17/02/18 170002815 160027195 170006953 CS1006 – Practical 1

Build Instructions

Mercurial Link: https://zz35.hg.cs.st-andrews.ac.uk/Project1/

This program has two options for searching. The first one is searching by prefix order, and the other is return the words by searching for first k characters where the r is chosen by users.

Please follow the instructions to run this program.

1. Compulsory Part (In Terminal).

Step 1: javac *.java

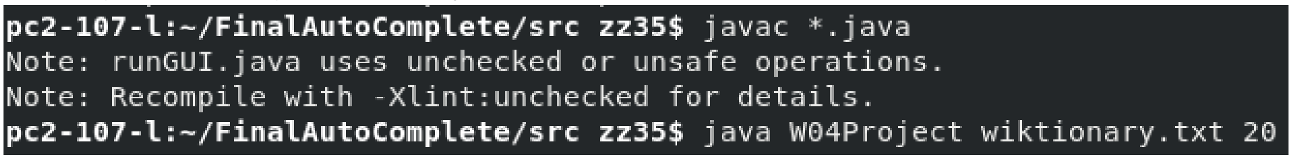
Step 2: java W04Project <filename> <the number of words are presented>

(Notice: there is a space between each other.)

Example:

javac *.java

java W04Peoject wiktionary.txt 20



```
pc2-107-l:~/FinalAutoComplete/src zz35$ javac *.java
Note: runGUI.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
pc2-107-l:~/FinalAutoComplete/src zz35$ java W04Project wiktionary.txt 20
```

2. Two optional Part (In Auto Complete Window).

(1) Sort by prefix order

Step 1: enter the query string.

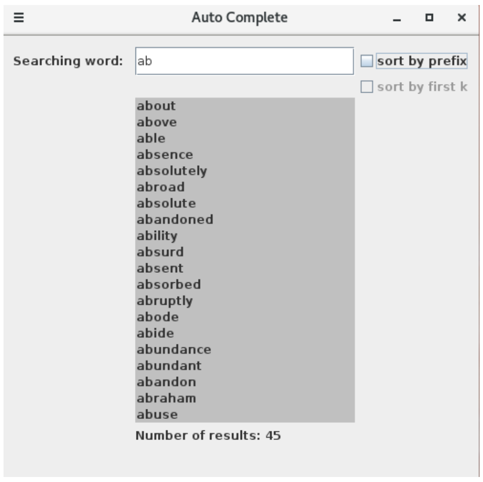
Step 2: click the checked box called "sort by prefix order".

Step 3: If you want to search another query string, delete the content in the text field and begin with step 1 again and again.

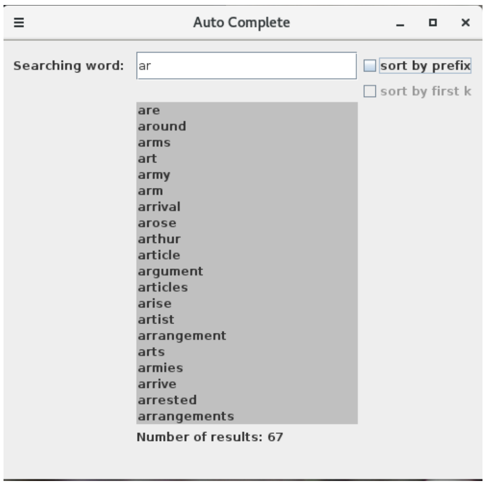
Step 4: If all the output are satisfied, close the window.

(Notice: If you want all the results with their weight, please open the terminal)

Example: searching "ab" and "ar" and present 20 words in the window



```
ab
about
above
able
absence
absolutely
abroad
absolute
abandoned
ability
absurd
absent
absorbed
abruptly
abode
abide
abundance
abundant
abandon
abraham
abuse
Number of results: 45
```



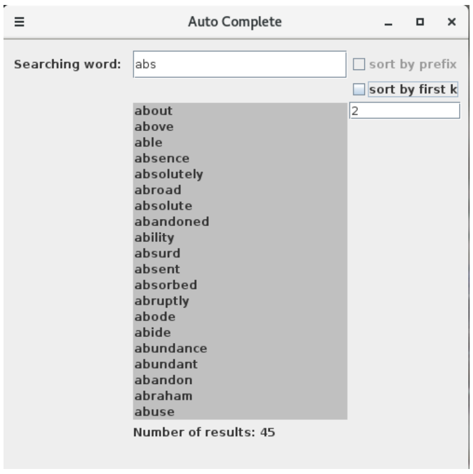
```
ar
are
around
arms
art
army
arm
arrival
arose
arthur
article
argument
articles
arise
artist
arrangement
arts
armies
arrive
arrested
arrangements
Number of results: 67
```

(2) Sort by first k

Step 1: Click the checked box called "sort by first k"

Step 2: A small text field is shown, and enter the value of k in that small field.
Step 3: Enter the query String in the big text field.
Step 4: Click the checked box again.
Step 5: If you want to try more different example, revise the content in the text fields, and click the checked box again.
Step 6: If all the output are satisfied, close the window.

Example: searching "abs" and "arb" by choosing first 2 characters, and presenting 20 words in the window.



