```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;

public class WordExtractor {
    public static void main(String[] args) {
        String vocabularyFile = "./google-10000-english-no-swears.txt";
        String inputFile = "./Input219.txt";

        ArrayList<String> validWords = loadVocabulary(vocabularyFile);
        ArrayList<String> extractedWords = extractWords(inputFile, validWords);

        for (String word : extractedWords) {
            System.out.println(word);
        }
    }

    private static ArrayList<String> loadVocabulary(String filename) { // loads the vocabulary words from the file specified by "vocabularyFile" into an ArrayList called "validWords" by calling the "loadVocabulary" method.
        ArrayList<String> validWords = new ArrayList<>();

        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = reader.readLine()) != null) {
                validWords.add(line.toLowerCase());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
        return validWords;
    }

    private static ArrayList<String> extractWords(String filename, ArrayList<String> validWords) { // the valid words from the input file specified by "inputFile" using the "extractWords" method, and the extracted words are stored in an ArrayList called "extractedWords.
        ArrayList<String> extractedWords = new ArrayList<>();

        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) { //"loadVocabulary" method takes a filename as input and returns an ArrayList of strings and reads the file line by line using a BufferedReader and adds each line (converted to lowercase) to the "validWords" ArrayList.
            String line;
            while ((line = reader.readLine()) != null) {
                String[] words = line.split("\\s+"); //reads the file line by line and splits each line into individual words using the regular expression "\\s+"
                for (String word : words) {
                    if (validWords.contains(word.toLowerCase())) {
                        extractedWords.add(word);
                    }
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        return extractedWords; //returns the "extractedWords" ArrayList containing the valid words extracted from the input file
    }
}
```

QUESTION 1A:

##

The code has a Java program called "WordExtractor" that takes two text files and extracts the valid words from the provided vocabulary.

The code imports necessary Java classes and packages for file I/0 operations and collections (ArrayList). The "WordExtractor" is a class defined with the main method, which is the entry point for the program. The two variables, "vocabularyFile" and "inputFile," are declared with the paths to the vocabulary file and the input file, respectively. This program loads the vocabulary words from the file specified by "vocabularyFile" into an ArrayList called "validWords" by calling the "loadVocabulary" method.

This extracts the valid words from the input file specified by "inputFile" using the "extractWords" method, and the extracted words are stored in an ArrayList called "extractedWords." A loop is used to iterate over the "extractedWords" list, and each word is printed to the console using the "System.out.println" statement. The "loadVocabulary" method takes a filename as input and returns an ArrayList of strings. It reads the file line by line using a BufferedReader and adds each line (converted to lowercase) to the "validWords" ArrayList.

The "extractWords" method takes a filename and an ArrayList of valid words as inputs and returns an ArrayList of extracted words. This reads the file line by line and splits each line into individual words using the regular expression "\\s+". It then checks if each word exists in the "validWords" list (after converting both to lowercase) and adds it to the "extractedWords" list if it is valid. If an exception occurs during file reading (IOException), the program prints the stack trace to the console. The program returns the "extractedWords" ArrayList containing the valid words extracted from the input file.

QUESTION 1B.

##

The use of ArrayLists in the given code affects the time complexity of the program as explained below:

(i). Loading the Vocabulary:

It reads the vocabulary file and add each word to the ArrayList "validWords" which has a time complexity of O(n), where n is the number of words in the vocabulary file. Adding an element to an ArrayList has an average time complexity of O(1), but when the ArrayList needs to resize its underlying array, it may take O(n) time to copy the existing elements to the new array.

(ii). Extracting of Words:

It reads the input file line by line and splitting each line into words has a time complexity of O(m), where m is the number of words in the input file. In order to check if each word exists in the "validWords" ArrayList using the "contains" method has an average time complexity of O(n), where n is the number of elements in the ArrayList. It is because the "contains" method iterates over the ArrayList to find a match. Therefore, the overall time complexity of extracting words from the input file is O(m * n), because both the "contains" check and the loop over the input words contribute to the time complexity.

(iii). Printing the extracted Words:

Iterating over the "extractedWords" ArrayList to print each word has a time complexity of O(K), where K is the number of extracted words. Overall, the time complexity of the program can be represented as O(n + m * n + K), where n represents the number of words in the vocabulary file, m represents the number of words in the input file, and K represents the number of extracted words.

EXAMPLES:

Let us give an instance that we have a vocabulary file with 1,000 lines (n = 1,000) and an input file with 10,000 words (m = 10,000).

