 10]) == [0, 1, 100, 10]
    >>> order_by_points([15, 6, 12, 3, 9]) == [12, 3, 15, 6, 9]
    >>> order_by_points([[0, -0, 1, -1]]) == [-1, 0, -0, 1]

    '''"""
```

new_stmt output:



```
Test case 1 passed.
Test case 2 passed.
Test case 3 passed.
Test case 4 passed.
Test case 5 passed.
Test case 6 passed.
Test case 7 passed.
Test case 8 passed.
Test case 9 passed.
Test case 10 passed.
```

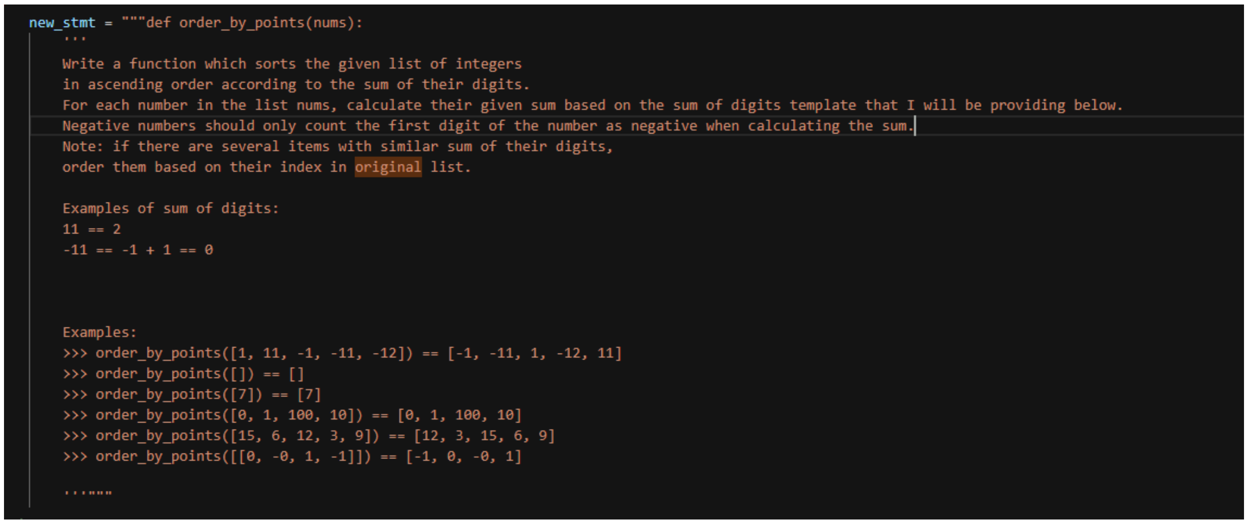
10 Test Cases:



```
"test_case_improved": [
  {
    "input": [1, 11, -1, -11, -12],
    "expected": [-1, -11, 1, -12, 11]
  },
  {
    "input": [1234, 423, 463, 145, 2, 423, 423, 53, 6, 37, 3457, 3, 56, 0, 46],
    "expected": [0, 2, 3, 6, 53, 423, 423, 423, 1234, 145, 37, 46, 56, 463, 3457]
  },
  {
    "input": [],
    "expected": []
  },
  {
    "input": [7],
    "expected": [7]
  },
  {
    "input": [0, 1, 100, 10],
    "expected": [0, 1, 100, 10]
  },
  {
    "input": [-10, -1, -100],
    "expected": [-10, -1, -100]
  },
  {
    "input": [9, 18, 27, 36],
    "expected": [9, 18, 27, 36]
  },
  {
    "input": [7, 70, 700, 7000],
    "expected": [7, 70, 700, 7000]
  },
  {
    "input": [15, 6, 12, 3, 9],
    "expected": [12, 3, 15, 6, 9]
  },
  {
    "input": [0, -0, 1, -1],
    "expected": [-1, 0, -0, 1]
  }
]
```

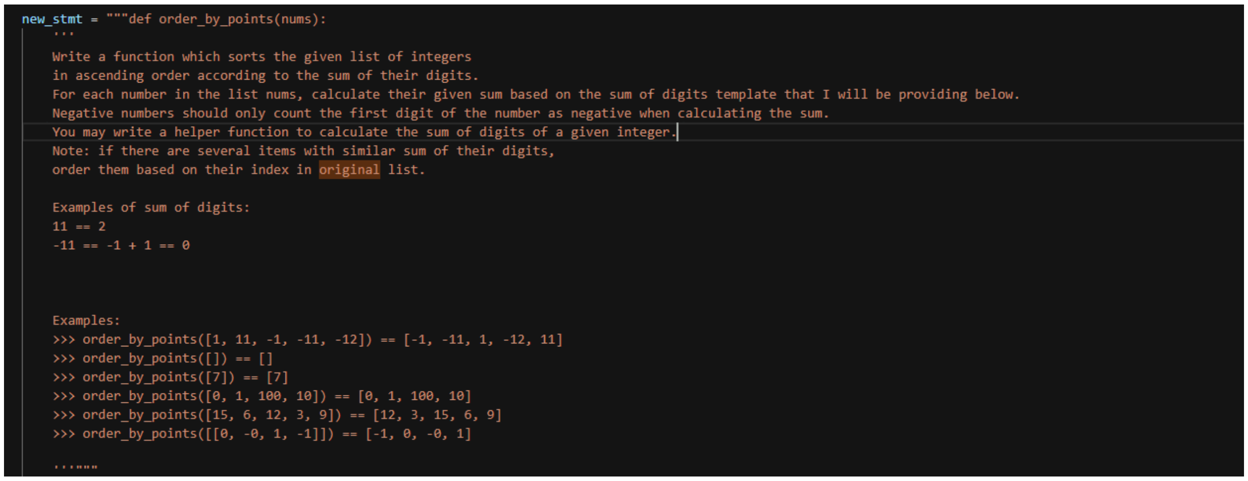
Prompts that improve performance:

1: Provide a visual example of parsing a number into sum of digits



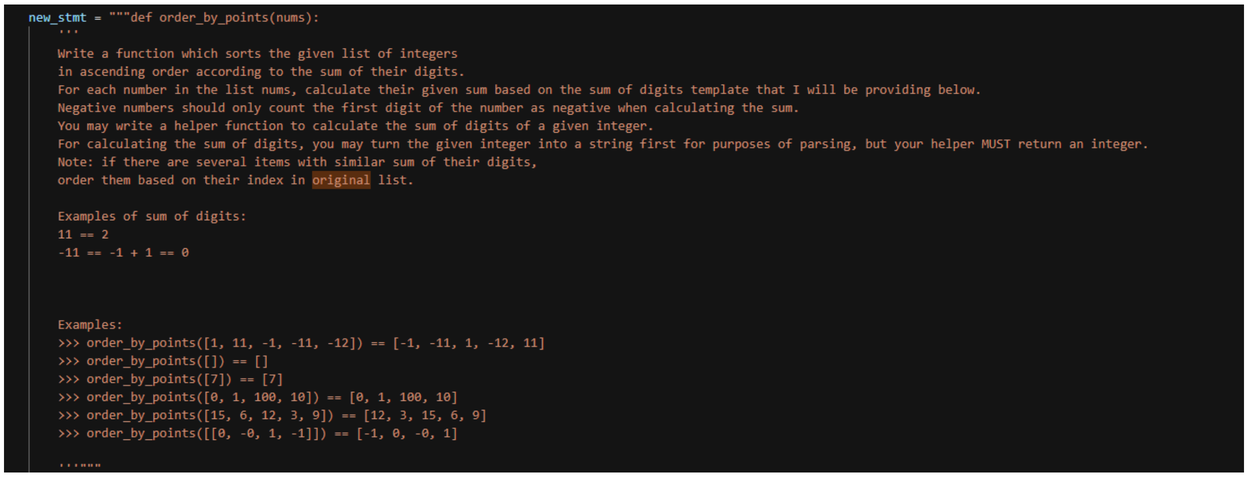
```
new_stmt = """def order_by_points(nums):
    '''
    Write a function which sorts the given list of integers
    in ascending order according to the sum of their digits.
    For each number in the list nums, calculate their given sum based on the sum of digits template that I will be providing below.
    Negative numbers should only count the first digit of the number as negative when calculating the sum.
    Note: if there are several items with similar sum of their digits,
    order them based on their index in original list.

    Examples of sum of digits:
    11 == 2
    -11 == -1 + 1 == 0



    Examples:
    >>> order_by_points([1, 11, -1, -11, -12]) == [-1, -11, 1, -12, 11]
    >>> order_by_points([]) == []
    >>> order_by_points([7]) == [7]
    >>> order_by_points([0, 1, 100, 10]) == [0, 1, 100, 10]
    >>> order_by_points([15, 6, 12, 3, 9]) == [12, 3, 15, 6, 9]
    >>> order_by_points([[0, -0, 1, -1]]) == [-1, 0, -0, 1]

    '''"""
```

2: Allow the use of a helper for calculating the sum of a singular digit



```
new_stmt = """def order_by_points(nums):
    '''
    Write a function which sorts the given list of integers
    in ascending order according to the sum of their digits.
    For each number in the list nums, calculate their given sum based on the sum of digits template that I will be providing below.
    Negative numbers should only count the first digit of the number as negative when calculating the sum.
    You may write a helper function to calculate the sum of digits of a given integer.
    Note: if there are several items with similar sum of their digits,
    order them based on their index in original list.