Auto Complete

Searching word: abs

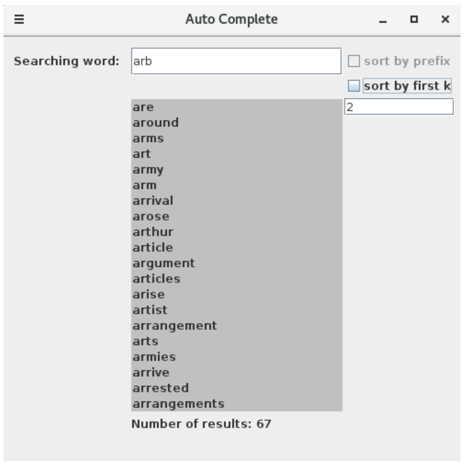
sort by prefix

sort by first k

2

```
about
above
able
absence
absolutely
abroad
absolute
abandoned
ability
absurd
absent
absorbed
abruptly
abode
abide
abundance
abundant
abandon
abraham
abuse
```

Number of results: 45



Auto Complete

Searching word: arb

sort by prefix

sort by first k

2

```
are
around
arms
art
army
arm
arrival
arose
arthur
article
argument
articles
arise
artist
arrangement
arts
armies
arrive
arrested
arrangements
```

Number of results: 67

Design

Overview:

Our program consists of 11 Java classes: 3 files included in the API (Term, BinarySearchDeluxe and Autocomplete), 4 comparators, 1 enumerator, 1 to implement all of these, and 2 files to combine the implementation with a GUI.

1. In Part 1, 2, 3

1.1 Term:

The design of the program begins with the creation of the three term classes that serve as the input file for the auto complete. The first, is all the terms by weight order. After reading in the txt file, a trim method removes the superfluous white-spaces between the beginning of the line, the weight and the word. Next, using a limited split method, the weights were separated from the words via a split at the white-space between them. Finally, by selecting only the odd members of the split string array, an array list is filled with just the words, ordered by their weight.

The term class implements all the API methods and constructors by instantiating the 4 comparators and initialising two public attributes (query and weight) respectively. None of the methods proved particularly useful, since the majority contain only line of code, leading it to be more efficient (and less confusing) to simply instantiate a new comparator, rather than calling the method multiple times.

1.2 Comparators:

The four comparators used in this practical are named Sorter, Lex_Comparator, LexR_Comparator and Weight_Comparator. All comparators are used to compare instances of the Term class.

The Sorter comparator compares simply returns the comparison of two strings by making use of the "compareTo" and "toLowerCase" String methods. This comparator was designed to be used when sorting the words in the text file into complete lexicographical order. "toLowerCase" is used, since real life implementations of autocomplete include suggestions which are different cases to the query, for example "CIA" is returned if the query is "cia".

The Lex_Comparator comparator is very similar to the Sorter comparator, however the String method "startsWith" is used rather than "compareTo", since this comparator will only consider certain sections of the strings, rather than the complete strings. To combat a StringOutOfBoundsException, an if-statement, along with appropriate return statements, is used to see if the string the search query is greater in length than that which it is going to be compared against is. This comparator was designed to be used during the binary search process if a complete lexicographical search was required.

The LexR_Comparator is very similar to Lex_Comparator, however it was designed for use during the binary search when only a constant of r characters are required to be searched; in order to do this, the compare method firsts creates a substring of the search term, before proceeding with the same search as the Lex_Comparator.

1.3 BinarySearchDeluxe:

Simple binary search is relatively easy to implement using Java, however with the added complexity of finding the first and last index of the requested search, the binary search became one of the trickiest parts of the project to implement.

Before the BinarySearchDeluxe class was created, a simple numeric binary search was designed in Java, since it was easier to see the principle phrases of code required in Java without the added complexity of comparators and extra comparisons. The main difference between a simple binary search and the "Deluxe" version is that more than one index is needed to be checked at the different conditionals, for example to check for a correct index, the index adjacent to the index being checked is also needed to be compared to the search term – this means several more conditionals are needed at each step to cover all possible situations which could occur.

To complete both the firstIndexOf and lastIndexOf methods, three extra checks after the iterative part of each search are used. Two checks are used for arrays sized 2 and one check is used for arrays sized 1.

1.4 Autocomplete:

The Autocomplete class brings together the Term and BinarySearchDeluxe classes. The two methods and constructor mentioned in the API are implemented and produce the desired output.

This class contains 3 attributes: searchType, terms and r. “terms” is an array of instances of terms and is initialised when this class is instantiated through the defined constructor. “r” and “searchType” are both used during the allMatches and numberOfMatches methods to decide on what type of search to complete; they are both static so that they can be accessed during the set-up of the program, before this class is instantiated.

An enumerator nested class called “SearchType” is contained within this class, and means that searchType can only be one of two values, therefore reducing potential for an error in our program.

1.5 Main:

This class implements and produces an output from all back-end classes. This class contains a main method and begins by setting up the program from user input. User input is gathered by a BufferedReader reading System.in, and the decisions are made by the program using switch statements, because switch statements are neat and are easy to edit if a change is needed to be made by a team member who did not created the Main class.

After retrieving the desired file to read from the user, this class reads the file, splits each line into respective words and weights (ignoring the first, since it doesn’t contain useful information), inputs this information into a list, converts this list into a list of newly instantiated Term objects, sorts this list by using a Sorter comparator, and then turns this sorted list into an array of Term objects for the Autocomplete class to handle.

