Assignment2 - Complete Matching Words via Search Key Report with LaTeX

Berk Gülay

07.04.2017

1 Brief Overview of the Program

1.1 Problem:

Writing a program that implements matching words with search key for a given N term as input. The program should predict k(another input) possible words as choices and while predicting program has to use Search Key from user and match this Search Key with given N terms to reach correct result as predicted output.

1.2 Used structures and base algorithms in my solution:

Structures:
1- **Term Class** with weight and word fields to store each query terms as item.
2- **Dict Class** with Term List,Term Num and File Name.
2.1) Term List: Includes all given queries as source to predict.
2.2) Term Num: Number of all given terms.
2.3) File Name: Name of input file that includes all queries in it as text lines.

* Dict Class also implements Completable Interface to guarantee that it can complete words according to Search Key. Completable Interface includes "word sort", "weight sort" and "found terms by Binary Search" methods.
3- **Completer Class** completes Search Key by using dict Class and it's terms as source. It's field is a "dict" object because of this. Fills this dict object and finds first k(user input) terms as result for Search Key.

Base Algorithms:
1- **Binary Search:** to search given search key among all query word's inside.
2- **Quick Sort:**
2.1) For lexicographic sort, adopted quick sort with compareTo() usage instead of less() to sort all words(Strings). Also adopted for usage of Term class that includes words we need in it.
2.2) For long sort, just adopted for usage of Term class that includes weights(Long) we need in it.

1.3 My Solution and Algorithm:

*** Firstly my program completes Search Key by taking it as case sensitive. So for instance, key "Go" , "go" , "GO" all will be considered differently and will be checked/completed as different words if possible of course.
*** Program asks for necessary inputs to user in "Console" with Scanner class usage at the beginning of the program.

Basic Steps:
1- Sort query words with adopted Quick Sort algorithm

2- Find first matching word according to Search Key by using Binary Search and then find all other matching words just by traversing up and down from first found term. All other matching terms will be next to first found matching term because they are in lexicographic order(sorted by words).
3-Sort found terms by their weights in descending order and take first k(input) terms among them and give as result.

Algorithms Details:
1- Take inputs from user (Input.txt as source to dict Class, k as number of output terms, Search Key)
2- If given text file is same as previous successful(error free) file, do not change source "dict" and "completer". Just find matching terms with new Search Key and k values by using same "completer".
3- If given text file is different, read it and fill completer's "dict" object with this text input source. Sort "dict" object in our "completer" object by it's Term's word with Quick Sort Algorithm usage in adopted way for Strings.(lexicographic sort A to Z)
4- Find first matching term with Search Key among all queries in "dict" object by Binary Search Algorithm. Then find all other terms that are matching with Search Key by simply checking upward and downward from this first matching term. Store all found terms in Terms array. (It saves time because it finds each term with just 1 check and if term doesn't match, method does not keep checking other terms after that time.)
* If Search Key is longer than checking term, they can not match. Keep searching in Binary Sort for other term in next part. But if you check upward/downward from first term(found with Binary Search above) in sorted Terms List, just stop checking up or down because no other term can match at upside/downside, if search key is longer than your present(checking) value.
5- Sort found terms by their weights in descending order with Quick Sort algorithm usage.
6- Give first k matching terms from found term's list that is weight sorted as result for Search Key. Search Key is completed with right k values(most used, most weighted) as result.
* If k is bigger than all found term's total num, user will take a warning about it and all found term's will be given as result to Search Key completion.

!* For extra knowledge about solution/algorithm please check source code. It also includes really sufficient explanation and information about all methods, classes, important/hard to understand lines and assigned values/fields.

1.4 Possible Warnings/Errors(All Caught):

1) IO Exception - If there is no such a file in directory. File name may be invalid/incorrect or File may not be exist in program directory.
! User will be warned about that and asked for changing file name.
2) k input Exception - If entered k input as output number is invalid.(Less than 1)
! User will be warned about that and asked for changing k number. (Must be bigger than 0)
3) Term's Query Weight Exception - If query weight is negative(less than 0)
! User will be warned about that and asked for changing input file or that input file's invalid query weight.)
4) Term's Query Word Exception - If query word is null(not exist)
! User will be warned about that and asked for changing input file or that input file's invalid query word.)
5) No matching item found Exception - If your Search Key can not be completable with any item/term in source data. Your search key may be too long or meaningless value. Another possibility is our source data(input text terms) may not be sufficient for completing your Search Key or compatible with your Search Key
! User will be warned about that and asked for changing either Search Key or Input Text File)

***If same input file will be used just with different lines in it, user should quit and run the program from beginning to be able to take new changes from same input file.(User will also see a warning about this in each successful program execution at bottom line with quit message.)