    Examples of sum of digits:
    11 == 2
    -11 == -1 + 1 == 0



    Examples:
    >>> order_by_points([1, 11, -1, -11, -12]) == [-1, -11, 1, -12, 11]
    >>> order_by_points([]) == []
    >>> order_by_points([7]) == [7]
    >>> order_by_points([0, 1, 100, 10]) == [0, 1, 100, 10]
    >>> order_by_points([15, 6, 12, 3, 9]) == [12, 3, 15, 6, 9]
    >>> order_by_points([[0, -0, 1, -1]]) == [-1, 0, -0, 1]

    '''"""
```

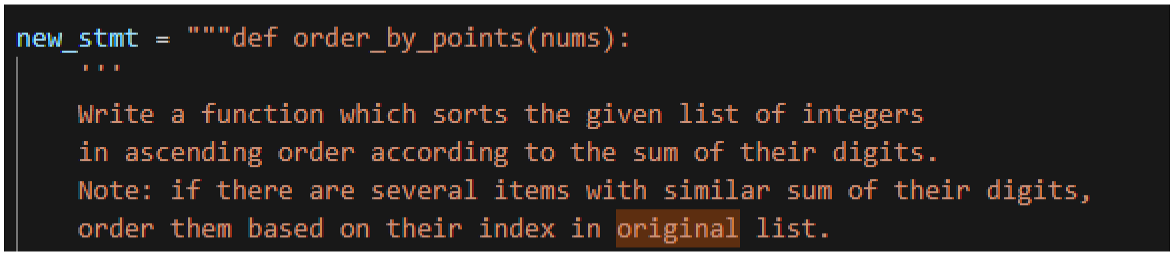
3: Allow the helper function to convert the number into a string for easier parsing as well as specifying that the helper must return an integer as opposed to an ambiguous return value



```
new_stmt = """def order_by_points(nums):
    '''
    Write a function which sorts the given list of integers
    in ascending order according to the sum of their digits.
    For each number in the list nums, calculate their given sum based on the sum of digits template that I will be providing below.
    Negative numbers should only count the first digit of the number as negative when calculating the sum.
    You may write a helper function to calculate the sum of digits of a given integer.
    For calculating the sum of digits, you may turn the given integer into a string first for purposes of parsing, but your helper MUST return an integer.
    Note: if there are several items with similar sum of their digits,
    order them based on their index in original list.

    Examples of sum of digits:
    11 == 2
    -11 == -1 + 1 == 0



    Examples:
    >>> order_by_points([1, 11, -1, -11, -12]) == [-1, -11, 1, -12, 11]
    >>> order_by_points([]) == []
    >>> order_by_points([7]) == [7]
    >>> order_by_points([0, 1, 100, 10]) == [0, 1, 100, 10]
    >>> order_by_points([15, 6, 12, 3, 9]) == [12, 3, 15, 6, 9]
    >>> order_by_points([[0, -0, 1, -1]]) == [-1, 0, -0, 1]

