```
#-------------------------------------------------
# 1111111111111111111111111111111111111111111111
# draw_histogram()
#-------------------------------------------------
#-------------------------------------------------
"""
Define the draw_histogram() function which is passed a Python
dictionary as a parameter. The keys of the dictionary are single
letters
and the corresponding values are integers, e.g., {'b': 5, 'a': 6,
'c': 3}. For each key:value pair in the dictionary the function
prints
the key, followed by ": ", followed by a series of stars. The
number of stars printed is given by the value corresponding to
the key.
The keys are printed in alphabetical order. Note that the key is
not printed if the corresponding value is a number less than 1.

For example, the following code:

print("1.")

draw_histogram({'a': 2, 'c': 7, 'b': 5})

print("2.")

draw_histogram({'a': 0, 'c': 5, 'b': 7, 'f': 0})

prints:

1.
a: **
b: *****
c: *******
2.
b: *******
c: *****
"""
#-------------------------------------------------
# 2222222222222222222222222222222222222222222222
# print_word_length_dictionary()
#-------------------------------------------------
"""
Define the print_word_length_dictionary() function which is
passed a tuple of strings as a parameter.
The print_word_length_dictionary() function creates a dictionary
where the keys are the lengths of the
words encountered in the tuple, and the values are a list of
words of that length. The function then
```

```
prints the keys and values in this dictionary such that:
- Each key/value pair is printed on a separate line.
- The keys (word lengths) are printed in ascending order of
magnitude.
- The values (list of words) are sorted in alphabetical order.

For example the following code:

print_word_length_dictionary(("hello","world","students","computer"
, "science", "auckland", "cats"))
prints

4: ['cats']
5: ['hello', 'world']
7: ['science']
8: ['auckland', 'computer', 'students']
"""
#-------------------------------------------------
# 3333333333333333333333333333333333333333333333
# print_most_common()
#-------------------------------------------------
#-------------------------------------------------
"""
Define the print_most_common() function which is passed two
parameters, a dictionary containing words and their corresponding
frequencies, e.g.,

 {"and":15, "talon":7, "frog":1, "cat":15, "tests":1, "dog":2,
 "bat":14, "rat":15}

and, an integer, the length of the key words to be considered.
The function first prints the length of the key words to be
considered, followed
by " letter words: ", then prints a sorted list of all the key
words of the required length from the dictionary which have the
highest frequency
followed by the frequency.

For example, the following code:

word_frequencies =  {"fish":9,  "parrot":8,  "frog":9,  "cat":9,
"stork":1,  "dog":4, "bat":9,  "rat":3}
print_most_common(word_frequencies, 3)
print_most_common(word_frequencies, 4)
print_most_common(word_frequencies, 5)
print_most_common(word_frequencies, 6)
print_most_common(word_frequencies, 7)

prints the following four lines of output:
```

```
3 letter words: ['bat', 'cat'] 9
4 letter words: ['fish', 'frog'] 9
5 letter words: ['stork'] 1
6 letter words: ['parrot'] 8
7 letter words: [] 0
"""
#----------------------------------------------------
# 4444444444444444444444444444444444444444444444
# get_names_marks_tuple_dict()
#----------------------------------------------------
#----------------------------------------------------
"""
Define the get_names_marks_tuple_dict() function which is passed
a filename as a parameter. The file contains lines of text
where each line is made up of a student name followed by a series
of numbers representing their marks.
An example input file is shown below ("ShortNamesAndNums.txt"):

The get_names_marks_tuple_dict() function returns a dictionary
with the student names as the keys and the corresponding
values which are tuples of their five best marks.  The marks in
the tuple must be sorted in ascending order. For example,
the line of text "Lara 65 54 79 83 25 58 76" becomes the
dictionary entry with the keyword "Lara" and the corresponding
value is a tuple made up of the 5 best marks in ascending order
"Lara": (58, 65, 76, 79, 83). You can assume that there
are always enough marks in each line of the input file.

Note: the testing code makes use of the
print_dict_in_key_order(a_dict) which prints the dictionary pairs
in sorted key order.

