Statistical analysis in Hangman

Part 1a. Statistical estimates of the number of words

HashSet only keeps unique words. Random selection of words from the Arraylist inevitably causes repeated selections of the same words, which will be added just once in a HashSet. Therefore, the number of elements in the HashSet is always smaller than the times the loop runs. I reasoned that the more "samplings" (looping), the bigger chance that the HashSet covers all the unique words of specific length in the text file. To test this idea, I want to see the trend of word number as opposed to expanding rounds of looping in my program `HangmanStats`. I created an array of integers `round` including numbers from 1,000 to 500,000. These numbers, when looped through in the program, are used as the limit of the inclusive looping filling in the HashSet `set` for each word length from 4 to 20. After looping over each limit `round[j]`, I printed out the numbers of words `set.size()` with a specified length `i` and cleared up the set.

Here are my codes for Part 1a.

```
import java.util.*;

public class HangmanStats {
    public static void main(String[] args) {
        HangmanFileLoader loader = new HangmanFileLoader();
        loader.readFile("lowerwords.txt");
        HashSet<String> set = new HashSet<String>();
        int[] round = {1000,5000,10000,20000,50000,100000,500000};
        for(int i = 4;i<21;i++){
            for (int j = 0;j < round.length; j++){
                for(int k = 0; k < round[j]; k += 1) {
                    set.add (loader.getRandomWord(i));
                    }
            System.out.printf("number of %d letter words = %d\n",
i,set.size());
            set.clear();
        }
        }
    }
}
```

The returned numbers for each word length are shown in Table 1 and Table 2. As I expected, the more times a word is randomly picked, the more words my program returns. Most of the numbers get fixed when the limit of loop reaches 100,000 (see the part labeled with yellow), suggesting that these numbers are likely to hit the peak value or the actual number of words. For each word length from 4 to 20, the final estimate of word number is the biggest number returned using different times of loop, as shown in the last rows of Table 1 and 2. However, it remains possible that the estimates are a little fewer than the actual number of words by random chance.

Table 1. Results of word number estimates from different times of loops

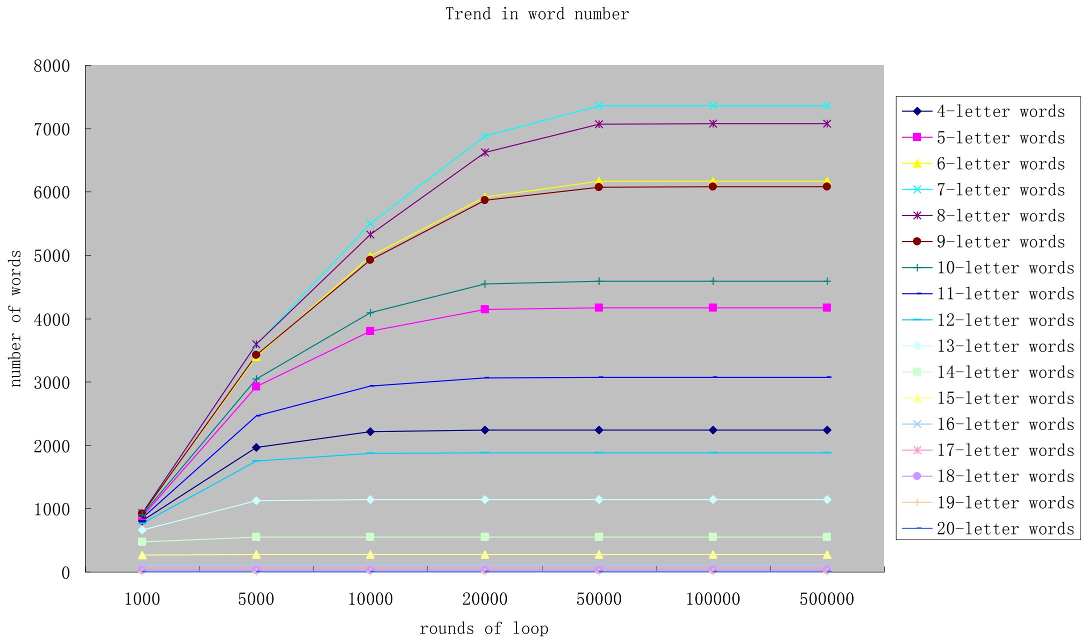
word length / **loop limit**	4	5	6	7	8	9	10	11
1000	809	876	924	938	929	916	900	845
5000	1963	2921	3392	3581	3592	3422	3040	2463
10000	2212	3800	4980	5499	5324	4920	4090	2935
20000	2235	4141	5917	6873	6620	5869	4544	3062
50000	2235	4170	6163	7353	7062	6075	4590	3069
100000	2235	4170	6166	7358	7070	6079	4591	3069
500000	2235	4170	6166	7359	7070	6079	4591	3069
final estimate	**2235**	**4170**	**6166**	**7359**	**7070**	**6079**	**4591**	**3069**