By instantiating an Autocomplete object and calling its methods in a while loop, the program then finds and outputs the desired results.

2. In the part 4 and the whole project

2.1 GUI: Layout and action listener:

Group Layout, action listener, and two command-line arguments are utilized in this part.

The GUI design begins by creating all of the components before adding them to a grouped layout. Next, a secondary window is created, with jLabels that will fill it with suggested words as soon as a query types. A Document Listener is used to detect when a query is being typed and serves to update the binary search function with what it should fill the suggestion window with. Finally, there are a series of tick boxes that allow the user to select which ordering system is used by the binary search.

Two command-line arguments are one is for the basic text file, and the other is for the number of words that user want to present.

If the “sortK” checked box is clicked, the other checked box will be disabled. At the same time, the option for user searching will be passed to the back-end, and the text field for entering the value of the first k will appear. On the other hand, if the user clicked the checked box called “sort by prefix order”, shares the similar situation.

After the checked box is clicked, the content of the user input will send it to the back-end by using the method “getText()” in action listener.

2.2 Dealing with exceptions:

In the back-end. There is one potential exception in this part. When there are no words are matched, the array tends to be null which means the length of the array is infinite, and the for loop is endless. Therefore, “ArrayIndexOutOfBoundException” is caught is this part. In this exception, the program will print out “Words no found!” in the terminal. In addition, if the user input is empty, the program will print out notice.

Furthermore, if the two command-line arguments are passed incorrectly, the usage notice will be print out in the terminal. Another exception is the users did not enter the number of first k values, which will cause “NumberFormatException”. By default, the k value is set to be 2.

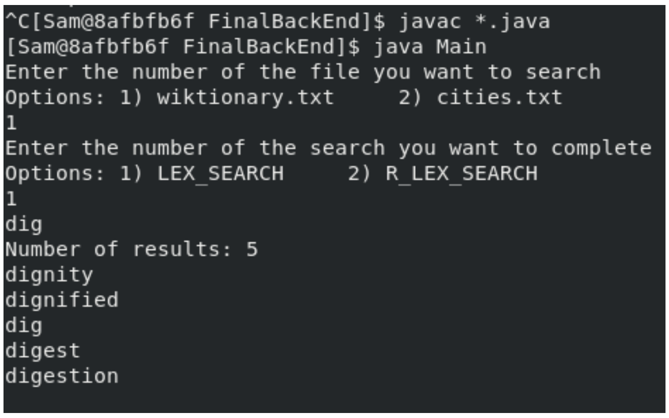
Testing

This part is divided into three parts, which are function testing, performance testing and final testing.

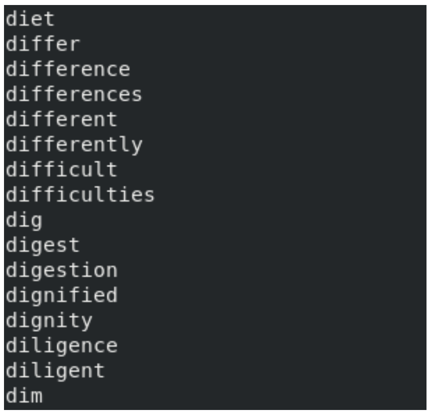
1. Function Testing

The main methods used to test the functionality of our program was trial and error and printing debug statements to the console. Trail and error ranged from inputting the program certain queries, such as “dig” and “ab”, which proved troublesome to the binary search algorithm and were used to correct faults, to simply executing the GUI main method to see how it looks and if it functions properly. To ensure our results were correct, a lexicographically sorted list of words from “wiktionary.txt” was created and compared to the results produced from the program. Some function testing examples are provided on the next few pages.

Example 1: Result from “dig” (left) compared to sorted wiktionary.txt



```
^C[Sam@8afbfb6f FinalBackEnd]$ javac *.java
[Sam@8afbfb6f FinalBackEnd]$ java Main
Enter the number of the file you want to search
Options: 1) wiktionary.txt     2) cities.txt
1
Enter the number of the search you want to complete
Options: 1) LEX_SEARCH     2) R_LEX_SEARCH
1
dig
Number of results: 5
dignity
dignified
dig
digest
digestion
```



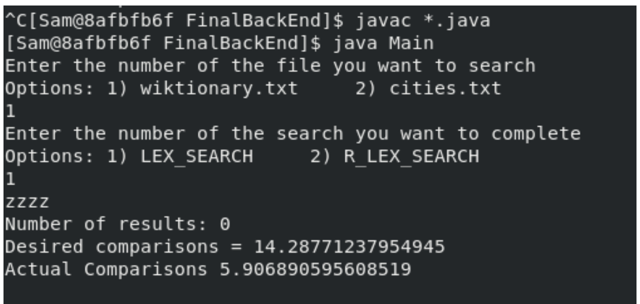
```
diet
differ
difference
differences
different
differently
difficult
difficulties
dig
digest
digestion
dignified
dignity
diligence
diligent
dim
```

It can be seen that all the words in the output match the required results in the sorted list (indicated by the green bar), therefore showing out program is successful for the query “dig”. The results are in a different order since they are sorted by weight in the program output. Although we have not tested and checked the almost limitless possibilities of queries with our program, we have tested about 20 different words from a range of letters throughout the alphabet

2. Performance Testing

Once we knew our program was working correctly in terms of functionality, we could test its performance and compare it to that required in the specification.

