Mini Search Engine

COSC 101, Fall 2014
Homework 12: the final episode
Due: Thursday, December 11, 11:55pm

1 Introduction

The last homework of the semester is a larger project to build a scaled-down Web search engine. There are three stages required for this homework for building your search engine:

Crawling and indexing. For this stage, you will write functions for traversing a set of connected web pages, and build Python data structures that model the words found on the pages (a "reverse index") and the connectivity structure of the pages.

Computing page rankings. With the connectivity information generated from the previous stage, you will use an idea developed by the founders of Google to compute "rankings" of web pages, in order to provide relevant results to search queries.

Accepting and processing search queries. Lastly, using the various data structures generated from the first two stages, you will accept search queries and print out search results ordered from most relevant to least relevant.

Each of these stages is described in detail below. For each stage, there are one or more Python functions that you will need to implement. **Some of these functions will be written during class.** The specific functions to be written in class are identified below so you can focus your efforts on writing the other functions.

For additional background on how search engines work, you are encouraged to read Chapters 1 and 2 of "Nine Algorithms That Changed the Future" by John MacCormick, available electronically through the Colgate library. This reading is entirely *optional*. It is a fun and easy read.

Because this homework requires an above average effort, you have more time to work on it than the usual weekly homework and it will be weighted more heavily, equivalent to 1.5 homework assignments.

2 Crawling and indexing

The first stage of this homework is to write functions to *crawl* and *index* a set of web pages. Crawling and indexing are two key activities that Web search engines perform in order to build the data structures used to answer search queries. There are two specific goals for this stage.

1. Build a Python dictionary that maps words found on web pages, to a list of web page names on which those words appear. This mapping is called a "reverse index".
2. Build a Python dictionary that maps web page names to a list of web pages to which a given page has links. This dictionary will essentially model the connectivity structure of the web pages you crawl, and is referred to as the "web graph".

There are 7 functions to write for this stage of the homework: `main`, `download_page`, `extract_links`, `remove_tags`, `normalize_word`, `index_page`, and `crawl_web`.

2.1 Function 0: `main`

Create a `main` function that takes no parameters and simply prints `Welcome to my mini search engine!` We will add more code to the main function as we continue.

2.2 Function 1: `download_page`

This function should accept a string parameter `pagename`, which should be a web page name of the form `"a.html"`, where `"a"` may be replaced by any alphabetic string. The function should retrieve and return the full contents of the web page with the full URL "`http://cs.colgate.edu/cosc101/testweb/`" + pagename.

To retrieve the full text of a web page as a string you can use the `urlopen` function in the `urllib2` module. For example, if the `pagename` is `"a.html"`, you should prepend the string `http://cs.colgate.edu/cosc101/testweb/` to the beginning to construct the full URL, as in:

```
fullurl = "http://cs.colgate.edu/cosc101/testweb/" + pagename
fileobject = urllib2.urlopen(fullurl)
```

The `urllib2.urlopen` function returns an object that behaves as a standard text file object: you can read from it using all the standard ways that you can read from a text file. The `download_page` function should just read *all* the text of the page as a string, and return the complete string.

2.3 Function 2: `extract_links`

This function should accept one parameter, which should be the full text of a web page as a string (*i.e.*, the result of calling `download_page`), and return a list of strings that refer to web page names that are linked from the current page.

Web pages are written using a language called HTML (HyperText Markup Language) and stored in plain text files[1]. Different "tags" in a web page are used to indicate different structural elements of a page, for example the title, an item in a bulleted list, or a table. Links on a web page that can be clicked on to "go to" another page are represented in HTML with <a> tags, also known as "address" tags. For this function we are just looking for address tags in a particular format.

The <a> tags to find on a page will have the following format:

```
<a href="a.html">
```

[1]See `http://www.w3.org/community/webed/wiki/HTML` if you'd like to learn more about HTML.

where the `a.html` component may change, depending on the name of the page linked to the current page. To be precise, a valid address tag always starts with `<a href="` and ends with `">` and the text between the quote marks is the link to extract. It is **not** sufficient to just look for `".html"`, you must explicitly look for address tags.

Your job for this function is to find all the address tags, and return a list of all the page names referred to by those tags.

While you will eventually use this function to process text from web pages, you should start by *testing* your function on small examples to make sure it works.

