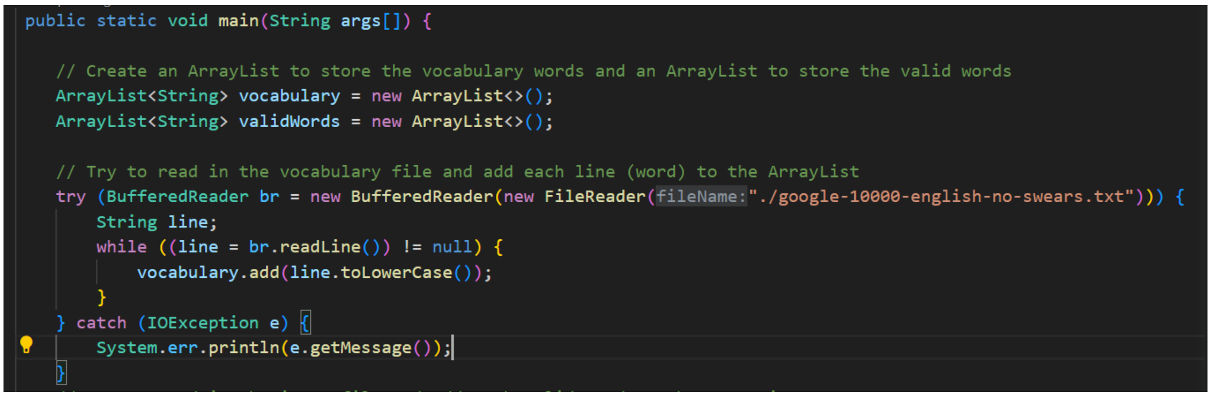
CMT219 Report

Question 1

1.
The program creates two ArrayLists (Figure 1). One is used to store the words from the vocabulary file, and the other to store the valid words from the 'input219.txt' file. Valid words are those that exist in both the 'input219.txt' and the vocabulary files.



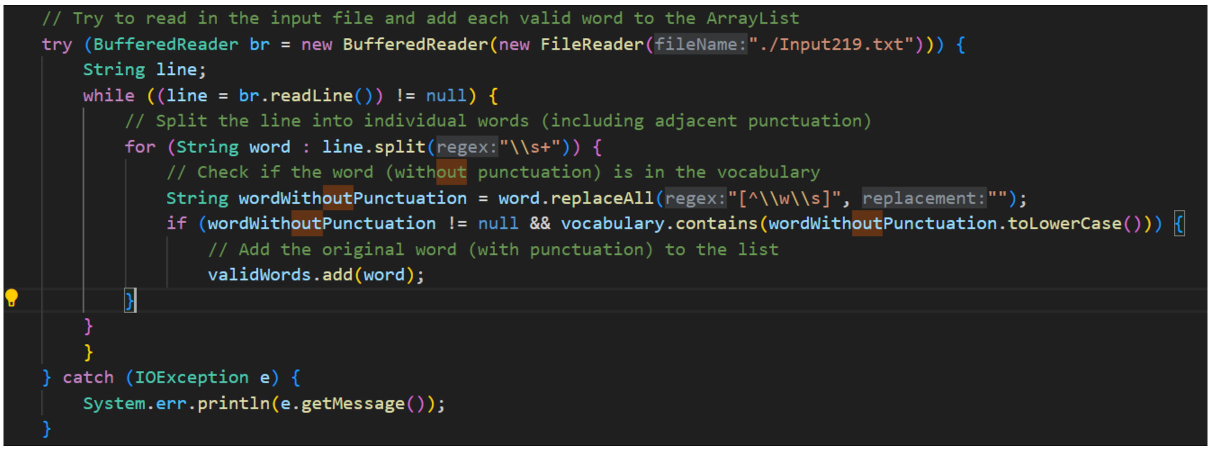
```java
public static void main(String args[]) {

    // Create an ArrayList to store the vocabulary words and an ArrayList to store the valid words
    ArrayList<String> vocabulary = new ArrayList<>();
    ArrayList<String> validWords = new ArrayList<>();

    // Try to read in the vocabulary file and add each line (word) to the ArrayList
    try (BufferedReader br = new BufferedReader(new FileReader(fileName:"./google-10000-english-no-swears.txt"))) {
        String line;
        while ((line = br.readLine()) != null) {
            vocabulary.add(line.toLowerCase());
        }
    } catch (IOException e) {
        System.err.println(e.getMessage());
    }
```

Figure 1. ArrayList data structurse used to set up vocabulary and valid words. Vocabulary file then read in and added to ArrayList.

