COMP 3270 FALL 2019

Programming Project: Autocomplete

Due before 11:59 pm Friday November 15

NO EXTENSIONS AND NO LATE SUBMISSIONS

Project submission is mandatory. If you do not, you will fail the course regardless of your numerical grade in other assignments and exams.

Overview

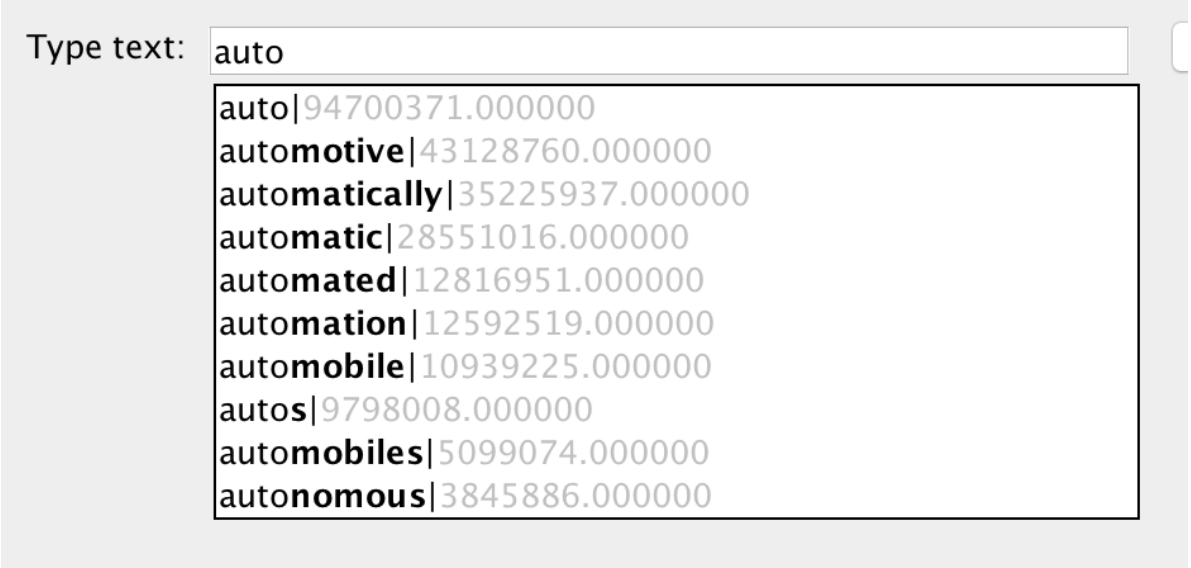
In this assignment, you will write a program to implement *autocomplete* for a given set of N terms, where a term is a query string and an associated nonnegative weight. That is, given a prefix, find all queries that start with the given prefix, in descending order of weight.

In this assignment, you will write a class, *TrieAutocomplete* to implement Autcomplete functionality using a *Trie* data structure.

The code attached with this assignment will include the following already completed classes:

- *Autocompletor* – the interface for which you will be writing implementations.
- *AutocompleteMain* – the class where you will launch the GUI. When testing your implementations of autocomplete, you will need to change the String AUTOCOMPLETOR_CLASS_NAME's value.
- *AutocompletorBenchmark* – once you have written an implementation of *Autocompletor*, this class will tell you how quickly it runs.
- *BruteAutocomplete* - A completed implementation of *Autocompletor*.
- *Node* – a node class for use in *TrieAutocomplete*. Nothing needs to be modified here, but definitely read through this class before starting *TrieAutocomplete*.
- *AutocompleteGUI* – the GUI for this project. You can ignore this class.
- *Term* – the class serves the following purposes:
    - The basic purpose is to encapsulate a term-weight pair.
    - More importantly, Term allows us to use Comparable and Comparator to sort terms in a variety of ways, which will make *Autocompletor* implementations much simpler.

