```
package forcomp
/** !!!!  DISCLAIMER !!!!
 *        The skeleton code for this was provided as an assignment for the
 *        coursera course - Functional Programming in Scala by Martin Odersky
 *        I am providing you with this code snippet to show you my understanding
 *        of functional programming principles
 *
 * !!!!   DISCLAIMER !!!!
 */


import common._

object Anagrams {

  /** A word is simply a `String`. */
  type Word = String

  /** A sentence is a `List` of words. */
  type Sentence = List[Word]

  /** `Occurrences` is a `List` of pairs of characters and positive integers
saying
   *  how often the character appears.
   *  This list is sorted alphabetically w.r.t. to the character in each pair.
   *  All characters in the occurrence list are lowercase.
   *
   *  Any list of pairs of lowercase characters and their frequency which is not
sorted
   *  is **not** an occurrence list.
   *
   *  Note: If the frequency of some character is zero, then that character
should not be
   *  in the list.
   */
  type Occurrences = List[(Char, Int)]

  /** The dictionary is simply a sequence of words.
   *  It is predefined and obtained as a sequence using the utility method
`loadDictionary`.
   */
  val dictionary: List[Word] = loadDictionary

  /** Converts the word into its character occurence list.
   *
   *  Note: the uppercase and lowercase version of the character are treated as
the
   *  same character, and are represented as a lowercase character in the
occurrence list.
   */
  def wordOccurrences(w: Word): Occurrences = w
                                                .toList
                                                .groupBy( x => x.toLower)
                                                .map(x => (x._1, x._2.length))
                                                .toList.sortBy(x => x._1)

  /** Converts a sentence into its character occurrence list. */
  def sentenceOccurrences(s: Sentence): Occurrences = wordOccurrences(s.mkString)

  /** The `dictionaryByOccurrences` is a `Map` from different occurrences to a
sequence of all
```

```
   *  the words that have that occurrence count.
   *  This map serves as an easy way to obtain all the anagrams of a word given
its occurrence list.
   *
   *  For example, the word "eat" has the following character occurrence list:
   *
   *     `List(('a', 1), ('e', 1), ('t', 1))`
   *
   *  Incidentally, so do the words "ate" and "tea".
   *
   *  This means that the `dictionaryByOccurrences` map will contain an entry:
   *
   *    List(('a', 1), ('e', 1), ('t', 1)) -> Seq("ate", "eat", "tea")
   *
   */
  lazy val dictionaryByOccurrences:
          Map[Occurrences, List[Word]] = dictionary.groupBy(x =>
wordOccurrences(x))

  /** Returns all the anagrams of a given word. */
  def wordAnagrams(word: Word): List[Word] =
dictionaryByOccurrences(wordOccurrences(word))

  /** Returns the list of all subsets of the occurrence list.
   *  This includes the occurrence itself, i.e. `List(('k', 1), ('o', 1))`
   *  is a subset of `List(('k', 1), ('o', 1))`.
   *  It also include the empty subset `List()`.
   *
   *  Example: the subsets of the occurrence list `List(('a', 2), ('b', 2))` are:
   *
   *    List(
   *      List(),
   *      List(('a', 1)),
   *      List(('a', 2)),
   *      List(('b', 1)),
   *      List(('a', 1), ('b', 1)),
   *      List(('a', 2), ('b', 1)),
   *      List(('b', 2)),
   *      List(('a', 1), ('b', 2)),
   *      List(('a', 2), ('b', 2))
   *    )
   *
   *  Note that the order of the occurrence list subsets does not matter -- the
subsets
   *  in the example above could have been displayed in some other order.
   */
  def combinations(occurrences: Occurrences): List[Occurrences] = {
    def _combin(occurrences: Occurrences): Set[Occurrences] = {
    if (occurrences.isEmpty) Set(List())
    else
     for {subs <- _combin(occurrences.tail)
          i <- List.range(0, occurrences.head._2+1)
       } yield {
         if (i > 0)
           (occurrences.head._1, i) :: subs
         else
           subs
       }
    }
    _combin(occurrences).toList
  }
  /** Subtracts occurrence list `y` from occurrence list `x`.
   *
   *  The precondition is that the occurrence list `y` is a subset of
   *  the occurrence list `x` -- any character appearing in `y` must
   *  appear in `x`, and its frequency in `y` must be smaller or equal
   *  than its frequency in `x`.
   *
   *  Note: the resulting value is an occurrence - meaning it is sorted
   *  and has no zero-entries.
   */
  def subtract(x: Occurrences, y: Occurrences): Occurrences = {
    def _sub(x: Occurrences, y: Occurrences): Occurrences = {
    val d = y.toMap
    for (kv <- x) yield {
           if (!d.contains(kv._1)) kv
           else (kv._1, kv._2 - d(kv._1))
         }
    }
    _sub(x, y).filter(x => x._2 > 0)
  }


}
```