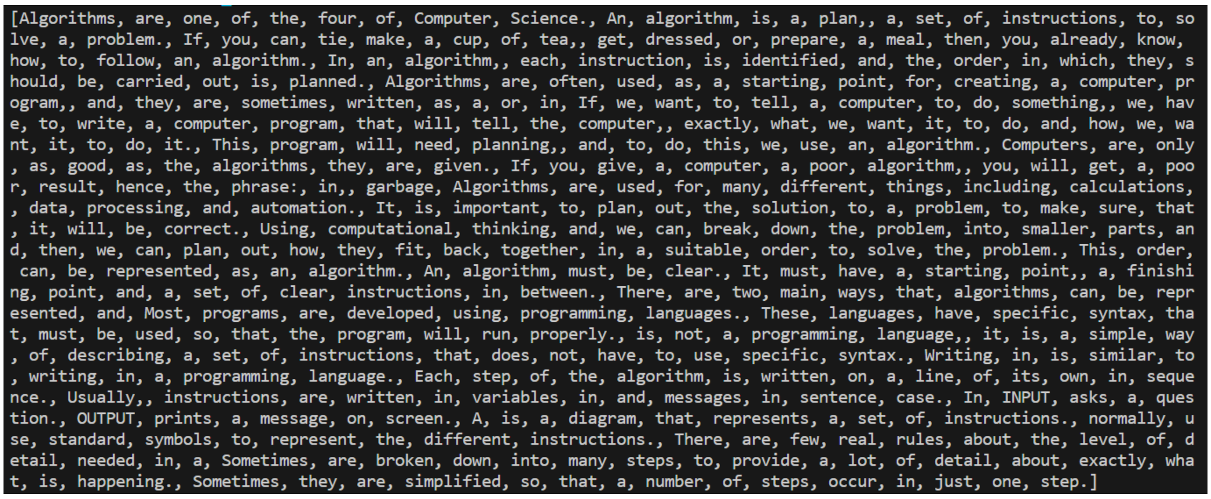
The program reads the vocabulary file (Figure 1), adds each word to the 'vocabulary' ArrayList in lower case, and then reads in the 'Input219.txt' file. Using a regular expression, it splits each line from the input file into words based on whitespace. The program removes any punctuation that's part of the word adjacent to a whitespace (Figure 2).



```java
    // Try to read in the input file and add each valid word to the ArrayList
    try (BufferedReader br = new BufferedReader(new FileReader(fileName:"./Input219.txt"))) {
        String line;
        while ((line = br.readLine()) != null) {
            // Split the line into individual words (including adjacent punctuation)
            for (String word : line.split(regex:"\\s+")) {
                // Check if the word (without punctuation) is in the vocabulary
                String wordWithoutPunctuation = word.replaceAll(regex:"[^\\w\\s]", replacement:"");
                if (wordWithoutPunctuation != null && vocabulary.contains(wordWithoutPunctuation.toLowerCase())) {
                    // Add the original word (with punctuation) to the list
                    validWords.add(word);
                }
            }
        }
    } catch (IOException e) {
        System.err.println(e.getMessage());
    }
```

Figure 2. Extracting and adding valid words (with punctuation) to an ArrayList using a vocabulary Arraylist, while ignoring invalid words and adjacent punctuation