For example, this string has just one link to extract:

```
>>> extract_links('abc <a href="a.html">xyz</a> def')
['a.html']
```

This string has two links:

```
>>> extract_links('A <a href="one.html">B</a> C <a href="two.html">D</a>')
['one.html', 'two.html']
```

This string has three links:

```
>>> page = '<a href="a.html">A</a><a href="b.html">B</a><a href="c.html">C</a>'
>>> extract_links(page)
['a.html','b.html','c.html']
```

In this example, `'a.html'` should **not** be extracted because `<a blah="` is not a valid start to an address tag.

```
>>> extract_links('abc <a blah="a.html">xyz</a>')
[]
```

In this example, `'one.html'` should **not** be extracted because it is not contained within an address tag:

```
>>> extract_links('A one.html <a href="two.html">D</a>')
['two.html']
```

In this example, `'two.html'` should **not** be extracted because it's a `<b>` tag not an `<a>` tag.

```
>>> extract_links('<b href="two.html">D</b>')
[]
```

You can assume that whenever you find the start of an address tag, it will always be followed by the end. So, for instance, you won't see something like this: `'<a href="a.html without end quote mark!'`

You can test your function even further by testing on real pages. First, call `download_page` to download the page `c.html` and save the result to a variable `text_c` and then call `extract_links` on `text_c`. If you do this, `extract_links` should return `['f.html', 'e.html']`. If you do the same on `a.html`, you should get `['c.html', 'd.html', 'b.html']`.

2.4 Function 3: `remove_tags`

This function should accept one parameter, which should be the full text of a web page as a string (*i.e.*, the result of calling `download_page`), and return a string that is the full text of the web page after HTML tags have been removed.

All tags start with the symbol `'<'` and end with the symbol `'>'`. Since strings are immmutable, you cannot actually "remove" a tag. The recommended approach is to make a copy of the text but *exclude* any text that falls between the characters `'<'` and `'>'`. In other words, copy each character in the string, except whenever the character `'<'` is encountered, don't copy it, find the next `'>'` character, and skip ahead and resume copying with the character following the `'>'`. Hint: the key ingredients are the find method on strings, a while loop, and possibly string slicing.

Again, you will eventually use this function to process text from web pages, but you should start by *testing* your function on small examples to make sure it works. Here are some examples.

```
>>> remove_tags('<b>abc</b>')
'abc'
>>> remove_tags('A<b>B</b>C')
'ABC'
>>> remove_tags('abc <a href="a.html">xyz</a>')
'abc xyz'
```

2.5 Function 4: `normalize_word`

This function should accept a string as a parameter, and return a new string. The function should remove any non-alphabetic characters from the string, and convert all letters to lower-case. Hint: you can re-use code from past lab assignments for this. Here are some examples for testing:

```
>>> normalize_word("That's")
'thats'
>>> normalize_word('NONE!')
'none'
>>> normalize_word('Hello, goodBYE!')
'hellogoodbye'
```

2.6 Function 5: `index_page`

This function should accept three parameters: a web page name (like `"a.html"`), the full contents of that page as a string, and a Python dictionary that stores a structure called a "reverse index". This function should *modify the reverse index dictionary* **in place**, and not return anything.

The reverse index dictionary simply maps words to lists of web pages on which those words are found. For example, if we built a reverse index by processing the two real web pages, `a.html` and `b.html`, the dictionary would contain these entries:

```
{
  'a': ['a.html', 'b.html'],
  'about': ['b.html'],
  'accustomed': ['b.html']
  'achiote': ['b.html']
  ...  and so on ...
}
```

This dictionary tells us that "a" occurs on both pages, but "about," "accustomed," and "achiote" only occur on `b.html`.

When processing the text of a web page, you should first remove all tags (by calling `remove_tags`). Then break up the resulting text into words and normalize each word (by calling `normalize_word`) before adding it to the dictionary (thus all keys of the dictionary will be alphabetic strings in lower-case). Remember that the `normalize_word` function may return an empty string if the word does not contain any alphabetical letters (e.g., a number). You should avoid adding the empty string to the reverse index. Also, if a word occurs twice on a web page, be careful not to add two occurrences of the web page to the list of pages.

Note: this function does **not** call `download_page`. As explained later, the `crawl_web` function will first call `download_page` and then pass the result of `download_page` as an argument to `index_page`.

