CS 211: Computer Architecture, Spring 2017
Programming Assignment 2: Dictionary Stat Generator
Due Date: March 1st, 5pm

Instructor: Prof. Santosh Nagarakatte

This assignment is designed to provide you some experience with C programming. Your task is to write a program that reads in a collection of dictionary and data file pairs and generates statistics about each pair. There are two parts to this assignment. The first part of the assignment is for regular credit and the second part of the assignment is extra credit.

First Part (100 points)

Your task in the first part is to write a C program called `first` that reads an input file, which contains a list of dictionary and data file pairs, and generates statistics for each pair.

Each line in your input file contains the names of the dictionary and data files. You have to read the files and generate the following statistics:

1. For every word w in the dictionary file, count the number of words w' that occur in the data file such that $w' = w$.
2. For every word w in the dictionary file, count the number of words w' that occur in the data file such that w is a *proper* prefix of w' (we shall say w' is a *superword* of w).

Write all the unique words in the dictionary along with these counts to the output file in lexicographical (i.e. alphabetical) order.

Definition of a word: Any string of characters can be a word. For example in the sentence:

"a&ab&abc234 dfg"

the words are:

a, ab, abc, dfg

They **do not** need to be meaningful. Words, in both dictionary and data files, correspond to the longest continuous sequence of letters read from the file. Another way to say this is that words are any sequence of letters separated by non-letter characters (punctuation, numbers, whitespace, etc.). Each unique word is case-insensitive. That is, "boOK", "Book" and "bOOk" are all occurrences of the same (unique) word "book".

Case-insensitivity also applies when matching prefixes: for example, both "Boo" and "bOo" are proper prefixes of "bOOk", which itself is a proper prefix of "booKING" (See the example below).

As an example, suppose the content of the dictionary file is:

boo22$Book5555bOoKiNg#bOo#TeX123tEXT(JOHN)

and that of the data file is:

John1TEXAN4isa1BOoRiSH%whohasa2bo3KING BOOKING bOoKIngs$12for a TEX-Text(BOOKS(textBOOKS)

Then, the various counts for the unique words in the dictionary file are:

Unique words	No. of occurrences	No. of superwords
boo	0	4
book	0	3
booking	1	1
tex	1	3
text	1	1
john	1	0

Input/Output specification

Usage interface

The program `first` should have the following usage interface:

```
first <mappingfile>
```

where `<mappingFile>` is the name of the mapping file. You can assume that the mapping file will exist and that it is well structured. So, unlike assignment1, you don't need to check the structure of the mapping file

If no argument or more than one argument is provided, or the file names provided are invalid, the program should print "`invalid input`" and abort.

Input specification

Here, and below, let `m` be the maximum number of words in either the dictionary or data files. Every word is of length at most `k`. Let `n` be the number of unique words in the dictionary file.

Your input will be a mapping file, which contains lines of the form: ⟨dictFile⟩ ⟨dataFile⟩, where dictFile and dataFile are the dictionary and data files for your program, respectively. An example of a mapping file is given below:

```
dict_1 data_1
dict_m data_m
```

The files: dict_1.txt and dict_m.txt are dictionary files and files: data_1.txt and data_m.txt are data files. They are plain text files with no special structure.

Output specification

Your program should generate several output files `outi.txt`, where `i` is the line number in mapping file. It means that you need to get mapping file as an argument to your program. Then each line in the mapping file has information about the dictionary file and the data file.
For example suppose line `j` in the mapping file is `dict_j data_j`. In this case you should produce `outj.txt`, which contains the described informations. Remember that you shouldn't have any spaces at the end of the lines in your output files. Also, you shouldn't have any empty lines in your outputs files . The program should write all the unique words (see definition above) that occur in the dictionary file along with their various counts (See above), in lexicographical order, one word per line to the output files i.e. the output should have exactly `n` lines:

```
<word1> <count11> <count12>
<word2> <count21> <count22>
                .
                .
                .
<wordn> <countn1> <countn2>
```

such that

1. `<word1>, <word2>, <word3>` ... `<wordn>` is the lexicographical ordering of the unique words occurring in the dictionary,

2. `<counti1>` denotes the number of occurrences of `<wordi>` in the data file,

3. `<counti2>` denotes the number of proper superwords of `<wordi>` in the data file,

4. `Line i` is `<wordi><space><counti1><space><counti2>`

Note that you can assume the dictionaries are non-empty.

Also note that if the mapping file has more than one line, you need to do same procedure for each line and create the result file (outi.txt) of every line.

For example, running `first` on the input described above should produce the following output:

```
boo 0 4
book 0 3
booking 1 1
john 1 0
tex 1 3
text 1 1
```

Second Part (Extra Credit - 50 points)

Your task in this second part is to write a C program called `second` that reads a input file, which contains a list of dictionary and data file pairs, and generates statistics for each pair.

