# Part 1a: Statistical/Forensic Analysis

**Methodology/Strategy:**
To estimate the number of words with certain length in the dictionary, the more random words we get by increasing calls of "HangmanLoader.getRandomWords" and putting the unique words in a set, the closer the size of the set generated will be comparing to the total number of words in the dictionary. Of course, when increased to a certain level, the number of calls is large enough to make the estimation accurate enough, and the set size will reach a "plateau" and any more calls of "HangmanLoader.getRandomWords" would be unnecessary. Thus, my strategy is to increase calls of "HangmanLoader.getRandomWords" and decide when the set size reaches "plateau".

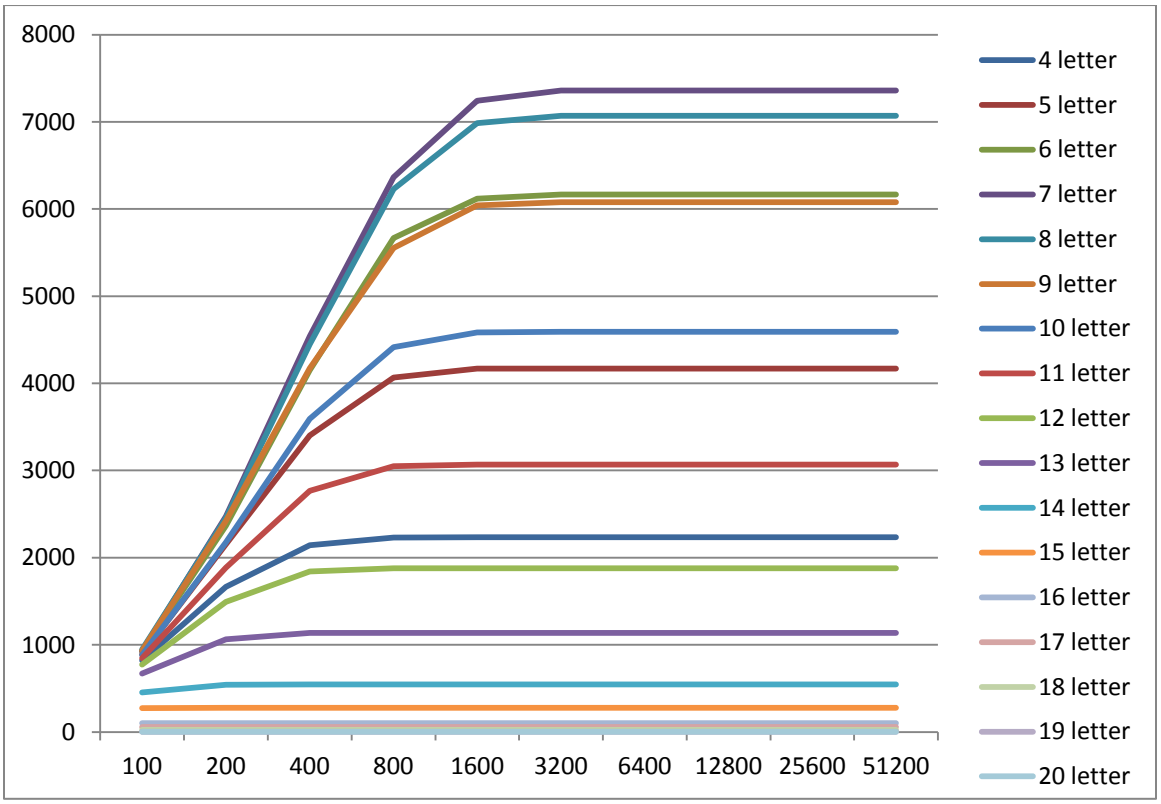
**My code is as follows (also saved in `HangmanStats`):**

```
import java.util.*;
public class HangmanStats {
	public static void main(String[] args) {
		HangmanFileLoader loader = new HangmanFileLoader();
		loader.readFile("lowerwords.txt");

		for(int i=4;i<21;i++){
			HashSet<String> set = new HashSet<String>();
			int[] number=new int[10];
			int limit=1000;
				for (int j=0;j<10;j++)
				{
					for(int k=0; k < limit; k += 1)
						{
						set.add(loader.getRandomWord(i));
						}
					number[j]=set.size();
					limit=limit*2;
				}

				for (int m=0;m<10;m++)

			System.out.printf("number of %d letter words = %d\n", i, number[m]);
		}
	}
}
```

