Autocomplete

Submission to website: Monday, 4/10, 10pm

Checkoff by LA/TA: Thursday, 4/13, 10pm

This lab assumes you have Python 3.5 or later installed on your machine. Please use the Chromium or Firefox web browser.

This lab has 30 tests and 6 coding points. The coding points will be assigned during check-off based on the organization and readability of your code.

Introduction

Type "aren't you" into Google search and you'll get a handful of search suggestions, ranging from "aren't you clever?" to "aren't you a little short for a stormtrooper?". If you've ever done a Google search, you've probably seen an autocompletion - a handy list of words that pops up under your search, guessing at what you were about to type.

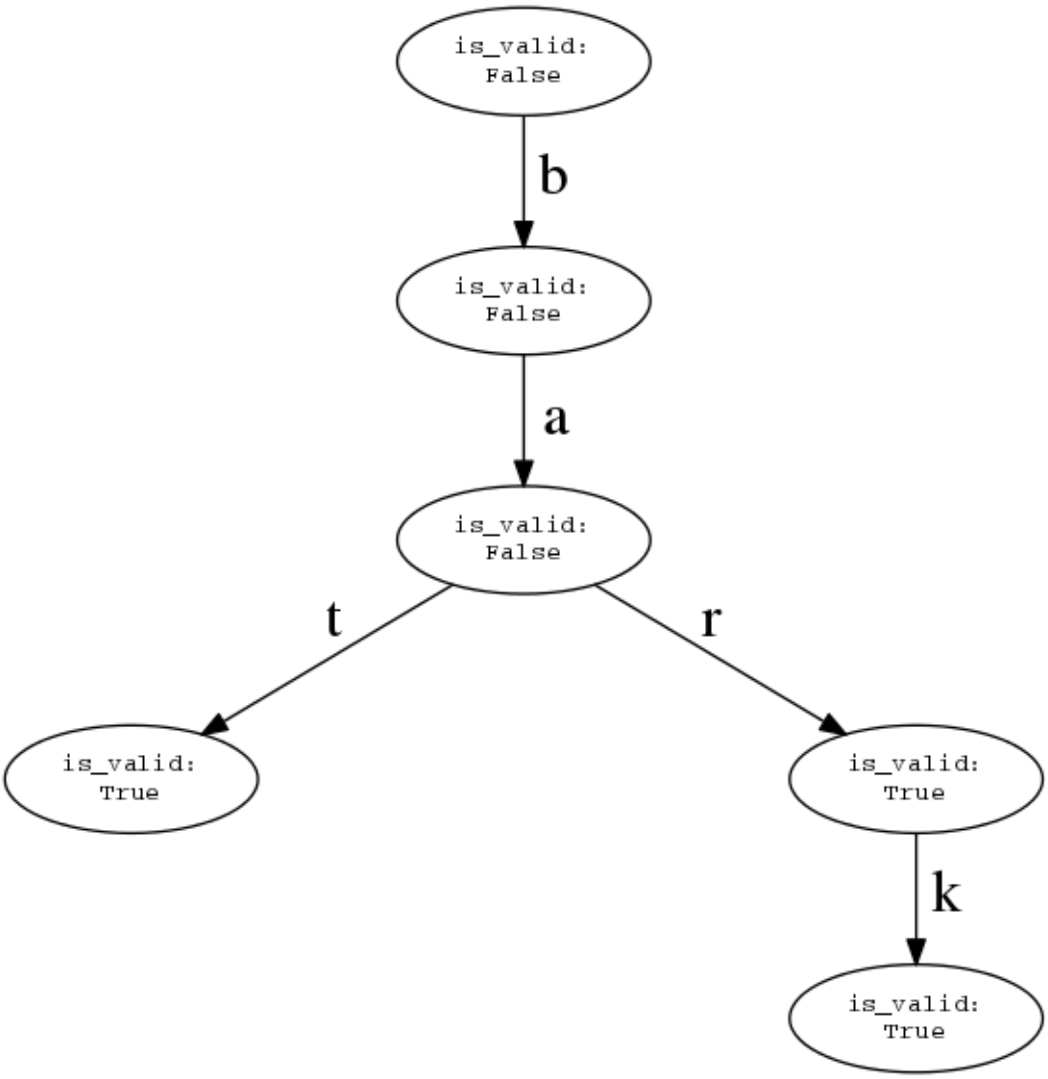
Search engines aren't the only place you'll find this mechanism. Powerful code editors, like Eclipse and Visual Studio, use autocomplete to make the process of coding more efficient by offering suggestions for completing long function or variable names.

