Programming Assignment 2 (Random Writer)

Atharva Pendse (aap3469)

Part 1

Introduction

The objective of this assignment is to create a ‘Random Writer’, which is a java program that predicts the next character by considering the probability with which that character appears after the preceding substring (which we shall call seed) in the given input string. This program takes a positive integer value ‘k’ which is the length of the seed, as its input. The program then considers the preceding substring of length ‘k’ and subsequently makes predictions for the character that follows.

The algorithm, when implemented correctly, with an appropriate value of k, can produce meaningful words which are actual words in the string used as the input. It has several useful applications, including the completion partial images by predicting the RGB values of the missing pixels based on the RGB values of the previous strings of k pixels in both dimensions. Another application is to Midi music files, where it can be used to predict the next unit of music based on the previous k units of music in the midi format.

My Goals

My goal was to finish the assignment and familiarize myself with the Java API, including array lists, which I had used previously, and Hash Maps, which I hadn’t. It turned out to be a fun project and I learned a lot.

Part 2:
Approach and Solution

2.1 Initial approach:

My initial approach was the one that the assignment handouts had suggested. Every time the next character was to be predicted, the code would scan the input and store all the characters immediately following the seed in an array list. Then the random number generator would be used to randomly choose a character stored in that array list.

This implementation worked; however, I found it to be too slow for large inputs. Since the code was scanning through the entire input for every character of output, the time required was proportional to:

(length of the input) x (length of the output)

Since the code was too slow and therefore impractical on larger inputs (more than 100,000 characters), I decided to use a faster approach. This approach involved using a data structure called Hash Map.

2.2 Second approach

A Hash Map stores key-value pairs and can access the stored data in constant time by virtue of the way in which it is set up to work. A Hash Map consists of an unordered set of key-value pairs. I implemented the Hash map using k-length substrings as the keys and an array list consisting of the set of letters following the particular substring as its value. For example, for k = 4, the input
"qwerty qwerty qwerty qwerty qwerty"
produces the Hash map:
hmap :

Key(string)	Value(Array List)
"qwer"	"t", "t", "t", "t", "t"
"wert"	"y", "y", "y", "y", "y"
"erty"	" ", " ", " ", " "
"rty "	"q", "q", "q", "q"
"ty q"	"w", "w", "w", "w"
"y qw"	"e", "e", "e", "e"
" qwe"	"r", "r", "r", "r"

After creating the basic hash map, the program predicts an initial random seed (discussed in Part 3.2). Next, it checks for that key in the hash map. If the seed is a valid key, then the program predicts the next character based on the array list of that seed and updates the current seed. This process repeats until we have an output string of the required length. If, however, the seed is not a valid key, the program generates a new random seed and proceeds until the output string has the required number of characters. This string is then written to the specified output file.

Part 3
Edge cases, Interesting Results and Reflection

3.1 Time efficiency of the Hash Map method:

Hash maps allow you to access any of the stored data in constant time due to the way in which they are set up to work. Therefore, we can get the array list for the current seed in constant time. The time taken for the execution of the program is the sum of time taken to set up the hash map structure with all its data and the time taken to create the output string using data stored in the hash map. The time taken to set up the hash map data structure is proportional to the number of characters in the input string. The time taken to produce the output string is proportional to the number of characters in the output string. The total time is proportional to the sum of both these times. Hence it depends on the value of
(length of input string + length of output string)
which is much faster than the array list method.

3.2 Generating Random Characters

Since we cannot randomly generate characters directly, we use the random number generation function. Suppose we have an array list ‘c ’ of 25 characters and we want to randomly pick a character from c. We first generate a random integer ‘randomint’ in the range of 0 to 24 and get the character stored in c at position ‘randomint’.

3.3 Value of K

The assignment handouts suggest that if we use a value of k in the range 5-7, the program produces meaningful words. Why is this true? The average word in the English language is in the range of 5 to 7 characters in length. If the value of k was too small, say k = 2, then the next character’s prediction would be based on only the previous two characters, and this would result in less meaningful words being formed. We can think of this is *underfitting*.

On the other hand, if the value of k is too large, say k = 15, then there will be *overfitting,* that is, the output string would be too similar to the input string. Thus, there would be no ‘randomness’ to the output string and there would be no point in executing the program at all.

3.4 Edge case k = 0

This edge case should be handled differently from the rest of the cases. If we have k = 0, the characters in the output string should appear with the same frequency as they do in the input string. Therefore, we select a random character from the input string ‘a‘ and add it to the output string ‘b’. Then we remove that character from a and proceed until the output string has the required number of characters.

Why should we remove the character from a once it is present in b? The answer is that when we already have a certain character in the output string, the probability of it appearing again in the output string decreases since it was already used once.

3.5 What I learned

While coding this assignment, I familiarized myself with the Java API. My initial code used the method specified in the assignment handouts, that is using array lists. However, I found this method to be too slow for large character inputs, so I changed my code to instead use Hash maps. In using Hash Maps, I got to learn about hash functions and why this approach is faster than the original method.

Part 4:
Testing

4.1 Testing for k >= 1

My initial intuition to check whether or not the program worked was to see if, given an input of meaningful words, the program outputted a set of meaningful words for k = 5. (Since the handout suggested that it should for a value of k such as 5, 6 or 7). Indeed, the outputted string had several meaningful words, almost all of which were part of the inputted string. This reassured me that my program was, at least partially, working alright.

The program works on the logic that the the probability of a character appearing after a particular string in the output file is equal to the probability with which that character appears after that string in the input file. Therefore, if we were to give the program a controlled input of the same string over and over again, it should give a predictable output.

This was the first testing method I used. My input file consisted of the string 'abcde ' over and over again.

"abcde "

In this input, for example, the character 'b' always appears after the character 'a' and the character 'c' always appears after the character 'b'. I ran the code with a k value of 6.

As expected, my output string was the same as the input string, except it started with a different character and had a different length. Why did it start with a different character? Because the initial seed was chosen randomly. So, the initial seed might have been "cde ab", leading to an output of

"cde abcde abcde abcde abcde abcde abcde abcde abcde abcde abcde abcde abcde abcde abcde abcde abcde abcd"

Why did it have a different length? Because I specified the length as the fourth argument to the main function.

The next input I used was

"aba bab aba bab aba bab aba bab aba bab aba bab aba bab aba bab aba bab aba bab aba bab aba bab aba bab aba bab aba bab aba bab aba bab aba bab"

and the output I got with k = 5 was
" bab aba bab aba bab aba bab aba bab aba"
which was just as expected because in the input string, the probability that b appears after a is twice the probability that b appears after a space. This probability should stay the same in the output string and indeed, it does.

The next input I tried was k = 2, length = 30, and the input string

"The quick brown fox jumped over the lazy dog."

The output it gave was
"uick brown fox jumped over the"
which I manually checked and confirmed that it was as expected.

4.2 Testing the edge case K = 0

For testing this case I used the special input:
"abcdefghijklmnopqrstuvwxyz"
I chose this string because it has all the letters in the English alphabet exactly once. When I ran the code with k = 0 and length = 26, the output I got was
"dkqrlvwpnahjgutocysziebxmf"
which was exactly what I had expected. Since each character appears exactly once in the input string, that is, equally frequently, it should appear with the same frequency in the output string. This output string also contains all the letters in the English alphabet exactly once, so it should be correct.
