Autocomplete Engine

Type "aren't you" into a search engine 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 web 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. For example, cell phones use autocomplete to predict words. Some IDEs use autocomplete to make the process of coding more efficient by offering suggestions for completing long function or variable names.

Your task will be to write an autocomplete engine. Your program will read in a corpus of text that will form the basis of your autocomplete engine. You will then read in a list of queries to perform autocompletion on, and output the results.

The Engine

Your autocomplete engine will suggest phrases based on the most commonly occurring phrases in the corpus. Your engine will take two pieces of information as input:

prefix
: the prefix to autocomplete

max_count
: the maximum number of autocomplete results to return

Your engine should take in a `prefix` and `max_count`, and output a list of the `max_count` most-frequently-occurring phrases in the corpus that start with or exactly match prefix. In the case of a tie, you may output any of the most-frequently-occurring phrases. If there are fewer than `max_count` valid phrases available starting with or matching prefix, output only as many as
there are. The outputted list may be in any order.

If `max_count` is zero (`0`), your list should contain all keys that start with `prefix`.

> **Note:**
> Comparisons between the provided prefix and the corpus of phrases should be done case insensitively.

For example, if the corpus contained (`bat`, `bat`, `bark`, `bar`), then the following are examples of possible inputs and expected outputs:

- `autocomplete("ba", 1)` outputs `["bat"]`
- `autocomplete("ba", 2)` might output either `["bat", "bark"]`, `["bark", "bat"]`, `["bat", "bar"]`` ``, or `["bar", "bat"]` since `"bark"` and `"bar"` occur with equal frequency
- `autocomplete("be", 1)` outputs `[]`

Input Format

Your program should accept the corpus filename as an input parameter. The corpus file will be a list of phrases, with one phrase per line. These phrases will only contain letters (A-Z, a-z) and spaces. For example, this could be a simple corpus file:

```
bat
Bat
bark
bar
```

Your program should then read the queries to perform on the corpus from `stdin`. Each query will be its own line and contain the following information separated by commas:

`query`
    The command to use. This will always be ``"complete"`` -- use your autocomplete engine to perform the query.
`prefix`
    The search phrase.
`max_count`
    The maximum number of results to return.

Here are some example input queries:

```
complete,ba,1
complete,ba,2
complete,be,1
complete,b,0
```

Output Format

Your program should output the results of each query as a comma-separated list of phrases to `stdout`. Each query result should be on its own line.

For example, the input above should create the following output:

```
bat
bat,bark

bar,bark,bat
```

Note that the third line is intentionally blank -- the input corpus does not have any phrases that begin with `"be"`.

Submission

Your submission should contain all of your code, build infrastructure, test assets, and a README explaining how to build and run your code. Your code must build and run in Linux. Your README should also contain a description of your solution including things like design choices you made and the rationale behind them, testing procedure, and what you might improve if you had more time.

Please submit your solution in a tarball

Note: Please do not post your solution to a public Github repository.

Evaluation Criteria

In software development, there are a variety of ways to solve any given problem. We are interested in your thought process, so please capture that in the README that you submit with your project. Also, rather than just give you the problem and have you solve it without understanding how we'll evaluate the project, we want to be clear about our expectations. We identified a few key characteristics that are important to us at Stateless as part of our every day engineering practices and we will evaluate your coding project based on these same principles.

correctness

The code should work as defined, including the examples we gave above, but for a broader set of inputs. Our product is highly-available in critical infrastructure -- it is survivable in the face of the unknown.

performance

We focus on two key aspects of performance in this project -- algorithmic efficiency and data structure choices. Note that your code may be run against large data sets; memory usage should be a consideration. At Stateless, our product is expected to operate in a highly-efficient manner, whether it is processing network traffic at high rates, quickly reacting to events, or creating a reactive user interface.

testing

How you test your code indicates your approach to driving for quality. At Stateless, our testing ranges from unit testing to full-system tests. While building a full CI/CD pipeline is overkill for this project, some amount of testing is critical.

readability

While this can be highly subjective, there are some base things to consider. Things like meaningful variable names, using straightforward logic and control flow, declaring variables to minimize scope, and avoiding repetitive code are evaluated; your choice of `CamelCase` or `snake_case`, tabs or spaces are not. High-quality code is read far more frequently than it is written.