The program filters out any words in "input219.txt" that do not exist in the vocabulary by converting them to lower case and checking them against the lower case words in the vocabulary ArrayList using the contains() method. If the resulting word, without punctuation, is not null and is contained in the vocabulary ArrayList, then the original word, with punctuation, is added to the validWords ArrayList using the add() method. Finally, the program prints the valid words from the ArrayList to the terminal (Figure 3).



```
[Algorithms, are, one, of, the, four, of, Computer, Science., An, algorithm, is, a, plan,, a, set, of, instructions, to, so
lve, a, problem., If, you, can, tie, make, a, cup, of, tea,, get, dressed, or, prepare, a, meal, then, you, already, know,
how, to, follow, an, algorithm., In, an, algorithm,, each, instruction, is, identified, and, the, order, in, which, they, s
hould, be, carried, out, is, planned., Algorithms, are, often, used, as, a, starting, point, for, creating, a, computer, pr
ogram,, and, they, are, sometimes, written, as, a, or, in, If, we, want, to, tell, a, computer, to, do, something,, we, hav
e, to, write, a, computer, program, that, will, tell, the, computer,, exactly, what, we, want, it, to, do, and, how, we, wa
nt, it, to, do, it., This, program, will, need, planning,, and, to, do, this, we, use, an, algorithm., Computers, are, only
, as, good, as, the, algorithms, they, are, given., If, you, give, a, computer, a, poor, algorithm,, you, will, get, a, poo
r, result, hence, the, phrase:, in,, garbage, Algorithms, are, used, for, many, different, things, including, calculations,
, data, processing, and, automation., It, is, important, to, plan, out, the, solution, to, a, problem, to, make, sure, that
, it, will, be, correct., Using, computational, thinking, and, we, can, break, down, the, problem, into, smaller, parts, an
d, then, we, can, plan, out, how, they, fit, back, together, in, a, suitable, order, to, solve, the, problem., This, order,
 can, be, represented, as, an, algorithm., An, algorithm, must, be, clear., It, must, have, a, starting, point,, a, finishi
ng, point, and, a, set, of, clear, instructions, in, between., There, are, two, main, ways, that, algorithms, can, be, repr
esented, and, Most, programs, are, developed, using, programming, languages., These, languages, have, specific, syntax, tha
t, must, be, used, so, that, the, program, will, run, properly., is, not, a, programming, language,, it, is, a, simple, way
, of, describing, a, set, of, instructions, that, does, not, have, to, use, specific, syntax., Writing, in, is, similar, to
, writing, in, a, programming, language., Each, step, of, the, algorithm, is, written, on, a, line, of, its, own, in, seque
nce., Usually,, instructions, are, written, in, variables, in, and, messages, in, sentence, case., In, INPUT, asks, a, ques
tion., OUTPUT, prints, a, message, on, screen., A, is, a, diagram, that, represents, a, set, of, instructions., normally, u
se, standard, symbols, to, represent, the, different, instructions., There, are, few, real, rules, about, the, level, of, d
etail, needed, in, a, Sometimes, are, broken, down, into, many, steps, to, provide, a, lot, of, detail, about, exactly, wha
t, is, happening., Sometimes, they, are, simplified, so, that, a, number, of, steps, occur, in, just, one, step.]
```

Figure 3. Unsorted validWords ArrayList printed to terminal.

2.

The program's time complexity can be assessed by analyzing the three main steps involved. Firstly, when reading in the vocabulary file, each word is added to an ArrayList, with a time complexity proportional to O(n), where n is the number of words in the file. Secondly, when reading in the input file, each word is checked against the vocabulary and added to an ArrayList if it is valid. The time complexity of checking whether a word is in an ArrayList is O(n), where n is the number of items in the ArrayList. Since this process involves a nested loop, iterating through each word in the input sentence and each word in the vocabulary, the time complexity of this step is O(n^2). Finally, the valid words are printed, with a time complexity of O(n) to print n items. Therefore, the overall time complexity of the program is dominated by the second step, which has a time complexity of O(n^2).

3.

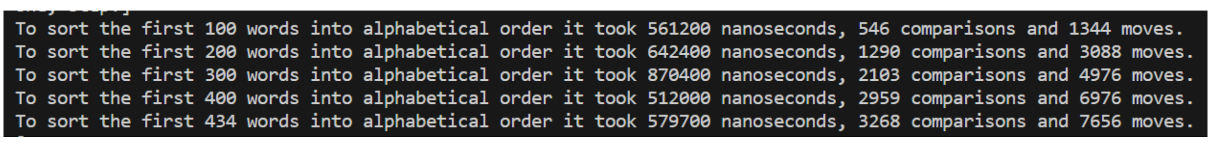
The program's efficiency could be improved by using a more efficient data structure than the ArrayList for storing the words and checking whether they exist in the vocabulary. Using either a HashSet or a LinkedList would improve the program's efficiency.

A HashSet would achieve faster lookups, as its time complexity for checking whether an item exists is O(1) on average. A linked list could also be used, as it dynamically allocates memory as elements are added to it (by updating pointers). Therefore, adding elements to a linked list would have a constant time complexity of O(1), regardless of the number of elements in the list.

Overall, the HashSet would be most appropriate for this program, as it only needs to check if a word is valid or not and does not need to maintain the order of valid words.