1. Loading Vocabulary: The "loadVocabulary" method reads each line from the vocabulary file and adds it to the "validWords" ArrayList. Since there are 1,000 lines, the time complexity of this operation would be O(n) = O(1,000). Adding an element to an ArrayList takes constant time on average.

2. Extracting Words: The "extractWords" method reads each line from the input file and splits it into individual words. For each word, it checks if it exists in the "validWords" ArrayList. Since there are 10,000 words, the time complexity of this operation would be O(m) = O(10,000). Searching for an element in an ArrayList has a time complexity of O(n) on average, where 'n' is the size of the ArrayList.

Therefore, the time complexity of the program in this example would be O(n + m) = O(1,000 + 10,000) = O(11,000).

QUESTION 1C:

##
#

Yes, I think we have few of them:

a.Use a Set for valid words: Instead of using an ArrayList for storing valid words, you can use a HashSet or TreeSet data structure from the Java Collections framework. Sets offer constant time complexity for the "contains" operation, which can significantly improve the performance of checking if a word is valid. The loadVocabulary method can be modified to return a Set<String> instead of an ArrayList<String>.

b.Use a StringBuilder for printing extracted words: Instead of directly printing each word using System.out.println in the loop, you can utilize a StringBuilder to construct the output string. Appending words to a StringBuilder and then printing the final result at once can be more efficient than multiple print statements.

```
1  Best product according to LeastExpensiveStrategy:
2  Vauxhall Nova
3  Best product according to MostPracticalStrategy:
4  Skoda Octavia
5
```

c.Employ parallel processing: If the size of the input file and the number of valid words are significantly large, you can consider using parallel processing techniques. For example, you can split the input file into chunks and process each chunk in parallel using multiple threads or utilize Java's parallel stream processing for handling the extraction and validation of words concurrently.

d.Utilize buffered reading for input files: While the code currently uses BufferedReader, it does not take full advantage of its buffering capabilities. You can increase the reading efficiency by

wrapping the FileReader with a BufferedReader while specifying a larger buffer size, such as BufferedReader reader = new BufferedReader(new FileReader(filename), BUFFER_SIZE). Adjusting the BUFFER_SIZE value can help optimize the reading performance.

EXAMPLES:

```java
import java.io.*;
import java.util.*;

public class WordExtractor {
    private static final int BUFFER_SIZE = 8192;

    public static void main(String[] args) {
        String vocabularyFile = "./google-10000-english-no-swears.txt";
        String inputFile = "./Input219.txt";

        Set<String> validWords = loadVocabulary(vocabularyFile);
        List<String> extractedWords = extractWords(inputFile, validWords);

        StringBuilder output = new StringBuilder();
        for (String word : extractedWords) {
            output.append(word).append("\n");
        }
        System.out.println(output.toString());
    }
```

```
private static Set<String> loadVocabulary(String filename) {
    Set<String> validWords = new HashSet<>();

    try (BufferedReader reader = new BufferedReader(new FileReader(filename), BUFFER_SIZE)) {
        String line;
        while ((line = reader.readLine()) != null) {
            validWords.add(line.toLowerCase());
        }
    } catch (IOException e) {
        e.printStackTrace();
    }

    return validWords;
}

private static List<String> extractWords(String filename, Set<String> validWords) {
    List<String> extractedWords = new ArrayList<>();

    try (BufferedReader reader = new BufferedReader(new FileReader(filename), BUFFER_SIZE)) {
        String line;
        while ((line = reader.readLine()) != null) {
            String[] words = line.split("\\s+");
            for (String word : words) {
                if (validWords.contains(word.toLowerCase())) {
                    extractedWords.add(word);
                }
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
```

```
        }

        return extractedWords;
    }
}
```

2:1.