Try setting AUTOCOMPLETOR_CLASS_NAME to BRUTE_AUTOCOMPLETE in *AutocompleteMain* and running it. If you load words-333333.txt and type in "auto" you should get the following result:



You are responsible for completing the following classes:

- *TrieAutocomplete* – an implementation of *Autocompletor* which performs the autocomplete algorithm using trie exploration.

In addition, after completing these classes, you will use following methods on your *Autocompletor* implementations:

- *AutocompleteBenchmark* to analyze the efficiency.

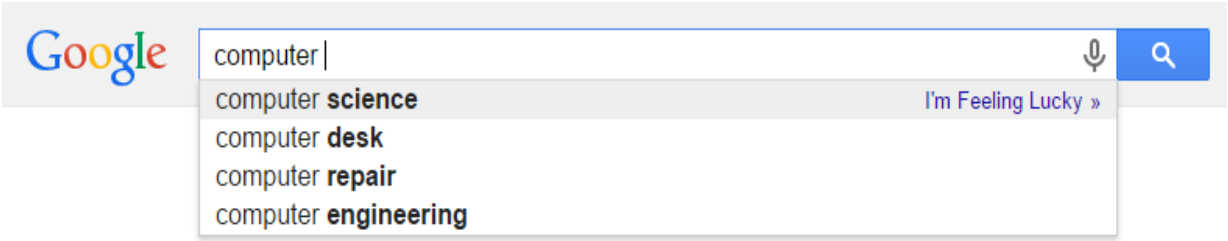
The incomplete methods come with headers detailing what the inputs and expected output are, as well as implementation details such as edge cases and exceptions to be thrown.

Acknowledgements

The assignment was developed and assigned for the Data Structures and Algorithms course at Princeton University and Duke University. This assignment has been adapted and updated to tailor the requirement for this course.

Introduction

Autocomplete is an algorithm used in many modern software applications. In these applications, the user types the text and the application suggests possible completions for that text:



While finding terms that contain a query is trivial, these applications need some way to select only the most useful terms to display (since users will likely not search through thousands of terms, nor will obscure terms like "antidisestablishmentarianism" be useful to most users). Thus, autocomplete algorithms need a way to find terms that start with or contain the prefix and a way of determining how likely each one is to be useful to the user and filtering out only the most useful ones.

According to one study, efficient autocomplete algorithms must do all this in at most 50 milliseconds. The user will be inputting the next keystroke, if it takes longer than that (while humans do not on average input one keystroke every 50 milliseconds, additional time is required for server communication, input delay, and other processes). Furthermore, the server must be able to run this computation for every keystroke, for every user.

In this assignment, you will be implementing autocomplete using several different algorithms and test which ones are faster in certain scenarios. The autocomplete algorithm for this assignment will be different than the industrial examples described above in two ways:

1. Each term will have a predetermined, constant weight/likelihood, whereas actual autocomplete algorithms might change a term's likelihood based on previous searches.
2. We will only consider terms which start with the user query, whereas actual autocomplete algorithms (such as the web browser example above) might consider terms which contain but do not start with the query.

Introduction to Tries

This section will cover the basics of Tries and their functionality.

What is a Trie?

A trie is simply a special version of a tree. In some trees, each node has a defined number of children, and a fixed way to refer to each of them. For example, in a binary tree, each node has two children (each of which could be null), and we refer to these as a left and a right child. Tries are different in that each node can have an arbitrary number of children, and rather than having named pointers to children, the pointers are the values in a key-value map.

