CMPS 12B
Introduction to Data Structures
Winter 2009

Programming Assignment 1

Due Friday January 23, 10:00 pm

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. 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 `s1==s2`, 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 which takes command line arguments giving the file to be searched and the target String(s) to search for. The executable jar file will be named Search so the command line will look like: `%Search inputFile target1 [target2 target3 ..]`. (As always '`%`' represents the Unix prompt, and you do not type it. The items in brackets `[]` are optional arguments.)

Input File Format and Program Output

You may assume that your program will be tested only on files formatted in the following way. The first line of the file will contain exactly one integer *n* and nothing else. The next *n* lines will each contain one String. Your program will determine whether or not the target string is amongst the *n* strings on lines 2 through (*n*+1) of the file. Your program will print a message to stdout stating whether or not the target was found, and (optionally) report the line on which the target is located in the case when it is found. For example suppose `file1` contains the following lines:

```
10
entire
beginning
possibly
specified
key
value
initial
before
dictionary
however
```

Running the program on `file1` with several targets results in:

```
% Search
Usage: Search file target1 [target2 ..]
% Search file1 entire key happy dictionary 10
entire found on line 2
key found on line 6
happy not found
dictionary found on line 10
10 not found
```

Observe that line numbering starts on line 1, even though the String (i.e. the number) on line 1 is not amongst the items being searched. If you choose the option to *not* report where the target is found, the corresponding output would be:

```
% Search file1 entire key happy dictionary 10
entire found
key found
happy not found
dictionary found
10 not found
```

Actually this is not an option if you aspire to get an A or better on this assignment. In other words, if your program does not state the line on which target is found, your maximum possible score will be 90 out of 100 (which is an A-). The functionality you choose to implement must be clearly stated in your README file. It is recommended that you begin by completing the project without reporting where the target is found, submit your project, then start working on the additional functionality.

Program Operation

Your program should read the integer *n* on first line of the input file, allocate a `String` array of length *n*, then read in the next *n* lines, storing them in the array. Your program will use Binary Search to find the target string(s). Recall that Binary Search requires the array to be in increasing order. As the above example indicates however, you cannot expect the input file itself 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. You can simply test this return value to determine if the target was found. Unfortunately the index returned by Binary Search is not the original index of the target in the *unsorted* array, which is what you need to determine the position of the target in the input file.

One possible approach to overcome this problem would be to simply do a *linear search* of the input array for the target. This would actually be the simplest way to write a program which transforms the input into the required output. Let there be no mistake however that this is *not* the task before you. You must sort the input array using Merge Sort, then search it using Binary Search, which you'll recall is more efficient than a linear search.

The most efficient way to determine the original index of the target is by altering the `mergeSort()` method to pass an integer array which keeps track of the original index of each of the strings in the array being sorted. Likewise the `merge()` method must also pass such an array. The signatures of `mergeSort()` and `merge()` would then be:

```
public static void mergeSort(String[] A, int[] I, int p, int r)
private static void merge(String[] A, int[] I, int p, int q, int r)
```

Observe that `merge()` is declared as `private` since it is a helper function for `mergeSort()`, and in fact you'll recall from our discussion in class that that it is where the real work of `mergeSort()` is done. When a call to `mergeSort()` returns, array `I` should contain in it's k^{th} position the original index of the string now located in the k^{th} position of `A`. For instance:

```
String[] word = {"ccc", "bbb", "ddd", "aaa"};
Int[] index = {0, 1, 2, 3};


// index[k] is the position of word[k]
```

```
mergeSort(word, index, 0, 3);
// index[k] is the original position of word[k] before the sort,
// i.e. word = {"aaa", "bbb", "ccc", "ddd"} and index = {3, 1, 0, 2}
```

Think of the sorting task as performing a permutation of the input array which places it in a certain order. The trick is to write `mergeSort(String[] A, int[] I, int p, int r)` so as to perform the same permutation on the subarray I[p..r] as is performed on subarray A[p..r]. Actually `mergeSort()` will just pass this problem along to `merge()`, which as usual is where the real work is done.

If you don't know how to deal with command line arguments or file input, don't worry. This will be the subject of lab assignment 2, which will be due before this programming assignment is due. Begin your project by manually initializing a `String` array in your function main as above, then adapt it to read from a file after you have completed lab2.

What to turn in
Submit the files README, makefile, and Search.java. Your makefile must make an executable jar file called Search, and must include a clean utility which removes all .class files and the executable jar file. See lab assignment 1 to see how to do this. Submit all files to the assignment name `pa1`.