Again, you can test this function with small examples that do not have to be real web pages. In this example, the web pages `'fake.html'` and `'fake2.html'` are, well, fake.

```
>>> myindex = {}
>>> index_page('fake.html', '<b>I</b> HearT "CHOCOlate!" chocolate', myindex)
>>> myindex
{'i': ['fake.html'], 'heart': ['fake.html'], 'chocolate': ['fake.html']}
>>> index_page('fake2.html', '123 <a href="a.html">chocolate</a>', myindex)
>>> myindex
{'i': ['fake.html'], 'heart': ['fake.html'], 'chocolate': ['fake.html', 'fake2.html']}
```

Notice a few things about this example: the index contains normalized words, "123" which normalizes to the empty string is not included, the word "chocolate" appears twice in the text of `'fake1.html'` but the string `'fake1.html'` only appears once in the index, the dictionary `myindex` is modified by `index_page` each time the function is called.

2.7 Function 6: `crawl_web`

We will write this function in class.

This function should take three parameters: a web page name (like `"a.html"`), a Python dictionary that stores a "reverse index" that maps words to lists of pages on which those words are found, and a Python dictionary that stores a representation of the web pages discovered, and which pages they are linked to (the "web graph"). This function should *recursively* "crawl" the web pages and modify the two dictionaries *in place*. It should not return anything. (Note: this is the **only** function that should be written recursively in this assignment.)

At this point, you will also want to add some lines to your `main` function. First, initialize two empty dictionaries, say `reverse_index` and `web_graph`. Then call `crawl_web` with the initial page `"a.html"`, and the two dictionaries. As the `crawl_web` function does its work, the two dictionaries will get populated with information.

Here's what this function must do, in pseudocode:

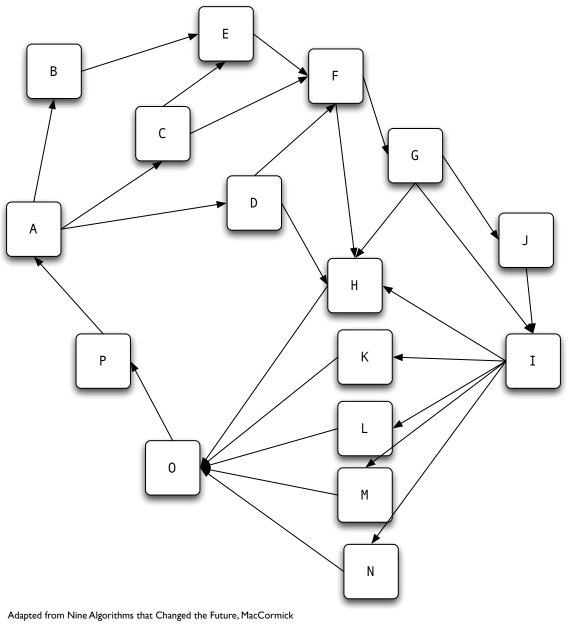
```
#
# pseudocode for crawl_web function
#
def crawl_web(pagename, reverse_index, webgraph):
    # 1) get contents of page named pagename using download_page
    # 2) extract links from page using extract_links
    # 3) index the page using index_page
    # 4) add the current page and links from this page to the
    #    webgraph dictionary
    # 5) for each page in the list of extracted links that we
    #    have not already visited, recursively call crawl_web
```

The first three steps should be fairly straightforward, given the functions you have already written in this stage.

The web graph dictionary built through the fourth step should simply have web page names as keys, and the value corresponding to each key should be the list of links available on that page. For the web pages available starting at `http://cs.colgate.edu/cosc101/testweb/a.html`, the *full* dictionary that you construct through the `crawl_web` function should be as follows:

```
{
  'j.html': ['i.html'], 'n.html': ['o.html'],
  'c.html': ['f.html', 'e.html'], 'l.html': ['o.html'],
  'b.html': ['e.html'], 'p.html': ['a.html'],
  'f.html': ['g.html', 'h.html'], 'd.html': ['f.html', 'h.html'],
  'i.html': ['h.html', 'k.html', 'l.html', 'n.html', 'm.html'],
  'o.html': ['p.html'], 'h.html': ['o.html'], 'm.html': ['o.html'],
  'a.html': ['c.html', 'd.html', 'b.html'],
  'g.html': ['i.html', 'j.html', 'h.html'], 'e.html': ['f.html'],
  'k.html': ['o.html']
}
```

Another way to view the collection of these pages is shown in the following diagram:


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Adapted from Nine Algorithms that Changed the Future, MacCormick

In this figure, arrows from one page name to another represent the fact that a page has a link to another page. For example, the page `"a.html"` has links to the pages `"b.html"`, `"c.html"`, and `"d.html"`.

