Comprehensive Practice

Section One

Goal

- In this section we will develop a program that will read two different text files and compare their vocabulary.
- In particular, we will determine the set of words used in each file and compute the overlap between them.
- Researchers in the humanities often perform such comparisons of vocabulary in selections of text to answer questions like, “Did Christopher Marlowe actually write Shakespeare’s plays?”

- We will develop the program in stages:
- 1. The first version will read the two files and report the unique words in each.
    - We will use short testing files for this stage.
- 2. The second version will also compute the overlap between the two files (i.e., the set of words that appear in both files).
    - We will continue to use short testing files for this stage.
- 3. The third version will read from large text files and will perform some analysis of the results
    - including the number of words in each list, the number of words of overlap, and the percentage of overlap.

Some Efficiency Considerations

- As you start writing more complex programs, you have to worry about programs running slowly because they are performing complex computations or handling large amounts of data.
- We need to explore it at least briefly for this exercise because otherwise we are likely to pursue an approach that won’t work well.
- For example, in the program we are about to write, we have to read in a file and come up with a list of the words from the file that doesn’t have any duplicates. One approach would be to test each word as we read it in to see if it is in the list, as described in the following pseudocode:

```
list = new empty list.
while (more words to process) {
    word = next word from file.
    if (list does not contain word) {
        add word to list.
    }
}
```

- The problem with this approach is that it would require us to call the ArrayList method called contains each time the program executes the loop.
- It turns out that the contains method can be fairly expensive to call in terms of time.
- To find out whether a particular value is in the list, the method has to go through each different value in the list.
- So as the list becomes larger and larger, it becomes more and more expensive to search through it to see if it contains a particular word.

- We will run into a similar problem when we get to the second version of the program and have to compute the overlap between the two lists.
- The simplest way to compute the overlap would be to write a method like this:

```
overlap = new empty list.
for (each word in list1) {
      if (word is in list2) {
           add word to overlap.
      }
}
```

- This approach will again require calling the contains method for a list that could potentially be very large. If both lists are large, then the approach will run particularly slowly.

- Both of these potential bottlenecks can be addressed by dealing with sorted lists.
- In a sorted list of words, all of the duplicates are grouped together, which makes them easier to spot.
  - And looking for the overlap between two sorted lists is easier than looking for overlap in two lists that are not ordered.
- Of course, sorting isn’t cheap either. It takes a nontrivial amount of time to sort a list.
  - But if we can manage to sort the list just once, it will turn out to be cheaper than making all of those calls on the contains method.

- Instead of trying to eliminate the duplicates as we read the words, we can just read all of the words directly into the list.
  - That way we won't make any expensive calls on the contains method.
- After we have read everything in, we can put the list into sorted order.
  - When we do that, all of the duplicates will appear right next to one another, so we can fairly easily get rid of them.
- Reading all of the words into the list and then eliminating duplicates will require more memory than eliminating the duplicates as we go, but it will end up running faster because the only expensive operation we will have is the sorting step.
  - This is a classic tradeoff between running time and memory that comes up often in computer science.
- We can make programs run faster if we're willing to use more memory or we can limit memory if we don't mind having the program take longer to run.

- Our approach to building up the list of words, then, will be as follows:

```
list = new empty list.
while (there are more words to process) {
    add word to list.
}
sort list.
eliminate duplicates.
```

- The task of eliminating duplicates also brings up an efficiency consideration.
- One obvious approach would be the following:

```
for (each word in list) {
    if (word is a duplicate) {
        remove word from list.
    }
}
```

- It turns out that remove is another expensive operation.
  - A better approach is to simply build up a new list that doesn’t have duplicates:

```
result = new empty list.
for (each word in list) {
    if (word is not a duplicate) {
        add word to result.
    }
}
```

- This code runs faster because the method that adds a word at the end of the list runs very fast compared with the method that removes a word from the middle of the list.
- As we write the actual code, we will refine the pseudocode presented here
  - For each file, read in all of the words into a list and sort it once.
  - Then use the sorted lists to build up lists that have no duplicates.
  - Finally use those two sorted lists to look for the overlap between the two lists.

