Building Indices

DS730

In this project, you will be working with input, output and threads. You are strongly encouraged to use Java for this project. If you are using Java, continue reading through page 5. If you are going to complete this project in Python, start reading on page 6. You are allowed to mix and match solutions if you would like (i.e. solving parts A and C in Java and part B in Python). If using Java, you are expected to investigate the Java library and use the classes and methods in the Collections library as much as possible. You will also need to look into the File class to see how to use folders and files. I would encourage you to use the Scanner class to read in data from the files as it is very easy to use. However, a BufferedReader and a StringTokenizer will provide you with better runtimes for very large files.

In short, you are creating a word index in a few different ways. An index helps you find information in your files faster. An index can also be very useful when comparing how similar two documents are to one another. One way of determining how similar two documents are is to compare the number of uncommon words they share. This might be useful in a recommender type system. For example, if I really like a book that often contains the words *California, surfing, sunshine* and *beach*, then odds are good I will like another book that contains a similar number of those keywords.

You must implement the following steps:

1. You will write a program that goes through a text file and creates a word index of every word in the file. The index will be the “page” that a particular word is found on. Since a text file only contains text and does not contain any metadata, “pages” will be created depending on the number of characters read in so far (not including delimiters). The number of characters that define a page will be specified at runtime.

2. The user will specify 3 command-line arguments. The first argument is the folder where all of the text files are saved, the second argument is the output folder that the output file(s) will be stored to and the third argument is the number of characters that represent a page.
    a. Assuming the java file is called Index.java, the following command:

```
java Index myFolder outputFolder 100
```

would indicate that all input files are stored in a folder called myFolder that is in the same folder as your Java class file. All of the input text files are in myFolder. All text files that you need to account for will have an extension of **.txt**. Only *.txt files will be in the input folder. The second argument is what folder you will store your output files to. You can assume this empty output folder has already been created. In the command, outputFolder will not have a slash attached to it. In other words, your code must insert the slash to make it work to put the files in the correct folder.

The third argument is the number of characters on a page. Assume that number is K. The number of characters on each page goes up to but doesn’t exceed K actual characters. For instance, if K is 100 and you have read in 98 actual characters so far and the next word is **badger**, the word badger would be the first word on the next page (do not split up the word and put part of it on one page and the other part on another page). To make this a bit easier, you must ignore delimiter characters with respect to the number of characters on a page. You can assume that no 1 word will be longer than the number of characters on a page. In this example, no word will be more than 100 characters.

3. Create a word index of each file and store that index into an output file. If your input file is called **a.txt**, then your output file must be called **a_output.txt** and your output will be stored in the output folder that was specified. You will create an output file for each input file. Your word index should be created in the following way:
    a. Read in all words. A word is any consecutive sequence of letters, numbers, apostrophes, special symbols, etc. The only things that delimit words are a space, tab and newline (i.e. whitespace). In my sample files, I am using the default delimiters specified by the Scanner class. If you use something different, you may get different results. You should store the words into one of the following Collections: TreeSet, HashSet, TreeMap or HashMap. If you don’t, your code will likely run very slow. All words are case insensitive. For example, the words **cat**, **Cat**, **CAT** and **cAT** are all considered the same word. Punctuation and other symbols are not to be filtered out. Therefore, **cat:** and **cat** are two separate words (note the colon after the first cat).
    b. Along with reading in all of the words, remember which “page” the word was on. We will start counting from page 1.

c. For each file, after you have read in all of the words, you should write out your word index to that file's output file. You should write out each word that appears in the file and for each word that you write out, you must also write what page(s) that word appears on. You must write out the words in alphabetical order and only put 1 word per line. Assume the word cat appears on pages 4, 10 and 16, your output should be formatted in the following fashion:

    cat 4, 10, 16

   In other words, it is the word, followed by a space, followed by the page(s) that word appeared on where each page is separated by a comma (see sample input/output). Page numbers must appear in ascending order. Words that appear multiple times on the same page should not show up multiple times in the final output. For example, if the word **cat** appears on page 4 a total of 3 times, page 4 should only show up once in the output.