The fifth step in `crawl_web` is where the term *web crawler* comes from: your program should behave like a web browser and *recursively* traverse the entire link structure shown in the figure above.[2]

3 Computing page rankings

The goal of the second stage of this homework is to build a Python dictionary that stores page ranks. The "page rank" dictionary maps a set of web page names to numeric rankings of those pages[3]:. A higher rank (higher numeric value) will mean that a page is more "relevant" when responding to search queries. Likewise, a smaller numeric rank will imply a page is of less relevance. **There is one function to write for this stage: `random_surfer_simulation`.**

To compute the web page rankings, you will just use the "web graph" dictionary you built in the previous stage. The "reverse index" is *not* used in this stage.

3.1 Function 7: `random_surfer_simulation`

To compute the PageRank of each web page, you will write a function to perform a series of "random web surfer" simulations to identify the key, authoritative pages[4]. The surfer simulation performs the following tasks.

1. Decide how many pages to visit for the entire simulation (for example, 100,000).
2. Pick a random page from all possible pages.
3. With probability $(1 - p)$, pick a random link from the current page and follow it. With probability p, pick a new random page. (In the original description of this algorithm, p was set at 0.15, so with 0.85 probability you choose a link from the current page, and with 0.15 probability you choose a new page. You can use the `random.random()` function to get a random floating point number between 0 and 1 to help with this process.)
4. Keep a count of every page visited using a Python dictionary. The dictionary should store web page names as keys and visit counts as values. At the end of the simulation, the ranking of each page is just its visit count divided by the total number of pages visited (e.g., 100,000).

Note that in the above simulation you are *not* actually requesting web pages using the `urllib2` module. In other words, you should **not** be calling `download_page` here. You will simply *simulate* following links by retrieving web page names and links from the web graph dictionary you built as part of the `crawl_web` function.

[2]The Wikipedia page on web crawling has some interesting detail on how real crawlers work: `http://en.wikipedia.org/wiki/Web_crawler`.

[3]The technique to compute the page rankings is actually *the* PageRank algorithm as described in a paper by Larry Page and Sergey Brin, the co-founders of Google: *The anatomy of a large-scale hypertextual Web search engine*, S. Brin and L. Page, *Computer Networks and ISDN Systems*, Vol. 30, No. 1, pp. 107–117, 1998. The paper is quite accessible to read and is available at `http://ilpubs.stanford.edu:8090/361/1/1998-8.pdf`.

[4]This is basically what the Google PageRank algorithm does, although the canonical Google implementation uses linear algebraic techniques rather than the simulation technique we will use.

Write a function called `random_surfer_simulation` that performs the above tasks. The function should take three parameters:

1. a dictionary (key is a string, value is a list of strings) representing the web graph dictionary
2. the parameter p, a float
3. the number of simulations, an int

It should return a new Python dictionary that contains web page names as the keys, and computed PageRanks as the values. Do **not** write this function recursively.

Remember that the web graph is just a dictionary, so for testing purposes, you can make up your own web graph by simply creating a dictionary with the names of fake web pages. In this example, we create a web graph dictionary that has only three pages: `fake1.html` has links to `fake2.html` and `fake3.html` and both `fake2.html` and `fake3.html` link back to `fake1.html`.

If you call your function with `p` equal to 0, then you should expect the rank of `fake1.html` to be around $\frac{1}{2}$ and the other two pages to have rank around $\frac{1}{4}$. (Your numbers may differ slightly.)

```
>>> web_graph = {'fake1.html': ['fake2.html', 'fake3.html'],
                 'fake2.html': ['fake1.html'],
                 'fake3.html': ['fake1.html']}
>>> random_surfer_simulation(web_graph, 0, 100000)
{'fake1.html': 0.5, 'fake2.html': 0.24903, 'fake3.html': 0.25097}
```

On the other hand, if `p` is equal to 1, then the simulation is choosing pages uniformly at random (i.e., links are ignored completely). In this case, all three pages should have rank around $\frac{1}{3}$.

```
>>> random_surfer_simulation(web_graph, 1, 100000)
{'fake1.html': 0.33258, 'fake2.html': 0.33357, 'fake3.html': 0.33385}
```

