# CS 345: Homework 5 (Due: Nov 09, 2015)

**Rules.** Teams of no more than three students as explained in syllabus (Handout 1). This is to be handed back in electronic form as a single tar or zip file decompressible on an AFS machine (afsconnect1.njit.edu or afsconnect2.njit.edu or osl11.njit.edu), where testing would take place. (At a minimum, the zipfile should contain a text file `HW5_ABC.txt` as explained below.) The code should be compilable/interpretable and executable on one or the other similar AFS machines in any language available there (emphasis in C, C++, Java, Python, Perl).

## 1 A file-based desktop search engine: FDSEE

You will implement the first components of a small file-based (the F of FDSEE), desktop (the D of fdsee) search engine (the SE from search and the last E from engine).

### 1.1 Objectives of the assignment

We assume that a crawler has already stored locally in a single directory a collection of files crawled from the Web; within that directory there might be subdirectories of files.

The first phase of this assigment is to go through all the files/directories and identify from their suffixes which ones are text searchable or not (a directory, or other files that are not searchable). A pathname stream of the text searchable files (path name relative to the top-level one) will be generated for verification of the successful completion of this objective.

In a second phase this stream of text searchable files will then be tokenized and in a subsequent homework indexed and searched.

### 1.2 Project deliverables

After compilation the executable unit produced should be named `fdsee`. If your code is interpreted the wrapper function should be `fdsee` possibly with an appropriate suffix as needed. We shall refer to that file as `fdsee` in the remainder. Every source file you submit must include in the form of comments in the first 5 lines the names of the members of the group including the last four digits of their NJIT IDs. In addition a file named `HW5_ABC.txt` needs to be included (that also conforms to the first-5-line convention) that includes instructions for compilation/interpretation, bugs, and anything else of interest (eg extensions). The `ABC` are the initials of the first names of the three members of a team. (Naturally if a team has fewer members this can be truncated.)

## 1.3 Introduction

The two phases of this homework involve determining which files are searchable, and then performing the parsing and tokenization of those searchable files. The first phase consists of Step1 and the second phase of Steps 2-4.

Step1: **Searchable Documents**. You will traverse the directory structure of the locally stored Web-pages (stored there as a result of web crawling) and identify the text searchable files among them. Some assistance is provided in the protected area of the course Web-page in the form of program `recdir.c` available as link L5 there. It's up to you whether you want to use it or not (it has only been tested on a Unix system). Searchable files will be uncompressed ASCII text files. After you have identified the searchable text files, you are then about to start the process of parsing these and only these files and tokenizing their contents and performing linguistic analysis in the form of stopword elimination and stemming.

Step 2: **Tokenization**. This step involves the tokenization of every searchable document. You identify and extract words/tokens of interest along with other useful information (such as word position in the text, font-size, and position in an html document). In this part you also convert keywords to numbers (wordIDs) and also URL (Uniform Resource Locators) or in fact URI (Uniform Resource Identifier) into numbers (docIDs). IMPLICIT in the tokenization is case-folding: everything becomes lower-case!

Step 3: **Stopword Elimination.** Steps 2-4 are part of the parsing phase: first identify the token and then convert the tokens into an index-term by eliminating some (stopwords) or obtain from several slightly different tokens one index-term (stemming). For the sake of this assignment we will concentrate only on elementary `stopword` removal. The output of this phase will be the tokens identified in Step 2 minus the stopwords; this stream of keywords will then be further processed in Step 4.

Step 4: **Stemming.** The fourth step uses the output of Step 3 to further eliminate and reduce the number of potential index terms by performing stemming.

This is the minimum implementation required to gain you the full points of this assignment. You can enrich this implementation by adding additional features such as searching more file types, do a more thorough parsing or more elaborate linguistic analysis.

### 1.3.1 Step 1: Searchable Document

The program `fdsee` will read the command line and behave as follows. (If you use Java, these options might follow a `java` invocation, and the dot slash might go away.)

```
% ./fdsee searchable filename
% ./fdsee searchable directoryname
```

The first argument in the command line (after the name of the executable) denotes the action. The next argument is an arbitrary file or directory name. For action `searchable` (searchable document/document list) you need to decide whether `filename`, or all the files under `directoryname` (and its subdirectories) are searchable text files or not and list them. A searchable text file `SETF` is a file with one of the following suffixes:

`.html , .htm , .txt , .cc , .cpp , .c , .h , .java .`

A `SETF` (for SEarchable Text File) file can be `HTML` or `TEXT`. An `HTML` file is a searchable text file with suffixes `.html` or `.htm`. A `TEXT` file is a searchable text file with any of the remaining suffixes. A file that is not a `SETF`, it is a `NSTF` (non-SETF abbreviation to four characters). The command argument `searchable` prints for every unit identified an `HTML` or `TEXT` for a `SETF` file. (Note that `SETF` is not printed, and there is no reason to list `NSTF` files or directories.) A sample output follows and ends with a `END-OF-LISTING`.

