CS361 Assignment 5

Due: Friday, July 7 by midnight.

The main part assignment is graded on a 10 point scale. For this assignment, it is possible to get up to 2 extra credit points. That information is at the bottom.

Submission: In order to get the first 10 points, you must submit a zip file, the name of which has to be UTEID_program5.zip (where UTEID is your UT EID). If you have a partner, please provide both your UTEIDs in the name of the zip file: UTEID1_UTEID2_program5.zip. In order to get the extra credit points, you need to submit another zip file: UTEID_program5_extra.zip file. Note these may be graded by different graders. You can't get the first 10 points based on UTEID_program5_extra.zip. Similarly, you can't get extra credit points based on UTEID_program5.zip. We'll be using a program to collect everyone's submission file. If you use an incorrect naming convention, you will have to go to the TA's office hours to handle this issue.

Partner/Team: If you're working in a team, ONLY one of the student in the team needs to submit the zip file using his/her account.

Grading README file: Your zip file must contain a README.txt file. Here's the README.txt file for this assignment. (Make sure to use this new one.) We're also using a program to grade your README.txt file. Any other names or formats of REAMDE file won't be accepted, e.g. readme.txt, readme, README, README.c, README.java. The README.txt file worths 3 (of the 10 points). If you use the incorrect REAMDE.txt file, you have to go to TA's office hour to handle this issue.

For the README.txt file, the maximum point is 3.

1. First Five Lines [0.25 point] have to match the following format

   UTEID: jd1234; bp5678;
   FIRSTNAME: Johnny; Brad;
   LASTNAME: Deep; Pitt;
   CSACCOUNT: johnyd; bradp;
   EMAIL: johnyd@cs.utexas.edu; bradp@cs.utexas.edu;

   Please don't remove "UTEID". Please don't remove ":". Also please don't remove ";".

2. Good description of your code [0.5 point]

   You have to write some sentences (at least 100 words) to explain high-level idea of program, some short description of the key/important function in your code.

3. You need to tell us how much you've finished [0.25 point]

   (1) Finished all requirements (2) Only finished parts of the requirements.

   If you're in the case (2), you have to list the unfinished things and write down the reasons for each, either no time or have some bugs. If grader thought you explanation is very good and detailed, you won't lose point even if you didn't finish many requirements of program.

4. Source of reference file [0.5 point]

   You may not upload your reference file if it is too big(>2MB). But you need to explain ``how" and ``where" do you find/create it. The file name should be reference.txt in case of you upload it. In README.txt, you need to write the number of lines and the number of English words in that reference.txt file.

5. Create enough of your own test cases [1 point]

   First, you have to write down one sentence to claim how many test cases have you created. If you have >=4 test cases, you will get 1 point. Second, you still need to attach all your own test cases and the output of that at the end of README.txt file. Third, you don't need to create some crazy test cases, for this assignment, choose some k < 50 and N < 20 is OK.

The grading scheme of this README.txt file has very little to do with the correctness of your program. Even if your program doesn't work well, please try your best to provide the details of the README.txt file, you still have a chance to get 3 out of 3 points.

Grading Program: For the program, the maximum point is 7.

1. You will get [1 points] If the program can be compiled. Before submission, make sure that your code can be compiled and executed on the Linux Machine (in our CS Lab). Even if your code is working on your own machine (Windows, ...), but not working on CS Lab machine, you won't get any points!

2. You will get [3 points] If the program also can be run.

3. The remaining 4 points come from our test cases. We will run 4 test cases (similar to the example that provided later in this page) and check the output of your program. If the output is reasonable, we treat it as correct. You will get 1 point for each correct output (4 points in total).

Deliverables: You will be submitting your code electronically. Be sure that you clearly identify the members of your team (one or two people). Include a README file so that we will understand clearly how to compile and run your code. For this assignment:

1. The primary file name should be `Passwords.java`. Students are welcome to organize their assignments into multiple files and must submit those files as well. However, the main method must be in `Passwords.java` and we should be able to compile your program with `javac *.java`.

2. The program should be executed via the command: `java Passwords reference-filename N k`

Background

One of the worst things about randomly generated passwords is that they are too hard to remember. As a result, users write them down and that introduces a significant security vulnerability. However, if users are allowed to choose passwords, they tend to choose strings that have too much semantic content. These are then susceptible to a dictionary attack. One approach to dealing with this is to generate pronounceable strings that have no semantic content, which is your task in this program.

One way to do that is to use a *Markov process*. You can think of a Markov process as a state machine where the next transition is a probabilistic function of the current state. You can also think of it as a model of the *nth*-order entropy of a language. For example, suppose your password generator program is producing a k-character password, one character at a time. If the current character is a T, the next character could be a vowel, H or R. It probably would not be a B or Q. The probabilities governing this choice somehow relate to the probabilities corresponding to the second-order entropy of English text in general (or of a specific English text).

If you want to see some excellent recent work on password cracking that seems to defeat this type of Markov modeling, see: Narayan and Shmatikov.