So, while a node in a Java binary tree might be defined as

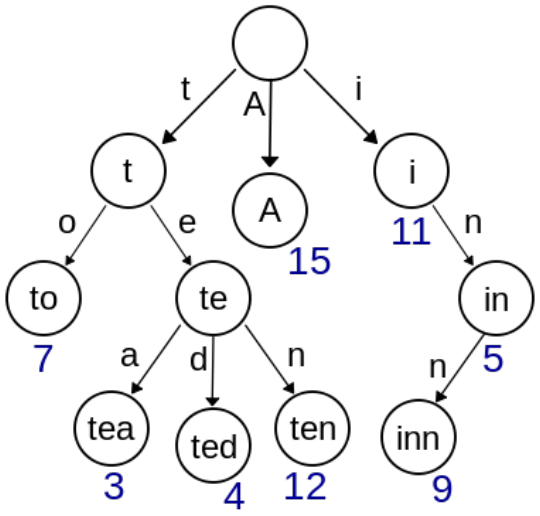
```
public class Node {
   Node myLeft, myRight;
}
```

A node in a Java trie might look like

```
public class Node {
   Map<Character, Node> children;
}
```

(Note that the Node class given to you has much more information than this)

The keys of the map will also correspond to some value. For example, for a trie that stores many Strings (as we wish to, in this assignment), the keys will be characters. In order to reach the node representing a word in a trie, we simply follow the series of pointers corresponding to the characters in the String. Below is a drawing of a sample word trie:



From the Wikipedia article on Tries

The top node is the root. It has three children, and to get to these children we have to use the keys t, A, and i. The word each node represents is the concatenation of the keys of pointers you have to take from the root to get to that node. So, to get to "tea" from the root, we have to follow the root's *t* pointer, then the *e* pointer, then the *a* pointer.

More generally, to get to a node representing a word in the String str, given a root pointer we might use the following code loop:

```
Node curr = root;
for (int k = 0; k < str.length(); k++) {
   curr = curr.children.get( str.charAt(k));
}
```

Trie Functionality and Utility

In creating a trie, we will have to add values to it. Adding a value to a trie is very similar to navigating to it. To add a value, simply try navigating to that value, but anytime a node on the path is missing, create that node yourself. The code for adding the word in str to a trie might look like this:

```
Node curr = root;
for (int k = 0; k < str.length(); k++) {
    if (!curr.children.containsKey(str.charAt(k))) {
        curr.children.put(str.charAt(i), new Node());
    }
    curr = curr.children.get(str.charAt(i));
}
```

(Again, please note that the Node class given to you is more detailed and this code alone is not a solution to this assignment)

We use tries because they have some useful properties, which we will take advantage of in this project:

- The time it takes to find an entry in a trie is independent of how many entries are in that trie - more specifically, to navigate to a node corresponding to a word of length *w* in a trie that stores *n* words takes O(*w*) time as opposed to O(f(*n*)) for some f(*n*). This improvement is possible because every navigation to the same word passes through the same set of nodes, and thus takes the same time regardless of what other nodes exist.
- All words represented in the subtrie rooted at the node representing some word start with that word. That is, the node representing "apple" will always be below the node representing "app" or more generally, every node below the node representing "app" represents a word starting with "app". For this reason, tries are sometimes called prefix trees. Given that the autocomplete algorithm is searching for words starting with a given prefix, this structure proves very useful.

TrieAutocomplete Overview/Methods

TrieAutocomplete implements the *Autocompletor* interface, which means it should, given a list of terms and weights for those terms, be able to find the top match(es) for a prefix amongst those terms. To do this, you will write methods to construct a trie, and then use the trie structure to quickly filter out all terms starting with a given prefix, and then find the highest weighted terms.

Within this class, you should:

1. Write the trie method add
2. Write the interface-required method topMatch
3. Write the interface-required method topMatches

The Node Class

For this entire section, it may be useful to have the *Node* class given to you open for reading. In addition, you should be comfortable with the basic concepts of a trie at this point.