SORTING:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;

public class WordExtractor {
    private static int moves = 0;
    private static int comparisons = 0;
    public static void main(String[] args) {
        String vocabularyFile = "./google-10000-english-no-swears.txt";
        String inputFile = "./Input219.txt";

        ArrayList<String> validWords = loadVocabulary(vocabularyFile);
```

```
        ArrayList<String> extractedWords = extractWords(inputFile, validWords);

       //  System.out.println("Original list:");
       // printWordList(extractedWords);

       // System.out.println("\nSorting...");
        long startTime = System.nanoTime();
        mergeSort(extractedWords, 0, extractedWords.size() - 1);
        long endTime = System.nanoTime();
        double elapsedTime = (endTime - startTime) / 1e6; // Convert to milliseconds

        System.out.println("\nSorted list:");
        printWordList(extractedWords);

        System.out.println("\nTime taken for sorting: " + elapsedTime + " ms");
        System.out.println("Moves: " + moves);
        System.out.println("Comparisons: " + comparisons);

    }

    private static ArrayList<String> loadVocabulary(String filename) { // `loadVocabulary` method reads the `vocabularyFile` line by line and adds each word to an `ArrayList` called `validWords`
        ArrayList<String> validWords = new ArrayList<>();

        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = reader.readLine()) != null) {
                validWords.add(line.toLowerCase());
            }
        } catch (IOException e) {
            e.printStackTrace();
```

```
    }

    return validWords;
  }

  private static ArrayList<String> extractWords(String filename, ArrayList<String> validWords) { //This `extractWords` method reads the `inputFile` line by line.
    ArrayList<String> extractedWords = new ArrayList<>();

    try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
      String line;
      while ((line = reader.readLine()) != null) {
        String[] words = line.split("\\s+");
        for (String word : words) {
          if (validWords.contains(word.toLowerCase())) {
            extractedWords.add(word);
          }
        }
      }
    } catch (IOException e) {
      e.printStackTrace();
    }

    return extractedWords;
  }

  // mergeSort implementation
  private static void mergeSort(ArrayList<String> words, int left, int right) {
    if (left < right) {
      int mid = (left + right) / 2;
      mergeSort(words, left, mid);
```

```
        mergeSort(words, mid + 1, right);
        merge(words, left, mid, right);
    }
}

// merge implementation

//takes two sorted sublists (`left` to `mid` and `mid + 1` to `right`) and merges them into a single sorted list. It uses an auxiliary `ArrayList` called `temp` to store the merged result.
private static void merge(ArrayList<String> words, int left, int mid, int right) {
    ArrayList<String> temp = new ArrayList<>();
    int i = left;
    int j = mid + 1;

    while (i <= mid || j <= right) {
        if (i > mid) {
            temp.add(words.get(j++));
        } else if (j > right) {
            temp.add(words.get(i++));
        } else if (words.get(i).compareTo(words.get(j)) <= 0) {
            temp.add(words.get(i++));
        } else {
            temp.add(words.get(j++));
        }
    }

    for (int k = 0; k < temp.size(); k++) {
        moves++;
        words.set(left + k, temp.get(k));
    }
}
```