Question 2

1.



```
To sort the first 100 words into alphabetical order it took 561200 nanoseconds, 546 comparisons and 1344 moves.
To sort the first 200 words into alphabetical order it took 642400 nanoseconds, 1290 comparisons and 3088 moves.
To sort the first 300 words into alphabetical order it took 870400 nanoseconds, 2103 comparisons and 4976 moves.
To sort the first 400 words into alphabetical order it took 512000 nanoseconds, 2959 comparisons and 6976 moves.
To sort the first 434 words into alphabetical order it took 579700 nanoseconds, 3268 comparisons and 7656 moves.
```

Figure 4. Calculated time, comparisons and moves program required for sorting at defined intervals.

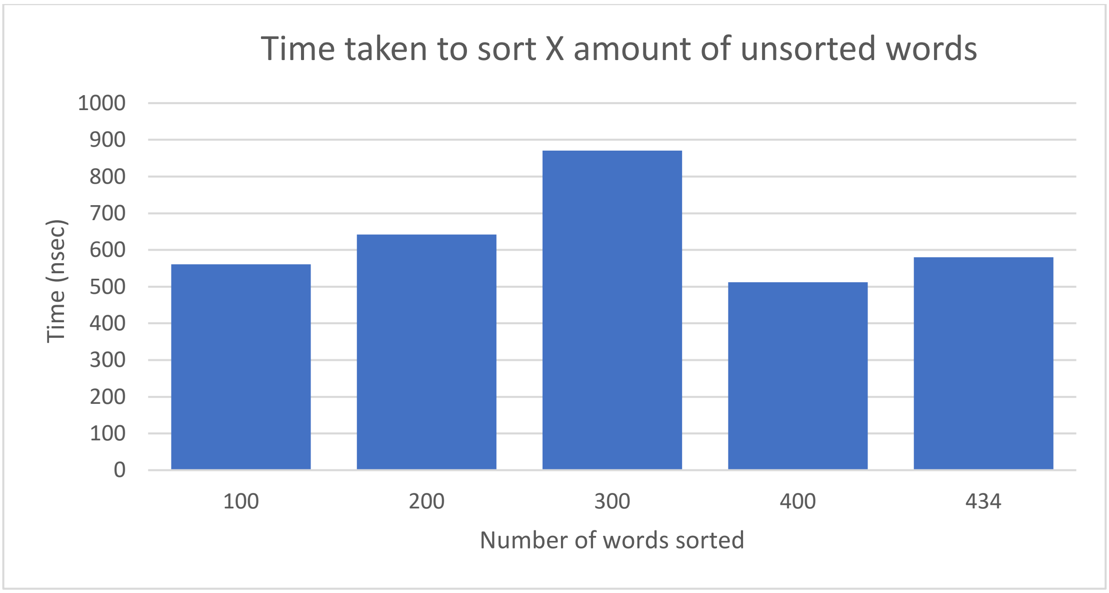


Figure 5. Graph showing time taken to sort amounts of validWords.

