ECE 242: Data Structures and Algorithms- Fall 2017

Project 1: FunWithWords (Arrays+ Simple Sorting+ Search)

Due Date: **Deadline: see website and moodle for submission**

Description

This project is intended to familiarize you with operations on arrays such as Simple Sorting and Searching Algorithms. The goal of the project is to design two different dictionary database and implement a variety of applications that make use of searching, inserting, or deleting. The dictionary can be customized using two different implementations: (i) using an array representation to store the whole dictionary, or (ii) using multiple dictionary arrays containing words of the same length.

How to start

The words of the dictionary will be read from an input file. The project comes with three 'unsorted' dictionaries:

1. a tiny dictionary "tiny.txt" of only 3 words (good for testing/debugging)
2. a short dictionary "short.txt" of only 20 words (good for testing/debugging)
3. a large dictionary "english.txt" that contains 99,171 english words (for production).
4. a large dictionary "french.txt" that contains 139,719 french words (for testing).

The project also includes multiple 'source' files:

1. `DictionaryApp1.java` to `DictionaryApp5.java`: six application files used for testing (provided). **You are not allowed to modify them**.
2. `Dictionary.java` file containing the class object Dictionary (to complete).
3. `Collection.java` file containing the class object Collection (to complete)
4. `EasyIn.java` file containing the class used to read user input from keyboard (provided)

All the functionalities of the application files (presented in detail below) should be successfully implemented to obtain full credit. You need to proceed step-by-step, application by application. A list of methods to implement is described at the end of each section. These (main) methods are required to make sure that the application run smoothly. You are welcome to implement any other (sub-)methods if needed (e.g. for a cleaner implementation).

`DictionaryApp1.java`

At first, the application is loading a dictionary (which is non-sorted by default). Two options are possible. By default, the user will be asked to enter the 'name of the file', and the maximum size of the array that will be used to store the words. For example if you enter 'short.txt' and '50', you will obtain the following result:

```
>java DictionaryApp1

Welcome to the Dictionary App 1
===============================

Enter dictionary filename:
short.txt
Enter max fixed size:
50
--------------------------------------------------
The dictionary named 'short.txt' is of size 20
the dictionary is not sorted
--------------------------------------------------

Which sorting algo? (1:bubble, 2:Selection, 3:Insertion, 4:Enhanced Insertion (optional))
```

For convenience, another option to specify your inputs is also possible using the command line argument:
`>java DictionaryApp1 short.txt 50`

We note that the code is returning information about the dictionary that has been loaded such as the number of words and its status (sorted or unsorted). By default a loaded dictionary is always considered "unsorted". **Remark:** the number of words is not supposed to be known in advance.

The code will then ask the user to select between four simple sorting algorithms: 1-bubble sort, 2-selection sort, 3-insertion sort, 4-enhanced insertion sort. The fourth algorithm is optional (see at the end of the project for extra credit).

Once the dictionary is sorted, updated information about the dictionary will be printed as well as the time it took to complete the sorting. Finally, the sorted dictionary will be saved in a new file with an extension that contains the added attribute "-sorted".

Example of execution:

```
>java DictionaryApp1 short.txt 50

Welcome to the Dictionary App 1
===============================

--------------------------------------------------
The dictionary named 'short.txt' is of size 20
the dictionary is not sorted
--------------------------------------------------

Which sorting algo? (1:bubble, 2:Selection, 3:Insertion, 4:Enhanced Insertion (optional))
1
--------------------------------------------------
The dictionary named 'short.txt' is of size 20
the dictionary is sorted
--------------------------------------------------

OK dictionary sorted in 0ms and save in file 'short.txt-sorted'

Goodbye!
```

We note that if the original 'short.txt' looks like:

```
morning
electrical
ate
dictionary
simon
class
of
tea
animal
school
this
cases
file
cat
code
eat
act
computer
sorting
screen
```

The file database 'short.txt-sorted' finally looks like:

```
act
animal
ate
cases
cat
class
code
computer
dictionary
eat
electrical
file
morning
of
school
screen
simon
sorting
tea
this
```