Modify your `main` function so that it now calls `random_surfer_simulation`. If you run the simulation using a web graph obtained from crawling the test web (`http://cs.colgate.edu/cosc101/testweb/`), and set `p` equal to 0.15 and run 100,000 simulations, the dictionary you compute and return from this function should look roughly like this:

```
 { 'j.html': 0.02431, 'n.html': 0.01742, 'c.html': 0.04367, 'l.html': 0.01709,
   'b.html': 0.04424, 'p.html': 0.13597, 'f.html': 0.10279, 'd.html': 0.04454,
   'i.html': 0.04559, 'o.html': 0.1499,  'h.html': 0.09572, 'm.html': 0.01749,
   'a.html': 0.12454, 'g.html': 0.05342, 'e.html': 0.06592, 'k.html': 0.01739 }
```

4 Accepting and processing search queries

The last stage of this homework is to use the reverse index and page rank dictionaries that were constructed as part of the `crawl_web` and `random_surfer_simulation` functions and respond to search queries with results in ranked order. **There are 7 functions to write for this stage**. Don't fret: some functions should be very short and we will write several of them in class.

The key function in this section is `search_engine`. This function is described at the end. Since `search_engine` performs a complex task, it relies on many helper functions to take care of different aspects of the process. All of the functions leading up to `search_engine` will ultimately be called directly (on indirectly) in the body of the `search_engine` function.

4.1 List processing functions

These three functions are general purpose functions that could be used on lists of any kind. They will be used later to process the results of a search query. **We will write these functions in class.**

4.1.1 Function 8: `list_union`

The function accepts two lists and returns a new list that contains all items that are found in either of the lists, but eliminating any duplicates. The elements in the resulting list can be in any order.

```
>>> list_union(['a.html', 'c.html'], ['a.html', 'b.html'])
['a.html', 'b.html', 'c.html']
```

4.1.2 Function 9: `list_intersection`

The function accepts two lists and returns a new list that contains only those items that are found in both of the lists. The elements in the resulting list can be in any order.

```
>>> list_intersection(['a.html', 'c.html'], ['a.html', 'b.html'])
['a.html']
```

4.1.3 Function 10: `list_difference`

The function accepts two lists and returns a new list that contains only those items that are found in the first list but not in the second list. The elements in the resulting list can be in any order.

```
>>> list_difference(['a.html', 'c.html'], ['a.html', 'b.html'])
['c.html']
```

4.2 Function 11: `get_query_hits`

This function should accept a *single* search term (exactly one word) and the reverse index dictionary as parameters. It should return a list of all the page names that match the search term. Notice that this list should just be the *value* corresponding to a given *key* in the reverse index dictionary. If no page contains the search term, it should return an empty list. In other words, if the the key is not present, then return an empty list. This function should be very short.

Here is a simple example on a "fake" reverse index.

```
>>> reverse_index = {'dog': ['fake.html']}
>>> get_query_hits('dog', reverse_index)
['fake.html']
>>> get_query_hits('cat', reverse_index)
[]
```

If you call your function passing in the *real* reverse index built from crawling the web, the search term "cacao" should return the list:

```
['a.html', 'f.html', 'h.html', 'o.html', 'e.html', 'd.html', 'b.html']
```

4.3 Function 12: `process_query`

We will write this function in class.

This function should accept two parameters: a query string and the reverse index dictionary. It should return a list in *any* order of all the page names matching the given query. This function should be called from the `search_engine` function.

Search queries should be formed from one or more words. You should treat all words in a query in the same way that you treat words found on a web page (i.e., in a case-insensitive manner, and excluding non-alphabetic characters). You can use the `normalize_word` function you wrote in the first stage to help with this process.