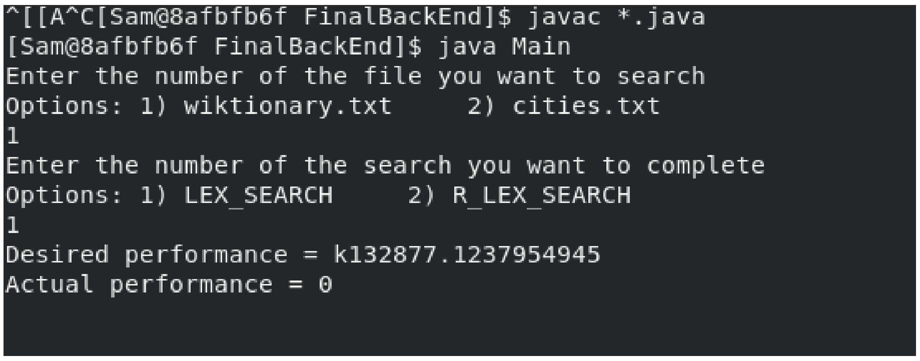
Binary search firstIndexOf perfomance:



```
^C[Sam@8afbfb6f FinalBackEnd]$ javac *.java
[Sam@8afbfb6f FinalBackEnd]$ java Main
Enter the number of the file you want to search
Options: 1) wiktionary.txt     2) cities.txt
1
Enter the number of the search you want to complete
Options: 1) LEX_SEARCH     2) R_LEX_SEARCH
1
zzzz
Number of results: 0
Desired comparisons = 14.28771237954945
Actual Comparisons 5.906890595608519
```

The worst case for a binary search is if the item being searched for isn't in the list being searched – in this scenario, this is achieved by typing in a query of a word which isn't in the term list ("zzzz" works for this). As shown by the output in the image above, the number of comparisons actually made is far below that of what is desired, and therefore our program is successful in this respect.

Autocomplete constructor performance:

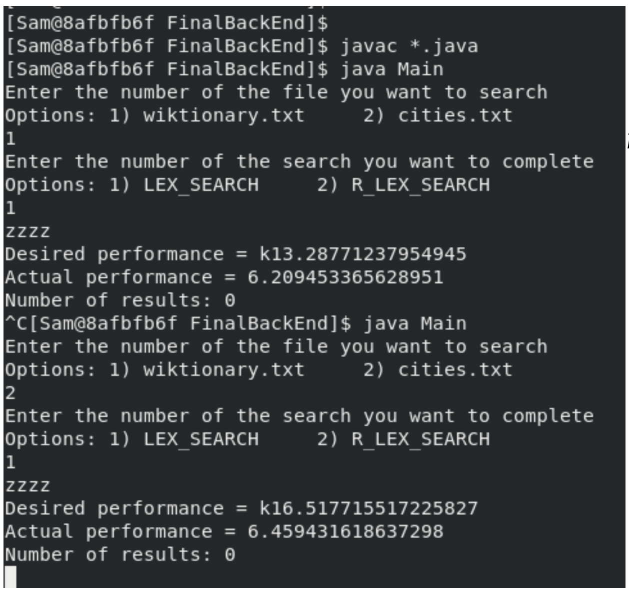


```
^[[A^C[Sam@8afbfb6f FinalBackEnd]$ javac *.java
[Sam@8afbfb6f FinalBackEnd]$ java Main
Enter the number of the file you want to search
Options: 1) wiktionary.txt     2) cities.txt
1
Enter the number of the search you want to complete
Options: 1) LEX_SEARCH     2) R_LEX_SEARCH
1
Desired performance = k132877.1237954945
Actual performance = 0
```

(k shows the proportionality mentioned in the practical specification)

The image above shows the constructor for the Autocomplete class actually uses no comparisons and therefore is very successful in this respect.

Autocomplete numberOfMatches performance:



```
[Sam@8afbfb6f FinalBackEnd]$
[Sam@8afbfb6f FinalBackEnd]$ javac *.java
[Sam@8afbfb6f FinalBackEnd]$ java Main
Enter the number of the file you want to search
Options: 1) wiktionary.txt     2) cities.txt
1
Enter the number of the search you want to complete
Options: 1) LEX_SEARCH     2) R_LEX_SEARCH
1
zzzz
Desired performance = k13.28771237954945
Actual performance = 6.209453365628951
Number of results: 0
^C[Sam@8afbfb6f FinalBackEnd]$ java Main
Enter the number of the file you want to search
Options: 1) wiktionary.txt     2) cities.txt
2
Enter the number of the search you want to complete
Options: 1) LEX_SEARCH     2) R_LEX_SEARCH
1
zzzz
Desired performance = k16.517715517225827
Actual performance = 6.459431618637298
Number of results: 0
```

k shows the proportionality mentioned in the practical specification

Desired performance decrease = k13.2877.../k16.5177… = 0.804 (3sf)