The Node class comes with several class variables for your use in completing *TrieAutocomplete*:

- *isWord* - Set to true if the current node corresponds to a word in the set of words represented by this trie. We need this because in creating the nodes representing words, we create nodes representing words that do not exist (e.g. in creating a node for "apple" we will create a node for "appl"). Thus, we need some way to distinguish between intermediary nodes (nodes between the root and word nodes that don't represent words) and word nodes.
- *myWord* - A convenience variable, which contains the word this node represents. Should be null if isWord is false
- *myInfo* - A convenience variable, which contains the character this node corresponds to
- *myWeight* - The weight of the word this node corresponds to. -1 if this node does not correspond to a word.
- *mySubtreeMaxWeight* - The maximum weight of a word in the subtree rooted at this node (the subtree includes the root node). Useful for navigating quickly to high-weight nodes. Tracking this extra piece of information will heavily speed up our autocomplete algorithm.
- *children* - The map from characters to the corresponding children nodes
- *parent* - A pointer to the parent of this node

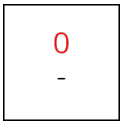
Add

Look at the constructor for *TrieAutocomplete*. It initializes the trie's root, and then calls the void add method on every word-weight pair in the arguments for the constructor. Your task is to write the add method such that it constructs the trie correctly. That is, if you write the method correctly, then every word-weight pair should be represented as a trie, and the descriptions of the class variables for Nodes listed above should be true for every node. More specifically, when add is called on a word-weight pair, you should:

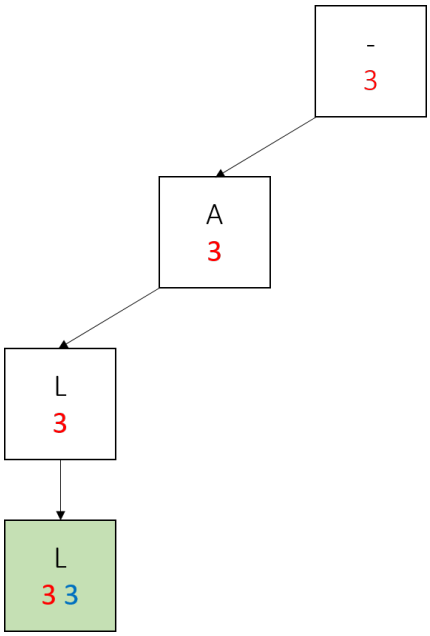
1. Create the node representing that word and any intermediary nodes if they do not already exist
2. Set the values of *myWord*, *myInfo*, *isWord*, and *myWeight* for all word nodes
3. Set the values of *mySubtreeMaxWeight* for all nodes between and including the root and the word node

To help you understand what all your add method should be doing, here's an example of a series of adds. In the trie below, each node's *mySubtreeMaxWeight* is red, the key to that node is the letter inside of it, and if the node represents a word, it has a green background and its weight is in blue.

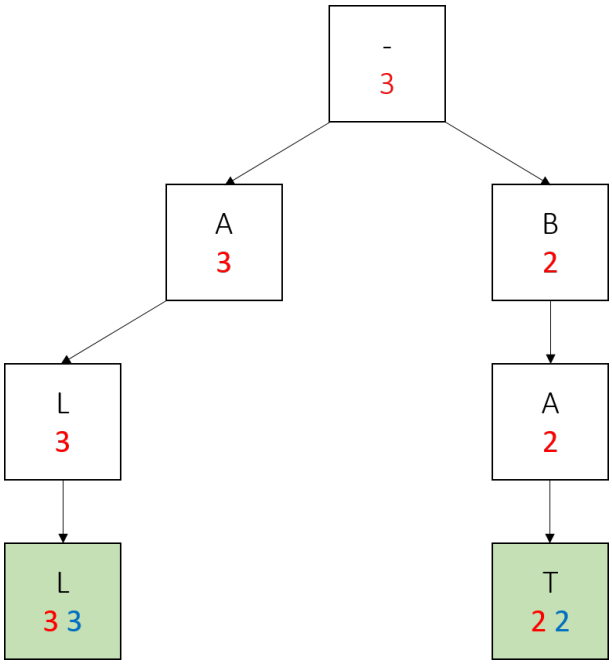
We start with just a root:



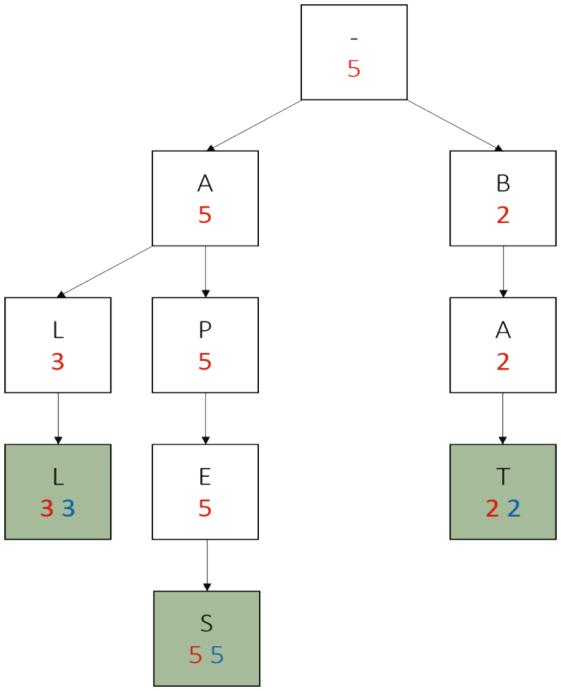
We call add(“all”, 3). Notice how since the only word at this point has weight 3, all nodes have mySubtreeMaxWeight 3.



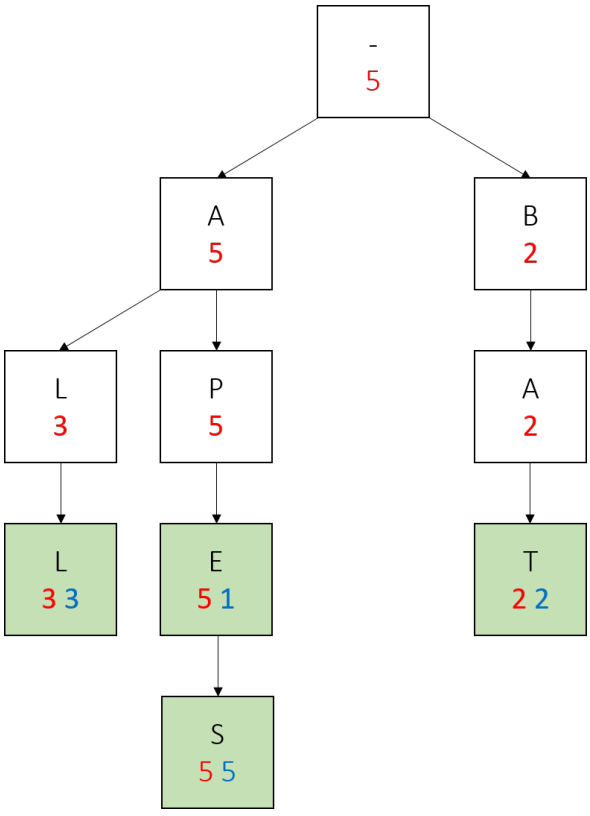
We call add(“bat”, 2). Only the nodes in the right subtree have mySubtreeMaxWeight 2, because the largest weight is still 3.