Remark: if the user select a maximum size array that is smaller than the number of items, the dictionary will be considered full and this info will be returned to the user. For example using maxsize=10 for the 'short.txt' file:

```
>java DictionaryApp1 short.txt 10

Welcome to the Dictionary App 1
```

```
===============================

--------------------------------------------------
The dictionary named 'short.txt' is of size 10
the dictionary is not sorted
Warning: the dictionary is full!
--------------------------------------------------

Which sorting algo? (1:bubble, 2:Selection, 3:Insertion, 4:Enhanced Insertion (optional))
1
--------------------------------------------------
The dictionary named 'short.txt' is of size 10
the dictionary is sorted
Warning: the dictionary is full!
--------------------------------------------------

OK dictionary sorted in 0ms and save in file 'short.txt-sorted'

Goodbye!
```

The file database 'short.txt-sorted' includes now only the first 10 words of 'short.txt' that have been loaded and sorted:

```
animal
ate
class
dictionary
electrical
morning
of
school
simon
tea
```

What you need to implement in `Dictionary.java`:

1. The `loadDictionary` method that loads the Dictionary (you can take inspiration of the methods in the 'ReadFilesExample.java' file provided in the website).

2. The `info` method that returns the information about the dictionary as requested (see examples above). **Remark:** a method `isFull` to check if the dictionary is full can be useful for both loadDictionary and info methods. You can also think of other methods if needed e.g. `getSize`, and `getName`.

3. The `sortBubble` method, the `sortSelection` method, and `sortInsertion` methods. The `sortEnhancedInsertion` is optional (if you do not plan to do it, just create a dummy/blank method since it is needed to compile). **Remark:** You can sort words lexicographically using the "compareTo" field for 'String' (it works also with the class 'Character'), example:

   ```
   String str1,str2;
   str1=Tesla;
   str2=Edison;
   if(str1.compareTo(str2)==0)
    System.out.println(str1+ is lexicographically equal to + str2);
   if(str1.compareTo(str2)>0) // that's the one for this example
    System.out.println(str1+ is lexicographically greater than + str2);
   ```

```
if(str1.compareTo(str2)<0)
 System.out.println(str1+ is lexicographically less than+ str2);
```

4. The `saveDictionary` method to save the dictionary in a file. **Example:** In order to save data into a file: here is an example on how to write a string and an integer into the file 'lecture':

```
String course=''ECE''
int n=242;
try {
            PrintWriter out = new PrintWriter(``lecture'');
            out.println(course);
            out.println(n);
            out.close();
        }
        catch (IOException e)
           {
               e.printStackTrace();
            }
    }
```

Other requirements: In a pdf file document, answer the following questions:

- Provide an 'informative' plot of Time vs N (number of items) for all algorithms (only one plot). To obtain these results, you could load multiple times the dictionary 'english.txt' while changing the maxSize input of the array (up to 99,171). Comments on running times and big-O?
- Let us assume we decide to run using the full dictionary 'english.txt-sorted' as input (which is sorted), report the times obtained for all sorting algorithms (you will be sorting a dictionary that is already sorted). Comments on running times and big-O?

DictionaryApp2.java

Like in Application 1, the code loads the dictionary as input file, it is asking the maximum size and also ask the question if the dictionary is sorted or not. Alternatively, you can also enter all the three information using the command line.

After printing the dictionary information on screen, it is asking how many words you would like to search at random. If the dictionary is not sorted to start with, it will search all the random words using a linear search. If the dictionary is sorted, the code will ask what types of search you would like (linear or binary).

When the search is done, the code returns the time it took to complete, as well as the average number of steps it took.