=> 19.6% performance change between files

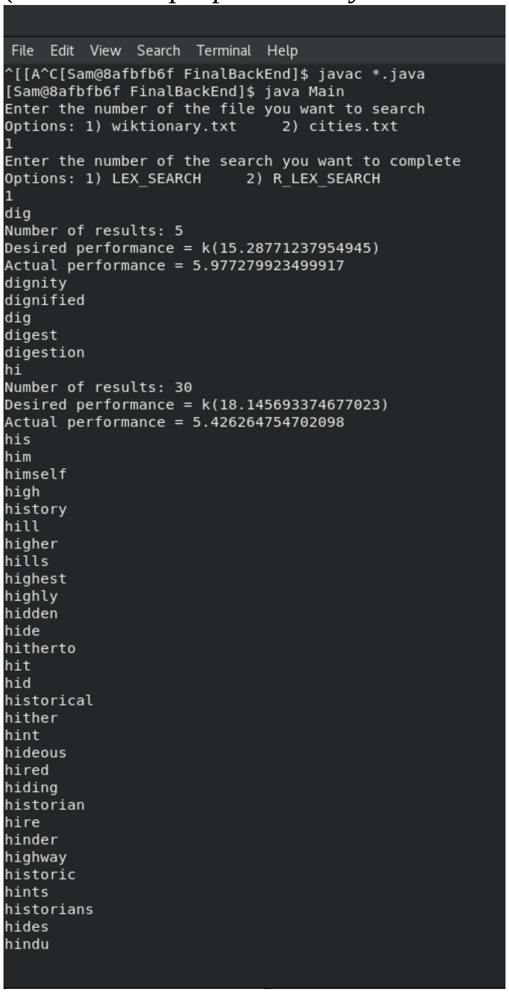
Actual performance decrease = 6.2093.../6.4594… = 0.961 (3sf)

=> 3.9% performance change between files

These calculations show that our program exceeds the required performance specifications, and therefore is very successful in this respect.

Autocomplete allMatches performance:

(k shows the proportionality mentioned in the practical specification)



```
^[[A^C[Sam@8afbfb6f FinalBackEnd]$ javac *.java
[Sam@8afbfb6f FinalBackEnd]$ java Main
Enter the number of the file you want to search
Options: 1) wiktionary.txt     2) cities.txt
1
Enter the number of the search you want to complete
Options: 1) LEX_SEARCH     2) R_LEX_SEARCH
1
dig
Number of results: 5
Desired performance = k(15.28771237954945)
Actual performance = 5.977279923499917
dignity
dignified
dig
digest
digestion
hi
Number of results: 30
Desired performance = k(18.145693374677023)
Actual performance = 5.426264754702098
his
him
himself
high
history
hill
higher
hills
highest
highly
hidden
hide
hitherto
hit
hid
historical
hither
hint
hideous
hired
hiding
historian
hire
hinder
highway
historic
hints
historians
hides
hindu
```

Desired performance decrease = 15.2877.../18.1456… = 0.843 (3sf)

=> 15.7% performance decrease between searches

Actual performance decrease = 5.9772.../5.4262… = 1.101 (3dp, to match others)

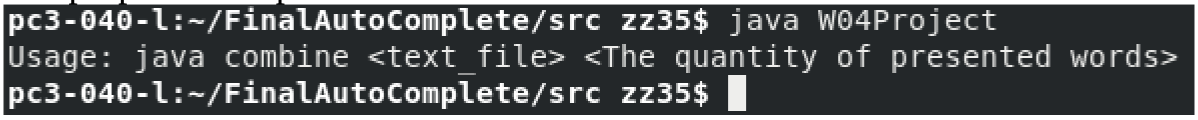
=> 10.1% performance **increase** between searches

As shown by the calculations, the performance of our program is actually increased when the number of results is increased. Although this seems counter-intuitive, the binary search can be very effective if the search entity lies in a certain position – this just happened to be the case for the query “hi”. We can still extrapolate that our program matches the project specification in relation to this performance, and is therefore successful at completing the task at hand.

3. Final Testing

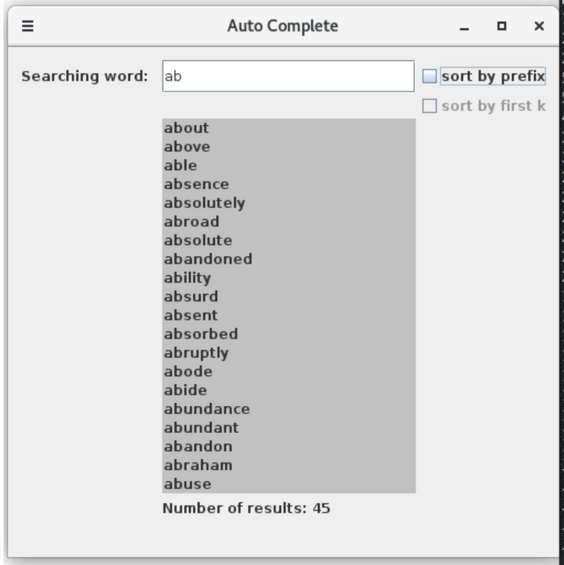
1.Sorting by prefix order testing (there are 5 types testing in this part)