2 Analyse of my Program

2.1 Time Complexity of my Program:

Measured Times for Program:
This program runs in 1 second with "alexa.txt" input file(24500 KB text file and 10^6 text lines) if file is not memorised (so program takes file to memory and sort it then gives completed Search Key results.)
But if file is memorised(so same as prev. file just Search Key or k value is different), program runs in much more quickly than 1 second. Run time is 2.10^{-2} *seconds* in this case.(Instant time)

Analyse:
This program's most costly part by time is Word Sorting as you can see from measured time results above. If no "word sort" is done(mentioned in second paragraph of measured times part above) program's time complexity comes really light for those kind of heavy input values.(Because just linear loops run and constant time operations done like "print" or "assign/reassign" except Quick Sort.)

For "Word Sort" operation as I mentioned in my algorithm part above Quick Sort alg. is used in adopted way but its complexity is same as usual Quick Sort. SO, most heavy part's time complexity of this program takes $N.logN$ average comparison and time value. Most probably it is not possible but in worst case $\sim N^2$ comparison would be done if terms are sorted/reverse sorted or almost sorted. Therefore program's time complexity value is O($N.logN$) in each execution. And program is really light for heavy input files(like "alexa.txt" file mentioned above) even they will be sorted(if they are not memorised and sorted already) for result. If they are memorised it takes instant time to see completed result even for really heavy files or inputs as mentioned in measured times part above.

2.2 Why I used this structures/algorithms in my Program?

Structure usage:
1- **Term Class** provides easy management for each Term's weight and word values. Also provides easy sort chance by weight or word as I wanted.
2- **Dict Class** provides access all terms at the same time and I can easily complete search keys by this "dict" object's terms which are in List. Also Term Num and File Name fields provides me check and print/access which file and how many terms I am managing right that moment.
* Dict Class also implements Completable Interface to guarantee that it can complete words according to Search Key. Completable Interface includes "word sort" method to sort all terms by their word value in "dict" object's List, "weight sort" method to sort by weight found terms from this "dict" object according to input Search Key and "found terms" method to find matching terms with Binary Search from "dict" object's List.
3- **Completer Class** provides to know which "dict" object is taken and filled as source. Also I can control program flow for Search Key completion with this classes methods. I can put in order other class methods according to needed operation and manage them easily to take result.
Algorithm usage:
1- **Binary Search:** provides O($logN$) complexity for searching and finding first matching term with Search Key input. So it is really light alg. complexity for searching, it is lighter than linear time and I can also find each other term in constant time after one Binary Search is done.
* After one Binary Search operation one matching term is found in logarithmic time and I continue to find other matching terms just checking next(upward and downward) items of this found term. This provides me constant time to find each other matching term which is the lightest possibility thanks to "word sorted" array.

2- **Quick Sort:**
Quick Sort is compatible for either integer sort or word sort(lexicographic sort) and sorts given array completely unlike Heap Sort. This provides me search all terms in constant time after 1 binary search operation because all terms are sorted by their word value and needed terms(by Search Key) comes next to each other always. Moreover quick sort is an in order sorting algorithm and does not need an auxiliary extra array which causes really high space complexity for big/heavy input values like ours. Most importantly it's most probable time complexity(O($N.logN$)) is lightest one among other efficient sorting algorithms. All of these reasons makes Quick Sort most proper algorithm for sorting arrays in this program.

!* For extra knowledge about program please check source code. It also includes really sufficient explanation and information about all methods, classes, each important/hard line and assigned values/fields

3 References:

1 - `https://www.tutorialspoint.com/java/`
2 - `https://docs.oracle.com/javase/7/docs/api/`
3 - `https://www.tutorialspoint.com/java/util`
4 - `Algorithms Fourth Edition, Sedgewick and Wayne`