In this lab, we are going to implement our own version of an autocomplete engine using a tree structure called a "trie," as described in this document. The staff have already found a nice corpus (list of words) for you to use - the full text of Jules Verne's "In the Year 2889." The lab will ask you first to generate the trie data structure using the list of words provided. You will then use the trie to write your own autocomplete and autocorrect, which select the top few words that a user is likely to be typing. Note that all words in the corpus and any string argument values in the tests will be in lower case.

The trie data structure

A trie, also known as a prefix tree, is a type of search tree that stores words organized by their prefixes (their first characters), with longer prefixes given by successive levels of the trie. Each node contains a Boolean (`true` / `false`) value stating whether this node's prefix is a word.

For example, consider the words `'bat'` , `'bar'` , and `'bark'` . A trie over these words would look like the following:



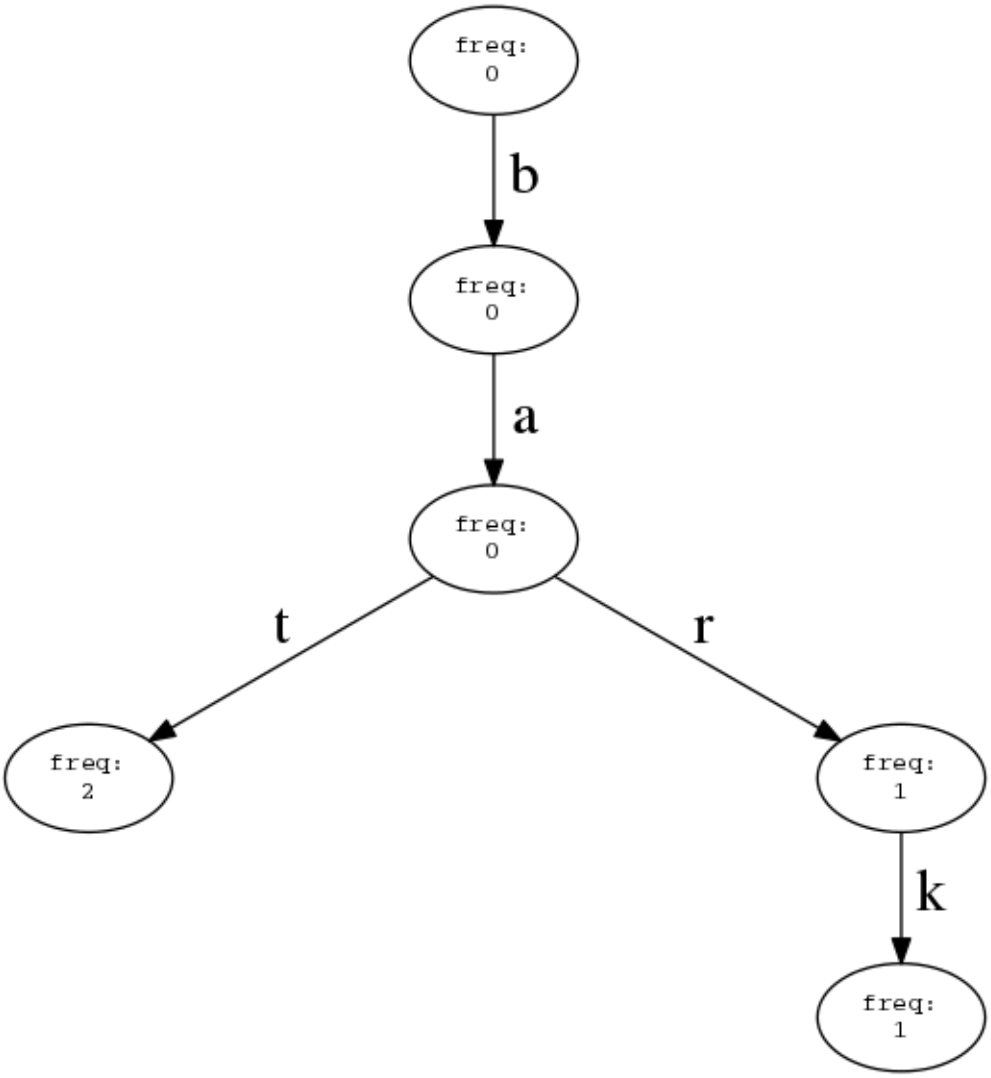
To list all words beginning with `'ba'`, begin at the root and follow the `'b'` and `'a'` edges to reach the node representing a `'ba'` prefix. This node is itself a trie and contains all words prefixed by `'ba'`. Enumerating all paths leading to `true` nodes (in this case, `'t'`, `'r'`, `'rk'`) produces the list of `'ba'` words: `'bat'`, `'bar'`, and `'bark'`.