The output becomes available as a side-effect into a file named `fdseeS1.txt` including the END-OF-LISTING line. (One line per SETF file identified.)

```
% ./fdsee searchable  alexg
alexg/courses/cs345/index.html  HTML
alexg/courses/cs345/my.txt      TEXT
alexg/courses/cs345/my.java     TEXT
alexg/courses/cs345/handouts.html  HTML
END-OF-LISTING
```

### 1.3.2 Step 2: Tokenization

If action is `token` then every SETF file (`HTML` or `TEXT`) is then parsed and tokenized. YOU DECIDE what constitutes a token: the simplest rule is everything between consecutive whitespace. A `tokendebug` provides a more informative output.

```
% ./fdsee token filename
% ./fdsee token directoryname

% ./fdsee tokendebug filename
% ./fdsee tokendebug directoryname
```

The tokenization phase is probably the most difficult part of this assignment and the most time consuming. You need to decide how to parse a document and what constitutes a token. You might make the job at hand easier if you use a parser program such as lex or yacc for this part (and the time involved to familiarize yourselves with them, if you have never used them before). An hour or two reading a manual page or the extensive documentation for the GNU equivalents names flex and bison might save you time building a tokenizer from scratch. Tokenizing a `TEXT` file is easier. Interesting tokens that will become index terms are going to be words (e.g. alphanumeric strings starting with a character) or non-trivial numbers. Single digit numbers can be discarded dates such as 1950 might become useful searchable terms. Collectively we will call all these interesting tokens `words` even if some of them are numbers.
I. `Words` will be represented by `wordID`s, i.e. a preferably 32-bit unsigned integer.
II. `Documents` will be represented by `docID`s, i.e. a preferably 32-bit unsigned integer.
Therefore if word `algorithms` is identified in a `TEXT` document a quadruplet of information is generated under `token` by the tokenizer.
`(docID,wordID,offset,attr)`
If however `tokendebug` is used intead, then the path of the document (docURL) rather than a docID, and the word or attribute value rather than a wordID or attribute code are generated. (However for `attrvalue` read also NOTE 2 that allows you to use `attr` instead.)
`(docURL,word,offset,attrvalue)`

**NOTE 1.** The tokenizer generates words that are lower case no matter what the original case was.
**NOTE 2.** You need to be able to convert a `wordID` into a `word` and the other way around and do so also for `docID` and `docURL`. Although `attr` is a value between 0 and 4 as explained below, `attrvalue` may be more informative. However YOU ARE ALLOWED to use `attr` rather than `attrvalue` with `tokendebug`.
**NOTE 3.** For the sake of this and remaining assignments a file in say directory
`alexg/courses/cs345/index.html` has the same URL/URI.

Under `token` the first element of the quadruplet above is a `docID` a number not a docURL. The second one is the unique `wordID` for the specified word. All files that have been searched should have a common lexicon (e.g. dictionary of words) and a single entry for each unique word. The third attribute `offset` is positional information about the identified word in `docID`. It identifies the word position of the word in the document, i.e. whether `algorithms` is the 10-th or 20-th word (or token) in the document. (The value of `offset` will depend on the tokenization method that you use.) The last element is an `attr`ibute value. For `TEXT` files all words have attribute 0. For `HTML` files certain words may have higher attribute values. Words surrounded by a `<TITLE>` tag will have attribute value 1. Words surrounded by an `<A>` which is an anchor tag will have attribute value 2. Words surrounded by a `<H1> to <H3>` tag have value 3, and `<H4> to <H6>` tags have value 4. Other values are possible.