```
    private static void printWordList(ArrayList<String> words) {
        for (String word : words) {
            System.out.println(word); //prints the sorted list of words, the time taken for sorting, the number of moves performed during sorting, and the number of comparisons made.
        }
    }
}
```

QUESTION 2:2

EXPLANATION:

The code above starts by specifying the paths to two files: `vocabularyFile`, which contains a list of valid words, and `inputFile`, which contains the input text from which words will be extracted. The `loadVocabulary` method reads the `vocabularyFile` line by line and adds each word to an `ArrayList` called `validWords`. The words are converted to lowercase before being added to ensure case-insensitive matching. This `extractWords` method reads the `inputFile` line by line. It splits each line into individual words using whitespace as the delimiter. For each word, it checks if it exists in the `validWords` list (after converting it to lowercase). If the word is valid, it is added to another `ArrayList` called `extractedWords`.

Also `mergeSort` method is an implementation of the merge sort algorithm. Its function is to recursively divides the `words` list into smaller sublists until each sublist contains a single element. Then, merges the sublists back together in a sorted order. The process above is repeated until the entire list is sorted. The `merge` method is a helper method used by `mergeSort`. It takes two sorted sublists (`left` to `mid` and `mid + 1` to `right`) and merges them into a single sorted list. It uses an auxiliary `ArrayList` called `temp` to store the merged result. In the `main` method, the program loads the `validWords` from the `vocabularyFile` and extracts the valid words from the `inputFile` using the `loadVocabulary` and `extractWords` methods, respectively.

The program prints the sorted list of words, the time taken for sorting, the number of moves performed during sorting, and the number of comparisons made.

Overall, the program reads valid words from a file, extracts words from another file based on the valid words, sorts the extracted words using merge sort, and displays the sorted list along with some statistics about the sorting process.

QUESTION 2:3

Sorted list:

A

Algorithms

Algorithms

Algorithms

An

An

Computer

Computers

Each

INPUT

If

If

If

In

In

It

It

Most

OUTPUT

Sometimes

Sometimes

There

There

These

This

This

Using

Writing

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

a

a

a

a

a

about

about

algorithm

algorithm

algorithm

algorithms

algorithms

already

an

an

an

an

and

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

back

be

be

be

be

be

be

break

broken

can

can

can

can

can

carried

clear

computational

computer

computer

computer

computer

creating

cup

data

describing

detail

detail

developed

diagram

different

different

do

do

do

do

does

down

down

dressed

each

exactly

exactly

few

finishing

fit

follow

for

for

four

garbage

get

get

give

good

have

have

have

have

hence

how

how

how

identified

important

in

in

in

in

in

in

in

in

in

in

in

in

in

in

including

instruction

instructions

instructions

instructions

instructions

into

into

is

is

is

is

is

is

is

is

is

is

is

it

it

it

it

its

just

know

languages

level

line

lot

main

make

make

many

many

meal

message

messages

must

must

must

need

needed

normally

not

not

number

occur

of

of

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

own

parts

plan

plan

point

point

poor

poor

prepare

prints

problem

problem

processing

program

program

program

programming

programming

programming

programs

provide

real

represent

represented

represented

represents

result

rules

run

sentence

set

set

set

set

should

similar

simple

simplified

smaller

so

so

solution

solve

solve

sometimes

specific

specific

standard

starting

starting

step

steps

steps

suitable

sure

symbols

syntax

tell

tell

that

that

that

that

that

that

that

that

the

the

the

the

the

the

the

the

the

the

the

the

the

the

then

then

they

they

they

they

they

things

thinking

this

tie

to

to

to

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

variables

want

want

want

way

ways

we

we

we

we

we

we

we

what

what

which

will

will

will

will

will

write

writing

written

written

written

you

you

you

you

Time taken for sorting: 2.3729 ms

Moves: 3418

Comparisons: 0

3.

Let's start with the LeastExpensiveStrategy:

```
public class LeastExpensiveStrategy implements ScoringStrategy {
    public int getScore(Product a) {
        // The score for the least expensive strategy should be higher for products with lower costs
        // You can calculate the score by subtracting the cost from a large constant value
        int score = Integer.MAX_VALUE - a.getCost();
        return score;
    }
}
```

let's move on to the MostPracticalStrategy:

```
public class MostPracticalStrategy implements ScoringStrategy {
    public int getScore(Product a) {
        // The score for the most practical strategy should be higher for products with higher practicality
        // You can directly return the practicality value as the score
        return a.getPracticality();
    }
}
```

A.

The purpose of the Strategy design pattern is to provide a way to dynamically change the behavior of an object at runtime. It allows you to define a family of algorithms or behaviors and encapsulate each one as a separate class. These classes, known as strategies, can be selected and used interchangeably based on the requirements or context of the application. Strategy pattern promotes the principle of composition over inheritance and enables the encapsulation of algorithms in separate classes. It allows for better code organization, flexibility, and extensibility. By decoupling the implementation

details of different strategies from the client code, the Strategy pattern promotes code reuse and simplifies maintenance.

From the code above, the Strategy pattern allows us to define multiple scoring strategies (`LeastExpensiveStrategy` and `MostPracticalStrategy`) that can be applied to different products. By encapsulating each strategy in a separate class, we can easily switch between strategies or add new ones without modifying the client code that uses the strategies. It also promotes flexibility and makes it easier to adapt to changing requirements or add new features to your program.

B.

```
public class LeastExpensiveStrategy implements ScoringStrategy {
    public int getScore(Product a) {
        // The score for the least expensive strategy should be higher for products with lower costs
        // You can calculate the score by negating the cost value to invert the ordering
        return -a.getCost();
    }
}
```

C.

```
public class MostPracticalStrategy implements ScoringStrategy {
    public int getScore(Product a) {
        // The score for the most practical strategy should be higher for products with higher practicality
        // You can directly return the practicality value as the score
        return a.getPracticality();
    }
}
```

D.

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

    int best_score = scoringStrategy.getScore(best_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);
        int current_score = scoringStrategy.getScore(current_product);

        if (current_score > best_score) {
            best_score = current_score;
            best_index = i;
        }
    }

    return products.get(best_index);
}
```

OUTPUT:

```
Best product according to LeastExpensiveStrategy:
Vauxhall Nova
Best product according to MostPracticalStrategy:
Skoda Octavia

```