EECS 233 Introduction to Algorithms and Data Structures
Fall 2014, Programming Assignment 3 ("P3")

Due Date: Tue Nov 18 by midnight.

Drafts: We will run our full test suite once a day until the final due date on any drafts submitted via blackboard. This allows you to receive feedback even if you have only implemented some of the methods. To run our test suite, your code needs to compile against the fundamental JUnit tests provided with the assignment. You should write blank skeleton methods for the parts you haven't yet implemented.

Compilation and correctness are 50% of the grade (see the Grading section below), which you will loose if your code doesn't compile. To avoid this, *it is important to submit your drafts sufficiently early, i.e. at least a couple days before the deadline, so you can get feedback in time to correct any errors*. In the event that your code does not compile, we will use your last working draft submission.

You can speed your implementation time and avoid compilation and test errors by writing your own JUnit tests to check the correctness of your code and edge cases as you implement each method.

Total Implementation Points: 107 (60% of total) + 20 points extra credit

In this assignment you will write a program that reads a text document as input and builds a data structure to efficiently calculate word occurrence and co-location statistics. These statistics form the basis for many for a wide variety of algorithms in areas ranging from spelling correction to natural language processing.

The general steps are:

- read the text file, extract and normalize the words
- compute word and word pair statistics
- implement functionality for a variety of queries:
  - word counts and rank of specific words or word pairs
  - k most (or least) common words or pairs
  - conditional word count given a preceding or following word (collocation)

Notes: Your implementation should also account for word pairs on separate lines, i.e. the last word of line n and the first word of line $n+1$ is also a valid word pair. You do not have to handle hyphenation. Most texts from Project Gutenberg do not have hyphenated words that span different lines, so this not an issue. However, hyphenated words like "grown-up" will be counted as the single word "grownup" by virtue of the word normalization, one step of which is removing the punctuation. In the specifications below, all the methods should be public unless otherwise noted.

1. Tokenizer

Write a class `Tokenizer` that provides the functionality to read lines of a text file, extract and normalize the words, and store them using Java's `ArrayList` class composed of `String` elements. You can use Java's `BufferedReader` or `Scanner` classes to read text files and the `ArrayList` class to store the word list. Refer to Java's regular expression documentation for the whitespace and punctuation character classes.

- ▷ A constructor with a `String` argument specifies the file from which to obtain the words. 4 P.
- ▷ A constructor with a `String[]` obtains the words directly from the `String` array. No file is read. 2 P.
- ▷ `ArrayList<String> wordList()` returns the word list created from the constructors. 2 P.
- ▷ Split lines from the text file into words using whitespace characters. You can use Java's `split` method in the `String` class. 3 P.
- Normalize words for both constructors as follows:
  - ▷ Convert words to lower case. 2 P.
  - ▷ Remove leading and trailing whitespace. 2 P.
  - ▷ Remove punctuation. 2 P.

2. HashEntry

Define a class `HashEntry` which you will use to define `HashTable` class below.

- ▷ The constructor `HashEntry(String key, int value)` should initialize a `HashEntry` with `key` and `value`. 2 P.
- ▷ Provide accessor methods `String getKey()` and `int getValue()` for the `key` and `value` fields. 4 P.
- ▷ `void setValue(int value)` should set or update the `value` field. 2 P.

3. HashTable

Write a class `HashTable` for String keys. You will use Java's built-in `hashCode()` function to compute hash codes for strings, but you should not use Java's built-in HashTable class to implement this class. The `hashCode` function can return negative numbers, so you also take the absolute value before modding it by the table size. The hash table entries should be defined using the `HashEntry` class above.

- ▷ A `HashTable` constructor with no arguments should initialize a table of a reasonable default size. 2 P.
- ▷ A `HashTable` constructor with an integer argument initializes the table to the specified size. 2 P.
- ▷ `void put(String key, int value)` should store the key-value pair in the hash table. In addition: 4 P.
  - ▷ It should handle collisions. You may use either separate chaining or a probing strategy. 4 P.
  - – It should resize the table when necessary. 4 P.
- ▷ `void put(String key, int value, int hashCode)` (note the different signature) should have the same functionality as `put` above, but use the provided hash code, which you should assume comes from a different `hashCode` function other than Java's. This is to allow for direct testing of collisions. 2 P.
- ▷ `void update(String key, int value)` should update `value` associated with `key` in the hash table. If `key` does not exist, it should be added to the table. 4 P.
- ▷ `int get(String key)` returns current the `value` associated with `key` if `key` exists and -1 if it does not. 4 P.
- ▷ `int get(String key, int hashCode)` should have the same functionality as `get` above, but use the provided hash code (which should then be modded by the table size). This is to allow for direct testing of collisions. 2 P.

