1 Project 1

Due: Sep 15 by 11:59p

Important Reminder: As per the course *Academic Honesty Statement*, cheating of any kind will minimally result in receiving an F letter grade for the entire course.

This document first provides the aims of this project. It then lists the requirements as explicitly as possible. This is followed by logs which should help you understand the requirements. Finally, it provides some hints as to how those requirements can be met.

1.1 Aims

The aims of this project are as follows:

- To get you to write a simple but non-trivial JavaScript program.
- To allow you to familiarize yourself with the programming environment you will be using in this course.
- To introduce you to text processing within JavaScript.

1.2 Requirements

This project requires you to write a crude document searching command-line application. The application will be provided with a file giving noise words which should be ignored while searching, as well as one-or-more additional files which constitute the documents being searched.

The user will specify a search by typing in a line containing search terms and the application should respond with the matches found for that search.

Both documents and search lines are preprocessed as follows:

1. They are split into words where a word is a maximal sequence of non-whitespace characters.
2. Each word is normalized:
   (a) The word is converted to lower-case.
   (b) The word is replaced by its stem. The only stemming currently performed is to delete any `'s` suffix.
   (c) All non-alphabetic characters (characters other than `a` .. `z`) are deleted.
3. The normalized word is ignored if it is empty or is a noise word.

The words resulting from preprocessing a search line are the search terms. Matching documents are those whose preprocessed contents contain one-or-more occurrences of at least one search term. Note that the order of the search terms in a search line do **not** matter; i.e. we are searching for each search term independently and not attempting any kind of phrase search.

Specifically, you are required to submit a `submit/prj1-sol` directory on your gitlab repository such that changing into this directory and running the command `npm install` will install all necessary external dependencies for your project (there will probably be no external dependencies for this project). It should then be possible to run the `index.js` file in that directory with two or more arguments:

NOISE_WORDS Specifies the name of a file which contains whitespace-separated noise words which should be ignored for searches.

DOC ... Specifies one-or-more files each of which constitutes a document to be searched.

When the program starts up, it should read the files specified by its arguments into memory. It should then go into an interactive loop:

1. Output a `'> '` prompt.
2. Read a line from the user.
3. Search all content documents for occurrences of the whitespace-separated search terms in the line read from the user.
4. Output a line summarizing the time take for the search in the form `time:` *mmm* `milliseconds`.
5. If no matching documents are found, output a line `no results`.
6. If matching documents are found, for each document which contains one or more of the search terms, output a result formatted as follows:
   (a) A single line containing:
      i. The name of the document. This is derived from its file name by removing any directory prefixes and `.txt` extension.
      ii. A single space.
      iii. The **score** for the document. This is the sum of the number of occurrences of each search term in the document.
   (b) One-or-more lines from the document which contain the **earliest** occurrence of each matching search term. These lines should occur in the same relative order as in the document. A line should occur only once even if it contains multiple search terms.

The match results should be ordered in non-ascending order by score. Results with the same score should be ordered in lexicographically increasing order by document name.

It should be possible for the user to type tab characters to request completion of the current search term while typing in a search line. The exact dynamic behavior for tab completion is illustrated in a *dynamic log*.

Other miscellaneous requirements:

- A non-functional requirement is that the cost of a search usually depends only on the number of search terms and not on the number of words in the documents. This means that it is not possible to read all the documents for each search.
- Your project **must** implement the API documented for the `DocFinder class` mentioned below.
- There may be available packages which implement the functionality required for this project. The use such packages is strictly forbidden.