- Once the list has been sorted, duplicates of any words will be grouped together.
- Remember that our plan is to build up a new list that has no duplicates.
- The simplest way to eliminate duplicates is to look for transitions between words.
  - For example, if we have 5 occurrences of one word followed by 10 occurrences of another word, most of the pairs of adjacent words will be equal to each other.
  - However, in the middle of those equal pairs, when we make the transition from the first word to the second word, there will be a pair of words that are not equal.
  - Whenever we see such a transition, we know that we are seeing a new word that should be added to our new list.

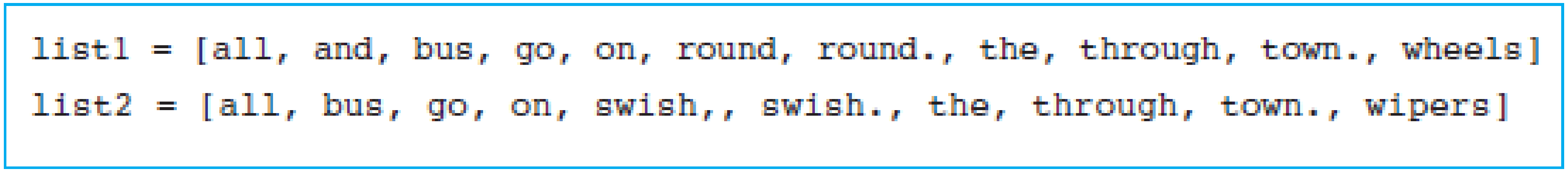
- Looking for transitions leads to a classic fencepost problem.
  - For example, if there are 10 unique words, there will be 9 transitions.
  - We can solve the fencepost problem by adding the first word before the loop begins.
  - Then we can look for words that are not equal to the words that precede them and add them to the list.
  - Expressed as pseudocode, our solution is as follows:

```
construct a new empty list.
add first word to new list.
for (each i) {
    if (value at i does not equal value at i-1) {
        add value at i.
    }
}
```

Version 2: Compute Overlap

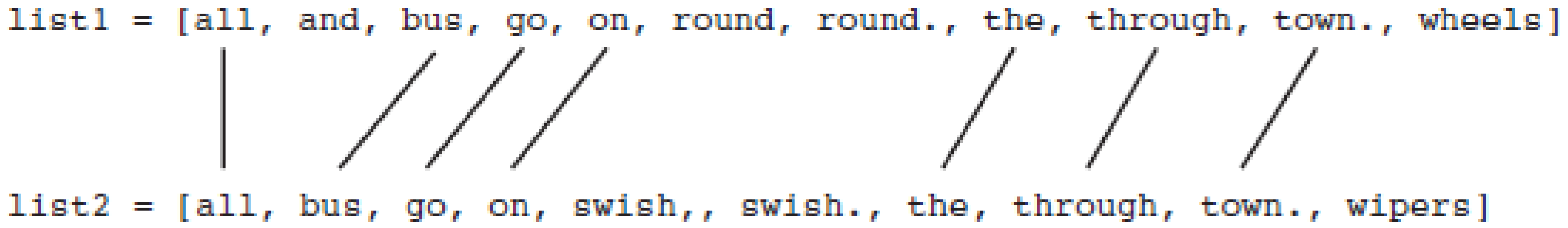
- The first version of the program produces two sorted ArrayLists containing sets of unique words.
- For the second version, we want to compute the overlap between the two lists of words and report it.
- This operation is complex enough that it deserves to be in its own method.
- So, we'll add the following line of code to the main method right after the two word lists are constructed:
  - ArrayList<String> common = getOverlap(list1, list2);

- The primary task for the second version of our program is to implement the getOverlap method.
- Look closely at the two lists of words produced by the first version:



