White rabbit

Table of Contents

Task

A secret message exist, but it is hidden. Find the secret phrase that will let you to see it. Here is a couple of important hints to help you out:

- An anagram of the phrase is: "poultry outwits ants"
- There are three levels of difficulty to try your skills with
- The MD5 hash of the easiest secret phrase is "e4820b45d2277f3844eac66c903e84be"
- The MD5 hash of the more difficult secret phrase is "23170acc097c24edb98fc5488ab033fe"
- The MD5 hash of the hard secret phrase is "665e5bcb0c20062fe8abaaf4628bb154"

A list of english words is given for you, it should help you out.

Programming language

I have decided to use Java 11 since I'm working with it right now and Java have a good performance comparing to Python or JavaScript when need to deal with a lot of operations and comparings.

Helper utils

In this task I have used some utils classes from other developer project (www.javagl.de) for combination and permutation stuff. Used my own file reading and cli parsing utils from my own project. At last, but not least I used apache common and codecs libraries for comparing md5 hashes, string operations (e.g. replacing, removing chars).

Problem statement

To solve this task we need to understand what anagram is and how we can detect anagrams. Two strings are said to be anagrams of one another if you can turn the first string into the second by rearranging its letters. For example, "table" and "bleat" are anagrams, as are "tear" and "rate". It is also true that "monkeys write" and "new york times" are anagrams. Since we have anagram for phrase "poultry outwits ants" and word list (dictionary) we need to iterate through all words and do combinations of those words with all possible permutations that creates phrase and is an anagram to given phrase. We also need to compare all detected anagrams with given one to one of the md5 hash. Another big headache in this problem is to operate through a big amount of words. The only solutions of optimization for that come to me is:

- Remove all words that do not have letters in given anagram. This will let for us to remove noises (extraneous words) in words list.
- From my practices, most of the algorithms works faster if input data is sorted, so we need to try sort everything :D
- Do not use recursive methods, since it will increase memory usage, can cause methods call frame overflow and honesty I didn’t see any algorithms with good performance that have used recursive approach.
- For word combination and permutation use java iterators since they are lazy and are memory optimized. It means that while elements are not requested they are not filtered and returned to you. Thanks for Marco Hutter for his work for implementation CombinationIterable and PermutationIterable classes for creating combinations and permutations of a certain number of elements of a given set in quite optimal way. This will also let for me to not invent a bicycle.
- Group all words (in a word list) with other words in a list if it is possible to create a phrase that have letters less or equal to length of given phrase. We also do not need to take into account spaces while comparing length of letters. Why it is a good idea to group every word with all other words that can make phrase? It will reduce an amount of data for CombinationIterable object, and we will need less time for creating and iterating thought combination of words that can create possible anagram phrase. This statement is very important since it decrease time of searching phrase drastically.

Solutions for anagram detection

One option is to list off all permutations of the first string and see if any of them are equal to the second string. You can do this recursively or by using a library function that lists permutations. This is considered a "poor" solution because it misses significantly easier approaches and takes at least n! time in the worst case.

Brute force example

```
class Checker {
    private boolean areAnagrams(String first, String second) {
        return areAnagramsRec("", first, second);
    }
    private boolean areAnagramsRec(String soFar, String remaining,
                                   String target) {
        if (remaining.length() == 0) {
            return soFar.equals(target);
        }
        for (int i = 0; i < remaining.length(); i++) {
            String whatsLeft = remaining.substring(0, i) +
                    remaining.substring(i+1);
            if (areAnagramsRec(soFar + remaining.charAt(i),
                    whatsLeft, target)) return true;
        }
        return false;
    }
}
```

Another approach would be to count up how many times each character appears in each string and confirm that each string has the same number of each character as the other. This approach will work, but is slower in practice than the histogram based approach outlined later on. If you see this, watch for an easy edge case: if you don't check that the string lengths are the same, it's easy to accidentally return true because every character present in the first string appears with the same frequency in the second string, but not the other way around. The time complexity of this approach depends on the particular implementation. Usually, they all use O(1) space.

Counting characters example

```
public class Checker {
    public boolean areAnagrams(String left, String right) {
        if (left.length() != right.length()) return false;
        for (int i = 0; i < left.length(); i++) {
            char currCh = left.charAt(i);
            if (numCopiesOf(currCh, left) != numCopiesOf(currCh, right)) {
                return false;
            }
        }
        return true;
    }

    private int numCopiesOf(char ch, String str) {
        int result = 0;
        for (int i = 0; i < str.length(); i++) {
            if (str.charAt(i) == ch) result++;
        }
        return result;
    }
}
```

Two strings are anagrams of one another if they're equal when their letters are sorted. This means that you can test for whether two strings are anagrams of one another by sorting the characters in each string and testing whether the sorted strings are equal. There are lots of different ways to sort an array of strings, some of which end up looking more like the approaches outlined later on. This type of solution is probably a "good" solution. With a standard sorting algorithm like quicksort or heapsort, it runs in time O(n log n). Using counting sort, this will run in time O(n) (though note that the counting sort solution ends up looking a lot more like the histogram approach we'll talk about later on).