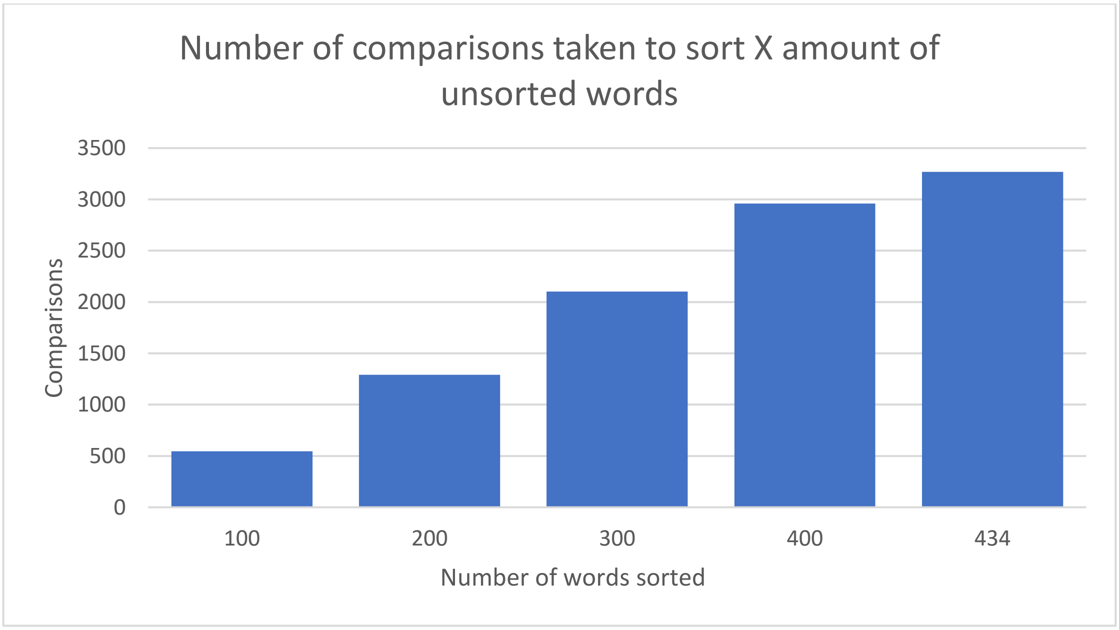


Figure 6. Graph showing number of comparisons taken to sort amounts of validWords.

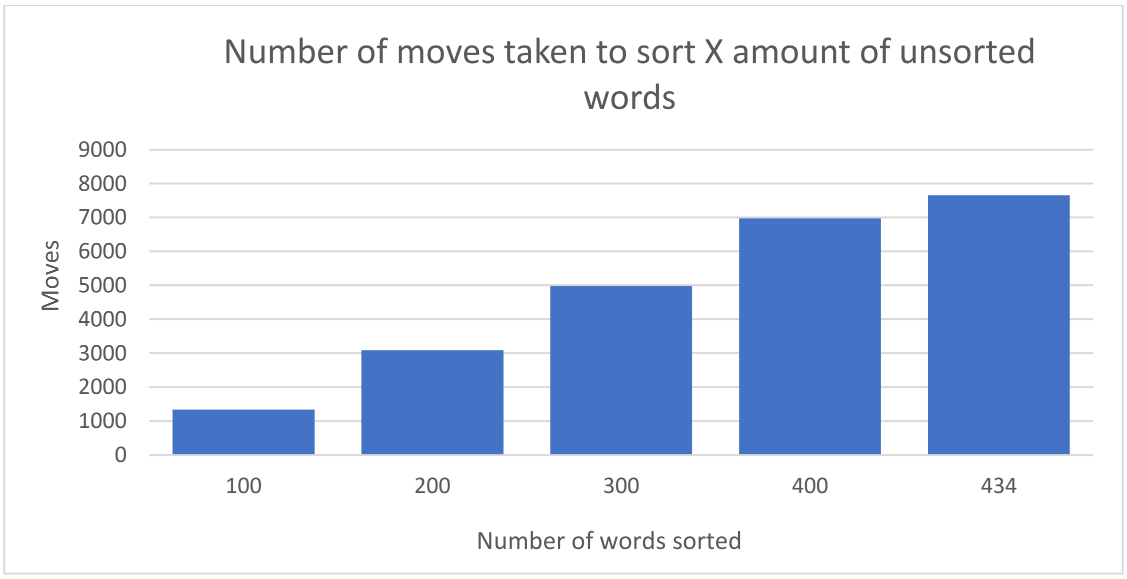
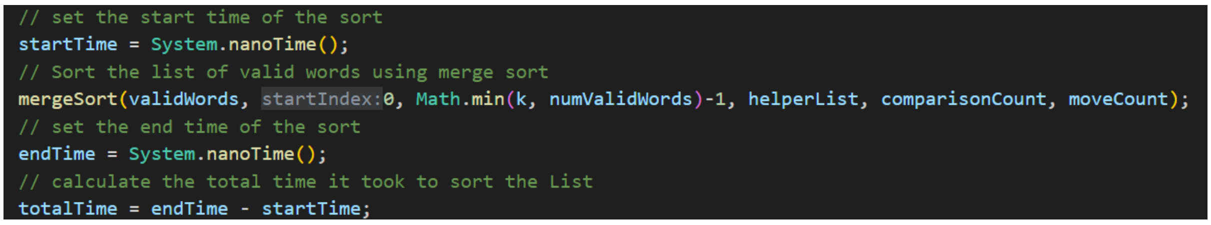


Figure 7. Graph showing number of moves taken to sort amounts of validWords.

2.

