Final Assignment – Word Find Game

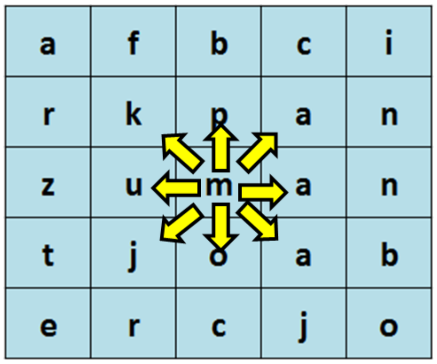
1 Introduction

On this assignment/lab you will create a word find game solver. The game is placed in a text file and your program shall read it and then automatically find the hidden words based on a set of pre-defined words from a given dictionary. In this game, the dictionary is also a text file containing thousands of English words. Dictionaries like this one can be easily found in the internet. After finding all the words, your program shall print how many words were found and list them, one-by-one in the terminal window.

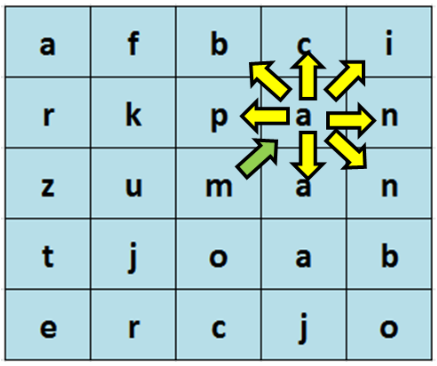
a	f	b	c	i
r	k	p	a	n
z	u	m	a	n
t	j	o	a	b
e	r	c	j	o

2 General Rules about this Game

How do we play this game? You pick a letter, such as “m” shown below, and then you can pick any neighbor letter (in this case, it could be p, a, a, a, o, j, u, or k).



Let’s say that you have picked the ‘a’ at north-east direction.



At this point, you have a word defined as “ma”. You should check with the list of words in the dictionary if there is a “ma” word. If not, pick another letter that is neighbor to the ‘a’ letter. One rule is that you cannot use a cell that you have used already, i.e., at the letter ‘a’ location, you cannot pick the letter ‘m’ again from where you came from. However there is no restriction to get another letter m if there is one neighbor to ‘a’ besides the other one. Examples should be ‘n’, ‘p’, and ‘c’, what would generate the words: man, map, and mac. All those three words are in the dictionary list, so they should be recorded as “found words”.

Even after finding a word, such as man, your program shall keep going on that path by choosing a neighbor letter since this word could be a subset of another word, such as “mana” (don’t ask me what mana means, but it is in the dictionary…).

3 Very important hint

How do you look for a word in the dictionary, such as banana? You open the dictionary in the first page and check that the first words start with letter “a”, then you quickly flip pages until you get words starting with “b”, then you start looking at words that start with “ba” and eventually words with ban and finally you find the banana. Easy hum?

But let’s step back and review this process. Imagine that your game has just constructed the word “banan” (missing the final ‘a’) and look at the dictionary for this word. It will not find it because banan is not a word. In the real word, you would be flipping pages until you find a word just before “banana”, which is “banal”. So your word “banan” is AFTER banal, but it is before the next dictionary word “banana”. In this specific example, your temporary word is a subset of an actual word (i.e., the actual word banana contains the substring banan). If this happens you must continue the process of picking another letter and checking if the new temporary word is in the dictionary (who knows you get lucky and get an “a” as next letter to form “banana”!).

However, if you get a letter, let’s say k, which will make your temp word be banank, when you go from the banal word (that is just before banana in the dictionary) to the banana, that is just after your temp word banank, and banank is NOT a subset of the banana (the next word), then your program should not getting more letters to this temp word, since we know that there will be none in the dictionary. Example: getting a new letter such as j would give banankj, that is also not part of the dictionary. Therefore there is no meaning to keep adding more letters to banank temp word.

Why am I discussing this? Because if you don’t stop adding letters to non-sense temp words your program will take hours to scan all possible combinations of temp words.

We will discuss a method implementation to check for those situations later on.

4 Developing the Game Solver Program

4.1 Reading the Dictionary Text File

To read a text file, use the Scanner class as we have used multiple times in this course.