There are certain rules you will need to follow for responding to queries, and two special syntaxes that affect how queries should be processed:

- Generally speaking, for queries that are formed from one or more words, the search results should include pages that match **any** of the terms. In other words, the results of the entire query should be the *union* of the results from each of the search terms.
- However, a user can use the special keyword `AND` as the *first* term in the search query. If the search query starts with `AND`, the search results should include pages that match **all** terms. In other words, the results of a query should be the *intersection* of search results from each of the terms. Please note, the word `AND` is a keyword and should **not** be treated like a search term.
- A user can prepend a minus sign (hyphen) as in `-cocoa` to a word to indicate that the search results should not include any pages that include the given term. Terms with a minus sign can appear anywhere in the search query.
- You should process words in a query *left to right*, separating them into two lists: include and exclude. The exclude list contains all of the terms that have a minus sign (be sure to remove the minus sign at this point). The include list contains all of the terms that do not have a minus sign. Process the include list first, being sure to either take the *union* or *intersection* depending on whether the search query started with the keyword `AND`. Now process the exclude list, building up a list of all pages that contains *any* of the terms in the exclude list. At this point, you should have two sets of results, one for the include and one for the exclude. Now remove all pages from the include results that also appear in the exclude results (using the `list_difference` function defined earlier).

For testing purposes, here are some examples using a fake reverse index. Note: each of these examples is testing a different requirement, so be sure to try all of them on your program.

```
>>> index = {'cat' : ['fake1.html','fake3.html'],
             'dog' : ['fake1.html','fake2.html','fake3.html'],
             'android'   : ['fake1.html']}
>>> process_query('CAT dOg!!', index)            # normalize
['fake1.html', 'fake2.html', 'fake3.html']
>>> process_query('AND cat dog', index)          # AND keyword
['fake1.html', 'fake3.html']
>>> process_query('dog cat -android', index)     # minus term
['fake2.html', 'fake3.html']
>>> process_query('AND cat dog -android', index) # AND keyword with minus
```

```
['fake3.html']
>>> process_query('dog -cat -android', index)    # multiple minus terms
['fake2.html']
>>> process_query('AND dog banana', index)       # non-matching term
[]
>>> process_query('ANDroid cat', index)          # starts with AND but not AND keyword
['fake1.html', 'fake3.html']
```

Some examples of queries on the real reverse index are given in Section 5.

4.4 Function 13: `print_ranked_results`

This function should accept a list of query results (a list of page name strings in any order), and the page ranks dictionary. It should print out the query results in ranked order, from highest to lowest rank, and not return anything. If the list of query results is empty, it should simply print "No matches for search terms."

Here are some examples using a fake results list and a fake page rank dictionary. In the first example, every page appears in results and so they are all displayed, ordered by page rank from largest to smallest.

```
>>> page_rank = {'fake1.html': 0.26, 'fake2.html': 0.24, 'fake3.html': 0.5}
>>> results = ['fake3.html', 'fake1.html', 'fake2.html']
>>> print_ranked_results(results, page_rank)
1: fake3.html (rank: 0.5)
2: fake1.html (rank: 0.26)
3: fake2.html (rank: 0.24)
```

Here is an example where the results list contains only two pages, `'fake1.html'` and `'fake2.html'`.

```
>>> page_rank = {'fake1.html': 0.26, 'fake2.html': 0.24, 'fake3.html': 0.5}
>>> results = ['fake1.html', 'fake2.html']
>>> print_ranked_results(results, page_rank)
1: fake1.html (rank: 0.26)
2: fake2.html (rank: 0.24)
```

If the results are empty, it should print a nice message: `No matches for search terms.`

```
>>> page_rank = {'fake1.html': 0.26, 'fake2.html': 0.24, 'fake3.html': 0.5}
>>> results = []
>>> print_ranked_results(results, page_rank)
No matches for search terms.
```

In Section 5 we show what the output should look when this function is used inside the complete search engine.

4.5 Function 14: `search_engine`

This function should take two parameters: the "reverse index" dictionary, and the "page rank" dictionary. It should:

1. ask the user for a search query,

2. process the query to obtain the search results, using the `process_query` function described above,
3. then print out the ranked results, using the `print_ranked_results` function, described above.

This function should continue to ask for a new search query until the user types 'DONE', at which point you should return from this function. Examples of this function's execution are shown in Section 5. To put everything together, modify your `main` function so that it now calls `search_engine`.