Note that we also check the `'ba'` node itself, though in this case the node is `false`, meaning `'ba'` is not known to be a word. Consider the words beginning with the string `'bar'`. Just as before, follow the `'b'`, `'a'`, and `'r'` edges to the `'bar'` node, then enumerate all paths to `true` nodes (`''` and `'k'`) to find the valid words: `'bar'` and `'bark'`.

This trie structure on its own, however, is not very useful. If we type only a few characters, for example `'b'`, the long list of words `b` generates is of little help to the user, who is only interested in the most likely candidates. To this end, we replace the Boolean flag in each node of our trie with a frequency metric, describing how often each word appears in our corpus. If the frequency of a trie node is 0, the prefix this node is not a word in the corpus. Assume that the more often a word appears in the corpus, the more likely it is to be

typed by our user. When using the trie to enumerate likely words, suggest only a few likely matches instead of the entire list.

Consider the following corpus: `"bat bark bat bar"`. The `example_trie` this corpus would generate is:



Assume we are interested in only the top result after autocompleting the string `'ba'`. Now instead of giving us all of `'bat'`, `'bark'`, and `'bar'`, we should just get the highest-frequency word - `'bat'`.

Note that in the tree above, the `'b'` and `'ba'` nodes have frequencies of `0`, meaning they're not valid words.

Trie class and basic methods

In `lab.py`, you are responsible for implementing the `Trie` class, which should support the following

methods:

__init__(*self*)

Initialize self to be an object with two instance variables:

- `frequency`, an integer frequency (number of times the word appears in corpus) of the word ending at this node. Initial value is 0.
- `children`, a dictionary mapping single-character strings to another trie node, i.e., the next level of the trie hierarchy (tries are a recursive data structure). Initial value is an empty dictionary.

insert(*self, word, freq*=None) **[tests 1-4]**

Add the given `word` to the trie, modifying the trie by adding child trie nodes as necessary. For the trie node that marks the end of the word, if `freq` is `None`, increment the node's frequency instance variable, otherwise set it to the specified value. This method doesn't return a value.

Examples (using `example_trie`):

- `t = Trie()` would create the root node of the `example_trie`.
- `t.insert("bat")` adds the two center nodes, with frequencies of 0, and the node to left of center, with a frequency of 1.
- `t.insert("bark")` adds the two nodes to the right of center, setting the frequency of the last node to 1.
- `t.insert("bat")` doesn't add any nodes and only increments the frequency of the node to the left of center.
- `t.insert("bar")` doesn't add any nodes and only increments the frequency of the node topmost node to right of center.

find(*self, prefix*) **[tests 5-8]**

Return the trie node for the specified `prefix` or `None` if the prefix cannot be found in the trie.