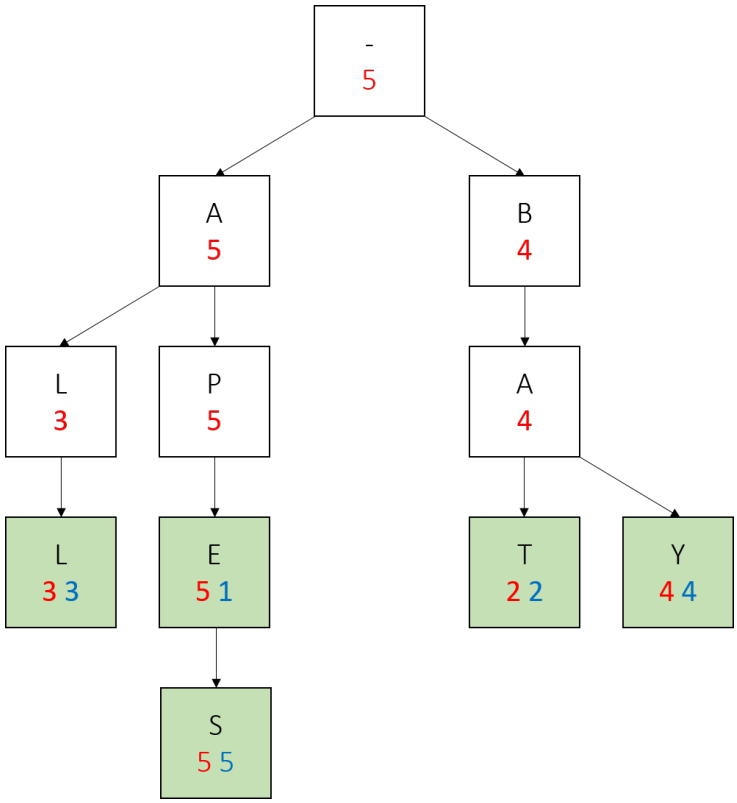
We call add(“apes”, 5). This time, we don’t create a node for the first letter. We update the mySubtreeMaxWeight of the root and the “a” node.



We call add(“ape”, 1). Notice how we create no new nodes, simply modify the values of an existing node.



Lastly we call add(“bay”, 4). Notice the changes in red numbers, and that we only created one new node here.



Also note how in the final tree, any word which has no larger-weight words below it (every word but apes) has the same red and blue values. This is true of any trie we construct in this project, if the parameters of nodes are updated correctly - a node with the same myWeight and mySubtreeMaxWeight has no children with a larger weight.

Most importantly, notice how on the path from a node representing a prefix to the largest weighted word in the subtree rooted at that node, all the red values are the same, and this value is also the weight of the largest weighted word. We will take advantage of this fact when writing topMatch().

This example of constructing a trie using add is a more simple one - we never added a word below an existing word, or a word which already exists. Your add should be able to handle any series of calls on word-weight pairs appropriately, including corner cases.

The most notable corner case is adding a word that is already in the trie, but with a lower weight. For example, if we were to call add(“apes”, 1). In this case, we would have to update the weight of apes, and then the mySubtreeMaxWeight of all its ancestors. This is especially tricky because not all the ancestors would not have the same new value. Be sure to take advantage of parent pointers when writing code specific to this case.

topMatch

Once add is written *topMatch()* becomes very simple. As noted in the above example, the mySubtreeMaxWeight of every node from a prefix node to the node under the prefix

node with the highest weight will be the same as that highest weight. Thus, the algorithm to find the top match is simply:

1. Navigate to the node corresponding to the prefix
2. Use the mySubtreeMaxWeight of the prefix node to navigate down the tree until you find the Node with the max weight

topMatches

topMatches() is similar, but not quite the same as *topMatch()*. We will still be taking advantage of mySubtreeMaxWeight to quickly navigate to high-weight nodes, but this time, we will have to go down multiple branches instead of just one.

To find the top k matches as quickly as possible, we will use what is known as a search algorithm - keep a PriorityQueue of Nodes, sorted by mySubtreeMaxWeight. Start with just the root in the PriorityQueue, and pop Nodes off the PriorityQueue one by one.

Anytime we pop a node off the PriorityQueue, or "visit" it, we will add all its children to the PriorityQueue. Whenever a visited node is a word, add it to a weight-sorted list of words.

When this list has k words with weight greater than the largest mySubtreeMaxWeight in our priority queue, we know none of the nodes we have yet to explore can have a larger weight than the k words we have found. At that point, we can stop searching and return those k words. If we run out of nodes in the PriorityQueue before we find k words, that means there are not k words in the trie.

