Project 1: **Search Engine**
COMP 475 – Web Science
100 points

You are to create a web search engine that works at the command line. To do this, you will write two Python scripts, indexer.py and search.py. Indexer.py should do the following:

1. After performing a crawl (using your other Python script), read all the HTML files that were stored in the "pages" directory. For each document, extract the title and the text from the body of the page (read the Beautiful Soup documentation to find out how). Beautiful Soup will include the text of the page in the content of the page, and that is OK. Beautiful Soup may also break on some pages and include HTML as text, but we will not worry about these exceptions or bugs.

2. All text should be converted to lowercase and non-alphanumeric characters should be ignored. So "123-456" would become "123" and "456", and "joe@yahoo.com" would become "joe", "yahoo", "com". Ignore the following stop words: a, an, and, are, as, at, be, by, for, from, has, he, in, is, it, its, of, on, that, the, to, was, were, will, with. Do not perform stemming.

3. A single inverted index should be created for the document corpus which maintains the document ID (numbered 1…*n* in order of the pages found in the "pages" directory), a 1 or 0 if the text is found in the title, and the term frequency from the body (normalized by the total number of tokens in the document *after* removing stop words).

4. After indexer.py has finished indexing all the web pages, it should output the index to **index.dat** which looks like this:

```
arkansas
   6 0 0.022
model
   1 0 0.309
   3 0 0.015
   5 1 0.001
tuesday
   2 0 0.082
white
   2 1 0.018
etc…
```

   Note that the indexed words are alphabetized, and there are 3 spaces before sets of three numbers (each separated by a single space) which are: doc ID, title (0 or 1), and normalized body TF (rounded to 3 decimal places). For example, the term *white* was found only in document 2; it was somewhere in the title and made up 1.8% of all the words in the document.

5. It may take some time for your program to run, so you should output information about the program's status as it indexes the crawled pages. Outputting what file is being worked on would be helpful to the user who is waiting for the program to finish its work.

After the index is written to index.dat, the search.py script will allow the user to search the corpus for specific words. Here is how it should operate:

1. First, read the search phrase at the command line. Examples:

```
> search.py bisons
> search.py "harding university"
```

If no command line argument is supplied, the program should tell the user a search term is required and terminate. Ignore any command-line arguments after the first.

2. Next, the program should read the index from index.dat into memory. Note that you may want to use similar data structures used in indexer.py, so you should write your programs in a way where you share code without having redundant code in each script. (It's OK to introduce new .py files to your project.)

3. For simplicity, all queries will be assumed to use boolean ANDs, and we will not implement phrase search. For example, the query *harding university* is should generate a boolean search for *harding AND university*, so only documents containing both terms should be considered a match.

4. Remove any stop words from the query as was done when indexing the documents.

5. After determining which documents match the search terms, calculate the relevancy score for each document:

   relevancy score = 0.9 * body TF + 0.1 * title score

   Do this for each term, and compute the average relevancy score for all terms. So if the search was for *harding university*, you would compute the score for *harding* and the score for *university* and compute the average to determine the overall relevancy score.

6. The total number of results should first be displayed. Then display every document ID and score (out to 3 decimal places) ordered by score, and number the results. Example:

```
Results: 4
1. docID 3, score 0.830
2. docID 1, score 0.814
3. docID 5, score 0.350
4. docID 8, score 0.108
```

Bonus: You can receive 5 bonus points by implementing phrase search. So when the user searches for "harding university", assume they want only documents with that exact phrase. To accomplish this, you will need to store the positions of the terms that are stored in the inverted index. Then use those positions to ensure the phrase matches successive positions.

Zip your entire project directory and submit it to Easel before it is due. Make sure your output matches the specifications *precisely* to avoid losing any points. If you use any code you find in the Web, you *must* document the source in your program.

Test Data

a.html

```
<title>cool!!! test!!!</title>
<body>
this 123-456.
</body>
```

b.html

```
<html>
<head>
<title>Go Cowboys!</title>
```

```
</head>
<body>
And another test and test!
</body>
</html>
```

c.html

```
<body>
This is a test.
</body>
```

```
Inverted index:
123
   1 0 0.200
456
   1 0 0.200
another
   3 0 0.200
cool
   1 1 0.200
cowboys
   3 1 0.200
go
   3 1 0.200
test
   1 1 0.200
   2 0 0.500
   3 0 0.400
this
   1 0 0.200
   2 0 0.500
```

Search for "test this" results in the following:

```
Results: 2
1. docID 2, score 0.450
2. docID 1, score 0.230
```

Search for "test cowboys go" results in the following:

```
Results: 1
1. docID 3, score 0.310
```

Search for "cool cowboys" results in the following:

```
Results: 0
```