Dictionary coding exercise:

1. Problem 1:

Count Word Frequency
Define a function to count the frequency of words in a given list of words.

Example:

```
words = ['apple', 'orange', 'banana', 'apple', 'orange', 'apple']
count_word_frequency(words)
```

Output:

```
{'apple': 3, 'orange': 2, 'banana': 1}
```

```
def count_word_frequency(words):
  word_count = {}
  for word in words:
     word_count[word] = word_count.get(word, 0) + 1
  return word_count
```

Explanation:

- Define a function **count_word_frequency(words)** that takes a list of words as its input.
- Initialize an empty dictionary **word_count** to store the frequency of each word in the list.
- Iterate through the list of words using a for loop.
- For each word, use the **get()** method to retrieve the current count of the word in the **word_count** dictionary. If the word is not yet present in the dictionary, **get()** returns the default value (0). Then, increment the count by 1 and update the dictionary.
- After iterating through all the words, return the **word_count** dictionary containing the word frequencies.

Time complexity:

The time complexity of this exercise is O(n), where n is the number of words in the input list. The loop iterates through each word in the list once, and the dictionary operations (get and update) take constant time, O(1), on average.

Space complexity:

The space complexity of this exercise is also O(n), where n is the number of unique words in the input list. In the worst case, all words are unique, and the **word_count** dictionary will have n entries.

2. Problem 2:

Common Keys

Define a function with takes two dictionaries as parameters and merge them and sum the values of common keys.

Example:

```
dict1 = {'a': 1, 'b': 2, 'c': 3}
dict2 = {'b': 3, 'c': 4, 'd': 5}
merge_dicts(dict1, dict2)
```

Output:

```
{'a': 1, 'b': 5, 'c': 7, 'd': 5}
```

```
def merge_dicts(dict1, dict2):
  result = dict1.copy()
  for key, value in dict2.items():
     result[key] = result.get(key, 0) + value
  return result
```

This code snippet defines a function **merge_dicts(dict1, dict2)** that merges two dictionaries and sums the values of common keys.

Explanation:

1. Create a copy of **dict1** named **result**. This ensures that the original **dict1** is not modified during the process. The **copy()** method takes O(n) time complexity, where n is the number of elements in **dict1**. The space complexity is also O(n) as a new dictionary is created with the same number of elements as **dict1**.
2. Iterate through the key-value pairs of **dict2** using a for loop. This loop runs for m iterations, where m is the number of elements in **dict2**. For each key-value pair:
   a. Use the **get()** method to retrieve the current value associated with the key in the **result** dictionary. If the key is not present in **result**, **get()** returns the default value (0).
   b. Add the value from **dict2** to the current value (or 0, if the key is not in **result**) and update the **result** dictionary with the new value for the key. The **get()** and update operations take O(1) average time complexity.
3. Return the merged dictionary **result**.

Time complexity:

The overall time complexity of this function is O(n + m), where n is the number of elements in **dict1** and m is the number of elements in **dict2**. The **copy()** method takes O(n) time, and the loop iterates m times with O(1) operations inside the loop.

Space complexity:

The space complexity of this function is O(n + m) in the worst case, where all keys in **dict1** and **dict2** are distinct, and the merged dictionary has n + m elements. In the best case, where **dict1** and **dict2** have the same keys, the space complexity is O(n) (or O(m), whichever is larger), as the merged dictionary has the same number of elements as the input dictionaries.

3. Problem 3:

Key with the Highest Value

Define a function which takes a dictionary as a parameter and returns the key with the highest value in a dictionary.

Example:

```
my_dict = {'a': 5, 'b': 9, 'c': 2}
max_value_key(my_dict))
```

Output:

```
b
```

```
def max_value_key(my_dict):
  return max(my_dict, key=my_dict.get)
```

Explanation:

Call the built-in **max()** function with the dictionary **my_dict** as its first argument and **key=my_dict.get** as its second argument.
The **max()** function iterates through the keys in the dictionary and compares the values associated with each key using the **get()** method. The **key** parameter specifies a custom function to extract a comparison key from each dictionary key. In this case, the **get()** method returns the value associated with each key, so the **max()** function compares the values of the dictionary.
The time complexity of the **max()** function is O(n), where n is the number of elements in the dictionary **my_dict**. The **max()** function iterates through all the keys in the dictionary once, and the **get()** method has an average time complexity of O(1).

Return the key with the highest value found by the **max()** function.
Time complexity:

The overall time complexity of this function is O(n), where n is the number of elements in the dictionary **my_dict**. This is determined by the **max()** function, which iterates through all the keys in the dictionary.

Space complexity:

The space complexity of this function is O(1), as it does not create any additional data structures or store any intermediate values. The **max()** function only keeps track of the current maximum value and its corresponding key, which requires constant space.

4. Problem 4:

Reverse Key-Value Pairs

Define a function which takes as a parameter dictionary and returns a dictionary in which everse the key-value pairs are reversed.