1.a improper user input notice

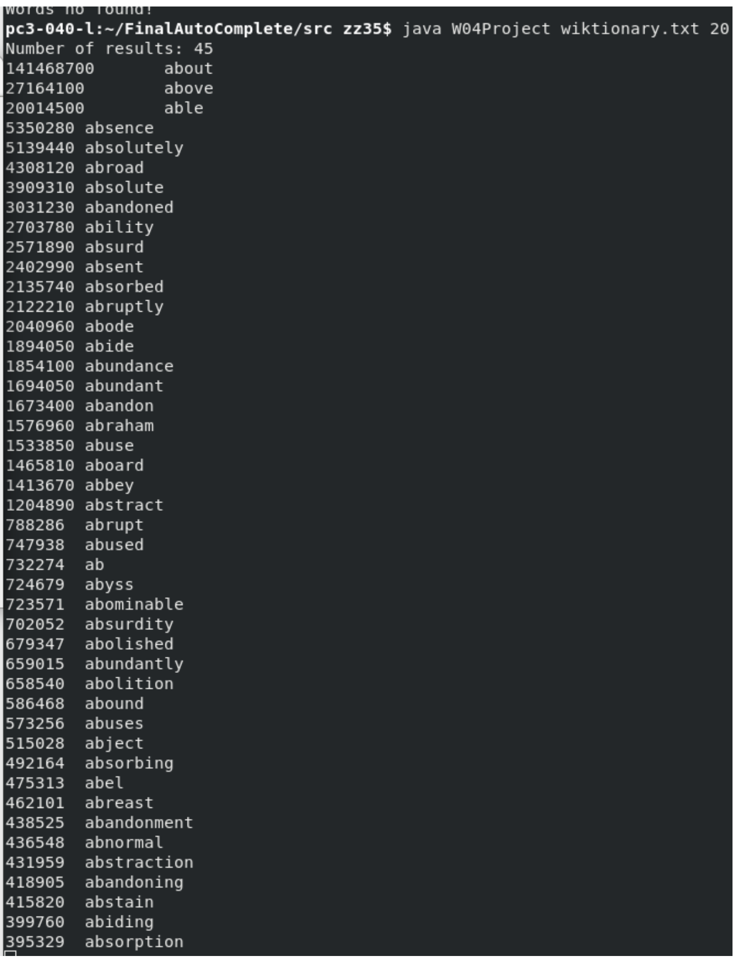


```
pc3-040-l:~/FinalAutoComplete/src zz35$ java W04Project
Usage: java combine <text_file> <The quantity of presented words>
pc3-040-l:~/FinalAutoComplete/src zz35$
```

1.b Searching “ab” as prefix order, and choose to present first 20 results.



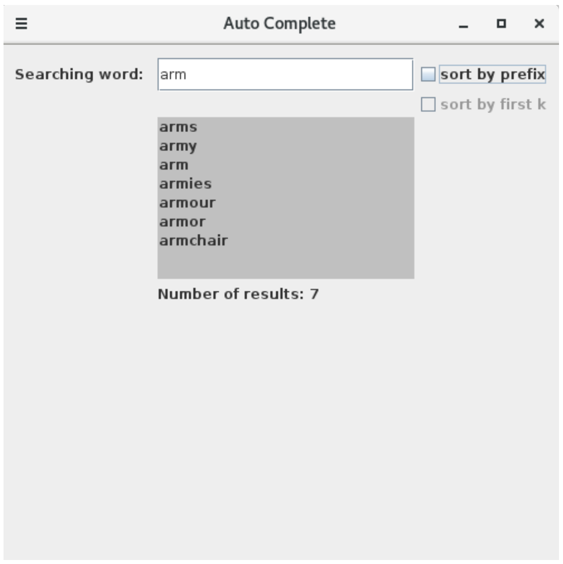
```
Searching word: ab

about
above
able
absence
absolutely
abroad
absolute
abandoned
ability
absurd
absent
absorbed
abruptly
abode
abide
abundance
abundant
abandon
abraham
abuse

Number of results: 45
```



```
words no found!
pc3-040-l:~/FinalAutoComplete/src zz35$ java W04Project wiktionary.txt 20
Number of results: 45
141468700       about
27164100        above
20014500        able
5350280 absence
5139440 absolutely
4308120 abroad
3909310 absolute
3031230 abandoned
2703780 ability
2571890 absurd
2402990 absent
2135740 absorbed
2122210 abruptly
2040960 abode
1894050 abide
1854100 abundance
1694050 abundant
1673400 abandon
1576960 abraham
1533850 abuse
1465810 aboard
1413670 abbey
1204890 abstract
788286  abrupt
747938  abused
732274  ab
724679  abyss
723571  abominable
702052  absurdity
679347  abolished
659015  abundantly
658540  abolition
586468  abound
573256  abuses
515028  abject
492164  absorbing
475313  abel
462101  abreast
438525  abandonment
436548  abnormal
431959  abstraction
418905  abandoning
415820  abstain
399760  abiding
395329  absorption
```

(Explanation: the terminal return all of the words with their weight, and the window show the amount of words that user require even though the quantity of results are larger than users expected.)

