Autocomplete Me

Background

Autocomplete is pervasive in modern applications. As the user types, the program predicts the complete *query* (typically a word or phrase) that the user intends to type. Autocomplete is most effective when there are a limited number of likely queries. For example, the Internet Movie Database uses it to display the names of movies as the user types; search engines use it to display suggestions as the user enters web search queries; cell phones use it to speed up text input.

In these examples, the application predicts how likely it is that the user is typing each query and presents to the user a list of the top-matching queries, in descending order of weight. These weights are determined by historical data, such as box office revenue for movies, frequencies of search queries from other Google users, or the typing history of a cell phone user. For the purposes of this assignment, you will have access to a set of all possible queries and associated weights (and these queries and weights will not change).

The performance of autocomplete functionality is critical in many systems. For example, consider a search engine which runs an autocomplete application on a server farm. According to one study, the application has only about 50ms to return a list of suggestions for it to be useful to the user. Moreover, in principle, it must perform this computation *for every keystroke* typed into the search bar and *for every user*!

Task

In this assignment, you will implement autocomplete using weighted tries. 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 k queries that start with the given prefix, in descending order of weight. Provide autocomplete functions in a form that a user could easily load into their program to support autocomplete in an application.

Once you have implemented the autocomplete method, write a command-line program which provides suggestions when given terms on the command line. For example,

```
python autocomplete.py "the" wiktionary.txt 5
```

would produce 5 completions of "the" from the terms in `wiktionary.txt`. Example output is shown in Examples. This command-line problem will naturally be slow, because it must load the input file from scratch each time; in an application, you would load the file only once and keep the data in memory to serve many autocomplete requests.

Several input files of various sizes, containing words and their weights, are provided in the `Data/autocomplete-me` directory: `movies.txt` is a list of movie titles from IMDB,

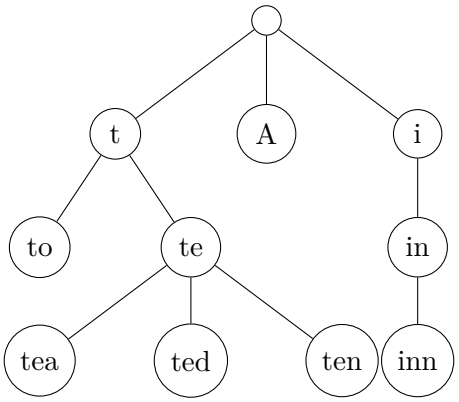
[0]Original problem written by Kevin Wayne.

`baby-names.txt` a list of the top US baby names, `wiktionary.txt` a list of common English words, and `pokemon.txt` a complete list of Pokémon. Each file starts with a line indicating the number of terms in the file; the following lines are pairs of terms and weights, separated by tabs.

Tries

Autocomplete needs to be fast, but the space of possible suggestions is huge: imagine how many different queries Google may need to suggest completions for. If n possible completions are stored in a large array, a linear search taking $O(n)$ time may be impractically slow, and even a binary search taking $O(\log n)$ time may be too slow.

Most autocomplete systems use a data structure like a ***trie***, a tree data structure that efficiently allows you to look up terms based on their prefixes. A very simple trie for English words starts with a root node with 26 children, one for each English character. Words starting with 'a' go into the 'a' child, and so on. Each child has child nodes for the second letter of terms, and their children correspond to the third letter, and so on until there are no terms long enough to require more child nodes. An example tree, for the words "A", "to", "tea", "ted", "ten", "i", "in", and "inn", is:



If the user starts typing "te", for example, we can traverse the tree from the root, from "t" to "te", then find that "tea", "ted", and "ten" are all possible completions.

Our task is harder because terms have weights. We need to find the top k suggestions by weight. We could obtain all suggestions, sort, then pick the top k, but this is inefficient. Instead, we can store weights with each node. Leaf nodes are marked with the weight of the word they represent, while branch nodes are marked with the *maximum* weight of their children. When traversing the tree, instead of using ordinary depth-first or breadth-first search we use a priority queue to traverse the nodes with highest weights first, stopping the traversal as soon as we have collected k suggestions. Hence if there are 10,000 possible completions of "the", but k is 5, we will not traverse all 10,000 possible completions in the trie.