The program initializes variables to track the performance of the sorting algorithm and creates a copy of the validWords list. The algorithm then enters a loop that sorts subsets of the list using the merge sort algorithm. The k variable tracks the number of words that have been sorted, starting at the first 100 and incrementing until the entire list has been sorted. The copy of the validWords ArrayList is used to reset it to unordered before sorting each subset of unsorted words. The mergeSort method is then called on the current subset, with the comparisonCount and moveCount arrays passed in to track the number of comparisons and moves made during the sort (Figure 8). The mergeSort() method implements the merge sort algorithm to sort the words. The method divides the ArrayList of words into two halves by recursively calling the mergeSort() until each half has only one element left. It then calls the merge() method to merge the two halves into a single, sorted ArrayList. The merge() method compares the first element of each half and selects the smaller one to be added to the new, sorted ArrayList. It continues comparing and adding elements until all elements are in the new ArrayList, incrementing the moves and comparison counters when an operation takes place.

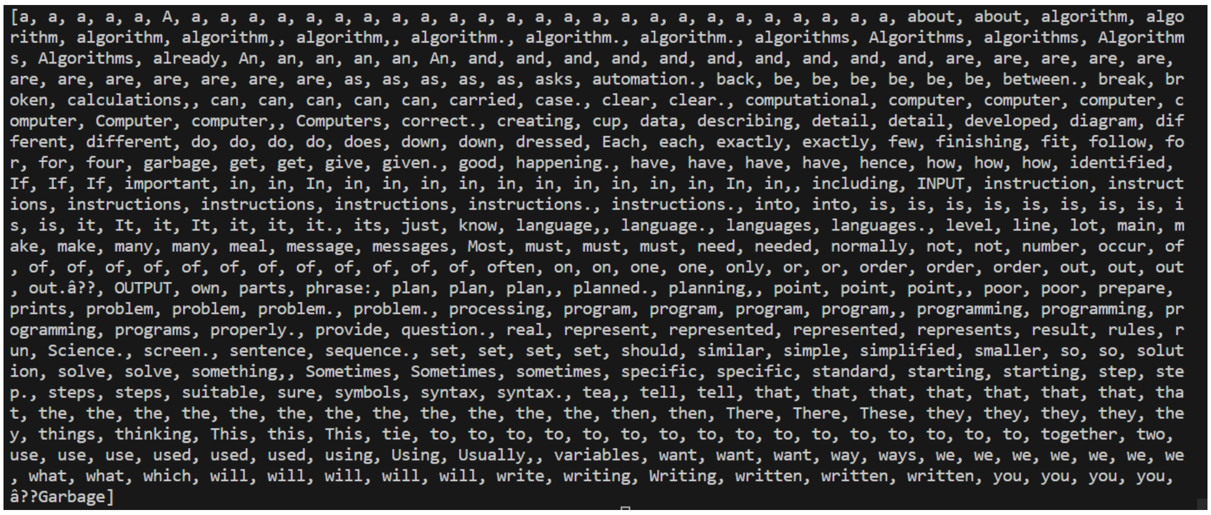
For the merge sort algorithm, the number of "moves" has been interpreted as the number of times elements are copied from one list to another during the merging process, unlike "swaps" which indicate the number of times elements are exchanged within the same array.



```
// set the start time of the sort
startTime = System.nanoTime();
// Sort the list of valid words using merge sort
mergeSort(validWords, startIndex:0, Math.min(k, numValidWords)-1, helperList, comparisonCount, moveCount);
// set the end time of the sort
endTime = System.nanoTime();
// calculate the total time it took to sort the List
totalTime = endTime - startTime;
```

Figure 8. Implementation of sorting time calculation.