Challenge: Spell Check

As a **challenge**, you can implement *spellCheck* to return the highest weighted matches within d (less than our equal to d) edit distance of the word. The challenge is neither required nor will you earn any additional points by completing it. Do it to improve your technical skills and for the pride it would bring you in being able to complete it!

Essentially, edit distance is the number of edits (deletions, insertions and replacements) that need to be made in order to change one word to another word. If the word is in the dictionary, then return an empty list.

Your implementation must be efficient. That is, it should run in better than brute-force time in the average case for large dictionaries. To be more specific, your program should not examine terms that are greater than edit distance d+1 from any prefix of the input word, and it should not repeat any computation that can be memorized. Outlined below is an efficient-enough solution to the dictionary edit distance problem that builds on your solution for part 1. You are free to use faster solutions, like DAWGs or BK-trees.

First, read the wikipedia article on Levenshtein distance. To make testing (and your implementation job!) easier this will be the only metric we use and test to determine edit distance; however if you are interested, feel free to look at edit distances that include transposition and phonetic matching algorithms that are widely used in search today. You can implement these on your own if you are interested. However, do not submit them!

Take note of the two-row solution. We will optimize that using a trie. Think about the matrix we construct as we iterate through. As the index in the word increases, only the last row of the matrix changes. We can avoid a lot of work if we can process the words in order, so we never need to repeat a row for the same prefix of letters.

For example, suppose our query word is *joshhug*, but the closest match in our dictionary is *boshbug*. Then when comparing *joshhug* against *boshbug*, the rows will look like this:

```
String  | Array
---------------------------------
        | [0, 1, 2, 3, 4, 5, 6, 7]
b       | [1, 1, 2, 3, 4, 5, 6, 7]
bo      | [2, 2, 1, 2, 3, 4, 5, 6]
bos     | [3, 3, 2, 1, 2, 3, 4, 5]
bosh    | [4, 4, 3, 2, 1, 2, 3, 4]
boshb   | [5, 5, 4, 3, 2, 2, 3, 4]
boshbu  | [6, 6, 5, 4, 3, 3, 2, 3]
boshbug | [7, 7, 6, 5, 4, 4, 3, 2]
```

Don't worry about the cases of characters. Uppercase and lowercase versions of the same letter will be treated differently.

For example, running an interactive spellcheck loop (by modifying the test client given in Autocomplete) showing the top 5 matches on *wiktionary.txt* of edit distance 1 gives:

```
whut
     1605908.0  what
       67063.6  shut
       21374.0  hut
        4155.8  whit
what
efect
      148795.0  effect
       20818.7  defect
       16929.4  erect
       13700.0  elect
heyy
gurl
      262689.0  girl
        4194.6  curl
```

Error Cases

For spellCheck, throw an IllegalArgumentException if dist is not positive, or if k is negative.

Benchmarking Autocomplete

The class *AutocompletorBenchmark* has been provided to you for timing your implementations of Autocompletor.

The benchmark class will time all of the following:

- The time it takes to initialize a new instance of the class.
- If the class is TrieAutocomplete, the number of nodes in the initialized Trie
- The time it takes to call *topMatch* and *topMatches*() for varied k on: A blank string, a random word from the source, prefixes of that word, and a prefix which does not start any words in the source

You can, of course, modify the benchmark class to add more tests.

For timing method calls, the benchmark class runs until either 1000 trials or 5 seconds have passed (to try to minimize variation in the results) and reports the average. Thus, it should not be surprising if one test takes 5 seconds to run, but if more than 5 seconds pass and a test has not completed, there is likely an infinite loop in your code.

Even for the largest source files (fourletterwords.txt and words-333333.txt) *TrieAutocomplete* should have some or most methods take times in the range of microseconds. This makes these methods' runtimes very susceptible to variation, even when we run 1000 trials of them - be sure to consider this when interpreting results and writing your analysis.