4. You must solve this project in 3 different ways:
   a. (22 pts) The first way is **without** using threads. You must call your file that does not use threads **Index.java**. You must time your code and determine how long it took to solve without using threads. Your program will create the appropriate files and then print out 1 thing to the terminal window: the amount of time it took to execute in milliseconds.
   b. (44 pts) Once you have your solution to 4a, modify it so that it uses threads in some fashion. The most natural way to use threads is to create a new thread for each file you read in. If you are testing this on a machine with multiple cores, you should notice a significant decrease in time (assuming you are using large enough files). You must call your file that uses threads **IndexRunner.java**. Note that this is not saying you are only allowed to create 1 file. Your main method must be in IndexRunner.java but you can create as many files as you want. Your program will create the appropriate files and then print out 1 thing to the terminal window: the amount of time it took to execute in milliseconds. Be sure to wait for your threads to finish before reporting a time.
   c. (44 pts) In short, you are creating a global word index. This differs from the previous question in that you need to *send* your results back to the master thread so that the master thread can produce the output. **Only the master thread is allowed to produce any output for this part.** You must name this master file **GlobalRunner.java**. You can create more than 1 Java file. Your main method must be in GlobalRunner.java but you can create as many Java files as you want.

Create a word index for each file and store each index into a single output file called **output.txt** that is stored in the specified output folder. The output will be a comma separated file. The first line of your output file must be this heading:
Word, first.txt, second.txt, third.txt, xfourth.txt, etc.
In other words, the first word should be Word. This is followed by the names of the input files in alphabetical order.
Your outputs will be combined. The words will be case insensitive and will appear in alphabetical order. For example, assume the word **cat** is in a.txt on pages 2, 4, 6, is in b.txt on 3, 5, 7 and is in c.txt on 2, 5, 7. Your output line for cat would be:
cat, 2:4:6, 3:5:7, 2:5:7
Since commas are used to delimit the files, colons will be used to delimit page numbers. If the word **dog** is in a.txt on pages 2 and 5, is not in b.txt but is in c.txt on page 8, then your output line for dog would be:
dog, 2:5, , 8
You are welcome to use whatever callback or synchronization solution you want but you have to make sure that only the master thread creates the output. No worker thread is allowed to print anything. Your program will create the appropriate output file and then print out 1 thing to the terminal window: the amount of time it took to execute in milliseconds.

A few things to be aware of:

1. Make sure you are doing this on your Hortonworks filesystem. Solving this problem on a Windows machine or some other OS may give you answers that are not consistent with my answers.
2. When your code is tested, we will only have your code, an arbitrary input folder and an arbitrary output folder created. Do not hardcode specific paths. There could be any number of files in the input folder.
3. You must choose appropriate data structures (e.g. TreeMap, HashSet, etc) for each problem. If you don’t, your code may take a long time to run. My solutions take a few seconds to run on the Hortonworks filesystem. For each problem using the input files of moby.txt, wap.txt and huckFinn.txt, your code must complete in under 30 seconds or it will be considered incorrect.

Sample Input/Output (Java)

You can find sample input and output files at (see README files for how to run them):

For problems 4a and 4b:
http://faculty.cs.uwosh.edu/faculty/krohn/ds730/JavaProj.zip

For problem 4c:
http://faculty.cs.uwosh.edu/faculty/krohn/ds730/MoreJavaProj.zip

Producing identical output to the posted output does not necessarily mean everything is correct. For example, if you ignore the multithreading requirements of 4b or ignore the printing requirements of 4c, your program may produce the correct output but still be incorrect.

What to Submit (Java)

You must submit all versions of your solution (i.e. steps 4a, 4b and 4c) in a single zipped file. When you are finished, submit a zipped file called `p4.zip` that contains your Java files and upload it to the dropbox.

The subsequent pages are only for those solving this project using Python. If you are solving this in Java, read the previous pages.

I will stress again that I strongly encourage you to solve this problem using Java instead of Python. You will see no improvement in runtimes with steps 8b nor 8c. In fact, your code will probably be slower because of the threading overhead. However, the goal is to understand the concept of threading. Having part (b) or part (c) be slower than part (a) is fine as long as the concept is implemented correctly.

The threading presentation in Python shows you how to create multiple threads. You should use this as your template for steps 8b and 8c. Since Python does not do any true multithreading, step 8c must be done in a specific fashion. Be sure to follow the rules of step 8c to ensure you are doing things correctly. Since Python doesn't do any true multithreading, it is possible to complete step 8c without worrying about shared access to variables because only 1 thread is executing at any 1 time.