Here are some examples of execution:

```
>java DictionaryApp2

Welcome to the Dictionary App 2
===============================

Enter dictionary filename:
short.txt
Enter max fixed size:
50
Is the dictionary sorted or unsorted (s or u)?:
u
--------------------------------------------------
The dictionary named 'short.txt' is of size 20
the dictionary is not sorted
```

```
--------------------------------------------------

How many random words you would like to search?:
10
Ok search done in 1ms
Average number of step is 12
Goodbye!
```

or

```
>java DictionaryApp2

Welcome to the Dictionary App 2
===============================

Enter dictionary filename:
short.txt-sorted
Enter max fixed size:
50
Is the dictionary sorted or unsorted (s or u)?:
s
--------------------------------------------------
The dictionary named 'short.txt-sorted' is of size 20
the dictionary is sorted
--------------------------------------------------

How many random words you would like to search?:
15
Which algo would you use: 1- linear search, or 2- binary search:
2
Ok search done in 1ms
Average number of step is 3
Goodbye!
```

What you need to implement in `Dictionary.java`:

1. The `setSorted` method that changes the status of the dictionary from sorted (if 'true' value) to unsorted (if 'false' value).

2. The `isSorted` method to check the status of the dictionary.

3. The `getRandomWord` method to select a word at random in the dictionary. **Example:** The Random class in java works this way:

   ```
   Random rnd= new Random();
   int index=rnd.nextInt(out); //select random number in [0:out-1]
   ```

4. The `linearSearch method` that will perform the linear search, and return true (if search is successful) or false. **Remark:** the search methods could update privately the variables `searchStepCounter`, and also `searchIndex`.

5. The `binarySearch method` that will perform the linear search, and return true (if search is successful) or false.

6. The `getStep` method that returns the number of steps it took to perform the search.

Other requirements: In a pdf file document, answer the following questions:

- For the 'english.txt-sorted' dictionary, provide a plot of Time vs M (number of word search). for both linear and binary search (only one plot). To obtain these results, you could load multiple times the dictionary 'english.txt-sorted' while changing the number of words to search (from M=1 to 10000) Comments on running times and give the big-O in function of N (number of words in the dictionary) and M?

`DictionaryApp3.java`

This application is similar to DictionaryApp2, but rather than searching a random list of words, the user would have to insert a new list of words inside the main dictionary.

Let us suppose that our main dictionary is 'short.txt' (unsorted) and one want to insert a list of words contains in 'tiny.txt', we get:

```
>java DictionaryApp3

Welcome to the Dictionary App 3
===============================

Enter dictionary filename:
short.txt
Enter max fixed size:
50
Is the dictionary sorted or unsorted (s or u)?:
u
--------------------------------------------------
The dictionary named 'short.txt' is of size 20
the dictionary is not sorted
--------------------------------------------------

Enter a file with a list of words to insert:
tiny.txt
--------------------------------------------------
The dictionary named 'tiny.txt' is of size 3
the dictionary is not sorted
--------------------------------------------------

--------------------------------------------------
The dictionary named 'short.txt' is of size 23
the dictionary is not sorted
--------------------------------------------------

Ok dictionary updated in 0ms and save in file 'short.txt-add'

Goodbye!
```

The code prints both the information of the main dictionary, the new temporary dictionary that contains the list of words and finally the information about the updated main dictionary. The updated dictionary will be saved in a new file with the extension '-add'.

Since we started with an unsorted dictionary in this example, the new words will be inserted at the end of the main dictionary. We then get for the 'short.txt-add' file:

```
morning
electrical
ate
dictionary
simon
class
of
tea
animal
school
this
cases
file
cat
code
eat
act
computer
sorting
screen
algorithm
numerical
feast
```

Remark: we note that the three last words are written in reverse of their original ordering in 'tiny.txt'. This is due to our implementation using the `deleteLast` method (see code).

In turn, if the user enters a sorted dictionary 'short.txt-sorted' with 'tiny.txt' for the list of words to be added, we get:

```

>java DictionaryApp3

Welcome to the Dictionary App 3
===============================

Enter dictionary filename:
short.txt-sorted
Enter max fixed size:
50
Is the dictionary sorted or unsorted (s or u)?:
s
--------------------------------------------------
The dictionary named 'short.txt-sorted' is of size 20
the dictionary is sorted
--------------------------------------------------

Enter a file with a list of words to insert:
tiny.txt
--------------------------------------------------
The dictionary named 'tiny.txt' is of size 3
the dictionary is not sorted
--------------------------------------------------

--------------------------------------------------
The dictionary named 'short.txt-sorted' is of size 23
the dictionary is sorted
```

```
--------------------------------------------------

Ok dictionary updated in 0ms and save in file 'short.txt-sorted-add'

Goodbye!
```