However, instead of using a regular String Array to store all the words from the dictionary, we will use a list, which is very similar to an array. The disadvantage of using an array is that you need to know the amount of words that you will store in that array so that JVM will allocate space in the computer’s memory before start storing them. Since we do not know how many words are in the dictionary, we can use the List class, which does not require a prior knowledge of the number of words to be stored. It will increase the memory allocation as needed, during run-time.

To create a List of Strings, you issue this command:

```
   List<String> dictionaryWordList = new ArrayList();
```

The above command creates an empty List of strings object. When reading the words from the text file, then you add word-by-word using the following command:

```
dictionaryWordList.add(readWord)
```

where `readWord` is the word that you just read from the dictionary using the Scanner object (inside a WHILE loop, right?)

Create a method that reads the dictionary and fills the List<String> object. This list of strings can be a global object in your program. The method signature and the global list of string object are given below:

```
private List<String> dictionary;
private void readDictionary(String dictionaryFilePath)
```

4.2 Reading the Letters File

The letters file contains a set of letters, organized in rows and columns as an actual game. The letters are separated by commas. An example is shown below (use this example to create the letters file and save it with whatever name you want and use the extension .txt).

```
5
a, f, b, c, i
r, k, p, a, n
z, u, m, a, n
t, j, o, a, b
e, r, c, j, o
```

To make you practice both types of ways of storing a data collection in memory, the letters above must be stored in a 2D array of chars (instead of Lists objects as done for the dictionary). Since we need to know the size of the 2D array before storing the letters, I provide the number 5 as first value of the text file.

Create a method that reads the puzzle game from your text file and fills in the 2D array (a global variable).

```
private char[][] puzzle;
private void readPuzzle(String puzzleFilePath)
```

4.3 Searching temp word in the dictionary List of strings

We need to create a method that will loop over the dictionary words and compare with the temporary word that your program has created. Use the String method ".compareTo" to analyze the three possible situations that your method should take care of (as discussed in Section 3):

a) It found the temp word in the dictionary (in this case, the compareTo will be 0), break the FOR loop and return a number that represents/means that the word has been found.
b) If the compareTo is less than 0, it means that you must keep looking in the dictionary (keep the FOR loop going on)
c) If the compareTo is greater than 0, then:
   - check if the dictionary word contains the temp word (such as "banana" contains "bana"). If so return a number that represents/means that your program should get another letter and keep going on the search.
   - If the dictionary word does not contain the temp work, it means that no further search on this temp word is necessary, such as banak. Return a number that represents this situation to the main caller.

The method and the variables that should be used to represent the search results are given below:

```
private final int FOUND_WORD = 0;
private final int KEEP_ADDING_LETTERS = 1;
private final int DISREGARD_WORD = -1;
private int getSearchCode(String word)
```

Hint: before implementing the rest of this program, test this method first, by calling it from your main method like this:

```
getSearchCode("bana");     // the method should return 1 (KEEP_ADDING_LETTERS)
getSearchCode("banana");   // the method should return 0 (FOUND_WORD)
getSearchCode("banakw");   // the method should return -1 (DISREGARD_WORD)
```

4.4 The Recursive Method

This problem is a typical problem that can be solved by applying the recursion theory: move to a location, add a neighbor to the word, check the dictionary, and repeat it over and over by asking the method to call itself as many times as necessary.

The method signature is given below:

```
private void findWordInPuzzle(int row, int col, String temp)
```

where the row and col are the row and column location of the letter to be added in the temporary word, defined as temp above.

Similarly to what we have discussed in class when we talked about the maze solver program, your method shall:

a) First of all check if the row and col are within the boundaries of the game (values of -1 or 6 are not permitted). If they are out-of-bounds, then return immediately.
b) Check if this cell (row, col) has been already used in the temp word. Do to so, I would advise you to temporarily replace the current cell letter with '*', meaning that it is being used by temp word, and when checking if it has been used, you would check against this '*' value. If this cell is being used already, return immediately.
c) If the two checks above did not return, then add the letter to temp word.
d) Call the getSearchCode with the temp word as input and analyze its return value.
   - If FOUND_WORD, add the temp word into a List<String> of found words (to be printed at the end of the game), and continue adding more letters to this temp word, since "bananas" is "banana" plus an "s" and should be found too, if possible.
   - If KEEP_ADDING_LETTERS, then as it says, keep adding more letters (situation: banan)
   - If DISREGARD_WORD, then return to the main caller by issuing "return;"
e) Save the current letter at row, col into a backup variable. We need this backup to revert the next step when done searching this path.
f) Temporarily set the current letter to '*'. This will tell your recursion method to not use this letter again in this current temp word (see item (b) above)