5 Examples of correct responses to queries

The following trace shows some example correct responses to queries which you can use for testing your final program. Note: due to the randomness of the ranking calculation, your rank numbers may differ slightly from the ones shown below.

```
Welcome to my mini search engine!
Search terms? (enter 'DONE' to quit) chocolat
1: p.html (rank: 0.135447)
Search terms? (enter 'DONE' to quit) chocolate
1: o.html (rank: 0.148689)
2: p.html (rank: 0.135447)
3: a.html (rank: 0.124522)
4: f.html (rank: 0.104021)
5: h.html (rank: 0.095885)
6: e.html (rank: 0.066301)
7: g.html (rank: 0.053378)
8: i.html (rank: 0.04513)
9: c.html (rank: 0.045028)
10: d.html (rank: 0.044844)
11: b.html (rank: 0.044291)
12: j.html (rank: 0.024352)
13: k.html (rank: 0.017155)
14: n.html (rank: 0.017024)
15: l.html (rank: 0.017023)
16: m.html (rank: 0.01691)
Search terms? (enter 'DONE' to quit) AND chocolat chocolate
1: p.html (rank: 0.135447)
Search terms? (enter 'DONE' to quit) cocoa
1: o.html (rank: 0.148689)
2: f.html (rank: 0.104021)
3: e.html (rank: 0.066301)
4: g.html (rank: 0.053378)
5: i.html (rank: 0.04513)
6: b.html (rank: 0.044291)
7: j.html (rank: 0.024352)
8: k.html (rank: 0.017155)
9: n.html (rank: 0.017024)
10: l.html (rank: 0.017023)
11: m.html (rank: 0.01691)
Search terms? (enter 'DONE' to quit) cocoa -chocolate
No matches for search terms.
Search terms? (enter 'DONE' to quit) chocolate -cocoa
```

```
1: p.html (rank: 0.135447)
2: a.html (rank: 0.124522)
3: h.html (rank: 0.095885)
4: c.html (rank: 0.045028)
5: d.html (rank: 0.044844)
Search terms? (enter 'DONE' to quit) AND chocolate cocoa
1: o.html (rank: 0.148689)
2: f.html (rank: 0.104021)
3: e.html (rank: 0.066301)
4: g.html (rank: 0.053378)
5: i.html (rank: 0.04513)
6: b.html (rank: 0.044291)
7: j.html (rank: 0.024352)
8: k.html (rank: 0.017155)
9: n.html (rank: 0.017024)
10: l.html (rank: 0.017023)
11: m.html (rank: 0.01691)
Search terms? (enter 'DONE' to quit) conquistador
1: b.html (rank: 0.044291)
Search terms? (enter 'DONE' to quit) AND chocolate conquistador
1: b.html (rank: 0.044291)
Search terms? (enter 'DONE' to quit) chocolate conquistador
1: o.html (rank: 0.148689)
2: p.html (rank: 0.135447)
3: a.html (rank: 0.124522)
4: f.html (rank: 0.104021)
5: h.html (rank: 0.095885)
6: e.html (rank: 0.066301)
7: g.html (rank: 0.053378)
8: i.html (rank: 0.04513)
9: c.html (rank: 0.045028)
10: d.html (rank: 0.044844)
11: b.html (rank: 0.044291)
12: j.html (rank: 0.024352)
13: k.html (rank: 0.017155)
14: n.html (rank: 0.017024)
15: l.html (rank: 0.017023)
16: m.html (rank: 0.01691)
Search terms? (enter 'DONE' to quit) chocolate cocoa conquistador -cocoa
1: p.html (rank: 0.135447)
2: a.html (rank: 0.124522)
3: h.html (rank: 0.095885)
4: c.html (rank: 0.045028)
5: d.html (rank: 0.044844)
Search terms? (enter 'DONE' to quit) DONE
```

6 Challenge problem

The challenge problem for this homework is to expand the capabilities of your query processing engine to support quoted phrases as a single search term. For example, the search query `"cacao bean" AND chocolate` should find all pages that have the two-word phrase "cacao bean" and the word chocolate on them, and print them in ranked order.

To solve the challenge problem, you may consider one of the two following approaches.

1. When accepting a search query, if you find that one of the terms is a quoted phrase, you may re-crawl the web pages in order to discover which pages have the set of words. Note that you should not need to recompute ranks or modify the existing (single-term) reverse index.

2. Another approach is to modify how you represent and store the reverse index. Instead of storing just a list of web page names as the dictionary value corresponding to a search term, you could store a list of tuples, which include the web page name, and the *position* (an integer index) of the term on the page. With this representation, you will have information regarding every instance of a search term on every page discovered in the web. You can use the location information to determine whether two or more words appear next to each other.

   If you take this approach, you should make a copy of your basic search engine code (i.e., without any challenge problem modifications) and submit the two versions separately.