Analysis

In addition to submitting your completed implementation of *TrieAutoComple*, you should benchmark them. Then, answer the following questions. Use data wherever possible to justify your answers, and keep explanations brief but accurate:

1. What is the order of growth (big-Oh) of the number of compares (in the worst case) that each of the operations in the *Autocompletor* data type make?
2. How does the runtime of *topMatches()* vary with k, assuming a fixed prefix and set of terms? Provide answers for *BruteAutocomplete* and *TrieAutocomplete*. Justify your answer, with both data and algorithmic analysis.
3. How does increasing the size of the source and increasing the size of the prefix argument affect the runtime of *topMatch* and *topMatches*? (Tip: Benchmark each implementation using fourletterwords.txt, which has all four-letter combinations from aaaa to zzzz, and fourletterwordshalf.txt, which has all four-letter word combinations

from aaaa to mzzz. These datasets provide a very clean distribution of words and an exact 1-to-2 ratio of words in source files.)

For details, see Programming Project Deliverables.

Notes:

jGrasp is recommended IDE, if you use other IDEs (e.g. Netbeans or Eclipse) please make sure to keep source files under **<default package>** otherwise you might face some errors.

Grading

For transparency's sake, below is a list of aspects of your code the automated tests will check. This may appear to be a large number of tests, but if you follow the guidelines set in the method headers and this writeup, you should already be passing most of, if not all of these tests:

- **75% Correctness:** for your implementation of *TrieAutocomplete* and passing the tests described above.
- **25.0% Analysis:** for your REFLECT, pseudocode, data from *AutocompletorBenchmark*. answers to the questions, and description of the tradeoffs.

Detailed Grading Rubric:

Criteria	Points (high)	Points (mid)		Points (low)
If there is some implementation and the code compiles	5 Full marks	-		0 No marks
Correct implementation of add()	20 Full marks	10 Some test cases fail, partially correct implementation		0 No marks
Correct implementation of topMatch()	20 Full Marks	10 Some test cases fail, partially correct implementation		0 No marks
Correct implementation of topMatches()	30 Full marks	20 Some test cases fail, partially correct autocomplete lists with weights	10 Autocomplete lists with no/0 weights	0 No marks
Correct Benchmarking	add() pseudocode and complexity	2.5 + 2.5 Full marks		0 No marks
	topMatch() pseudocode and complexity	2.5 + 2.5 Full marks		
	topMatches() pseudocode and complexity	2.5 + 2.5 Full marks		
	Analysis Q/A	1 + 0.5 + 0.5 Full marks		
	Graphical analysis	8 Full marks		

Caution

Your submission should include the following statement verbatim. If not included, your project will automatically receive a zero grade.

NO-PLAGIARISM CERTIFICATION: I certify that I wrote the code I am submitting. I did not copy whole or parts of it from another student or have another person write the code for me. Any code I am reusing in my program from the web or some other source is clearly marked as such with its source clearly identified in comments.

Your work should follow the academic integrity guidelines stated in the syllabus. Do your own work; do not copy that of another. We may run code plagiarism detection software on your submissions. If we find evidence of copying, you will automatically receive a zero for this homework and we will refer your plagiarism to the appropriate university committee and your no plagiarism certification will be used in the proceedings.

Submission:

Your submission must include exactly the following:

1. Your implementation of AutoComplete.java (no other source code/class file)
2. Complete Programming Project Deliverable (that contains your benchmarking)
3. NO-PLAGIARISM CERTIFICATION (Create a text file with your name, your email address, the no plagiarism certification, and what IDE or compiler you used).

Submit to canvas all three abovementioned files as a single zipped folder named yourID.zip (i.e. xzy0099.zip) before the deadline on the due date.

If you have any doubts/questions, ask the TA before you code. Our expectation is that you know how to program by now. So, the TA will not debug your source code for you.

If there is any trouble with your submission, it is up to you to sort out any such problem cooperatively with the TA. If this is not done in a timely manner, your homework will not be graded.