Each line in your input file contains the names of the dictionary and data files. You have to read the dictionary and data files and generate the following statistics:

1. For every word w in the dictionary file, count the number of words w' that occur in the data file such that $w' = w$.

2. For every word w in the dictionary file, count the number of words w' that occur in the data file such that w' is a *proper* prefix of w.

Write all the unique words in the dictionary along with these counts to the output file in lexicographical (i.e. alphabetical) order. A word is define the same way as in the first part.
Case-insensitivity also applies when matching prefixes: for example, both “Boo” and ”bOo” are proper prefixes of “bOOk”, which itself is a proper prefix of “booKING” (See the example below).

As an example, suppose the content of the dictionary file is:

boo22$Book5555bOoKiNg#bOo#TeX123tEXT(JOHN)

and that of the data file is:

John1TEXAN4isa1BOoRiSH%whohasa2bo3KING BOOKING bOoKIngs$12for a TEX-Text(BOOKS(textBOOKS)

Then, the various counts for the unique words in the dictionary file are:

Unique words	**No. of occurrences**	**No. of prefixes**
boo	0	1
book	0	1
booking	1	1
tex	1	0
text	1	1
john	1	0

Input/Output specification

Usage interface

The program `second` should have the following usage interface:

```
second <mappingFile>
```

where `<mappingFile>` is the name of the mapping file. You can assume that the mapping file will exist and that it is well structured. So, unlike assignment1, you don't need to check the structure of the mapping file.

In case no argument or more than one argument is provided, or the file names provided are invalid, the program should print "`invalid input`" and abort.

Input specification

Here, and below, let `m` be the maximum number of words in either the dictionary or data files. Every word is of length at most `k`. Let `n` be the number of unique words in the dictionary file.
Your input will be a mapping file, which contains lines of the form: `<dictFile> <dataFile>` dictFile and dataFile are the dictionary and data files for your program, respectively. They are plain text files with no special structure.

Output specification

Your program should generate several output files `outi.txt`, where `i` is the line number in mapping file. It means that you need to get mapping file as an argument to your program. Then each line in the mapping file has information about the dictionary file and the data file.
For example suppose line `j` in the mapping file is `dict_j data_j`. In this case you should produce `outj.txt`, which contains the described informations. Remember that you shouldn't have any spaces at the end of the lines in your output files. Also, you shouldn't have any empty lines in your outputs files .
The program should write all the unique words (see definition above) that occur in the dictionary file along with their various counts (See above), in lexicographical order, one word per line to the output files i.e. the output should have exactly `n` lines:

```
<word1> <count11> <count12>
<word2> <count21> <count22>
                .
                .
                .
<wordn> <countn1> <countn2>
```

such that

1. `<word1>, <word2>, <word3>` ... `<wordn>` is the lexicographical ordering of the unique words occurring in the dictionary,
2. `<counti1>` denotes the number of occurrences of `<wordi>` in the data file,
3. `<counti2>` denotes the number of proper prefixes of `<wordi>` in the data file,
4. `Line i` is `<wordi><space><counti1><space><counti2>`

Note that you can assume the dictionaries are non-empty.

Also note that if the mapping file has more than one line, you need to do same procedure for each line and create the result file (outi.txt) of every line.

For example, running `second` on the input describe above should produce the following output:

```
boo 0 1
book 0 1
```

```
booking 1 1
john 1 0
tex 1 0
text 1 1
```

Design & Implementation

Design

We recommend using the data structure "trie" (`http://en.wikipedia.org/wiki/Trie`), though you are free to use whatever data structures and algorithms you want as long as your implementation is reasonably efficient (See the Grading Section for detailed efficiency requirements).

Implementation

As an additional requirement, your code should implement and use the following functions:

- `void readDict(FILE *dict_file)`: The function takes a file pointer to the dictionary file and reads the unique words from the dictionary file and stores them in an appropriate data structure.
- `void matchStr(char* str)`: This function will take a word and search the data structure that holds the unique dictionary words in order to find matches and to update the various counts. This function should be used while scanning the data file for occurrences of dictionary words and their prefixes and superwords.
- `void printResult()`: This function will produce the output of the program.

You are free to add any other functions you need.

You are allowed to use functions from standard libraries (e.g., `strcmp()`) but you cannot use third-party libraries downloaded from the Internet (or from anywhere else). If you are unsure whether you can use something, ask us.

We will compile and test your program on the iLab machines so you should make sure that your program compiles and runs correctly on these machines. You must compile all C code using the gcc compiler with the `-Wall -Werror -fsanitize=address` flags.