In this code, an int array is generated by loop to keep the set number. In each loop cycle, limit, the number of calls of "HangmanLoader.getRandomWords" is increased by two-folds. After a defined 10 cycle loop, the numbers stored in the int array is printed. Finally, this process is done to words containing 4-20 letters in length. As is shown in the table and figure, although in general smaller word number become stable more quickly, starting from 64000 calls of "HangmanLoader.getRandomWords", the numbers of words with different lengths all become stable. And the stable numbers which represent the "plateau" can be treated as final estimation.

## The data generated:

| | | limit number of calls (n=10^3) | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1n | 2n | 4n | 8n | 16n | 32n | 64n | 128n | 256n | 512n | Conclusion |
| number of %d Letter words | 4 | 821 | 1667 | 2144 | 2232 | 2235 | 2235 | 2235 | 2235 | 2235 | 2235 | 2235 |
| | 5 | 890 | 2152 | 3403 | 4067 | 4168 | 4170 | 4170 | 4170 | 4170 | 4170 | 4170 |
| | 6 | 924 | 2368 | 4150 | 5668 | 6117 | 6165 | 6166 | 6166 | 6166 | 6166 | 6166 |
| | 7 | 927 | 2472 | 4541 | 6366 | 7241 | 7359 | 7359 | 7359 | 7359 | 7359 | 7359 |
| | 8 | 945 | 2454 | 4450 | 6230 | 6984 | 7069 | 7070 | 7070 | 7070 | 7070 | 7070 |
| | 9 | 936 | 2420 | 4173 | 5552 | 6043 | 6079 | 6079 | 6079 | 6079 | 6079 | 6079 |
| | 10 | 885 | 2174 | 3592 | 4417 | 4584 | 4591 | 4591 | 4591 | 4591 | 4591 | 4591 |
| | 11 | 846 | 1884 | 2768 | 3051 | 3068 | 3069 | 3069 | 3069 | 3069 | 3069 | 3069 |
| | 12 | 775 | 1492 | 1840 | 1880 | 1880 | 1880 | 1880 | 1880 | 1880 | 1880 | 1880 |
| | 13 | 669 | 1063 | 1135 | 1137 | 1137 | 1137 | 1137 | 1137 | 1137 | 1137 | 1137 |
| | 14 | 453 | 543 | 545 | 545 | 545 | 545 | 545 | 545 | 545 | 545 | 545 |
| | 15 | 274 | 278 | 278 | 278 | 278 | 278 | 278 | 278 | 278 | 278 | 278 |
| | 16 | 103 | 103 | 103 | 103 | 103 | 103 | 103 | 103 | 103 | 103 | 103 |
| | 17 | 57 | 57 | 57 | 57 | 57 | 57 | 57 | 57 | 57 | 57 | 57 |
| | 18 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 |
| | 19 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| | 20 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |



## Part1b: How many calls are needed before returning a word that was previously returned?

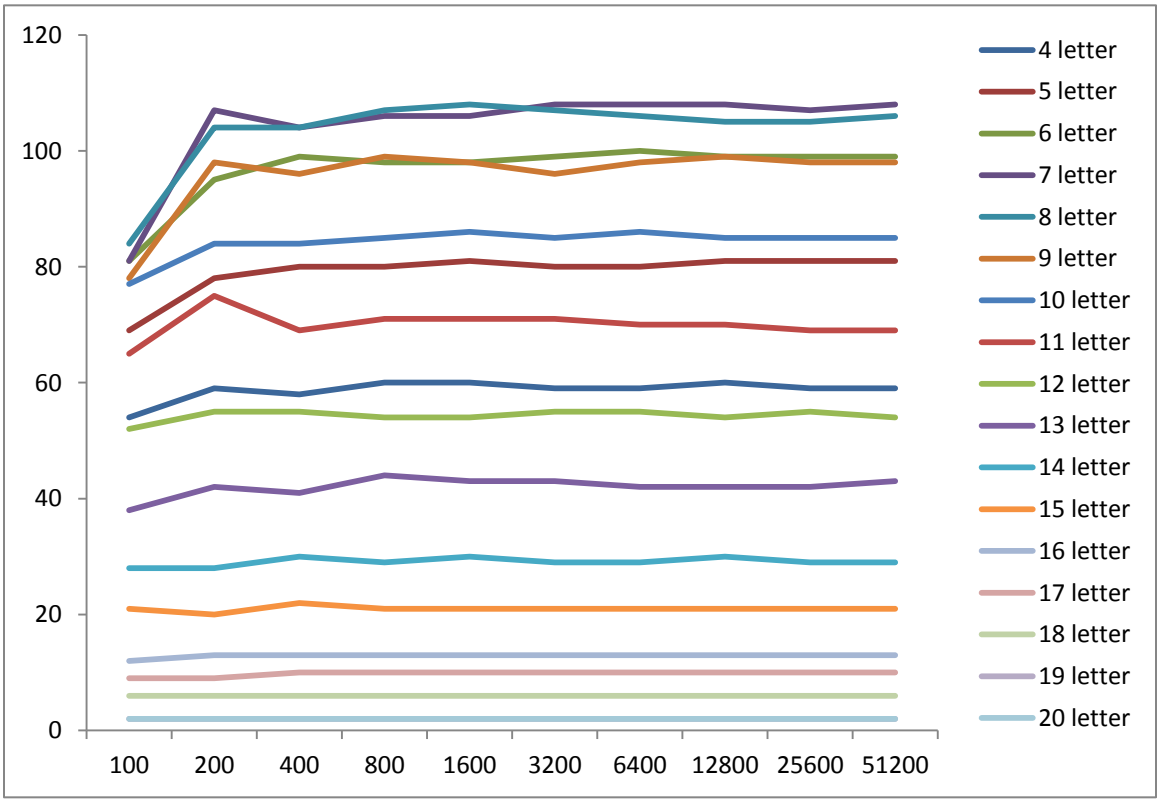
**This question is asking approximately after how many calls there will appear a random word that s not unique, which means it has appeared before. This problem can be solved by repeatedly calling of "**getRandomWords**" and record the number of calls needed to get a word that has appeared before. A loop is used for recording number of calls. However, one record may not be accurate, for you can either be too lucky or too unlucky that the number is either too small or too large. Thus, another loop to increase the number of records is needed to make the result more accurate, and different records can be stored in an int array. Although the number is likely to fluctuate, the more tests we make, the more accurate the result will be as an average of the records. Also, to determine when the tests are enough, I use another loop the increase the range limit of the test. My code is as follows: (also saved as** `HangmanTimes.java`)

```
import java.util.*;

public class HangmanTimes {
	public static void main(String[] args) {
		HangmanFileLoader loader = new HangmanFileLoader();
		loader.readFile("lowerwords.txt");

		for(int i=4;i<21;i++)
		{
			int limit=100;
			for (int m=0;m<10;m++){

			int[]calls=new int[limit];
			for (int j=0;j<limit;j++)
			  {
				ArrayList<String> wordlist=new ArrayList<String>();
				for(int k=0; k < limit; k ++)
							{
							String loopword=loader.getRandomWord(i);
							if(!wordlist.contains(loopword))
								wordlist.add(loopword);
							else break;
							}
				calls[j]=wordlist.size()+1;
			  }
			int total = 0;
			for(int temp:calls)
				total+=temp;

			double ave=total/limit;

	System.out.println(limit+ " tests show that " +ave+" calls of "+i+"-letter words
are needed ");
	limit*=2;
				}
		}
	}
}
```