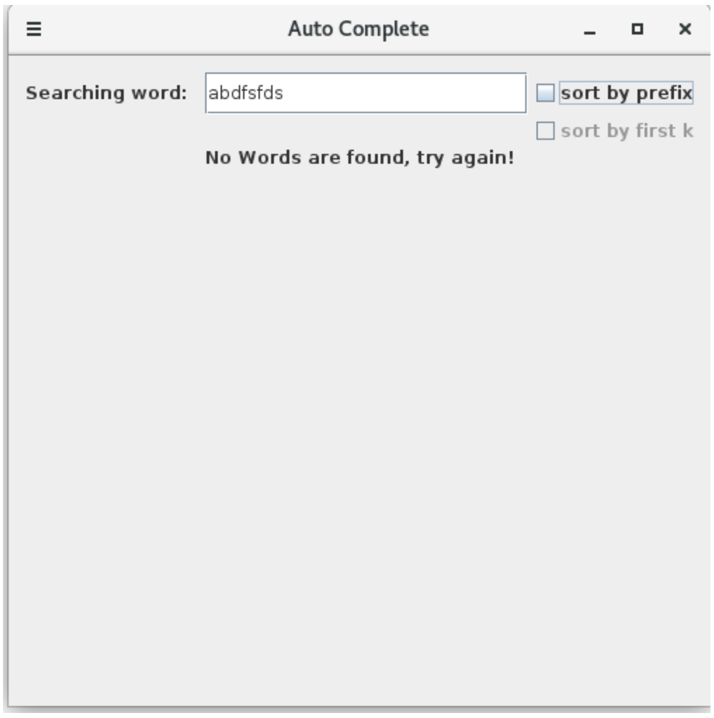
1.c The following shows users require the quantity of words is larger than the result.



```
arm
```

```
arms
army
arm
armies
armour
armor
armchair
Number of results: 7
```

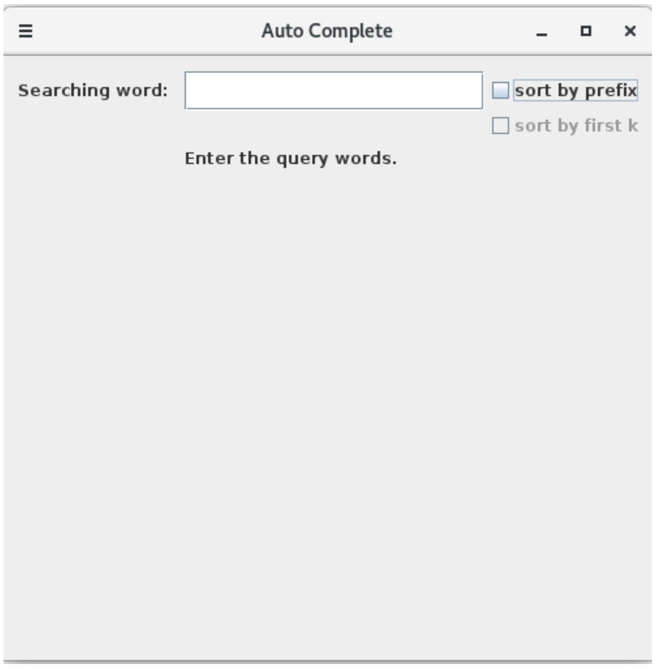
1.d Showing there is no result



```
abdfsfds
```

```
No Words are found, try again!
```

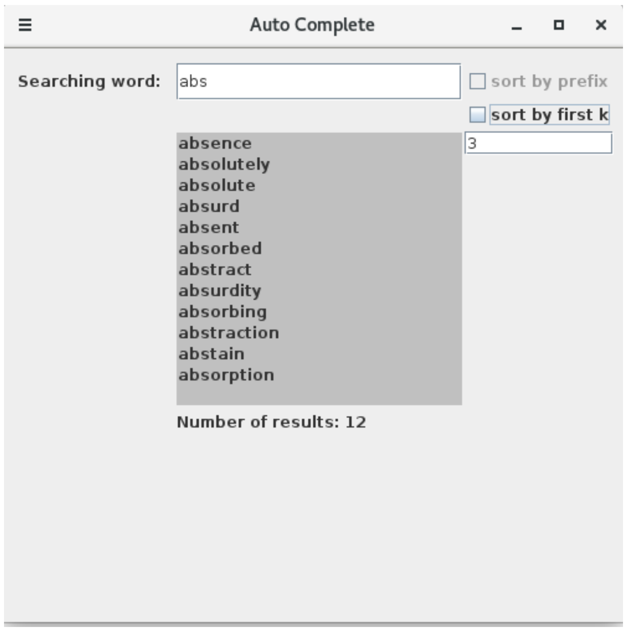
1.e showing there is no input query string



Enter the query words.

2. Sorting by first k characters

2.a function testing

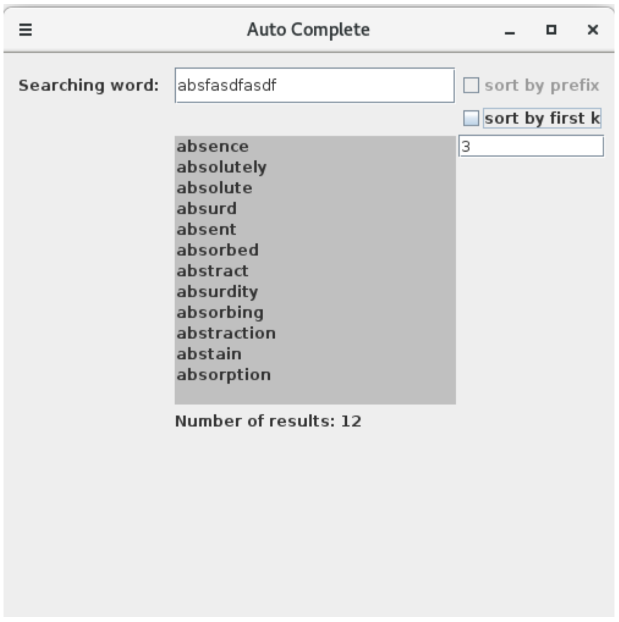


abs

3

```
absence
absolutely
absolute
absurd
absent
absorbed
abstract
absurdity
absorbing
abstraction
abstain
absorption
```