Examples (using `example_trie`):

- `t.find("")` returns `t`, the root node.
- `t.find("ba")` returns the bottommost of the three center nodes, i.e., `t.children["b"].children["a"]`.

__contains__(*self, word*) **[tests 9-13]**

Return True if `word` occurs with a non-zero frequency in the trie. This is the special method name used by Python to implement the `in` operator. For example,

```
word in trie
```

is translated to

```
trie.__contains__( word )
```

Hint: use `self.find(word)` to do the hard work!

Examples (using `example_trie`):

- `"ba" in t` returns False since that interior node has a frequency of 0.
- `"bar" in t` returns True
- `"barking" in t` return False since "barking" can't be found in trie.

__iter__(*self*) **[tests 14-15]**

Return an iterator that produces `[word, freq]` pairs for each word stored in the trie. The pairs can be produced in any order. This is the special method name used by Python when it needs to iterate over a data object, i.e., the method invoked by the `iter()` built-in function. For example, the following Python code will print all the words in a trie:

```
print(word for word,freq in trie)
```

Hint: see the slides for Lecture 8. You'll want to return a generator function that uses `yield` and `yield from` to produce the required sequence of values one at a time. See https://docs.python.org/3/howto/functional.html#generators.

Examples (using `example_trie`):

- `list(t)` returns `[['bat', 2], ['bar', 1], ['bark', 1]]`. Note that the `list` function has an internal `for` loop that uses `iter(t)` to iterate over each element of the sequence `t`.
- `list(t.find("bar"))` returns `[['', 1], ['k', 1]]`. This may seem a bit weird, but rememeber that we were treating the interior node returned by `t.find("bar")` as the root of its

own mini-trie.

Autocomplete method

autocomplete(*self, prefix, N*) **[tests 16-20]**

`prefix` is a string, `N` is an integer; returns a list of up to `N` words. Return a list of the `N` most-frequently-occurring words that start with `prefix`. In the case of a tie, you may output any of the most-frequently-occurring words. If there are fewer than `N` valid words available starting with `prefix`, return only as many as there are. The returned list may be in any order.

Return `[]` if prefix is not in the trie.

Hint: `self.find` is useful in finding the trie node at which to start your enumeration.

Examples (using `example_trie`):

- `t.autocomplete("ba",1)` returns `['bat']`.
- `t.autocomplete("ba",2)` might return either `['bat', 'bark']` or `['bat', 'bar']` since "bark" and "bar" occur with equal frequency.
- `t.autocomplete("be",1)` returns `[]`.

Autocorrect method

You may have noticed that for some words, our autocomplete implementation generates very few or no suggestions. In cases such as these, we may want to guess that the user mistyped something in the original word. We ask you to implement a more sophisticated code-editing tool: autocorrect.

autocorrect(*self, prefix, N*) **[tests 21-24]**

`prefix` is a string, `N` is an integer; returns a list of up to `N` words. `autocorrect` should invoke `autocomplete`, but if fewer than `N` completions are made, suggest additional words by applying one **valid edit** to the prefix.

An **edit** for a word can be any one of the following:

- A single-character insertion (add any one character in the range a-z> at any place in the word)
- A single-character deletion (remove any one character from the word)
- A single-character replacement (replace any one character in the word with a character in the range a-z

- A two-character transpose (switch the positions of any two adjacent characters in the word)

A **valid edit** is an edit that **results in a word in the trie without considering any suffix characters**. In other words we don't try to autocomplete valid edits, we just check if `valid_edit in self` is True.