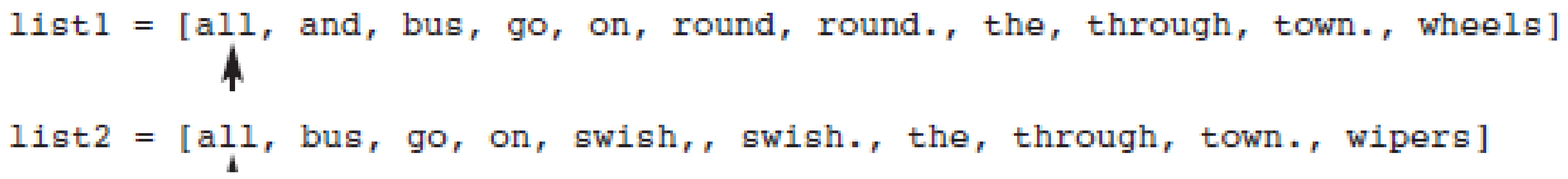
```
list1 = [all, and, bus, go, on, round, round., the, through, town., wheels]
list2 = [all, bus, go, on, swish,, swish., the, through, town., wipers]
```

- People are pretty good at finding matches, so you can probably see exactly which words overlap.
  - Both lists begin with "all", so that is part of the overlap.
  - Skipping past the word "and" in the first list, we find the next match is for the word "bus".
  - Then we have two more matches with the words "go" and "on".
  - Next there are a couple of words in a row in both lists that don’t match, followed by matches with the words "the", "through", and "town.",
  - and a final unique word in each list.
  - The complete set of matches is as follows:



```
list1 = [all, and, bus, go, on, round, round., the, through, town., wheels]
list2 = [all, bus, go, on, swish,, swish., the, through, town., wipers]
```

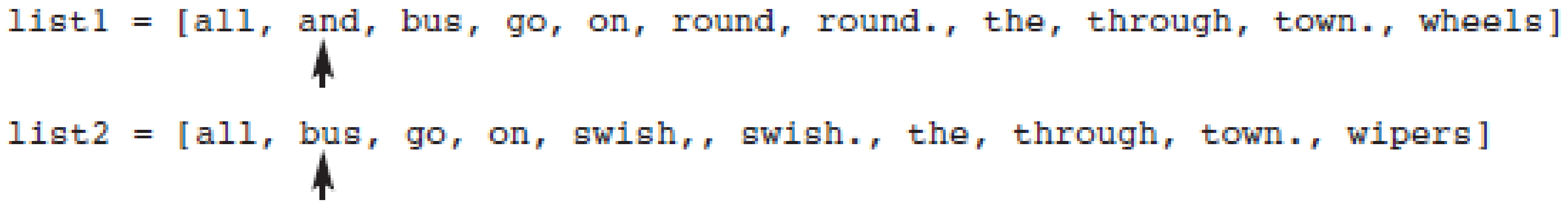
- We want to design an algorithm that parallels what people do when they look for matches.
- Imagine putting a finger from your left hand on the first list and putting a finger from your right hand on the second list to keep track of where you are in each list.
- We will compare the words at which you are pointing and, depending on how they compare, move one or both fingers forward.
- We start with the left finger on the word "all" in the first list and the right finger on the word "all" in the second list.



```
list1 = [all, and, bus, go, on, round, round., the, through, town., wheels]

list2 = [all, bus, go, on, swish,, swish., the, through, town., wipers]
```

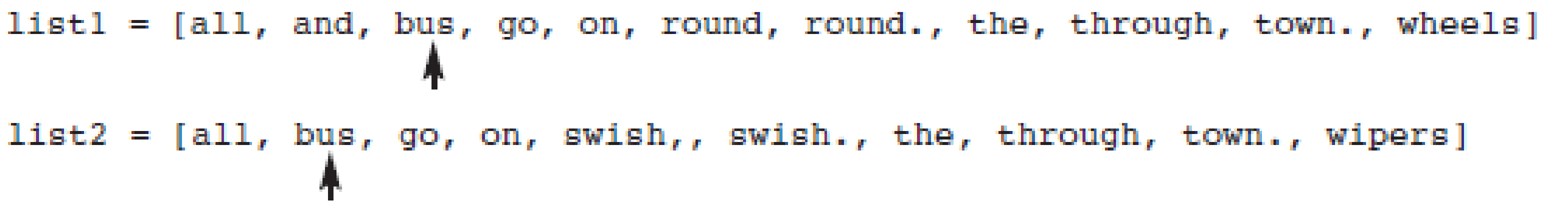
- The words match, so we add that word to the overlap and move both fingers forward:



```
list1 = [all, and, bus, go, on, round, round., the, through, town., wheels]

list2 = [all, bus, go, on, swish,, swish., the, through, town., wipers]
```