Number of results: 12



absfasdfasdf

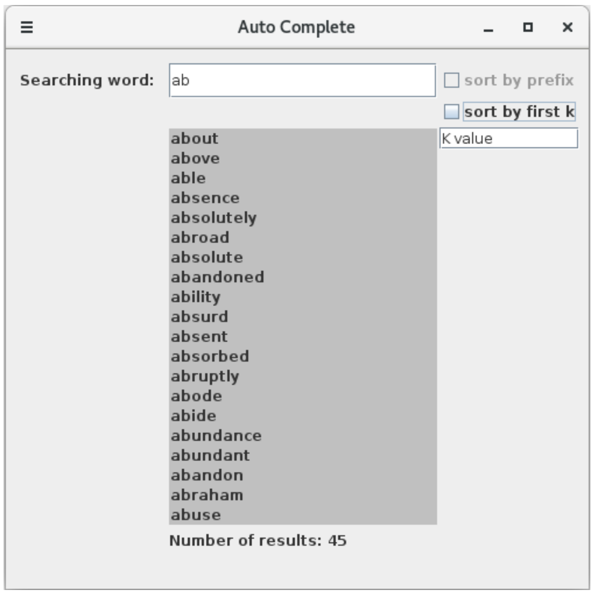
3

```
absence
absolutely
absolute
absurd
absent
absorbed
abstract
absurdity
absorbing
abstraction
abstain
absorption
```

Number of results: 12

(In this part “abs” and “absfasdfasdf” are entered separately, and both of them are sort by the same first k, which is 3. Therefore, the output is the same.)

2.b user enter without k value, and the program choose k equal 2 as default.



Searching word: ab

☐ sort by prefix

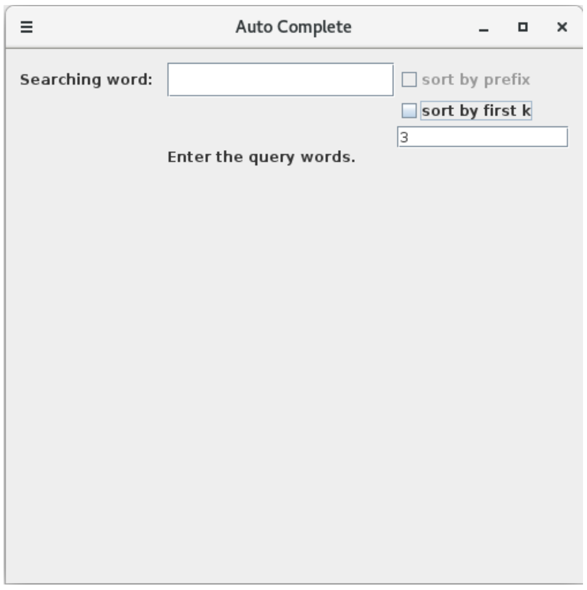
☐ sort by first k

K value

```
about
above
able
absence
absolutely
abroad
absolute
abandoned
ability
absurd
absent
absorbed
abruptly
abode
abide
abundance
abundant
abandon
abraham
abuse
```

Number of results: 45

2.c No query word is entered.



Searching word:

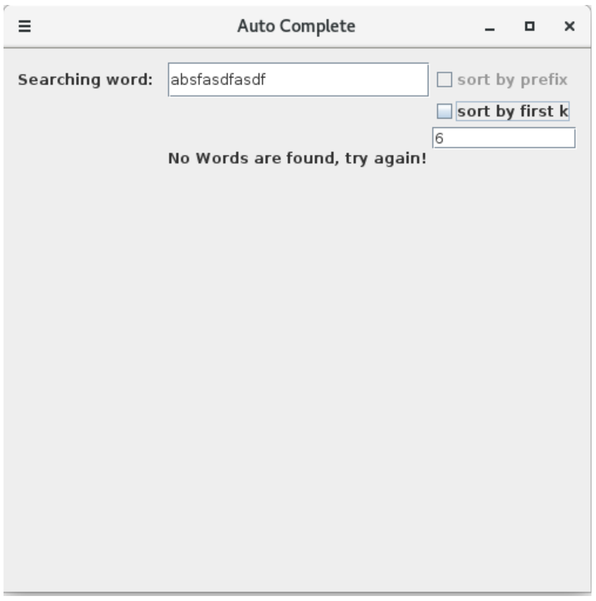
☐ sort by prefix

☐ sort by first k

3

Enter the query words.

2.d No words is found.



absfasdfasdf

6

No Words are found, try again!

Conclusion

This program has successfully achieved both sorting by prefix order and sorting by first k characters with given a text file with query words and their weight. This program catch also catch the user enter problem, such as forget to enter query string so on.