If you are using Python, it is assumed you have enough knowledge to solve the non-threaded parts of this project. Therefore, we will not be answering questions about Python regarding non-threaded code. If you need to learn new things about Python to complete this project, then why not learn a new language that actually does multithreading? With that said, we assume most of you have never seen command line arguments nor have had to time your code. Examples of this are given below.

If you have never used command line arguments with Python before, this is a short example on how to use them:

```
import sys
print(sys.argv[1] + " " + sys.argv[2] + " " + sys.argv[3])
```

Store the above code into args.py and run the code using the following command:

```
./args.py one two three
```

You will notice that **one two three** prints out. Can you guess what's in sys.argv[0]? If you want to time your code, use the following code as an example:

```
import time
start = time.perf_counter()
#code to do a bunch of stuff
time.sleep(1)
end = time.perf_counter()
total = end - start
print("Took " + str(total) + " seconds to finish.")
#should print about 1 second
```

In short, you are creating a word index in a few different ways. An index helps you find information in your files faster. An index can also be very useful when comparing how similar two documents are to one another. One way of determining how similar two documents are is to compare the number of uncommon words they share. This might be useful in a recommender type system. For example, if I really like a book that often contains the words *California, surfing, sunshine* and *beach*, then odds are good I will like another book that contains a similar number of those keywords.

You must implement the following steps:

5. You will write a program that goes through a text file and creates a word index of every word in the file. The index will be the “page” that a particular word is found on. Since a text file only contains text and does not contain any metadata, “pages” will be created depending on the number of characters read in so far (not including delimiters). The number of characters that define a page will be specified at runtime.

6. The user will specify 3 command-line arguments. The first argument is the folder where all of the text files are saved, the second argument is the output folder that the output file(s) will be stored to and the third argument is the number of characters that represent a page.
    a. Assuming the Python file is called index.py, the following command:
    ./index.py myFolder outputFolder 100
    would indicate that all input files are stored in a folder called myFolder that is in the same folder as your Python file. All of the input text files are in myFolder. All text files that you need to account for will have an extension of **.txt**. Only *.txt files will be in the input folder. The second argument is what folder you will store your output files to. You can assume this empty output folder has already been created. In the command, outputFolder will

not have a slash attached to it. In other words, your code must insert the slash to make it work to put the files in the correct folder.

The third argument is the number of characters on a page. Assume that number is K. The number of characters on each page goes up to but doesn’t exceed K actual characters. For instance, if K is 100 and you have read in 98 actual characters so far and the next word is **badger**, the word badger would be the first word on the next page (do not split up the word and put part of it on one page and the other part on another page). To make this a bit easier, you must ignore delimiter characters with respect to the number of characters on a page. You can assume that no 1 word will be longer than the number of characters on a page. In this example, no word will be more than 100 characters.

7. Create a word index of each file and store that index into an output file. If your input file is called **a.txt**, then your output file must be called **a_output.txt** and your output will be stored in the output folder that was specified. You will create an output file for each input file. Your word index should be created in the following way:
    a. Read in all words. A word is any consecutive sequence of letters, numbers, apostrophes, special symbols, etc. The only things that delimit words are a space, tab and newline (i.e. whitespace). All words are case insensitive. For example, the words **cat**, **Cat**, **CAT** and **cAT** are all considered the same word. Punctuation and other symbols are not to be filtered out. Therefore, **cat:** and **cat** are two separate words (note the colon after the first cat).
    b. Along with reading in all of the words, remember which “page” the word was on. We will start counting from page 1.
    c. For each file, after you have read in all of the words, you should write out your word index to that file's output file. You should write out each word that appears in the file and for each word that you write out, you must also write what page(s) that word appears on. You must write out the words in alphabetical order and only put 1 word per line. Assume the word cat appears on pages 4, 10 and 16, your output should be formatted in the following fashion:

        cat 4, 10, 16

       In other words, it is the word, followed by a space, followed by the page(s) that word appeared on where each page is separated by a comma (see sample input/output). Page numbers must appear in ascending order.

Words that appear multiple times on the same page should not show up multiple times in the final output. For example, if the word **cat** appears on page 4 a total of 3 times, page 4 should only show up once in the output.