4. WordStat

Here you will write a `WordStat` class to compute various words statistics. You should design the class and use your classes above so that the statistics are computed once upon construction for a given source, rather than for each query.

- a constructor with a `String` argument computes the statistics from the file name. 2 P.
- a constructor with a `String` array argument computes the statistics from the words in the `String` array. 2 P.
- `int wordCount(String word)` returns the count of the `word` argument, which you should normalized before returning the count. Return zero if the word is not in the table. 3 P.
- `int wordRank(String word)` returns the rank of `word`, where rank 1 is the most common word. Also take into account the following 3 P.
  - – Normalize the word before returning its rank. 1 P.
  - – Words with the same count should return the same rank. 2 P.
  - – The rank should account for duplicate ranks. For example, if there are only three words w1, w2, w3 with counts 20, 20, 10, their rank would be 1, 1, 3 (not 1, 1, 2). 2 P.
- `String [] mostCommonWords(int k)` returns a `String` array of the `k` most common words in descending order of their count. For words with the same rank, the method should return them in alphabetical order. For example, if a text has (only) the words dog, cat, bat, ape, and eel with counts 20, 10, 20, 20, and 10, `mostCommonwords(2)` would return `{"ape", "bat"}`. 3 P. 2 P.

- `String [] leastCommonWords(int k)` returns a `String` array of the `k` least common words in increasing order of their count. 3 P.
- `int wordPairCount(String w1, String w2)` returns the count of the word pair `w1 w2`. 3 P.
- `int wordPairRank(String w1, String w2)` returns the rank of the word pair `w1 w2`. 3 P.
- `String [] mostCommonWordPairs(int k)` returns the `k` most common the word pair in a `String` array, where each element is in the form `"word1 word2"`, i.e. separated by a single space. 5 P.
- `String [] mostCommonCollocs(int k, String baseWord, int i)` returns the `k` most common words at a given relative position `i` to `baseWord`. These are called "collocations." The relative position can be either `+1` or `-1` to indicate words following or preceding the base word. For example, 5 P.

  ```
  mostCommonCollocs(10, "crash", -1)
  ```

  would return the 10 most common words that precede "crash." You do not need to implement relative positions other than +1 or -1.
- (Extra credit) `String [] mostCommonCollocsExc(int k, String baseWord, int i, String [] exclusions)` has the same functionality as `mostCommonCollocs` except that it excludes from consideration any words in the `String` array `exclusions`. This provides a means to obtain collocations that exclude common word pairs such as "of the" or "in a". +10 P.
- (Extra credit) `String generateWordString(int k, String startWord)` returns a string composed of `k` words of the form `startWord` w_2 w_3 ... w_k. The string is generated by finding w_2, the most common collocation of the `startWord`, then w_3, the most common collocation of w_2, and so on. Each word should be separated by a single space. +10 P.
- Go online and find some electronic text that interests you (Project Gutenberg is a good place to start). Use your text to illustrate your methods. Describe your results in the `examples.txt` using the output of your code and include this file with your submission. Feel free to use more than one example text. Be sure to also illustrate the extra credit functions if you have done them. 15 P.

5. Grading

Your grade will consist of the following:

compilation	20%
correctness	30%
effort	30%
testing	10%
design	5%
style	5%
Total	100%

a) *Compilation* (20%) Does it compile or not? This is given a weight of 20% for two reasons: 1) If your code doesn't compile, we can't check its correctness. Non-compiling code implies zero points for correctness.[1] 2) Debugging compilation errors can be difficult and time consuming, but it is a fundamental aspect of programming. To avoid this, develop incrementally and compile often to make sure your code remains in a working state.

b) *Correctness* (30%) We will check correctness automatically for each requirement with our JUnit test suite using the assignment points as listed above. This is usually pass/fail, but we will give partial credit where feasible. We will provide you with a basic JUnit test suite that checks fundamental functionality and that your code has the correct method names and argument types. Note that even simple formatting errors can cause test failures, so follow the specifications carefully. If we cannot run the tests because your code does not finish in a reasonable amount of time, i.e. you have an infinite loop or inefficient algorithm, you will not receive a correctness grade.