Example:

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
reverse_dict(my_dict)
```

Output:

```
{1: 'a', 2: 'b', 3: 'c'}
```

```
def reverse_dict(my_dict):
  return {v: k for k, v in my_dict.items()}
```

Explanation

Use dictionary comprehension to create a new dictionary by iterating through the key-value pairs in the input dictionary **my_dict** using the **items()** method. The **items()** method returns an iterable that produces key-value pairs as tuples.
For each key-value pair (k, v) from the input dictionary, the dictionary comprehension creates a new entry in the reversed dictionary with the value **v** as the key and the key **k** as the value. The syntax is **{v: k for k, v in my_dict.items()}**.
The time complexity of this operation is O(n), where n is the number of elements in the dictionary **my_dict** The dictionary comprehension iterates through all the key-value pairs in the input dictionary once.

Return the new dictionary with the reversed key-value pairs.
Time complexity:

The overall time complexity of this function is O(n), where n is the number of elements in the dictionary **my_dict**. This is determined by the dictionary comprehension, which iterates through all the key-value pairs in the input dictionary.

Space complexity:

The space complexity of this function is O(n), where n is the number of elements in the dictionary **my_dict**. This is because the function creates a new dictionary with the same number of elements as the input dictionary but with reversed key-value pairs.

5. Problem 5:

Conditional Filter

Define a function that takes a dictionary as a parameter and returns a dictionary with elements based on a condition.

Example:

```
my_dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
filtered_dict = filter_dict(my_dict, lambda k, v: v % 2 == 0)
```

Output:

```
{'b': 2, 'd': 4}
```

```
def filter_dict(my_dict, condition):
  return {k: v for k, v in my_dict.items() if condition(k, v)}
```

Explanation:

Use dictionary comprehension to create a new dictionary by iterating through the key-value pairs in the input dictionary **my_dict** using the **items()** method. The **items()** method returns an iterable that produces key-value pairs as tuples.
For each key-value pair (k, v) from the input dictionary, the dictionary comprehension checks if the condition is met by calling the **condition(k, v)** function. The condition function takes a key and a value as arguments and returns a boolean value.
If the condition is met, the dictionary comprehension creates a new entry in the filtered dictionary with the same key-value pair. The syntax is **{k: v for k, v in my_dict.items() if condition(k, v)}**.
The time complexity of this operation is O(n), where n is the number of elements in the dictionary **my_dict**. The dictionary comprehension iterates through all the key-value pairs in the input dictionary once.

Return the new dictionary containing the filtered key-value pairs.

Time complexity:

The overall time complexity of this function is O(n), where n is the number of elements in the dictionary **my_dict**. This is determined by the dictionary comprehension, which iterates through all the key-value pairs in the input dictionary.

Space complexity:

The space complexity of this function depends on the number of elements in the filtered dictionary, which in turn depends on the condition function. In the worst case, when all key-value pairs meet the condition, the space complexity is O(n), where n is the number of elements in the dictionary **my_dict**. In the best case, when no key-value pairs meet the condition, the space complexity is O(1) as the function creates an empty dictionary.

6. Problem 6:

Same Frequency

Define a function which takes two lists as parameters and check if two given lists have the same frequency of elements.

Example:

```
list1 = [1, 2, 3, 2, 1]
list2 = [3, 1, 2, 1, 3]
check_same_frequency(list1, list2)
```

Output:

False

```
def check_same_frequency(list1, list2):
  def count_elements(lst):
    counter = {}
    for element in lst:
      counter[element] = counter.get(element, 0) + 1
    return counter

  return count_elements(list1) == count_elements(list2)
```

Explanation:

check_same_frequency(list1, list2) checks if two given lists have the same frequency of elements.

Define an inner function **count_elements(lst)** that counts the frequency of elements in a list **lst**. The function creates an empty dictionary **counter**, iterates through the list, and increments the count for each element using the **get()** method. The **get()** method takes O(1) average time complexity. The function returns the **counter** dictionary. The time complexity of **count_elements()** is O(n), where n is the number of elements in the input list **lst**. The space complexity is also O(n), as the dictionary **counter** has as many keys as there are distinct elements in the list.

Call the **count_elements()** function on both input lists **list1** and **list2**. This operation has a time complexity of O(n1 + n2), where n1 and n2 are the lengths of **list1** and **list2**, respectively. The space complexity is O(m1 + m2), where m1 and m2 are the numbers of distinct elements in **list1** and **list2**, respectively.

Compare the resulting dictionaries using the == operator. This operation has a time complexity of O(min(m1, m2)), as it compares the keys and values of both dictionaries.

Return the result of the comparison (True if the dictionaries are equal, False otherwise).

Time complexity:

The overall time complexity of this function is O(n1 + n2 + min(m1, m2)), where n1 and n2 are the lengths of **list1** and **list2**, and m1 and m2 are the numbers of distinct elements in **list1** and **list2**, respectively. This is determined by the time complexity of the **count_elements()** function and the dictionary comparison.

Space complexity:

The space complexity of this function is O(m1 + m2), where m1 and m2 are the numbers of distinct elements in **list1** and **list2**, respectively. This is because the function creates two dictionaries with as many keys as there are distinct elements in the input lists.

7.