**NOTE 4a: Behavior with `token` argument.** So a command such as the one depicted below will generate a stream of parenthesized numbers (the example below shows only a single quadruplet for the single TEXT file containing only one word).

```
% ./fdsee token alexg/courses/cs345/my.html
(10,20,1,4)
```

**NOTE 4b: Behavior with `tokendebug` argument.** If however the command is `tokendebug` instead of `token`, more useful information will be printed instead of `docID, wordID` as explained earlier.

```
% ./fdsee tokendebug  alexg/courses/cs345/my.html
(alexg/courses/cs345/index.html,algorithms,1,<H4>)
```

**NOTE 5.** A reminder for NOT 4a and 4b. Tag `<H4>`, per NOTE 3, has attribute value 4 and so do `<H5>, <H6>`.

The output becomes available as a side-effect into a file named `fdseeS2.txt` for `token` and `fdseeS2d.txt` for `tokendebug`.

### 1.3.3 Step 3: Stopword Elimination

This assumes as input the output of Step 2 for `token` and eliminate from the stream those tokens/words identified as stopwords. The list of stopwords for the sake of this assignment is given below. Thus `stopword` behaves similarly to `token` rather than `tokendebug`. It generates an output stream with fewer tuples; yet offset and other information does NOT change.

```
     I a about an are as at be by com en for from how in is it  of on or that the
   this to was what when where who will with www
```

```
% ./fdsee stopword filename
% ./fdsee stopword directoryname
```

The effect of command `stopword` is incremental. It is equivalent to `token` with the addition of stopword elimination. (In the generated stream, stopwords will not be listed.) It's up to you whether you want to insert them into the lexicon (the easier thing to do).

The output becomes available as a side-effect into a file named `fdseeS3.txt`.

### 1.3.4 Step 4: Stemming

The stream input for this step is the output of Step 3 (if implemented) or Step 2 otherwise.

```
% ./fdsee stem filename
% ./fdsee stem directoryname
```

For action `stem` you need to apply Harman's stemming algorithm that eliminates very simple suffixes (eg. plural). The output is a stream similar to the input.

Note that it is quite possible that say wordID `25` already corresponds to word `algorithm` and there is another wordID say `35` that corresponds to word `algorithms`. If as a result of the stemming method we have a translation of `algorithms` into `algorithm` you need to decide what to do with wordIDs 25 and 35. The design decision is yours. One way to deal with it is to convert for example `35` into `25`; an issue is what to do with wordID `35` that is left unused.

```
  HarmanStemmingAlgorithm(word)
 1. If word ends in -ies but not -eies or -aies
     then -ies --> -y;
 2. If word ends in -es  but not -aes, -ees or -oes
     then -es  --> -e;
 3. If word ends in -s but not -us or -ss
     then -s   --> -;
```

The output becomes available as a side-effect into a file named `fdseeS4.txt`.

### 1.3.5 Side Effects

Each one of the commands `searchable`, `stopword`, `token` and `tokendebug` and `stem` will have the side-effects that have already been mentioned, i.e. a text file as named above. Moreoever the following files will also be generated.

**SideEffect 1:** `fdseeDL.txt`. A file with this name will be generated in the directory in which `fdsee` was invoked. Its contents would be quadruplets one per line consisting of a `docID, docURL` and a `SETF, NSTF` attribute as the third element. For a `SETF` the last (fourth) attribute is an `HTML` or `TEXT`. For `NSTF` no fourth attribute needed. You may use parentheses or commas to separate attributes of tuples. The list is ordered by `docID` and is ASCII and printable, one tuple per line. `docID`'s do not need to be consecutive nor start from zero or one for that matter.

**SideEffect 2:** `fdseeLX.txt` . The generation of such a file in the directory in which `fdsee` was invoked. It lists all words indexed and ordered by wordID. They (wordID) do not need to be consecutive nor start from zero or one for that matter. Although the first two attributes (wordID, word) are required to be present, other information is up to you. This includes a `STOPWORD, NONSTOPWORD` attribute value, and a stemmed wordID. (For example algorithms has a wordID 20, but after stemming it might get mapped to algorithm whose wordID is 15.)

```
% cat ./fdsee\_lexicon
 0  alex        NONSTOPWORD   0
 1  the         STOPWORD      1
 15 algorithm   NONSTOPWORD   15
 ...
 20 algorithms  NONSTOPWORD   15
 ....
```

**NOTE 6.** We do not specify how the lexicon incorporates information related to stemming. To keep things simple, do not incorporate such info into the lexicon. ■

Date Posted: 9/08/2015