- Now we are pointing at the word "and" from the first list and the word "bus"
- from the second list.
  - The words don’t match.
  - So what do you do?
- It turns out that the word "bus" in list2 is going to match a word in list1.
  - So how do you know to move the left finger forward?
  - Because the lists are sorted and because the word "and" comes before the word "bus", we know that there can’t be a match for theord "and" in the second list.
  - Every word that comes after "bus" in the second list will be alphabetically greater than "bus",
  - so the word "and" can’t be there.
  - Thus, we can move the left finger forward to skip the word "and":



```
list1 = [all, and, bus, go, on, round, round., the, through, town., wheels]

list2 = [all, bus, go, on, swish,, swish., the, through, town., wipers]
```

- This gets us to the second match, for "bus", and the algorithm proceeds.
- In general, we find ourselves in one of three situations when we compare the current word in list1 with the current word in list2:
  - The words might be equal, in which case we’ve found a match that should be included in the overlap and we should advance to the next word in each list.
  - The word from the first list might be alphabetically less than the word from the second list, in which case we can skip it because it can’t match anything in the second list.
  - The word from the second list might be alphabetically less than the word from the first list, in which case we can skip it because it can’t match anything in the first list.

- Thus, the basic approach we want to use can be described with the following pseudocode:

```
if (word from list1 equals word from list2) {
    record match.
    skip word in each list.
} else if (word from list1 < word from list2) {
    skip word in list1.
} else {
    skip word in list2.
}
```

Punctuation

- This also seems like a good time to think about punctuation.
- The earlier versions allowed words to contain punctuation characters such as commas, periods, and dashes that we wouldn’t normally consider to be part of a word.
- We can improve our solution by telling the Scanner what parts of the input file to
- ignore.
- Scanner objects have a method called useDelimiter that you can call to tell them what characters to use when they break the input file into tokens.
  - When you call the method, you pass it what is known as a *regular expression.*
  - Regular expressions are a highly flexible way to describe patterns of characters.
- For our purposes, we want to form a regular expression that will instruct the Scanner to look just at characters that are part of what we consider words.
  - That is, we want the Scanner to look at letters and apostrophes.
  - The following regular expression is a good starting point:
    - [a–zA–Z']

- [a–zA–Z']
- This regular expression would be read as
  - "Any character in the range of a to z, the range of A to Z, or an apostrophe."
  - This is a good description of the kind of characters we want the Scanner to include.
  - But we actually need to tell the Scanner what characters to *ignore*
    - so we need to indicate that it should use the opposite set of characters.
  - The easy way to do this is by including a caret (∧) in front of the list of legal characters:
    - [∧a-zA-Z']
  - This regular expression would be read as, "Any character other than the characters that are in the range of a to z, the range of A to Z, or an apostrophe."

- Even this expression isn’t quite right, though
  - because there might be many such characters in a row.
  - For example, there might be several spaces, dashes, or other punctuation characters separating two words.
  - We can indicate that a sequence of illegal characters should also be ignored by putting a plus after the square brackets to indicate “Any sequence of one or more of these characters”:
    - [^a-zA-Z']+

useDelimiter

- We pass this regular expression as a String to a call on useDelimiter.
- We can add this at the beginning of the getWords method:

```
public static ArrayList<String> getWords(Scanner
input) {
        input.useDelimiter("[^a-zA-Z']+");
        …
}
```