Rules. The first option of this assignment is programming based. This is to be handed back no later than midnight on the Monday it is due (see information below) in electronic form as a single tar or zip file decompressible on an AFS machine. For the paper-based option the deadline is the standard one. The code should be compilable and executable on an AFS machine (if done in C, C++ or Java), or interpreted there (if done in Perl, Python or other interpreted language supported by AFS). **A group of at most two students can work collectively to submit one assignment; both students of the group will be assigned the same numeric grade.**

Due date: No later than Oct 31,2011.

1 OPTION 1 (125 points): A file-based desktop search engine

1.1 Objectives of the assignment

We assume that a crawler has stored locally in a single directory a collection of files already crawled from the Web. This first phase of this programming assignment will ask you to go through these files (and the directory structure) and determine whether a file is text searchable or not. The input to this first phase is a directory name and the output a listing of files in the directory and its (possible) subdirectories that are text searchable. This stream of file names can then be tokenized (in a follow-up phase) and indexed and searched (in the phases of a follow-up assignment).

1.2 Introduction

The first two phases of an indexing step are undertaken. This involves determining which files are indexable, and then performing the parsing of those files. This task is accomplished in three steps.

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 on how to traverse a directory structure in a UNIX file-system. You can use this information as a base for this OPTION of the assignment. For the sake of this assignment 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 files and tokenizing their contents.

Tokenization The second step involves the first phase of parsing, that of tokenization. For every searchable document, you identify and extract keywords/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 locators to numbers (docIDs).

Linguistic Analysis The third part involves the second phase of parsing, the linguistic analysis of the keywords/tokens that would be turned into index terms. For the sake of this assignment we will concentrate only on elementary `stopword` removal. The output of this phase will be then fed into the indexer.

This is the minimum implementation required to gain you the full points of this assignment. You can enrich this implementation by adding additional features.

1.2.1 Project deliverables

A single executable (interpretable) file will be the result of your source code into the form of a file named `hw4`. `hw4` can be the result of a compilation, or just an interpretable source file. We shall refer to that file under as `hw4` in the remainder. We shall assume that the file resides in directory `homework`, i.e. `homework/hw4`. (We use a UNIX-like notation of forward slashes to describe directories.)

1.2.2 OPTION 1a: Searchable Document

The executable file `hw4` will read the command line and behave as follows.

```
% ./hw4 searchable file-name
% ./hw4 searchable directory-name
```

The first argument in the command line (after the name of the executable) denotes the action. The second argument is a file or a directory name.

For action `searchable` (searchable document list) you need to decide whether `file-name`, or al the files under `directory-name` (and its subdirectories) are searchable text files or not and list them. A searchable text file `STF` is a file with one of the following suffixes:

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

An `STF` 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 either a directory name or a non-searchable file is `NSTF` (non-STF).

The command argument `searchable` prints for every file identified as an `STF` whether it is `HTML` or `TEXT` along with the full path name relative to the directory name listed as the second argument during program invocation. A (partial) output may look like as follows in the case where there are two HTML files in subdirectory `cs485` of directory `courses` which is the only directory of `alexg`. An `END-OF-LISTING` concludes this listing.

```
% ./hw4 slist  alexg
alexg/courses/cs485/index.html  HTML
alexg/courses/cs485/handouts.html HTML
END-OF-LISTING
```

Note that if you call the program with a directory name prefix that includes `/home` and a user-account such as `u1` the whole prefix is trimmed away and replaced by the default URL as shown in the previous example.

```
% ./hw4 slist  /home/u1/alexg
```

Therefore this latter invocation will also generate the same output as before. In general one of the following labels will be printed `HTML, TEXT` for `STF` files, and optionally `NSTF` for non-STF files. (You might only list `HTML, TEXT` files, and ignore NSTF files.)