[1] This also implies that simply submitting blank methods that compile (against the whole test suite) will earn you 20%, which is better than 0%, but still not advisable.

c) *Effort* (30%) The total points listed at the top of the assignment constitute 60% of the total grade: 30% is correctness; the other 30% is effort. We endeavor to design the assignments and JUnit tests so that the requirements can be checked independently, but it can still happen that a small bug or design error leads to a cascade of test suite failures. It is not feasible for us to debug your code and assign credit according to the nature of your error(s). Instead, we use a simple approach to give you credit for effort: We inspect your code pertaining to each required method and decide (yes or no) if it is reasonably complete. Passing a particular requirement automatically give you full credit for effort. Thus, even if you fail certain tests, you can still get half the points if it looks like you made a reasonable effort to implement the functionality. Code that is incomplete or obviously wrong will receive zero credit for those methods.

d) *Testing* (10%) Scored on a scale of 0 to 10. Your code should include JUnit tests to check that the code is correct and performs as specified. They should be well thought-out with clear reasoning behind them, which you should explain in your comments. Having a large number of tests that are essentially equivalent will receive deductions. Remember that the goal of testing is to make the process of implementation and debugging faster and more efficient. So don't waste your time writing a bunch of unnecessary tests. Here are some guidelines from the *Practice of Programming*:

- Test as you write the code. This goes along with writing incrementally. Once you implement a part of the assignment, write a test to check its correctness.
- Test code at its boundaries. This is also referred to as testing "corner cases." Do conditions branch in the right way? Do loops execute the correct number of times? Does it work for an empty input? A single input? An exactly full array? Etc.
- Test systematically. Does your test code check every condition? Does every line get executed? Especially test parts you're going to reuse. Otherwise, debugging can be much more difficult.
- Test automatically. Have test suite that automatically runs all the tests. JUnit helps provides this functionality. This helps identify errors as soon as they are introduced, which would otherwise can be very difficult to debug.
- Stress test. Does your code work on large inputs? Random inputs? It's easy to make implicit assumptions that aren't always valid. Stress testing can help uncover these.

e) *Design* (5%) Scored on a scale of 0 to 5. Design and implementation[2] considers how you chose to structure your program to implement the specification. It also concerns how you structure the code within a method and your choice of data structures and algorithms. For grading we will consider things like, how you chose to break down a problem into smaller components, how elegantly your code handles edge cases, etc.

f) *Style* (5%) Scored on a scale of 0 to 5. Style considerations include (from *Practice of Programming*):

- Variable name choices. Are they descriptive and do they convey information about their purpose? Note for simple concepts, it perfectly acceptable, even preferable to use, for example, `i` for an index or `n` for an integer.
- Expressions and statements. Are they clear? Do they make the meaning as transparent as possible? Well-written code should not need much commenting, if any. Is it formatted in a way that aides readability?
- Consistency and Idiom. Is formatting consistent? Are common operations consistent with programming idiom? (the textbook, for example)
- Comments. Do they aid the reader in understanding the code? Comments that restate what is already clear the code are redundant and not helpful. Nor are comments that are not consistent with the code. For more complex methods, the comment should state the running time of the method (or running times if there are different cases).

[2]The design, style, and testing considerations follow the guidelines and ideas in *The Practice of Programming* by Kernighan and Pike.

General Instructions and Guidelines

Turning in your code. You should submit a zipped folder of your java source and JUnit tests files via Blackboard. Follow these instructions:

- Make a separate submission folder with the following name: `EECS233_abc123_P3` where `abc123` is replaced by your own Case Network ID.
- The folder should contain your java source code, your JUnit tests, and any additional files (e.g. if you have a README.txt).
- Your JUnit filenames should have the form `ClassNameTest.java`.
- Do *not* include the java `.class` files. They take up too much space, and we will compile your files ourselves.
- Your submission folder should not contain subfolders.
- Compress your submission folder using the `.zip` format (do not use `.rar` or some other format)
- The compressed filename should be the same as the folder name, but with the `.zip` extension.
- Submit this file via Blackboard.