Table 2. Results of word number estimates from different times of loops (cont.)

word length / **loop limit**	12	13	14	15	16	17	18	19	20
1000	763	662	468	267	103	57	23	3	3
5000	1748	1124	545	278	103	57	23	3	3
10000	1870	1137	545	278	103	57	23	3	3
20000	1880	1137	545	278	103	57	23	3	3
50000	1880	1137	545	278	103	57	23	3	3
100000	1880	1137	545	278	103	57	23	3	3
500000	1880	1137	545	278	103	57	23	3	3
final estimate	**1880**	**1137**	**545**	**278**	**103**	**57**	**23**	**3**	**3**

The general trend of how word number estimates change vs. times of loop is shown in Diagram 1. It's obvious that all of the word numbers hit the plateau eventually when the times of loop are big enough. This indicates it is reasonable to use the plateau value of in estimating the number of words.

Diagram 1. Trend of how number of words change vs. times of loop



Part 1b. My own question

The question I asked is how many calls are needed before one word previously returned is returned again.

To answer this question, I want to find out the number of loops needed to return a word already returned before. First I created an empty ArrayList of strings **`wordlist`** and added random words of specific lengths in it. Each time a new word is added, I check if it was picked before by looping through the old elements in **`wordlist`**. If it was, then I record the difference **`duplicateCall`** between the new index of the added word **`m`** and the last index of the same word **`n`** and print out **`duplicateCall`**. I found that this number is different every time the code was run. This is actually expected because picking a word twice is a random event and thus the number of calls in between varies by chance. I then decide to run this module 1000 times and get the average value **`average`** from the whole set of **`duplicateCall`** instead. I expect the values of **`average`** to be close to each other across different running.

Here are my codes for Part 1b.

```
import java.util.*;

public class HangmanMyStats {

  public static void main(String[] args) {
    HangmanFileLoader loader = new HangmanFileLoader();
    loader.readFile("lowerwords.txt");
    ArrayList<String> wordlist = new ArrayList<String>();
    for(int i = 4;i < 21;i++){
        double total = 0.0;
        for (int j = 0;j < 1000; j++){
              int duplicateCall;
              boolean duplicate = false;
              for(int m = 0; m < 10000; m += 1) {
                  wordlist.add(loader.getRandomWord(i));
                  for(int n = 0; n < m; n += 1) {
                      if(wordlist.get(n).equals(wordlist.get(m))){
                          duplicateCall=m-n;
                          total += duplicateCall;
```

```
                        duplicate = true;
                        break;
                    }
                }
                if (duplicate)
                    break;
            }
            wordlist.clear();
        }
        double average = total/1000;
        System.out.printf("On average %f calls are needed before
returning the same %d letter words again\n",average,i);
            }

    }

}
```

I ran the above program three times and the results are shown in Table 3 and 4. The numbers have been rounded off. As I expected, the average numbers of calls needed for each word length are pretty close among results of three runs, suggesting that my program identifies the central range in distributions of these numbers.

The averages of the returned values from three runs are put in the last row of Table 3 and 4, which are used as an estimate of average calls needed for one word to be returned again. Notice that these estimates well correlate with the corresponding estimates of total number of words with the same lengths. This phenomenon makes sense because the more words in a list, the smaller the chance a word from the list will be picked twice, further justifying that my program is reasonable.

Table 3. Results of call number estimates in three runs

word length / run	4	5	6	7	8	9	10	11
1st	31.388	41.967	51.376	53.656	50.698	51.37	41.278	36.335
2nd	29.682	40.124	49.427	53.808	51.645	49.759	43.264	33.552
3rd	30.186	40.294	50.693	54.452	54.504	49.438	42.615	34.78
average estimate	**30.42**	**40.80**	**50.50**	**53.97**	**52.28**	**50.19**	**42.39**	**34.89**

Table 4. Results of call number estimates in three runs (cont.)

word length / run	12	13	14	15	16	17	18	19	20
1st	27.665	21.838	14.86	10.668	6.76	5.049	3.464	1.47	1.455
2nd	27.938	21.476	14.721	10.76	6.631	4.939	3.331	1.452	1.427
3rd	27.022	21.055	14.373	11.016	6.662	5.019	3.402	1.42	1.434
average estimate	**27.54**	**21.46**	**14.65**	**10.81**	**6.68**	**5.00**	**3.40**	**1.45**	**1.44**

Extra Credit:

For extra credits, I created following Java classes and text files:
HangmanMyLoader.java;
HangmanMyGame.java;
HangmanMyExecuter.java;
GRE words.txt
Country names.txt

Basically, I prompt users for the choice of the text file to load. Then load the file and pick a random word from the file. Following parts are essentially the same as the regular word-oriented Hangman game.