For example, the following code:

names_and_marks_dict =
get_names_marks_tuple_dict("ShortNamesAndNums.txt")
print_dict_in_key_order(names_and_marks_dict)

prints:

Elaine : (47, 49, 52, 53, 61)
Lara : (58, 65, 76, 79, 83)
Ryu : (80, 81, 82, 85, 91)
Tom : (29, 44, 45, 49, 54)
Wayne : (71, 82, 83, 96, 97)
"""
#----------------------------------------------------
# 5555555555555555555555555555555555555555555555
```

```
# remove_long_synonyms()
#--------------------------------------------------
"""
Define the remove_long_synonyms() function which is passed a
dictionary as a parameter.  The keys of the parameter dictionary
are words and the corresponding values are lists of synonyms
(synonyms are words which have the same or nearly the same
meaning).
The function removes all the synonyms which have 7 or more
characters from each corresponding list of synonyms.  As well,
the function sorts each corresponding list of synonyms.

For example, the following code:

synonyms_dict = {'look': ['gaze', 'see', 'glance', 'watch',
'peruse'],

   'put': ['place', 'set', 'attach', 'keep', 'save', 'set aside',
   'effect', 'achieve', 'do', 'build'],

   'beautiful': ['pretty', 'lovely', 'handsome', 'dazzling',
   'splendid', 'magnificent'],

   'slow': ['unhurried', 'gradual', 'leisurely', 'late',
   'behind', 'tedious', 'slack'],

   'dangerous': ['perilous', 'hazardous', 'uncertain']

   }

remove_long_synonyms(synonyms_dict)

print("1.")

print_dict_in_key_order(synonyms_dict)


synonyms_dict = {'come': ['approach', 'advance', 'near',
'arrive', 'reach'],

   'show': ['display', 'exhibit', 'present', 'point to',
   'indicate', 'explain', 'prove', 'demonstrate', 'expose'],

   'good': ['excellent', 'fine', 'superior', 'wonderful',
   'grand', 'superb', 'edifying'],

   'bad': ['evil', 'immoral', 'wicked', 'contaminated',
   'spoiled', 'defective',  'substandard', 'faulty', 'improper',
   'inappropriate']
```

```
   }

remove_long_synonyms(synonyms_dict)

print("2.")

print_dict_in_key_order(synonyms_dict)

prints:

1.
beautiful : ['lovely', 'pretty']
dangerous : []
look : ['gaze', 'glance', 'peruse', 'see', 'watch']
put : ['attach', 'build', 'do', 'effect', 'keep', 'place',
'save', 'set']
slow : ['behind', 'late', 'slack']
2.
bad : ['evil', 'faulty', 'wicked']
come : ['arrive', 'near', 'reach']
good : ['fine', 'grand', 'superb']
show : ['expose', 'prove']
"""
#-------------------------------------------------
# 6666666666666666666666666666666666666666666666
# contains_keys_and_values()
#-------------------------------------------------
"""
Define the contains_keys_and_values() function which is passed
two dict objects as parameters, dict1 and dict2.
The two parameter dictionaries both have corresponding values
which are lists of elements (numbers or strings).
The function return True if the following two conditions are met:

dict1 contains all the keys which are in dict2 (dict1 may contain
extra keys),

and,

the elements in all the corresponding value lists of dict2 are
also elements in one or more of the corresponding
value lists of dict1.  Note: when testing this part of the
condition, the elements can be in any order and in any
of the corresponding value lists, e.g., if one of the
corresponding values lists of dict2 is [4, 2] and any one
of the corresponding value lists of dict1 contains the element 4
and any one of the corresponding value lists of
dict1 contains the element 2, this part of the condition is
```

```
satisfied.
The function returns False in all other cases.