Submission

You have to e-submit the assignment using Sakai. Your submission should be a tar file named `pa2.tar`. To create this file, put everything that you are submitting into a directory named `pa2`. Then, `cd` into the directory containing `pa2` (that is, `pa2`'s parent directory) and run the following command:

```
tar cvf pa2.tar pa2
```

To check that you have correctly created the tar file, you should copy it (`pa2.tar`) into an empty directory and run the following command:

```
tar xvf pa2.tar
```

This should create a directory named `pa2` in the (previously) empty directory.

The `pa2` directory in your tar file must contain 2 folders(first and second). If you don't want to do extra credit part you can leave second folder empty. Each folder should have:

- `readme.pdf`: This file should briefly describe the main data structures being used in your program, a big O analysis of the run time and space requirements of your program in terms of the parameters `k,m,` and `n` (See the First Input/Output and Second Input/Output sections above), and any challenges that you encounter in this assignment.

- `Makefile`: There should be at least two rules in your Makefile:
  1. `first`: build your `first` executable. (Or `second`, if you are doing the extra credit.)
  2. `clean`: prepare for rebuilding from scratch.
- source code: all source code files necessary for building your programs. Your code should contain at least two files: `first.h` and `first.c`.
- If you want to do extra credit part, you need to do all of the previous tasks for second part.

Grading guidelines

The grading will be done in two phases, an automated phase and a manual phase.

Automated grading phase

This phase will be based on programmatic checking of your program. We will build a binary using the Makefile and source code that you submit, and then test the binary for correct functionality and efficiency against a set of inputs. Correct functionality and efficiency include, but are not limited to, the following:

1. We should be able build your program by just running `make`.
2. Your program should follow the format specified above for the usage interface.
3. Your program should **strictly** follow the input and output specifications mentioned above. (**Note: This is perhaps the most important guideline: failing to follow it might result in you losing all or most of your points for this assignment. Make sure your program's output format is *exactly* as specified. Any deviation will cause the automated grader to mark your output as "incorrect". REQUESTS FOR RE-EVALUATIONS OF PROGRAMS REJECTED DUE TO IMPROPER FORMAT WILL NOT BE ENTERTAINED.**)
4. There will be three types of test cases: test cases for checking I/O specifications (as discussed in points 1-3) (class A test cases), small test cases for checking correctness (class B test cases), and big test cases that are designed to test the efficiency of your program (class C test cases): if you are simply storing all the dictionary words in an array and then matching every word in the data file against every dictionary word by doing a linear search, your program will most likely exceed the time limit when run on the big test cases.
5. We recommend that the running time and space complexity of your programs be roughly $\mathcal{O}(\texttt{mk})$ and $\mathcal{O}(\texttt{nk})$ respectively. Any correct program whose complexity matches the recommended values is guaranteed to work on all of the big test cases (**Note: Requests for re-evaluations of programs that fail on big test cases will not be entertained**).

Note: This phase will also involve detecting copying, if any. If two submissions are found to be identical, they will instantly be awarded zero points.

Manual grading phase

This phase will involve manual inspection of your code and the `readme.pdf` file provided by you along with the code. **Note that only programs that pass the class A and class B test cases will be eligible for grading in the manual phase**. Here are some guidelines you should follow in order to score well in this phase:

1. Explain your design (choice of data structure and algorithms) and the big O analysis carefully in the `readme.pdf` file.
2. Modularize your code i.e. split your code into pieces that make sense, where the pieces are neither too big nor too small.

3. Make sure your code is well-documented with comments. This does not mean that you should comment every line of code. Common practice is to document each function (the parameters it takes as input, the results produced, any side-effects, and the function's functionality) and add comments in the code where it will help another programmer figure out what is going on.
4. Use variable names that have some meaning (rather than cryptic names like strA).
5. Additionally, the following will make it easier for us to go through your code:
   - Define prototypes for all functions.
   - Place all prototype, `typedef & struct` definitions in header (.h) files.

Autograder

We provide the AutoGrader to test your assignment. AutoGrader is provided as autograder.tar.gz. Executing the following command will create the autograder folder.

```
$tar zxvf autograder.tar.gz
```

There are two modes available for testing your assignment with the AutoGrader.

First mode

Testing when you are writing code with a `pa2` folder

1. Lets say you have a `pa2` folder with the directory structure as described in the assignment.
2. Copy the folder to the directory of the autograder
3. Run the autograder with the following command
   $python auto_grader.py
   It will run the test cases and print your scores.

Second mode

This mode is to test your final submission (i.e, pa2.tar)

1. Copy pa2.tar to the autograder directory
2. Run the autograder with pa2.tar as the argument. The command line is
   $python auto_grader.py pa2.tar