**Summary**

**Algorithm:**

**General process:**

Run index.py first to build posting lists before start query-ing

(Posting lists are stored in postingFile.txt)

My TF-IDF calculation is without “log” component.

After that, depends whether we are:

**Processing query without weight:**

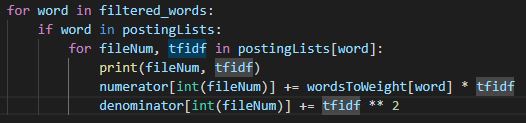


**Or**

**Processing query with weight:**

**Regardless of choices, the steps are almost identical:**

* Remove stopwords and downcase words in query
* Load posting lists from disk into main memory
* Record weights for each word in the query (for processing without weight, all weights would be 1). Meanwhile, calculate the square root of sum for each word (so we can multiply it with square root of sum for matching words in a particular document)
* The method that I use is Term-at-a-time:
* For each word that has been filtered in the query, check if that word exists in the index. If yes, access the posting list of that word and add weight(query\_word) \* (TF\*IDF of matching word) to an array called numerator.



+ Meanwhile, add (TF\*IDF) ^ 2 to the square root of sum for

matching words in a particular document

* Applying formulas provided from Professor’s email:







* I only apply these formulas for documents that have at least one matching. Otherwise, we may run into division by zero or unnecessary documents for display (Documents with score = 0)
* Put documents with positive scores into the max heap. Now I just simply pop the max heap to obtain top 10 (or less) documents.

**Note:** My program also displays top 10 tf\*idf terms per document for each of top 10 documents by computing top ten words for each document during indexing. Then for every document in the top 10, I simply look into that file and display them.

**Running Time Complexity:**

Let **m** be the number of query words and let **n** be the number of documents in the index. Then for each word, the program needs to look into all documents in the posting list of that word and compute the weight for each document.

→ O(m \* n)

After that, I put the score for each document in an array and heapify it

→ O(n)

To extract top 10 documents, I pop each element in the max heap. The running time for each pop is:

→ O(log n)

There are only a constant number of documents needed to pop (10) → O(1)

**Therefore, the total running time is:**

O(m \* n) + O(n) + O(log n) \* O(1) = O(m \* n)

Note: output queries from the sample are stored in the file “query output”