Drafts. Draft submissions allow you to check your code for correctness against our own suite of JUnit tests. This will be the same test suite used for the final version.[3] It is in your best interest that your draft be as complete possible. This will allow you to benefit from feedback from the full test suite and give you time to identify problems, fix compilation errors, or get help. We cannot provide feedback at the last minute or allow you to fix compilation problems after the deadline that could have been avoided by starting earlier.

- **Fundamental items**. Assignment items marked with "▷" are *fundamental*. These should be simple and straightforward to implement and represent the minimal functionality we need to test your code. We will provide you with our JUnit tests for these items, so that you can check the correctness of your code yourself. Needless to say, do not modify them, but if you find errors or potential issues, please bring them to our attention as soon as possible. You can use these JUnit tests as templates to write your own test suite for the rest of the assignment, which you should do in separate files, one for each class.
- **Use exact method names**. Use the exact method names specified, including the character case. This is required to check your code. We cannot make modifications to your code when grading.
- **Do not declare your source as a package**. This isn't necessary and it prevents compilation with the grading test suite.
- **Complete skeleton methods are required**. In order for the JUnit tests to compile, you need to write skeletons for *all* the required methods, even extra credit items. These can be empty, i.e. without any code, but they must have properly typed arguments and return values. If you do not this, we will not be able to test your code.
- **Ensure your code compiles**. You must ensure that your code is in a stable state and compiles before submitting your code. Our test suite script will automatically using the latest submitted version. We will only run the test suites on new submitted drafts once a day, so do not ask for your code to be regradedÊunless there is a specific issue you are resolving with the TA. You should be writing your own JUnit test suites as you code to check correctness.
- **Throw unchecked exceptions**. Java has two types of exceptions: checked and unchecked. Unless otherwise specified, you should throw unchecked exceptions, which means you do not put a "throws" statement in your method declaration. The assignment will specify which type of exception you should throw.

[3] Although we reserve the right to correct errors or shortcomings in the test suites when we find them. If this happens, we will try to be fair, so that students are not led to have a false sense of completion.

Version Control. You should always write your code incrementally, with a short write-test-debug cycle. Once you have a component written (which could be just a single method), write JUnit tests that check that your implementation is correct. Since we cannot grade what we cannot compile, you need to use some kind of version control to ensure that you have a saved working version once each step is completed. This could be as simple as copying the directory or using a sophisticated version control system like "git". Anything is fine, as long as you always maintain a working copy of your code that you can turn in (or revert to if you introduce a bug that you can't track down).

Use an Automated Backup System. Everyone should be using a backup system that backs up your files automatically and seamlessly, such as Dropbox or Google Drive, among many others. They are free services for small space requirements, so there is no excuse anymore for loosing your work to a hard drive failures or accidentally deleted files. If you haven't done so already, set up a such a system now.

Final Version. As for your drafts, make sure your code compiles and is in a stable state (i.e. your best working version) before the deadline.

General Strategies. It is helpful to work through the logic of the implementation for the problems in the assignment on a piece of paper, *before* you write any code. This can include things like data structure design, implementation, and small examples you use to work through the steps of your algorithms. You do not need to turn this in, but it can be helpful for grading in you include a README.txt file or a pdf scan of your notes that explains your design choices to the grader. Your code comments should describe clearly your high-level logic, design choices, and the running time and/or memory complexity of your algorithms.

Cheating and Plagiarism. It should go without saying that you should solve the problems and write the code on your own. You can consult the book and the internet for java questions, but do not look for solutions to the problems given in the assignments. You are encouraged to get help from your classmates regarding basic programming issues, debugging problems, features of Eclipse, etc., but like I mentioned in class, be careful about giving (or receiving) too much help. Not only might it spoil someone's learning efforts, it may appear as plagiarism. To ensure fairness, we will run software to check for cheating, so please do not copy code. Also keep in mind that your goal should be to develop the ability to think through problems, find solutions, and implement them in code. Some exam questions will be designed to test to what extent you have achieved this using examples drawn from the assignments.