1.3 Example Logs

The behavior of the program is illustrated by the following annotated static log:

```
prj1-sol> $ DATA=$HOME/cs580w/data #set up data dir
prj1-sol> $ GBS=$DATA/corpus/gbs   #set up contents dir

prj1-sol> $ ./index.js #show usage message
usage: index.js NOISE_WORDS DOC...

#start application
prj1-sol> $ ./index.js $DATA/noise-words.txt $GBS/*.txt

#application outputs a '> ' prompt; user types in a search.
> superman  #single word search
time: 2 milliseconds
man-and-superman: 14
The Project Gutenberg EBook of Man And Superman, by George Bernard Shaw
back-to-methuselah: 9
of Man and Superman. As it was, she knew quite well what he wanted; for
misalliance: 2
TARLETON.  Still, you know, the superman may come.  The superman's an
the-doctors-dilemma: 1
a Superman; but still, it's an ideal that I strive towards just as any

#search for two words; note ordering of results having the same score.
> flower girl
```

```
time: 5 milliseconds
pygmalion: 234
wrought by Professor Higgins in the flower girl is neither impossible
mrs-warrens-profession: 26
Parisian girl in Brieux's Les Avaries come on the stage and drive into
back-to-methuselah: 25
HASLAM. Silly girl! Going to marry a village woodman and live in a hovel
settles again on the flower, and repeats the performance every time the
caesar-and-cleopatra: 21
BELZANOR. Command! A girl of sixteen! Not we. At Memphis ye deem her a
ladies, all in their gayest attire, are like a flower garden. The facade
man-and-superman: 21
she's a wonderfully dutiful girl. Her father's wish would be sacred to
confection by the apron and feathers of a flower girl, strike all the
the-devils-disciple: 21
alone. A girl of sixteen or seventeen has fallen asleep on it. She is a
misalliance: 18
because my mother thinks a girl should know what a man is like in the
you off into the flower bed, and then lighted beside you like a bird.
heartbreak-house: 11
girl, slender, fair, and intelligent looking, nicely but not expensively
the-doctors-dilemma: 9
been talking to that poor girl. It's her husband; and she thinks it's
major-barbara: 8
soul like a flower at a street corner; but she bought it for
girl.
getting-married: 7
every flower and change every hour, as their fancy may dictate, in
girl's chances by making her a slave to sick or selfish parents, its
the-philanderer: 6
her sister--a girl under twenty--spending half their time at such a
candida: 3
lake for bathers, flower beds with the flowers arranged carefully in
at one side, a miniature chair for a boy or girl on the other, a nicely
how-he-lied-to-her-husband: 2
sweet enough, is it not? [He takes the flower from the table]. Here are
the-man-of-destiny: 2
and eating it uncooked. Nevertheless one girl of bad character, in whom

> prj1-sol> $ #typed Ctrl-D to terminate program
```

The completion behavior of the program is illustrated by the following *dynamic log*.

1.4 Provided Files

The prj1-sol directory contains a start for your project. It contains the following files:

doc-finder.js This skeleton file constitutes the guts of your project. You will need to flesh out the skeleton, adding code as per the documentation. You should feel free to add any auxiliary function or method definitions as required.

index.js This file provides the complete command-line behavior which is required by your program. It requires doc-finder.js. You **must not** modify this file; this ensures that your `DocFinder` meets its specifications and facilitates automated testing by testing only the `DocFinder` API.

README A README file which must be submitted along with your project. It contains an initial header which you must complete (replace the dummy entries with your name, B-number and email address at which you would like to receive project-related email). After the header you may include any content which you would like read during the grading of your project.

Additionally, the course data directory contains data which can be used for testing your project. Specifically, it contains:

Noise Words A file `noise-words.txt` containing noise words.

"Real" Documents Some George Bernard Shaw plays which can be searched.

Test Documents Short documents which may be useful during development and testing.

1.5 Hints

The following steps are not prescriptive in that you may choose to ignore them as long as you meet all project requirements.

1. Read the project requirements thoroughly. Look at the sample logs to make sure you understand the necessary behavior.
2. Look into debugging methods for your project. Possibilities include:
   - Logging debugging information onto the terminal using console.log() or console.error().
   - Use the chrome debugger as outlined in this article.
3. Consider what kind of indexing structure you will need to represent the documents. For each unique non-noise normalized word in all the documents you will need to track:
   - The documents the word occurs in.

- For each document the word occurs in, track the number occurrences of the word in that document.
- For each document the word occurs in, track the smallest offset of the word in that document. Note that given the offset of a word occurrence, you can recover the line for that occurrence by simply looking for the last newline before the offset and the first newline after the offset (this assumes that you have stored the contents of the entire document).

When designing your indexing structure, you should think in terms of the abstract data structures **Maps** and **Sets**. Possibilities for implementing these data structures in JavaScript include:

- For implementing sets, use the new Set addition to the standard library. It would work but has the disadvantage that JSON conversion (as may be necessary for future projects) requires some work.
- Adapt JavaScript Array's to implement sets. The API may not be particularly set-like and you will need to take care to avoid duplicates, but JSON conversion is trivial.
- For implementing maps, use the new Map addition to the standard library. An advantage of this approach is that it preserves insertion order. It may also be lighter than the `Object` alternative suggested below. Disadvantages include a clumsy API (`get()` and `set()`) and non-trivial work required for JSON conversion.
- Use standard JavaScript objects as maps. Advantages include the ability to using simple `[]`-based access and trivial JSON conversion. Disadvantages include the lack of any defined order for property keys and the overhead of all the machinery associated with objects like inheritance.

4. Start your project by creating a `work/prj1-sol` directory. Change into that directory and initialize your project by running `npm init -y`. This will create a `package.json` file; this file should be committed to your repository.

5. Copy the provided files into your project directory:

   ```
   $ cp -p $HOME/cs580w/projects/prj1/prj1-sol/* .
   ```

   This should copy in the `README` template, the `index.js` and the `doc-finder.js` skeleton file into your project directory.