Your Assignment

First, compute a ``followers'' table. Take a file containing an arbitrary but fairly large English language ``reference'' text in ASCII. You might use a text version of a paper you've written or something that you grab from a website. (Note: if you use too short a reference text it doesn't work well because you may have letters that have no followers and your computation will break down.) Compute a 26 x 26 table with the frequency with which each letter follows each other letter in the text. There's an example below. Ignore non-letters, case, and white space. However, don't count the first letter of a word as following the last letter of the previous word. Also, maintain two other one-dimensional arrays: COUNTS contains the total counts of each letter (minus the ending letters of words since you didn't record any follower for them), and STARTERS containing the count of the number of times each letter begins a word.

Eliminating following between words reduces the problem of unpronounceable combinations. However, there is still a problem of inter-syllable following. You don't need to fix it, but perhaps consider how you might.

The name of the file containing your reference text should be input from the command line. I.e., we should be able to run your program on our own reference text and get the letter counts expected. Your program should print out a nice tabular representation of your followers table. Below is an example of the first two rows of such a display.

```
         A     B     C     D     E     F     G     H     I     J     K     L     M     N     O     P     Q     R     S     T     U     V     W     X     Y     Z
A:      13   161   411    90     2    40   131     9    48     2    30   424   177   627     1   130     0   353   289   723    41    84    12     4    42     0
B:      25     8     1     0   233     0     0     0    51     2     0    91     6     1    90     0     0    21   102     7    42     1     0     0    51     0
....
```

Second, using these tables write a routine to generate N k-character passwords where each successive character in the password is computed as a Markov process from the previous character. That is, the ith character is selected probabilistically, based on how likely it is to follow whatever you selected for character (i-1). Thus, if the previous character was a B, it's much more likely that the next character chosen will be an E than a D.

The first character in each password should be generated probabilistically from the frequencies of the initial letters of words in the text (as stored in the STARTERS array). Thus, more passwords will start with T than with Q. (I don't know why my example below seems a bit odd in that respect.)

How do you compute the probabilities? Imagine that you had a map divided into regions of various sizes, and that you throw a dart to select a point *randomly* on the map. The probability of the point falling within a given region should be proportional to the area of the region. You can use a similar idea here. For each letter L1, the number of times that it follows another letter L2 in your reference text is a fixed proportion of the total times that letter L1 appears in the text. You can use this fact to implement your Markov process.

Notice that the ith row in the followers table represents the relative frequency of the letters that follow the ith character in your reference text. Consider the two row of the followers table displayed above. For the reference text from which this table was generated, there are 13 places in the text where A follows A, 161 where B follows A, etc. We find that there are a total of 3844 A's in the text (discounting the ending letters of words). So, if the current letter is an A, we should follow it with an A, 13/3844 of the time; with a B, 161/3844 of the time, etc. (I think my table was somewhat skewed because I didn't eliminate inter-word following when I generated it, but you should do so.)

Suppose we have generated an A and now are trying to select the next letter in our password. Then to choose that next letter, select an integer randomly in the range [0..3843]. Let's say we select 1023. Accumulating totals down the A row, we see that this number corresponds to an L. That is, L is the first character for which the cumulative count exceeds 1023.

You can think of this process as dividing the unit interval [0..1] into 26 regions with the size of each region being the relative frequency of the corresponding character. A random number in the range [0..1] selects the corresponding character, with probability depending on the percentage of the interval devoted to that character.

To select the starting character for each password, choose an initial letter based on the relative frequency of starting letters of words in your reference text as stored in your STARTERS array following an analogous process. That is, your passwords will tend to start with letters that begin words in English (or at least in your reference text).

Your program should be able to generate an arbitrary number of passwords of arbitrary fixed-size. The resulting passwords should generally be pronounceable. For example, a given run of my implementation (when asked for 15 9-character passwords) generated the following:

```
Passwords are:
 dintelaug
 tominosed
 rorecrdil
 tobecallo
 zanderong
 vemndfmse
 nsolyprea
 ncungecin
 rmormoron
 nchaniona
 ftimiront
 cbapedtro
 alsasttcr
```

```
nandecoft
ondexanth
```

However, my version didn't ignore interword following, so these aren't quite as pronounceable as they should be.

Suggestion: Instrument your program so that the selection of the random number will be unpredictable for a "production run" but will follow a reproducible sequence (given a supplied seed) for a testing run. This will make it possible for you to reliably generate the same set of passwords over and over so that you can better debug your program. This is easy to do using the Java Random class with a supplied seed.

Extra Credit

The above is all that you must do to complete the assignment. However, there are two extra steps you can complete for up to two extra points (1 for each part). In your README file, describe what you did to improve your program in either of the following two ways.

1. This password generator takes into account the second-order entropy of English. Take into account the third-order entropy. That means maintaining a three dimensional table (or the equivalent) and taking into account the likelihood of a given character following the previous digram.

2. It may happen by chance that your password is or contains an English word, rendering it more susceptible to a dictionary attack. Many password generators preclude this possibility. Detect and flag any passwords that contain an English word of at least four characters. Unix systems have a dictionary of some 45,000 words that you can use for this purpose (try /usr/share/dict/words), but allow the capability to add in an arbitrary additional file of words containing, for example, your first name, your last name, etc. Display any passwords that you reject along with the substring that led to the rejection.