8. You must solve this project in 3 different ways:
    a. (22 pts) The first way is **without** using threads. You must call your file that does not use threads **index.py**. You must time your code and determine how long it took to solve without using threads. Your program will create the appropriate files and then print out 1 thing to the terminal window: the amount of time it took to execute in seconds.
    b. (44 pts) Once you have your solution to 8a, modify it so that it uses threads in some fashion. The most natural way to use threads is to create a new thread for each file you read in. Unfortunately, because of the global interpreter lock, you will see no speedup with using threads in Python. You must call your file that uses threads **indexRunner.py**. Your program will create the appropriate files and then print out 1 thing to the terminal window: the amount of time it took to execute in seconds. Be sure to wait for your threads to finish before reporting a time.
    c. (44 pts) In short, you are creating a global word index. This differs from the previous question in that you need to *send* your results back to the master thread so that the master thread can produce the output. **Only the master thread is allowed to produce any output for this part.** In other words, if you call a function using the threading.Thread(...) command, that function should not be creating any output. If you have any variables that are used by the worker threads, you must use locks to ensure that only 1 thread is accessing the variable at once. Because of the non-threaded nature of Python, this step is likely unnecessary because no 2 threads could access the same data structure at any 1 time. You could skip this requirement and your code would still work. However, the goal is to simulate what *would* happen in a multithreaded environment. Therefore, if the worker threads are accessing anything that is shared, you must surround that access with a lock. Ask if this requirement does not make sense[1].
       You must name this master file **globalRunner.py**. Create a word index for each file and store each index into a single output file called **output.txt** that is stored in the folder specified. The output will be a comma separated file. The first line of your output file must be this heading:
       Word, first.txt, second.txt, third.txt, xfourth.txt, etc.

[1] This requirement is not stated in the Java portion because Java code will almost certainly fail if critical regions are not synchronized.

In other words, the first word should be Word. This is followed by the names of the input files in alphabetical order.
Your outputs will be combined. The words will be case insensitive and will appear in alphabetical order. For example, assume the word **cat** is in a.txt on pages 2, 4, 6, is in b.txt on 3, 5, 7 and is in c.txt on 2, 5, 7. Your output line for cat would be:
cat, 2:4:6, 3:5:7, 2:5:7
Since commas are used to delimit the files, colons will be used to delimit page numbers. If the word **dog** is in a.txt on pages 2 and 5, is not in b.txt but is in c.txt on page 8, then your output line for dog would be:
dog, 2:5, , 8
You are welcome to use whatever callback or synchronization solution you want but you have to make sure that only the master thread creates the output. No worker thread is allowed to print anything. Your program will create the appropriate output file and then print out 1 thing to the terminal window: the amount of time it took to execute in seconds.

A few things to be aware of:

1. Make sure you are doing this on your Hortonworks filesystem. Solving this problem on a Windows machine or some other OS may give you answers that are not consistent with my answers.
2. When your code is tested, we will only have your code, an arbitrary input folder and an arbitrary output folder created. Do not hardcode specific paths. There could be any number of files in the input folder.
3. You must choose appropriate data structures for each problem. If you don't, your code may take a long time to run. My solution with index.py takes a few seconds to run on the Hortonworks filesystem. Because there is no multithreading with Python, the code that uses threading does not see any speedup. In fact, it is actually a little bit slower. However, it is not that much slower. For each problem using the input files of moby.txt, wap.txt and huckFinn.txt, your code must complete in under 30 seconds or it will be considered incorrect. Since you will not see any speedup with parts 8b and 8c, it will be hard to know whether or not you did the threading portion correctly. If you are concerned about whether or not you did the threading correctly, be sure to ask specific questions about your threading code. We do not generally answer “is this correct” type questions. We want to point you in the right direction without giving away the answer.

Sample Input/Output (Python)

You can find sample input and output files at (see README files for how to run them):

For problems 8a and 8b:
http://faculty.cs.uwosh.edu/faculty/krohn/ds730/JavaProj.zip

For problem 8c:
http://faculty.cs.uwosh.edu/faculty/krohn/ds730/MoreJavaProj.zip

Producing identical output to the posted output does not necessarily mean everything is correct. For example, if you ignore the multithreading requirements of 4b or ignore the printing requirements of 4c, your program may produce the correct output but still be incorrect.

What to Submit (Python)

You must submit all versions of your solution (i.e. steps 8a, 8b and 8c) in a single zipped file. When you are finished, submit a zipped file called `p4.zip` that contains your Python files and upload it to the dropbox.