The resulting new file 'short.txt-sorted-add' will contain the sorted list of words:

```
act
algorithm
animal
ate
cases
cat
class
code
computer
dictionary
eat
electrical
feast
file
morning
numerical
of
school
screen
simon
sorting
tea
this
```

What you need to implement in `Dictionary.java`:

1. The `getSize` and `isFull` methods (you may have implemented them with Application1).
2. The `deleteLast` method that is removing the last word of the dictionary and decrease the number of active words by 1.
3. The `insert` method that first checks if the dictionary is unsorted or sorted; if unsorted, the new word would be inserted at the end; if sorted, the code must perform a **binary search** to find the right position, then the word is copied while the rest of the words are shifted up. **Remark 1:** if the word is not found using binary search, it would be useful for the `binarySearch` method to return the `searchIndex` with the value of `lowerBound` (that should be the correct position for inserting the new word).
   Remark 2: For the `insert` method, you may want to consider separately the special case of an empty dictionary (would be useful for the next application).

Other requirements: In a pdf file document, answer the following questions:

- Let us assume that we have a list of M words to insert into a dictionary containing N words. What would be the big-O for the insert method (i) if the dictionary is unsorted, (ii) if it is sorted.

DictionaryApp4.java

Here we propose to create a collection of dictionaries of same size words from an original input dictionary file. The original dictionary will be considered unsorted by the code (even if you may choose a sorted dictionary as input) and it is assumed to be of maximum size of 300,000.

The user is asked to enter the name of the main dictionary, and the code is then creating the collection of dictionaries, print some information and saved all the new multiple dictionaries into their respective files with extension "i" where "i" represents the number of letter in the words. Similarly to Application2, the code will use a random list of words but it will now search them within the collection.

Here an example of execution with 'short.txt':

```
>java DictionaryApp4

Welcome to the Dictionary App 4
===============================

Enter the main dictionary filename (maxsize<500000):
short.txt
--------------------------------------------------
The dictionary named 'short.txt' is of size 20
the dictionary is not sorted
--------------------------------------------------

--------------------------------------------------
The collection named 'short.txt' contains 10 dictionaries
dict --> size
1 --> 0
2 --> 1
3 --> 5
4 --> 3
5 --> 3
6 --> 3
7 --> 2
8 --> 1
9 --> 0
10 --> 2
--------------------------------------------------

How many random words you would like to search?:
10
Ok search done in 0ms
Average number of step is 1
Goodbye!
```

For this example, 10 is the maximum number of letters one can find in a word while considering all the words in 'short.txt'. As a result, the collection will contain 10 new dictionaries. The files 'short.txt1' to 'short.txt10' will also be created. Here 'short.txt1' and 'short.txt9' will be empty (meaning that there is no word containing only 1 or 9 letters in the main 'short.txt' dictionary). As an example the file 'short.txt6' looks like:

```
animal
school
screen
```

Finally, we note that for creating the collection, the main dictionary may have been erased (emptied), so we would need to reload it before pursuing with the search.

What you need to implement in `Collection.java`:

1. A constructor that creates an array of dictionaries from the main dictionary as argument. The number of dictionaries could be obtained from a new `maxSizeWord` method in the `Dictionary` class that scans linearly all the words. We will consider a fixed size of 300,000 as well for each dictionary (to ease the implementation). We also forced the new created dictionaries to all be **sorted** by default. The collection could be filled up using the `deleteLast` and `insert` methods of `Dictionary` used/implemented in Application3. **Remark:** Because the sorted flag is 'true', the resulting new dictionaries will all be sorted (due of the insertion procedure in a sorted array).

2. The `info` method that is printing out all the information as requested (see example above).

3. The `saveCollection` method that will create all the new files with their proper extension from 1 to maxSizeWord.

4. The `search method` that acts in two steps: (i) identify the number of letters that composed the word to search, (ii) search the word within its corresponding sub-dictionary.