The time taken to sort each distinct number of words is recorded (Figure 8) by measuring the start time before the mergeSort() method is called and the end time once it has completed. The performance measurements are then printed to the terminal for each iteration of the loop (Figure 4). Finally, once the sorting loop has terminated, the program prints the alphabetically ordered words of the complete validWords ArrayList to the terminal.



```
[a, a, a, a, a, A, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, about, about, algorithm, algo
rithm, algorithm, algorithm,, algorithm,, algorithm., algorithm., algorithm., algorithms, Algorithms, algorithms, Algorithm
s, Algorithms, already, An, an, an, an, an, An, and, and, and, and, and, and, and, and, and, and, are, are, are, are, are,
are, are, are, are, are, are, are, as, as, as, as, as, asks, automation., back, be, be, be, be, be, be, between., break, br
oken, calculations,, can, can, can, can, can, carried, case., clear, clear., computational, computer, computer, computer, c
omputer, Computer, computer,, Computers, correct., creating, cup, data, describing, detail, detail, developed, diagram, dif
ferent, different, do, do, do, do, does, down, down, dressed, Each, each, exactly, exactly, few, finishing, fit, follow, fo
r, for, four, garbage, get, get, give, given., good, happening., have, have, have, have, hence, how, how, how, identified,
If, If, If, important, in, in, In, in, in, in, in, in, in, in, in, in, in, In, in,, including, INPUT, instruction, instruct
ions, instructions, instructions, instructions, instructions., instructions., into, into, is, is, is, is, is, is, is, is, i
s, is, it, It, it, It, it, it, it., its, just, know, language,, language., languages, languages., level, line, lot, main, m
ake, make, many, many, meal, message, messages, Most, must, must, must, need, needed, normally, not, not, number, occur, of
, of, of, of, of, of, of, of, of, of, of, of, of, often, on, on, one, one, only, or, or, order, order, order, out, out, out
, out.â??, OUTPUT, own, parts, phrase:, plan, plan, plan,, planned., planning,, point, point, point,, poor, poor, prepare,
prints, problem, problem, problem., problem., processing, program, program, program, program,, programming, programming, pr
ogramming, programs, properly., provide, question., real, represent, represented, represented, represents, result, rules, r
un, Science., screen., sentence, sequence., set, set, set, set, should, similar, simple, simplified, smaller, so, so, solut
ion, solve, solve, something,, Sometimes, Sometimes, sometimes, specific, specific, standard, starting, starting, step, ste
p., steps, steps, suitable, sure, symbols, syntax, syntax., tea,, tell, tell, that, that, that, that, that, that, that, tha
t, the, the, the, the, the, the, the, the, the, the, the, the, then, then, There, There, These, they, they, they, they, the
y, things, thinking, This, this, This, tie, to, to, to, to, to, to, to, to, to, to, to, to, to, to, to, to, together, two,
use, use, use, used, used, used, using, Using, Usually,, variables, want, want, want, way, ways, we, we, we, we, we, we, we
, what, what, which, will, will, will, will, will, write, writing, Writing, written, written, written, you, you, you, you,
â??Garbage]
```

Figure 9. Alphabetically sorted validWords ArrayList.

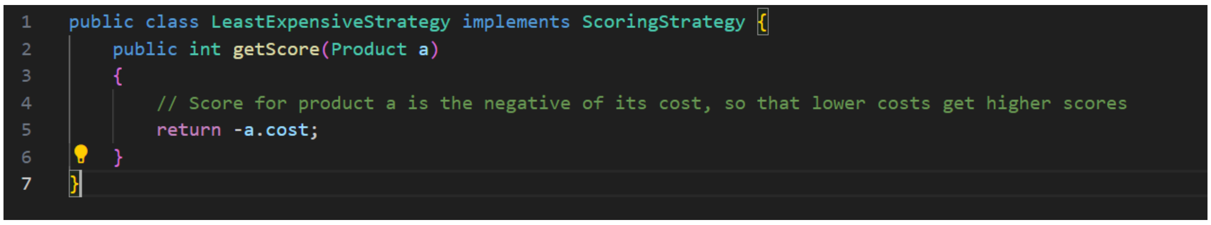
Question 3

A)

The purpose of the Strategy design pattern is to define a family of algorithms, encapsulate each one as an object, and make them interchangeable. This allows us to use all the algorithms in the same way, and the algorithm can vary independently of the 'users' that utilize it, allowing the algorithm to be selected at runtime.

In this program, the ScoringStrategy interface encapsulates the scoring algorithm. The MostPracticalStrategy and LeastExpensiveStrategy classes implement that interface to provide two different ways of scoring the products. The ProductRecommender class uses the getBestProduct() method to determine which product in the ArrayList has the best score according to the specified ScoringStrategy.

