Slang

Challenge:
Engineering Intern

A coding challenge for the engineering intern role at Slang

Overview

This document describes a coding challenge for engineering students looking to work as an intern with Slang. We expect that you spend a maximum of two hours to develop a solution that answers the problem listed below.

Feel free to submit your work quickly if you feel that you've adequately demonstrated your abilities. Be sure to balance code quality, performance and productivity, and keep in mind that we take attention to detail very seriously!

The task is based on a two-step real-world task that we've had to solve, and while it's been boiled down into specifications appropriate for a short development assessment, it is still representative of some of the work you'd be exposed to in this role. We suggest that you read through everything here — it's a quick read — before starting. And, as always, let us know if you have any questions or comments! I will be available to answer any questions at ricardo@slangapp.com at any point throughout the day/night.

Specifications

The descriptions here are intentionally a bit free-form — we'd like you to make most of the decisions yourself.

First Step

We'd like you to write a simple function that given a piece of text and an integer *n*, it returns the n-grams for that text. For background, "an n-gram is a contiguous sequence of n items from a given sample of text or speech" [Wikipedia]. It is commonly used in the field of Natural Language Processing (NLP). Here are some examples to help you understand this concept:

```
1-gram of "Slang" → ["S", "l", "a", "n", "g"]

2-gram of "Slang" → ["Sl", "la", "an", "ng"]

3-gram of "Slang" → ["Sla", "lan", "ang"]

4-gram of "Slang" → ["Slan", "lang"]

5-gram of "Slang" → ["Slang"]
```

The signature of your function should be something similar to this:

```
calculateNGrams(text, n);
```

The result of running it against one of our examples above should return the same output

```
calculateNGrams("Slang", 3);
["Sla", "lan", "ang"]
```

Second Step

We'd like you to extend the functionality of your algorithm to return only the most frequent n-gram, not all of them. Let's run a more interesting example through our first function:

```
calculateNGrams("to be or not to be", 2);

["to", "o ", " b", "be", "e ", " o", "or", "r ", " n", "no", "ot", "t ", " t", "to",
"o ", " b", "be"]
```

As you can see, there are several 2-grams that are repeated, our new task is to find the most repeated one. In this example it is “to” (as it’s the first one to be repeated twice). Your function should look something like this:

```
mostFrequentNGram("to be or not to be", 2);

"to"
```

What we’re expecting

We’re expecting an algorithm that handles these two steps in a nice, cleanly organized and efficient way. It should handle edge cases. You can choose any language you feel most comfortable with to implement it. Make sure you follow conventions and idiomatic guidelines for your chosen language! We move fast but are fanatical about writing clean, idiomatic, maintainable code. You should, as a comment within your functions, correctly explain the running time complexity of your algorithm. Extra credit if it’s better than quadratic.

Submission

You should submit your algorithms as a single file in a GitHub repository. You should share your repository with me (username **ricardovj**), and you should use the repository as you would for any other project — commit early and often! We’re interested in seeing the history for your work, too.

All right — that’s it. Thanks for reading through this, and let me know if you have any questions or comments. We’re excited to see what you come up with!