5. The `getStep` method that returns the number of steps it took to search the sub-dictionary in the collection.

DictionaryApp5.java

Similar to Application4, but we are now using the collection to crack the word code of a lock. Let us suppose we have a lock composed of 3 letters with the following two options by letters:
(c,e) for letter 1
(a,p) for letter 2
(t,i) for letter 3
We could form $2^3 = 8$ combinations (number of options to the power the number of letters). This could increase significantly if we add more letters or more options, for example 5 letters and 8 options gives $8^5 = 32768$. However, most of locking systems are using words that can be found in the dictionary (easy to remember), so we could take advantage of this in application5.

Example of execution using the dictionary 'short.txt':

```
>java DictionaryApp5

Welcome to the Dictionary App 5
===============================

Enter the main dictionary filename (maxsize<500000):
short.txt
--------------------------------------------------
The dictionary named 'short.txt' is of size 20
the dictionary is not sorted
--------------------------------------------------


--------------------------------------------------
The collection named 'short.txt' contains 10 dictionaries
dict --> size
1 --> 0
2 --> 1
3 --> 5
4 --> 3
```

```
5 --> 3
6 --> 3
7 --> 2
8 --> 1
9 --> 0
10 --> 2
---------------------------------------------------

How many letters to crack?
3
How many possible options by letter ?
2
Enter all the options for letter 0 :c e
Enter all the options for letter 1 :a p
Enter all the options for letter 2 :t i
cat
eat
Number of words found 2
Goodbye!
```

We see that the beginning of the execution is similar to application4, then the code is asking to enter all the options for the respective letters of the lock. After a search, the code is returning some possible words that are in the dictionary.

What you need to implement in `Collection.java`:

1. The `crackLock` method that is going to consider $c = (\text{nboption})^{\text{nbletter}}$ combinations, search them in the collection (the corresponding sub-dictionary composed of 'nbletter' words), and return all the words found. Since it could be quite complicated to generate all these combinations (without using recursion), you will be using a stochastic approach where the code will generate $5 * c$ random combinations of letters (for example: 1st letter chosen at random among all its possible options, etc.). This technique may not be able to find all the acceptable words at the end but it generally works well if the number of random combinations is large enough. In addition, as soon as a word is found you will remove it from the dictionary (to avoid duplicates if you get the same random combination). As shown in the example above, the code must print out all the words found and the number that has been found.

What you need to implement in `Dictionary.java`:

1. The `delete` method that is needed by the `crackLock` method in the `Collection` class. This works similarly to the insert method using a binary search to search of the word, and if it is found, shift down the rest of the array.

Other requirements: In a pdf file document, answer the following questions:

- Report all the words found (and how many) using the dictionary 'english.txt' and the following combination of 6 for 4 letters: (q, e,t,u,l,s); (r, u,f, l,o,q); (k,m,s,a,r,w);(w,x,s,o,p,g);
  Remark: You must run the code a few times, to make sure you can randomly generate all of them.

Submission

Submit a single zip file on moodle by the due date (no extension, no exception) composed of: All your source files (*.java) including EasyIn.java, all (*.txt) files, a README (text) file, your "pdf" report file that answer all the questions asked all along this document. Include all information in the README file e.g.: Your name, ID, e-mail, what have you done or what does not work: Indicate which applications have been fully completed (or what has been partially completed in each application), other special features or assumptions made in your program we should know about, and everything else you want to say. Note: Your zip file must be self-contained so the code will nicely compiled without the need of extra-files.

Grading Proposal

This project will be graded out of 100 points:

1. Your program should implement all basic functionality and run correctly. (75 points).
2. Overall programming style: source code should have proper identification, and comments. (10 points).
3. Report and answering to questions (15 points).

Extra Credit (no help from TA or instructors)

(5pts) Implement Enhanced Insertion sort as a fourth option for the sorting algorithms. Update all the plots/reports with results/timings.
(5pts) If you insert the french dictionary inside the english one, how many words are the same?