```java
import java.util.*;

/**
 * This class Search for Anagrams of a Word given a Dictionary.
 *
 *  It first store a dictionary for word reference.
 *  Then pre-process data from the dictionary, forming an
 *  Anagram Dictionary for faster anagram look-up.
 *
 *  When Given a word, the print function looks for
 *  all anagrams from the Dictionary
 *  and print anagrams from the Anagram Dictionary
 * *
 * <ul>
 * <li> Name: AnagramSolver.java
 * <li> Description: Anagram Solver
 * <li> Class: Java 145
 * <li> Instructor: Ken Hang && Janet Ash
 * <li> Date: March 10 2015
 * </ul>
 * @author  Hai H Nguyen (Bill)
 * @version Winter 2015
 */
public class AnagramSolver {
    private List<String> dictionary;

    private Map<String, List<String>> anagramDictionary =
            new HashMap<String, List<String>>();

    /**
     * Constructor, given a list, does the following:
     * <ul>
     * <li> Initialize the dictionary </li>
     * <li> Extract an Anagram Dictionary</li>
     * <li> Break if the Dictionary is Empty</li>
     * </ul>
     * @param dictionary                    List of words
     * @throws IllegalArgumentException     If Dictionary is Empty
     */
    public AnagramSolver(List<String> dictionary) {
        if (dictionary.isEmpty()) {
            throw new IllegalArgumentException("Dictionary is Empty!");
        } else {
            this.dictionary = dictionary;

            prepareAnagramDictionary();
        }
    }

   /**
    * Store all combinations of words from the dictionary
    * into an Anagram dictionary
    */
    private void prepareAnagramDictionary(){
        for (String word : dictionary) {
            // Extract a Letter Inventory for each word
            LetterInventory key = new LetterInventory(word);
         // Store them into the Anagram Dictionary map
            if (anagramDictionary.containsKey(key.toString())){
                anagramDictionary.get(key.toString()).add(word);
            } else  {
                List<String> values = new ArrayList<String>();
            // Register a new value set
                values.add(word);
            // Assign the key with the new value set
                anagramDictionary.put(key.toString(),values);
            }
        }
    }

   /**
    * Extract a Letter Inventory list from the given letter inventory
    * Each element in the list serves as key for Anagram Dictionary map
    */
    private List<LetterInventory> allAnagramsOf(LetterInventory sLi){
        List<LetterInventory> anagramList = new ArrayList<LetterInventory>();

        for (String word : dictionary) {
           // Extract a Letter Inventory for each word
            LetterInventory wordLi = new LetterInventory(word);
            // Extract a Letter inventory of leftover letters
            LetterInventory leftOverLi = sLi.subtract(wordLi);
            // If leftover words is not negative, store the word
            if (leftOverLi != null){
                anagramList.add(wordLi);
            }
        }

        return anagramList;
    }

   /**
    * Recursively print all of the anagrams that
    * forms the first passed letter inventory.
    * @param out           Stack of Chosen Strings
    * @param root          Letters to use
    * @param choices       Choices available
    * @param max           Maximum size of Chosen String Stack
    */
    private void printAnagrams(Stack<String> out, LetterInventory root,
                               List<LetterInventory> choices,
                               int max){
       // The recursion continues if there are letters to use AND
       // max is 0 OR size of Stack is less than or equal to max
        if (root!=null && (out.size()<=max||max==0)){
         /*
         // Useful debug lines, use for Small test only!
           debugLog("Letters to use: " + root);
            debugLog("Choices: " + choices);
           debugLog("Chosen: " + out);
         */
         // If Letters to use is Empty AND max is 0 OR size of Stack equals max:
           if (root.isEmpty() && (out.size()==max||max==0)){
              debugLog(out);
           } else {
              for (LetterInventory choice : choices) {
               // For each choice, get a set of leftover letters
                 LetterInventory leftOverLi = root.subtract(choice);
               // Get the list of word mapped to each choice
                 List<String> words = anagramDictionary.get(choice.toString());
               // For each word in the word list:
                 for (String word : words) {
                    out.push(word);
                  // Recursive with the new Stack:
                    printAnagrams(out, leftOverLi, choices, max);
                  // Pop the word, Back track to Previous Stack
                    out.pop();
                 }
              }
            }
        }
    }

    /**
     * Print all anagrams series of a given words. Each series are
     * restricted to a maximum word, and a given dictionary.
```

```
     * @param s                             String to search for anagram
     * @param max                           Maximum words for each anagram series
     * @throws IllegalArgumentException     If max is smaller than 0
     */
    public void print(String s, int max) {
        if (max < 0){
            throw new IllegalArgumentException("Max < 0");
        } else {
            LetterInventory lettersToUse = new LetterInventory(s);
         // Extract a list of keys from the word
            List<LetterInventory> choices = allAnagramsOf(lettersToUse);

            //debugLog("Choices are: " + choices);

            Stack<String> answerStack = new Stack<String>();
         // Begin the Back track Recursion Loop:
            printAnagrams(answerStack, lettersToUse, choices, max);
        }
    }

    private void debugLog(Object o){
        if(o!= null) {
            System.out.println(o.toString());
        }
    }
} //IS29
```