6. You should be able to run the project:

   ```
   $ data=$HOME/cs580w/data
   $ gbs=$data/corpus/gbs
   $ tst=$data/corpus/tests
   $ ./index.js #show usage message
   ```

```
$ ./index.js $data/noise-words.txt $tst/*.txt
```

You should be able to type in search terms but all the searches will fail without any results since the code in your `doc-finder.js` file is a NOP.

7. Replace the `XXX` entries in the `README` template. Commit your project to gitlab:

```
$ git add .
$ git commit -a -m 'started prj1'
```

8. Open the copy of the `doc-finder.js` file in your project directory. It is set up as follows:

   (a) Preliminaries: Importing of a inspect() function; note the use of the destructuring syntax. A `strict` declaration.

   (b) A skeleton `DocFinder class` definition with `@TODO` comments. Note that the `class` syntax is a recent addition to JavaScript and is syntactic sugar around JavaScript's more flexible object model. It was used in this project because JavaScript's object model has not yet been covered in the course. Note that even though the use of this `class` syntax may make students with a background in class-based OOPL's feel more comfortable, there are some differences worth pointing out:

   - No data members can be defined within the `class` body. All "instance variables" must be referenced through the `this` pseudo-variable in both `constructor` and methods. For example, if we want to initialize an instance variable `indexes` in the `constructor()` we may have a statement like:

```
this.indexes = new Map();
```

   - There is no implicit `this`. If an instance method needs to call another instance method of the same object, the method **must** be called through `this`.
   - There is no easy way to get private methods or data. Instead a convention which is often used is to prefix private names with something like a underscore and trust `class` clients to not misuse those names.

   (c) An assignment of the `DocFinder class` to `module.exports`. This makes `DocFinder` the only declarations available outside the `doc-¬finder.js` file; all other declarations are local to the file.

   (d) Miscellaneous top-level definitions including complete code for word normalization and a definition of a `Result` class for returning search results. These definitions can be used in the rest of the file, including within methods in the `DocFinder class`. Having definitions like the `const WORD_REGEX` outside the `class` is necessary since the `class`

syntactic sugar does not allow such a definition within the `class` body.

9. Decide on the data members you will need for your `DocFinder` instance and initialize them within the `constructor()` through `this`.

10. Start out by thinking about how to implement the `words()` method. Some thoughts:

    - You need to split `content` on whitespace and normalize the words. So something like

      ```
      content.split(/\s+/).
        map((w) => normalize(w)).
        filter((w) => !is_noise_word(w))
      ```

      seems to work (the regex `/\s+/` matches one-or-more whitespace characters; `is_noise_word()` is pseudo-code).

    - Even though the above meets the specification for the `words()` function, a bit more thought reveals that usually when we get the words from document content we will also need the offset of each word within the content. So it may be better to have another "private" method `_wordsLow()` which returns a list of pairs: `[` *word*`,` *offset* `]` and define `words()` as a simple wrapper around `_wordsLow()`; something like:

      ```
      this._wordsLow(content).map(pair => pair[0])
      ```

    - When implementing `_wordsLow()`, the `String` `split()` method may not be the best choice as obtaining the offsets of words will be extremely clumsy. Instead, a regex match would work with code which uses a *regex exec()*:

      ```
      let match;
      while (match = WORD_REGEX.exec(content)) {
        const [word, offset] = [match[0], match.index];
        //do stuff with word and offset
      }
      ```

11. Use `words()` to implement `addNoiseWords()`. A set is an ideal data structure for the noise words. You can simply use `forEach()` on `words()` to add each word to a noise words set.

    At this point you can test your code by running your program with a noise words file and verifying (using console or chrome debugging) that the noise words have been added correctly.

12. Start getting to the guts of your project by implementing the `addContent¬ ()` method. You will need to add each of the non-noise normalized words

in the document content to your indexing structure. The actual code will depend on the design of your indexing structure. Feel free to define "private" helper methods.

13. Implement the `find()` method. Again, the details will depend on the design of your indexing structure. At this point, you have implemented the main functionality of the project.

14. Implement the `complete()` method. Again, you will need some kind of indexing structure for all the words in your document collection. You can simply choose to index off the first character of each word. Instead of updating this index structure whenever a new word is added, you can do it lazily in a batch mode when it is needed (i.e. when there is a call to `complete()`) and it is not up-to-date; simply recompute the entire completion indexing structure.

15. Iterate until you meet all requirements.

It is a good idea to commit and push your project periodically whenever you have made significant changes. When it is complete please follow the procedure given in the *gitlab setup directions* to submit your project using the `submit` directory.