CMPS 12B
Introduction to Data Structures
Programming Assignment 1

In this assignment you will implement the Binary Search and Merge Sort algorithms discussed in class. You may begin by studying the examples posted on the webpage at:

https://classes.soe.ucsc.edu/cmps012b/Winter19/Examples/Lecture/Recursion/

Your task will be to adapt these methods to operate on String arrays rather than int arrays. The key operation to alter is the comparison of array elements. Given Strings `s1` and `s2`, the expression `s1.compareTo(s2)<0` returns true if and only if `s1<s2` in the lexical ordering induced by the Unicode Character Set, `s1.compareTo(s2)==0` if and only if the Strings are identical, and `s1.compareTo(s2)>0` if and only if `s1>s2` in the same ordering. Go through the Binary Search and Merge Sort examples and replace `int` comparison with `String` comparison where appropriate. You will write a program called Search.java that takes command line arguments giving a file to be searched and target word(s) to search for. The executable jar file will be named Search so the command line will look like:

```
% Search file target1 [target2 target3 ..]
```

As always `%` represents the Unix prompt. The items in brackets `[]` represent optional arguments.

Input File Format and Program Output
Each line of the input file for this project will contain a single word, i.e. a string containing no spaces or tabs. Your program will determine whether or not the target word is amongst the words in the input file, print a message to stdout stating whether or not the target was found, and state the line on which the target was found, if it is found. For example suppose `in1` contains the following lines:

```
entire
beginning
possibly
specified
key
value
initial
before
dictionary
however
```

Suppose also that no file named `blah` exists in the current working directory. The following test runs illustrate correct program operation.

```
% Search
Usage: Search file target1 [target2 ..]
% Search in1
Usage: Search file target1 [target2 ..]
% Search blah before
blah (No such file or directory)
Usage: Search file target1 [target2 ...]
% Search in1 before happy possibly
before found on line 8
happy not found
possibly found on line 3
```

Observe that if the program is invoked with fewer than 2 command line arguments, it will halt with a usage message. If there are at least 2 command line arguments, but the first is not the name of a valid file, then the error message associated with `FileNotFoundException` is printed, followed by the same usage message. Note also that line numbering begins at 1. You may assume that your program will be tested only on input files that are properly formatted.

Program Operation

Your program should begin by determining the number of lines in the input file. See the examples `LineCount.java` and `LC.java` on the webpage at

https://classes.soe.ucsc.edu/cmps012b/Winter19/Examples/Programs/pa1/

to see some ways to do this. If the number of lines (and therefore the number of words) is *n*, allocate a `String` array of length *n* and store each word in that array. Your program will use Binary Search to find the target word(s). Write a function with the following heading to do this.

```
static int binarySearch(String[] A, int p, int r, String t)
```

Recall however that Binary Search requires the array to be in increasing order. As the above examples indicate, you cannot expect the input file to be sorted. Therefore you must first call Merge Sort on the array before you search it. Binary Search returns -1 if the target is not found, and a non-negative integer giving it's index in the array if it is found. But this will be the index of the target in the *sorted* array, not its index in the original *unsorted* array, which is what you need to determine its position in the input file. A tempting approach to overcome this problem is to simply do a *linear search* of the word array for the target. This would actually be the simplest way to write a program that transforms the input into the required output. *However this is not the task before you and not the point of the exercise.* You must sort the word array using Merge Sort, then search it using Binary Search, which you'll recall is more efficient than a linear search.

To determine the line number on which the target is found, alter the `mergeSort()` method so as to pass in an extra int array that keeps track of the original line number for each of the words in the array being sorted. Likewise, the `merge()` method should also pass such an array. Write functions `mergeSort()` and `merge()` with the following headings.

```
static void mergeSort(String[] word, int[] lineNumber, int p, int r)
static void merge(String[] word, int[] lineNumber, int p, int q, int r)
```

Recall from our discussion in class that `merge()` is where the real work of `mergeSort()` is done. When a call to `mergeSort()` returns, `lineNumber[k]` should be the line number where `word[k]` is located in the input file. For instance:

```
String[] word = {"ccc", "bbb", "ddd", "aaa"};
int[] lineNumber = {1, 2, 3, 4};

mergeSort(word, lineNumber, 0, 3);
// now:
// word is {"aaa", "bbb", "ccc", "ddd"}
// lineNumber is {4, 2, 1, 3}
```

Think of the sorting action as performing a permutation of the input array that places it in a certain order. The trick is to write `mergeSort()` and `merge()` so as to perform the same permutation on the subarray

`lineNumber[p....r]` as is performed on subarray `word[p....r]`. Actually `mergeSort()` will just pass this problem along to `merge()`, which as usual, is where the real work is done.

See the project description for lab2 to learn how to deal with command line arguments and file input. It is recommended that you begin your project by manually initializing a `String` array in function `main()`, write functions `mergeSort()` and `merge()` to sort that array, adapt function `main()` to read a file given on the command line, then work on getting the output to display line numbers of found targets.

What to turn in

Submit the files `README`, `Makefile`, and `Search.java`. Your Makefile must create an executable jar file called `Search`, and must include a `clean` utility that removes all `.class` files as well as `Search` itself. See Lab Assignment 1 to learn how to do this. Submit all files to the assignment name `pa1`.