For example, the following code:

dict1 = {}
dict2 = {}
print("1.", contains_keys_and_values(dict1, dict2))

dict1 = {"a": [4, 3] , "d": [6, 2], "z": [], "t": [2, 23]}
dict2 = {"z": [2, 3, 6, 23], "a": [4]}
print("2.", contains_keys_and_values(dict1, dict2))

dict1 = {"a": [6, 3], "p": []}
dict2 = {"a": [3, 6, 3], "p": [6, 1]}
print("3.", contains_keys_and_values(dict1, dict2))

dict1 = {"a": [6, 3], "p": []}
dict2 = {"a": [3, 6, 3], "p": ['a']}
print("4.", contains_keys_and_values(dict1, dict2))

dict1 = {"a": [6, 3], "p": ['a'], "t": ['s']}
dict2 = {"a": [3, 6, 3], "p": ['a'], "s": ['a']}
print("5.", contains_keys_and_values(dict1, dict2))

prints:


1. True
2. True
3. False
4. False
5. False
"""
#-----------------------------------------------
# 7777777777777777777777777777777777777777777
# get_triples_dict()
#-----------------------------------------------
#-----------------------------------------------
"""
Define the get_triples_dict() function which is passed a string
of text as a parameter.  The function first
converts the parameter string to lower case and then returns a
dictionary with keys which are all the unique
consecutive three alphabetic characters from the text, and the
corresponding values are the number of times
the three consecutive alphabetic characters appear in the text.
Use the isalpha() method to check if a
character is alphabetic or not.  The dictionary should only
```

contain entries which occur more than once.
After your dictionary has been created and populated, you need to remove any key-value pairs which have a
corresponding value of 1. For example, if the text is "Super, duper" the algorithm proceeds as follows:

```
Character 's':  String is "s",  Dictionary is {}
Character 'u':  String is "su", Dictionary is {}
Character 'p':  String is "sup", change string to "up",
                         Dictionary is {'sup': 1}
Character 'e':  String is "upe", change string to "pe",
                         Dictionary is {'sup': 1, 'upe': 1}
Character 'r':  String is "per", change string to "er",
                         Dictionary is {'sup': 1, 'upe': 1, 'per':
                         1}
Character ',':  String is "er",  Dictionary is {'sup': 1, 'upe':
1, 'per': 1}
Character ' ':  String is "er",  Dictionary is {'sup': 1, 'upe':
1, 'per': 1}
Character 'd':  String is "erd", change string to "rd",
Dictionary is
                           {'sup': 1, 'upe': 1, 'per': 1, 'erd': 1}
Character 'u':  String is "rdu", change string to "du",
Dictionary is
                           {'sup': 1, 'upe': 1, 'per': 1, 'erd':
                           1, 'rdu': 1}
Character 'p':  String is "dup", change string to "up",
Dictionary is
               {'sup': 1, 'upe': 1, 'per': 1, 'erd': 1, 'rdu': 1,
               'dup': 1}
Character 'e':  String is "upe", change string to "pe",
Dictionary is
               {'sup': 1, 'upe': 2, 'per': 1, 'erd': 1, 'rdu': 1,
               'dup': 1}
Character 'r':  String is "per", change string to "er",
Dictionary is
               {'sup': 1, 'upe': 2, 'per': 2, 'erd': 1, 'rdu': 1,
               'dup': 1}
Remove all entries with a value of 1: Dictionary is {'upe': 2,
'per': 2}
```

For example, executing the following code::

```
print("1.")
print_dict_in_key_order(get_triples_dict('super, duper'))
print("\n2.")
print_dict_in_key_order(get_triples_dict("ABC ABC ABC"))
print("\n3.")
print_dict_in_key_order(get_triples_dict("Sometimes the smallest
things make more room in your heart"))
```

```
print("\n4.")
print_dict_in_key_order(get_triples_dict("My favourite painting
is the painting i did of my dog in that painting in my den"))
prints (output is shown here in four separate columsn):

1.          2.          3.          4.
per - 2     abc - 3     est - 2     ain - 3
upe - 2     bca - 2     sma - 2     epa - 2
            cab - 2                 gin - 2
                                    ing - 3
                                    int - 4
                                    myd - 2
                                    ngi - 3
                                    nti - 3
                                    pai - 3
                                    tin - 3
"""
```