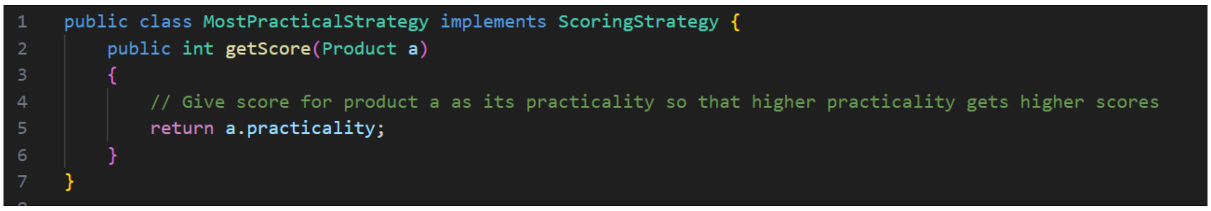
B)



```java
public class LeastExpensiveStrategy implements ScoringStrategy {
    public int getScore(Product a)
    {
        // Score for product a is the negative of its cost, so that lower costs get higher scores
        return -a.cost;
    }
}
```

Figure 10. Implementation of the LeastExpensiveStrategy class that implements the ScoringStrategy interface. This class defines the getScore() method, which returns the negative cost of a given product, assigning higher scores to products with lower cost.

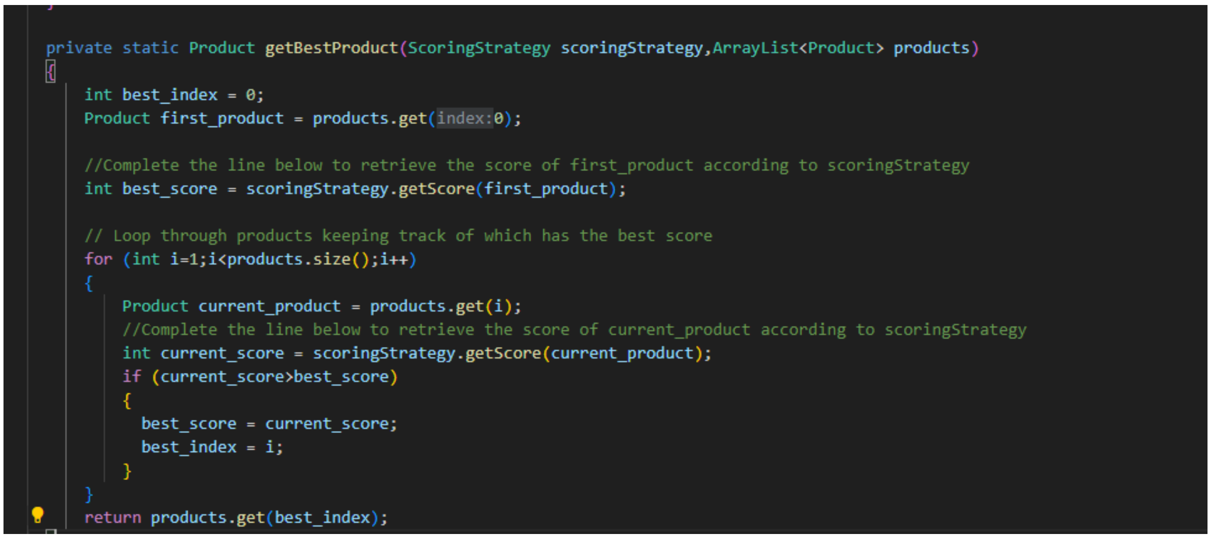
C)



```java
public class MostPracticalStrategy implements ScoringStrategy {
    public int getScore(Product a)
    {
        // Give score for product a as its practicality so that higher practicality gets higher scores
        return a.practicality;
    }
}
```

Figure 11. Implementation of the MostPracticalStrategy class that implements the ScoringStrategy interface. This class defines the getScore() method, which returns the practicality of a given product, assigning higher scores to products with higher practicality.

D)



```java
private static Product getBestProduct(ScoringStrategy scoringStrategy,ArrayList<Product> products)
{
    int best_index = 0;
    Product first_product = products.get(index:0);

    //Complete the line below to retrieve the score of first_product according to scoringStrategy
    int best_score = scoringStrategy.getScore(first_product);

    // Loop through products keeping track of which has the best score
    for (int i=1;i<products.size();i++)
    {
        Product current_product = products.get(i);
        //Complete the line below to retrieve the score of current_product according to scoringStrategy
        int current_score = scoringStrategy.getScore(current_product);
        if (current_score>best_score)
        {
          best_score = current_score;
          best_index = i;
        }
    }
    return products.get(best_index);
```

Figure 12. getBestProduct method that finds the best product from an ArrayList based on a given scoring strategy using a loop that iterates through each product and retrieves its score

The two lines of required completed code in Figure 12 play a key role in allowing the ProductRecommender class to use different scoring strategies, as the getScore() method of the scoringStrategy object is called dynamically at runtime.