    '''"""
```

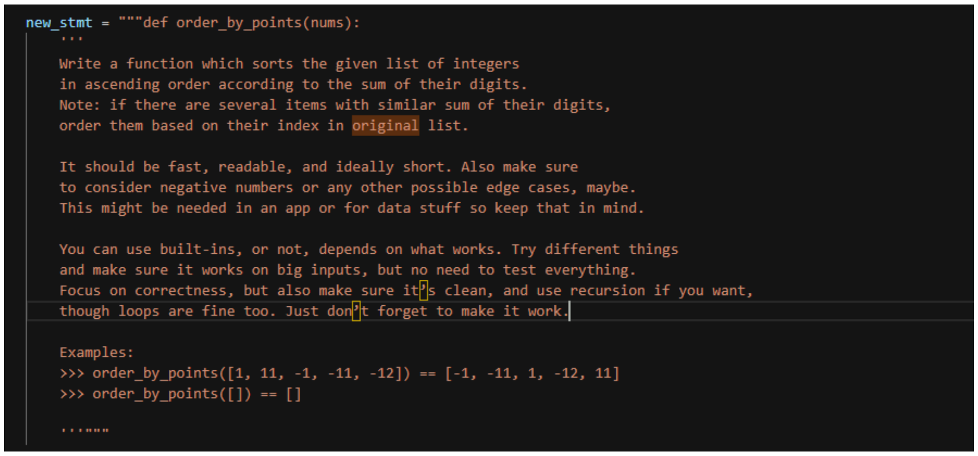
Prompts that decrease performance:

1: No example test cases



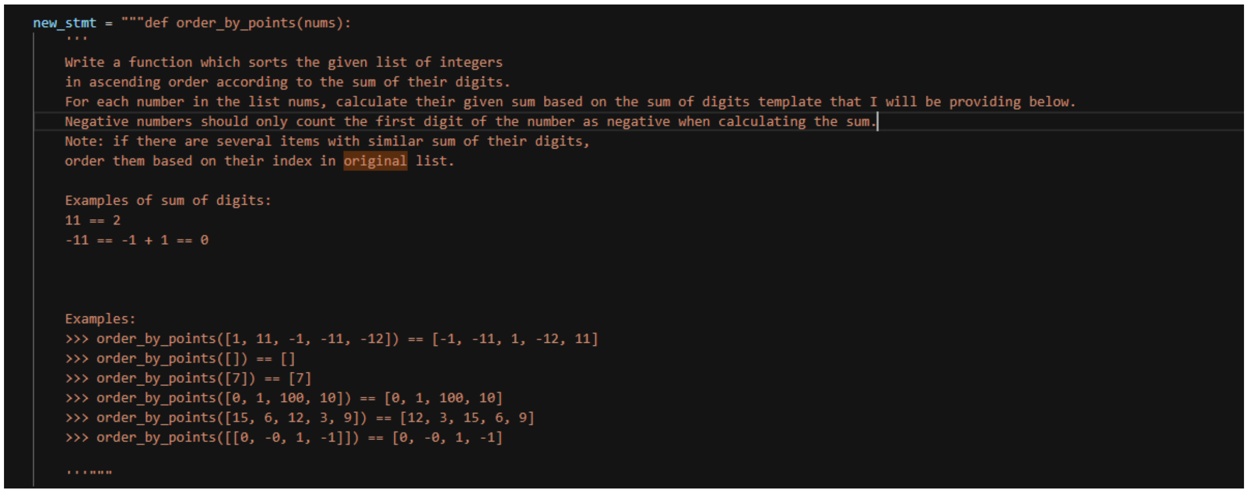
```
new_stmt = """def order_by_points(nums):
    '''
    Write a function which sorts the given list of integers
    in ascending order according to the sum of their digits.
    Note: if there are several items with similar sum of their digits,
    order them based on their index in original list.
```

2: Overloading the LLM with contradictory information



```
new_stmt = """def order_by_points(nums):
    '''
    Write a function which sorts the given list of integers
    in ascending order according to the sum of their digits.
    Note: if there are several items with similar sum of their digits,
    order them based on their index in original list.

    It should be fast, readable, and ideally short. Also make sure
    to consider negative numbers or any other possible edge cases, maybe.
    This might be needed in an app or for data stuff so keep that in mind.

    You can use built-ins, or not, depends on what works. Try different things
    and make sure it works on big inputs, but no need to test everything.
    Focus on correctness, but also make sure it's clean, and use recursion if you want,
    though loops are fine too. Just don't forget to make it work.

    Examples:
    >>> order_by_points([1, 11, -1, -11, -12]) == [-1, -11, 1, -12, 11]
    >>> order_by_points([]) == []

    '''"""
```

3: For some reason confuses the LLM into calculating average



```
new_stmt = """def order_by_points(nums):
    '''
    Write a function which sorts the given list of integers
    in ascending order according to the sum of their digits.
    For each number in the list nums, calculate their given sum based on the sum of digits template that I will be providing below.
    Negative numbers should only count the first digit of the number as negative when calculating the sum.
    Note: if there are several items with similar sum of their digits,
    order them based on their index in original list.

    Examples of sum of digits:
    11 == 2
    -11 == -1 + 1 == 0



    Examples:
    >>> order_by_points([1, 11, -1, -11, -12]) == [-1, -11, 1, -12, 11]
    >>> order_by_points([]) == []
    >>> order_by_points([7]) == [7]
    >>> order_by_points([0, 1, 100, 10]) == [0, 1, 100, 10]
    >>> order_by_points([15, 6, 12, 3, 9]) == [12, 3, 15, 6, 9]
    >>> order_by_points([[0, -0, 1, -1]]) == [0, -0, 1, -1]

    '''"""
```