1.2.3 OPTION 1b: Tokenization

If action is `token` then for every file identified as `STF` (with `HTML, TEXT` printed in the output) a parsing function is called that tokenizes each file into its individual tokens.

```
% ./hw4 token file-name
% ./hw4 token directory-name
```

```
% ./hw4 debug-token file-name
% ./hw4 debug-token directory-name
```

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. The choice of C/C++/Java is up to you. However you might make the job at hand easier if you explore using 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.
`words` will be represented by `wordID`s, i.e. a preferably 32-bit unsigned number.
`Documents` will be represented by `docID`s, i.e. a preferably 32-bit unsigned number.
Therefore if word `algorithms` is identified in a `TEXT` document a quadruplet of information is output by the `tokenizer` (an explanation is provided after the NOTES):
`(docID,wordID,wpos,attr)`

NOTE. The tokenizer generates words that are lower case no matter what the original case was.
NOTE. You need to have ways to determine the word represented by say `wordID` 25 or the full URL name of the document with `docID` 100.
NOTE. For the sake of this and remaining assignments a file in say directory
`alexg/courses/cs485/index.html` has a full URL which is
`http://www.cs.njit.edu/~alexg/courses/cs485/index.html`.

The first element of the quadruplet above is a `docID` and not the full-URL of the document in which the named word appears. The second one is the unique `wordID` for the specified word (e.g. `algorithms`). All files that have been searched should have a common lexicon and a single entry for each unique word. The third attribute is positional information about the identified word in the document. It identifies the word position of the word in the document, i.e. whether `algorithms` is the 10-th or 20-th word in the named document. (The value of `wpos` will depend on the tokenization method that you use.) The last element is an `attribute` 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; your implementation is open-ended.

So a command such as the one depicted below will generate in the output a stream of parenthesized numbers (the example below show only a single quadruplet).

```
% ./hw4 token alexg/courses/cs485/index.html
(10,20,30,4)
```

NOTE. We do not specify how big the attribute or positional values are going to be. It's up to you to decide so.
If however the command is `debug-token` instead of `token`, more useful information can be printed such as

```
% ./hw4 debug-token alexg/courses/cs485/index.html

( http://www.cs.njit.edu/~alexg/courses/cs485/index.html,algorithms,30,<H4>)
```

1.3 OPTION 1c: Linguistic Analysis

The linguistic analysis module will take as input the output of Part 1b and eliminate those keywords that are identified as stopwords. The list of stopwords for the sake of this assignment is given below.

```
    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
```

```
% ./hw4 stopwords file-name
% ./hw4 stopwords directory-name
```

The effect of command `stopwords` is incremental. It is equivalent to `token` with the addition of stopword elimination. (In the generated stream, stopwords will not be listed.)

1.3.1 Side Effects

Each one of the three commands `stopwords`, `token` and `debug-token` will have the following common side-effect. The generation of a file `lexicon` in the directory in which `hw4` was issued and the generation of a file `doclist` also in the same directory. The first file `lexicon` lists all words indexed by wordID and the latter `doclist` lists all URL/documents indexed by docID. The two files are ASCII printable/viewable files. For example the output might look like as follows. No indentation or pretty printing is required. However one item per line should be printed.

```
% cat ./lexicon
 0  alex
 ...
 20 algorithms
 ....
```

■

2 OPTION 2 (125 points): Problem solving

Problem 1. (40 points) Is the state of Google's dictionary (lexicon) in 1997/1998 (use paper L1 and Subject 2 for reference) consistent with Heaps' Law i.e. how many words does the lexicon maintain and how many unique words does Heaps' Law predict about the 147GB or so of the document collection? Justify your claims.

Problem 2. (40 points) Provide Unary, Elias-γ, Elias-δ and Golomb-5 codes for $k = 51$ and $k = 40$.

Problem 3. (30 points) We have the string of 20 characters A, C, G, T shown below.
`AACC GGTT GGAA AACC AAGT`
(Ignore spaces, they are provided for clarity only.) If you compress this text using Huffman coding, what are the prefix codes for each one of the four characters? What is the total number of bits used to encode just the text (consisting of A, C, G, T and ignoring the spaces or other auxiliary information)?

Problem 3. (15 points) What is the entropy of the text in Problem 3?