**My answer and justification**

**As is shown in table and figure below, as the test number increases, the result become stable and is approximately the conclusion.**

| Tests number | 100 | 200 | 400 | 800 | 1600 | 3200 | 6400 | 12800 | 25600 | 51200 | conclusion |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 letter | 54 | 59 | 58 | 60 | 60 | 59 | 59 | 60 | 59 | 59 | 59 |
| 5 letter | 69 | 78 | 80 | 80 | 81 | 80 | 80 | 81 | 81 | 81 | 81 |
| 6 letter | 81 | 95 | 99 | 98 | 98 | 99 | 100 | 99 | 99 | 99 | 99 |
| 7 letter | 81 | 107 | 104 | 106 | 106 | 108 | 108 | 108 | 107 | 108 | 108 |
| 8 letter | 84 | 104 | 104 | 107 | 108 | 107 | 106 | 105 | 105 | 106 | 106 |
| 9 letter | 78 | 98 | 96 | 99 | 98 | 96 | 98 | 99 | 98 | 98 | 98 |
| 10 letter | 77 | 84 | 84 | 85 | 86 | 85 | 86 | 85 | 85 | 85 | 85 |
| 11 letter | 65 | 75 | 69 | 71 | 71 | 71 | 70 | 70 | 69 | 69 | 69 |
| 12 letter | 52 | 55 | 55 | 54 | 54 | 55 | 55 | 54 | 55 | 54 | 54 |
| 13 letter | 38 | 42 | 41 | 44 | 43 | 43 | 42 | 42 | 42 | 43 | 43 |
| 14 letter | 28 | 28 | 30 | 29 | 30 | 29 | 29 | 30 | 29 | 29 | 29 |
| 15 letter | 21 | 20 | 22 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 |
| 16 letter | 12 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 |
| 17 letter | 9 | 9 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| 18 letter | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| 19 letter | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 20 letter | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |



**Extra' question:**

1) Based on a set of words containing known number, what is a safe number of calling "getRandonWords" in order to get every word out of the set? That is, what is the relationship between the set size and the number of calls.

**Extra Credit**

To offer users the choice of playing hangman in more than one category, I provide three files named "fruits.txt", "animals.txt" and "colors.txt". Based on user's input, one of these three files will be loaded correspondingly. This is achieved in `HangmanCategory class` by the following code:

```
public void playcategory() {
      String choice= readString("Please enter'a','b'or'c' to choose game category:
      [a]fruits [b]animals [c]colors");
            HangmanFileLoader data = new HangmanFileLoader();

            if (choice.equals("a"))
             data.readmyFile("fruits.txt");
                  if(choice.equals("b"))
                        data.readmyFile("animals.txt");
                  if(choice.equals("c"))
                        data.readmyFile("colors.txt");
                  else System.out.println("Please choose again!");
```

Rather than load words to HashMap based on word-length, in the game of `HangmanCategory`, words from the chosen catergory are loaded to ArrayList regardless of their lengths. To achieve this, I add a new method in `HangmanFileLoader` called `readmyFile` :

```
ArrayList <String> categorylist=new ArrayList <String> ();

      public boolean readmyFile(String categoryfilename) {
            try {
                  FileReader dataFile = new FileReader(categoryfilename);
                  BufferedReader bufferedReader = new BufferedReader(dataFile);
                  String currentLine = bufferedReader.readLine();

                  while(currentLine != null) {
                        String trimmedWord = currentLine.trim();
                        categorylist.add(trimmedWord);
                        currentLine = bufferedReader.readLine();
                  }
                  bufferedReader.close();
            }
            catch (IOException e) {
                  System.err.println("A error occured reading file: " + e);
                  e.printStackTrace();
                  return false;
            }
            return true;
      }
```

**And to get a random word from the ArrayList, I also add to `HangmanFileLoader` a new method called `getcategoryWord`.**

```
	public String getcategoryWord(){

		Random CatRandom= new Random();
		int randomindex=CatRandom.nextInt(categorylist.size());
		String secretWord= categorylist.get(randomindex);

		return secretWord;
	}
```

**Other parts of `HangmanCategory` are very similar with `HangmanGame` in strategy. Finally I create a new class to play the game of `HangmanCategory`.**

```
public class HangmanCatExecuter {
	public static void main(String[] args) {
		HangmanCategory mygame = new HangmanCategory();
		mygame.playcategory();
	}
}
```