g) At this point, your program would be ready to add any neighbor letter to your temp word. Remember that your program must investigate all possible paths surrounding your current letter as shown in the two pictures shown in Section 2. To do so, you should recursively call your findWordInPuzzle method 8 times, one for each direction discussed in Section 2, and adjusting the inputs row, col accordingly (Example to visit the cell right above the current position, you should use (row-1, col).

h) After visiting all the possible paths, don't forget to revert the cell letter from '*' to the value you have backed up in item (e) above.

4.5 The Main Method

The recursion method that you wrote above will start at a given cell and from that cell it will explore all possible paths. So if you start the recursion method with cell (1,1), all words that starts with K will be checked since K is the first letter in cell (1,1).

To make your program go over all possible words in this puzzle, you will need to call findWordInPuzzle inside a nested FOR loop in your main program, so each time this recursive method will starts its quest with a different letter from the puzzle.

And finally, when all letters and paths have been visited, then your program shall tell how many words it found and list them in the terminal window. Use the following code as basis for yours.

```
public static void main(String[] args) {
    new WordFinderApp();
}

public WordFinderApp() throws InterruptedException {
    dictionary = readDictionary();
    puzzle = readPuzzle();
    for (int row=0; row < puzzle.length; row++) {
        for (int col=0; col< puzzle[0].length; col++) {
            findWordInPuzzle(row, col, "");
        }
    }

    System.out.println("There are " + foundWordList.size() + " words found");
    for (String foundWord : foundWordList) {
        System.out.println(foundWord);
    }
}
```

5 Test your program

My program (that might be not completely right) gave the following results.

```
ar          pa          ana         manic       macro
arf         pac         am          manna       mo
ark         pain        amp         manna       moa
arum        pan         ama         mana        moan
frump       panic       ama         manana      moan
fa          panama      amort       man         moa
far         pan         amour       manna       moan
ba          panama      amu         mana        mojo
ban         panama      ab          mana        moc
bani        pam         na          map         mor
banana      pa          naan        ma          more
ban         pan         nan         man         mort
baa         panic       nana        mania       mu
bam         panama      nana        maniac      mut
bap         pan         nab         manic       mute
cain        panama      naan        manna       muter
can         panama      nam         manna       mura
canna       pam         nap         mana        murk
canna       puma        na          manana      an
can         puma        nan         man         ani
canna       puma        nana        manna       anna
cam         put         nana        mana        anna
camp        putz        naan        mana        ana
cap         pur         nam         map         an
caput       ai          nap         ma          anna
cab         ain         nab         man         ana
in          an          up          manna       ana
inn         ani         um          manna       ab
rum         anna        ump         mana        abo
rump        anna        ut          manana      aba
rumor       ana         ma          mana        abamp
rut         an          mac         manana      am
ka          anna        main        major       amp
kaf         ana         man         mac         ama
```

```
amain
ama
amort
amour
amu
na
naan
nab
nacre
nam
na
naan
nan
nana
nab
nam
nap
na
nan
nana
naan
nam
nap
nab
tup
tump
tumor
turf
turk
tromp
jump
jut
jute
jura
jo
jojoba
jojoba
jouk
jet
om
oca
or
orc
orca
ore
ort
out
outre
outer
our
an
anna
anna
ana
ana
ab
abo
aba
abamp
acre
am
amp
ama
amain
ama
amort
amour
amu
bo
boa
ba
baa
ban
banana
banana
bam
ba
baa
ban
bani
banian
banana
ban
baa
bam
bap
et
er
rom
romp
roman
roman
roman
roman
roman
roan
roan
roam
roan
roam
roc
roup
rout
route
re
ret
comp
compute
computer
coma
coma
coma
cor
core
coup
couter
can
canna
canna
cab
cabana
cabana
cam
camp
croup
jab
jam
jo
job
jo
jouk
There are 301 words found
BUILD SUCCESSFUL (total time: 4 seconds)
```