For example, editing `"te"` to `"the"` is valid, but editing `"te"` to `"tze"` is not, as "tze" isn't a word. Likewise, editing `"phe"` to `"the"` is valid, but `"phe"` to `"pho"` is not because "pho" is not a word in the corpus, although many words beginning with "pho" are.

In summary, given a prefix that produces C completions, where C < N, generate up to N-C additional words by considering all valid single edits of that prefix (i.e., corpus words that can be generated by 1 edit of the original prefix), and selecting the most-frequently-occurring edited words. Return a list of suggestions produced by including **all** C of the completions and up to N-C of the most-frequently-occuring valid edits of the prefix; the list may be in any order. Be careful not to repeat suggested words!

Example (using `example_trie`):

- `t.autocorrect("bar",3)` returns `['bar', 'bark', 'bat']` since "bar" and "bark" are found by autocomplete and "bat" is valid edit involving a single-character replacement, i.e., "t" is replacing the "r" in "bar".

Selecting words from a trie

It's sometimes useful to select only the words from a trie that match a pattern. That's the purpose of the `filter` method.

filter(*self, pattern*) **[tests 25-30]**

`pattern` is a string. Return a list of `[word, freq]` pairs for those words whose characters match those of `pattern`. The characters in `pattern` are matched one at a time with the characters in each word stored in the trie. If all the characters in a particular word are matched, the `[word, freq]` pair should be included in the list to be returned.

The characters in `pattern` are interpreted as follows:

- `'*'` matches a sequence of **zero or more** of the next unmatched characters in `word`.
- `'?'` matches the next unmatched character in `word` no matter what it is. There must be a next unmatched character for `'?'` to match.
- otherwise the character in the pattern must exactly match the next unmatched character in the word.

Pattern examples:

- `"*a*t"` matches all words that contain an "a" and end in "t". This would include words like "at", "art", "saint", and "what".
- `"year*"` would match both "year" and "years"
- `"*ing"` matches all words ending in "ing"
- `"???"` would match all 3-letter words
- `"?ing"` matches all 4-letter words ending in "ing"
- `"?*ing"` matches all words with 4 or more letters that end in "ing"

Filter examples (using `example_trie`):

- `t.filter("*")` returns `[['bat', 2], ['bar', 1], ['bark', 1]]`, i.e., listing all the words in the trie.
- `t.filter("???")` returns `[['bat', 2], ['bar', 1]]`, i.e., listing all the 3-letter words in the trie.
- `t.filter("*r*")` returns `[['bar', 1], ['bark', 1]]`, i.e., listing all the words containing an "r" in any position.

Hint: the matching operation can implemented as a recursive search function that attempts to match the next character in the pattern with some number of characters at the beginning of the word, then recursively matches the remaining characters in the pattern with remaining unmatched characters in the word.

Note: you **cannot** use any of the built-in Python pattern-matching functions, e.g., functions from the `regex` module -- you are expected to write your own pattern matching code. Copying code directly from stackoverflow is also not appropriate.

Testing your lab

We've included a 6.009-autocomplete-powered search bar so you can see your code in action. Run `server.py` and open your browser to localhost:8000 and type into the search bar to see the top 5 results from your `autocomplete` and `autocorrect` function, using the corpus of words from Jules Verne's "In the Year 2889."

In the search box, try typing "when", checking after each letter to see the suggested words:

- "w" suggests "was", "with", "which", "will", "we"
- "wh" suggests "which", "what", "when", "why", "who"
- "whe" suggests "when", "where", "whether", "whenever", "whence"

- "when" suggests "when", "whenever", "whence"

In the autocorrection box, try typing "thet" then Ctrl+Space to see a list of suggested corrections: "the", "that", "they", "then", and "them".

As in the previous labs, we provide you with a `test.py` script to help you verify the correctness of your code.

Does your lab work? Do all tests in `test.py` pass? You're done! Submit your `lab.py` on web.mit.edu/6.009 and get your lab checked off by a friendly staff member.