The trie represents a different choice of tradeoffs from, say, using binary search to scan through all possible autocomplete suggestions. The binary search method makes it very easy to load the data—just read in the words and weights and you're basically done. But it takes $O(\log n)$ time to search for words and $O(n \log n)$ time to sort the results. The trie requires

additional processing to load in the data and build the trie, which may be rather slow, but finding a word in a trie takes $O(m)$ time, where m is the length of the word, *regardless* of the number of words in the trie. If we can reuse the same trie to answer many thousands of autocomplete queries, the extra trie-building processing is worth it.

Implement your autocomplete algorithm using a trie. You may use any reference materials necessary on the construction of tries, but *you must implement your trie on your own*. Key issues to address in your implementation:

- How do you represent the trie? A simple functional programming method uses a **`Node`** data type (struct, record, or equivalent) containing an array of child nodes, a maximum weight, and other appropriate data, plus functions for querying the tree. An object-oriented implementation might use a **`Trie`** class with query methods.
- How do you insert words into the trie? You will need some kind of recursive method that updates the maximum weights of branch nodes, if necessary, as it descends the tree to find the appropriate place to put the word.
- What if a node contains words but also has children? For example, the above trie contains "in" and "inn", a child of "in"; your autocompleter must correctly suggest both terms when provided the query "i". But if the trie contained "inn" but not "in", you must not suggest "in". Write tests for this case.
- How do you test your tree code? Simple manual tests are likely not sufficient; you will need to write extensive randomized tests to catch edge cases and subtle bugs.

R users may also want to explore the `data.tree` package from CRAN, which provides tools for creating trees in R. But remember that your trie implementation must be your own.

Example

If you write a *`read.terms`* function to ingest a file and produce a data structure containing the terms, and *`autocomplete`* to do the searching and suggesting, then we might expect

```
words <- read.terms("wiktionary.txt")
autocomplete("the", words, 5)
```

to produce something like

```
5627187200 the
 334039800 they
 282026500 their
 250991700 them
 196120000 there
```

Make sure *`autocomplete`* returns results in a useful form which could be used by an application to do completion, and that it can be called many times with a trie loaded only once.

Questions

In your GitHub pull request, or in a text file included with your submission, explain the performance of your code. You should address the following points in detail:

- For the largest input file (`movies.txt`), how quickly can your matcher return matches for typical queries? How long does it take to load the data (and build the trie, if necessary)? You should separately time loading the data (which only has to be done once) and finding matches, which can happen many times after the data is loaded.

  Note: R does not provide efficient tools to build large data structures, and so loading 200,000 movies may take a prohibitively long time. If so, you can use a smaller test file to answer this question.

- How does performance scale with the number of words? Test on various input sizes and graph the results. Try to estimate the complexity (in big O notation) of your algorithm.

- What seems to be the slowest part of your code? What could you improve? Point to specific functions or portions of code that take a disproportionate amount of time to run.

Use profiling where appropriate to justify your claims. (The `microbenchmark` package for R may also be useful; Python provides a `timeit` module which is similar.) Provide source code for any benchmarks or tests you run. Testing on a single input isn't sufficient—you must test on a range of inputs to be sure your results are representative. Analyze performance like a statistician.

Requirements

A **Mastered** submission will:

☐ Provide the searching, matching, and output functions described above. Ensure they are well-documented with comments explaining how they work.

☐ Supply a complete set of tests to ensure your matcher finds the correct terms in the correct order. Write supplemental tests for the component pieces, like the trie algorithms.

☐ Use a randomized test of the trie matching algorithms to ensure they function correctly for a wide range of inputs.

☐ Write a command-line program to find suggestions for a given input, as described above.

☐ Answer the questions specified above.

A **Sophisticated** submission will additionally:

☐ Use excellent variable names, code style, organization, and comments, so the code is clear and readable throughout.

☐ Be efficient, making the best use of its data structures with minimal copying, wasteful searching, or other overhead.

☐ Have a comprehensive suite of tests to ensure that the individual components function correctly on corner cases, error cases, and unexpected inputs.

☐ Give particularly thorough and detailed answers to the questions above.

☐ Supply a command-line driver that runs all your tests.

You should also remember that **peer review is an essential part of this assignment.** You will be asked to review another student's submission, and another student will review yours. You will then revise your work based on their comments. You should provide a clear, comprehensive, and helpful review to your classmate.