```
import collections
import math

############################################################
# Problem 3a

def findAlphabeticallyLastWord(text:str) -> str:
    """
    Given a string |text|, return the word in |text| that comes last
    alphabetically (that is, the word that would appear last in a dictionary).
    A word is defined by a maximal sequence of characters without whitespaces.
    You might find max() and list comprehensions handy here.
    """
    # BEGIN_YOUR_CODE (our solution is 1 line of code, but don't worry if you
deviate from this)
    return max(sorted(text.split(' ')))
    # END_YOUR_CODE

############################################################
# Problem 3b

def euclideanDistance(loc1, loc2) -> int:
    """
    Return the Euclidean distance between two locations, where the locations
    are pairs of numbers (e.g., (3, 5)).
    """
    # BEGIN_YOUR_CODE (our solution is 1 line of code, but don't worry if you
deviate from this)
    return math.sqrt((loc1[0] - loc2[0])** 2 + (loc1[1] - loc2[1])** 2)
    # END_YOUR_CODE

############################################################
# Problem 3c

def mutateSentences(sentence:str):
    """
    Given a sentence (sequence of words), return a list of all "similar"
    sentences.
    We define a sentence to be similar to the original sentence if
      - it as the same number of words, and
      - each pair of adjacent words in the new sentence also occurs in the
original sentence
        (the words within each pair should appear in the same order in the
output sentence
         as they did in the orignal sentence.)
    Notes:
      - The order of the sentences you output doesn't matter.
      - You must not output duplicates.
      - Your generated sentence can use a word in the original sentence more
than
        once.
    Example:
      - Input: 'the cat and the mouse'
      - Output: ['and the cat and the', 'the cat and the mouse', 'the cat and
the cat', 'cat and the cat and']
                (reordered versions of this list are allowed)
    """
    # BEGIN_YOUR_CODE (our solution is 17 lines of code, but don't worry if you
deviate from this)
    words = sentence.split(' ')
    length = len(words)
    adjacentWordDict = {}

    def recurse(adjacentWordDict, sentenceLength, similarSentences, temp):
```

```
        if len(temp) == sentenceLength:
            similarSentences.add(' '.join(temp))
            return
        last = temp[-1]
        if last in adjacentWordDict:
            for value in adjacentWordDict[last]:
                temp.append(value)
                recurse(adjacentWordDict, sentenceLength, similarSentences,
temp)
                temp.pop()
        return

    for i in range(length - 1):
        if words[i] not in adjacentWordDict:
            adjacentWordDict[words[i]] = set()
        adjacentWordDict[words[i]].add(words[i + 1])

    similarSentences = set()
    for _ in adjacentWordDict:
        recurse(adjacentWordDict, length, similarSentences, [_])
    return similarSentences
    # END_YOUR_CODE

############################################################
# Problem 3d

def sparseVectorDotProduct(v1, v2) -> float:
    """
    Given two sparse vectors |v1| and |v2|, each represented as
collections.defaultdict(float), return
    their dot product.
    You might find it useful to use sum() and a list comprehension.
    This function will be useful later for linear classifiers.
    Note: A sparse vector has most of its entries as 0
    The keys of the sparse vector can be any type. For example, an input could
be {'a': 1.0, 'b': 2.0}
    """
    # BEGIN_YOUR_CODE (our solution is 4 lines of code, but don't worry if you
deviate from this)
    result = 0
    for element in v2:
        if element in v1:
         result = result + v1[element] * v2[element]
    return result
    # END_YOUR_CODE

############################################################
# Problem 3e

def incrementSparseVector(v1, scale, v2):
    """
    Given two sparse vectors |v1| and |v2|, perform v1 += scale * v2.
    Do not modify v2 in your implementation.
    This function will be useful later for linear classifiers.
    """
    # BEGIN_YOUR_CODE (our solution is 2 lines of code, but don't worry if you
deviate from this)
    for element in v2:
        v1[element] += scale * v2[element]
    return v1
    # END_YOUR_CODE

############################################################
# Problem 3f
```

```
def findSingletonWords(text:str):
    """
    Splits the string |text| by whitespace and returns the set of words that
    occur exactly once.
    You might find it useful to use collections.defaultdict(int).
    """
    # BEGIN_YOUR_CODE (our solution is 4 lines of code, but don't worry if you
deviate from this)
    c = collections.Counter(text.split(' '))
    return set(k for k,v in c.items() if v==1)
    # END_YOUR_CODE

############################################################
# Problem 3g

def computeLongestPalindromeLength(text:str) -> str:
    """
    A palindrome is a string that is equal to its reverse (e.g., 'ana').
    Compute the length of the longest palindrome that can be obtained by
deleting
    letters from |text|.
    For example: the longest palindrome in 'animal' is 'ama' and it's length is
3.
    Your algorithm should run in O(len(text)^2) time.
    You should first define a recurrence before you start coding.
    """
    # BEGIN_YOUR_CODE (our solution is 14 lines of code, but don't worry if you
deviate from this)
    cache = {}

    def findRecursively(text, cache):
        if (len(text) <= 1):    # Base case
            return len(text)
        if (text in cache):     # Base case
            return cache[text]

        if (text[0] == text[-1]):
            temp = 2 + findRecursively(text[1:-1], cache)
            cache[text] = temp
            return temp
        else:
            len1 = findRecursively(text[1: ], cache)
            len2 = findRecursively(text[ :-1], cache)
            temp = max(len1, len2)
            cache[text] = temp
            return temp

    return findRecursively(text, cache)
    # END_YOUR_CODE
```