Sorting characters example

```
public class Checker {
    public boolean areAnagrams(String left, String right) {
        var leftChars = left.toCharArray();
        var rightChars = right.toCharArray();
        Arrays.sort(leftChars);
        Arrays.sort(rightChars);
        return Arrays.equals(leftChars, rightChars);
    }
}
```

The final approach is to build a frequency histogram of the characters in each string and checking whether those histograms are the same. There are lots of variations on this theme: you can build the histogram as an array or as a hash table, you can build histograms for each string and compare them, or build a histogram for one and then destructively modify it for the second, etc. You can even think of counting sort as belonging to this family. These approaches typically use O(n) time

and O(1) space, making them among the fastest approaches to solving this problem.

Histogramming example

```
public class Checker {
public boolean areAnagrams(String left, String right) {
        if (left.length() != right.length()) return false;

        Map<Character, Integer> frequencies = new HashMap<>();

        for (int i = 0; i < left.length(); i++) {
            if (!frequencies.containsKey(left.charAt(i))) {
                frequencies.put(left.charAt(i), 1);
            } else {
                frequencies.put(left.charAt(i), frequencies.get(left.charAt(i)) + 1);
            }
        }

        for (int i = 0; i < right.length(); i++) {
            if (!frequencies.containsKey(right.charAt(i)) ||
frequencies.get(r.charAt(i)) == 0) return false;
            frequencies.put(right.charAt(i), frequencies.get(right.charAt(i)) - 1);
        }
        return true;
    }
}
```

Decided to use histogram based anagram detection since it is most effective according literature (https://web.stanford.edu/class/cs9/sample_probs/Anagrams.pdf)

Solutions for word combinations and permutations

Word combinations

It is important to have a chance to create all possible combination of words subset in given words set. If we have words set: $S = \{ A,B,C,D,E, \dots \}$, $n = |S| = \infty$, we need to create all possible phrases consisting of 2, 3, 4, 5 words ($k = 2, 3, 4, 5$):

Words combination example for n=3, k=2

```
[A, A]
[A, B]
[A, C]
```

Word permutation

By using word combination we will create possible combination of words in a phrase, but it will not create all possible words positions in that phrase. It means that we need to create a permutations. For a set S with $n=|S|$, there are $m=n!$ different permutations:

Words permutation example for S = { A,B,C }, n = |S| = 3

```
[A, B, C]
[A, C, B]
[B, A, C]
[B, C, A]
[C, A, B]
[C, B, A]
```

Decided to use *CombinationIterable*, *PermutationIterable* classes that were implemented by Marco Hutter. After analysing his code I decided that it is good enough, optimized, not depend on anything else, and I will not implement in better way. The fun fact is, that *CombinationIterable* class founds not only all words combinations for the phrase, but all permutations as well :), but we still use *PermutationIterable* class since it will work on a small subset of words that already meets anagram definition and such combination will decrease search time of phrase (when we have a big amount of data is not clear when we will get second combination of words that is a permutation for words combination that was found before while using *CombinationIterable* only).

Solution for words grouping

Grouping all words (in a word list) with other words in a list if it is possible to create a phrase that have letters less or equal to length of given phrase is quite easy and straightforward. For grouping, it is a good idea to use hash map collection since for key we can use a word and as value we can use array list of all other words that in combination with first one do not have more letters than given anagram phrase. For example, if we have a phrase, `group all`, and set of words *S={me, you, bicycle, tea, breakfast, all, words ....}* we can group them in this way:

Words grouping

```
{
    me: [you, tea, words, ...],
    you: [tea, all, words, ...],
    bicycle: [...],
    tea: [all, words ...],
    breakfast: [...],
    ...
}
```

Example of pseudo code

```
var possibleCombinations = new HashMap<>();

for (var i = 0; i < words.size(); i++) {
    var word = words.get(i);
    var combinations = new ArrayList<>();

    for (var j = i + 1; j < words.size(); j++) {
       var combination = words.get(j);
       var possiblePartOfPhrase = word + combination;

       if (possiblePartOfPhrase.length() <= anagramLength && inAnagrams(anagramPhrase,
possiblePartOfPhrase)) {
           combinations.add(combination);
       }

   }

   if (combinations.size() > 0) {
       possiblePartOfPhrase.put(word, combinations);
   }
}

possibleCombinations.put(word, combinations);
```

It would be nice to find a way to make the same grouping with java stream and lambda functions, but I'm afraid it can make code less readable.

Combination of all solutions

Find a secret phrase is a painless task, since we divided all problem into smaller pieces and have a solutions for anagram detection, combinations, permutations and grouping. Steps will be: * Filter all words that have all letters in given anagram; * Group all words in list with combination of other words that have letters in sum less or equal to given anagarm phrase; * Iterate through group with all words with combination and find all combinations; * Iterate through all combinations and for every combination create permutations; * Iterate through all permutations and check if words from permutation can create the anagram for given phrase; * If permutation words can create the anagram to given phrase, check if md5 hash is equal to the given;

Implementation of all these steps can be found in *Solver* class. It should be easy to follow code and logic.

Example of simplified code from Solver class

```
public class Solver {
    public static void solve(
            String anagramPhrase, String wordlistLocation, String md5hash) {

        System.out.printf(
                "Trying to solve with args `%s`, `%s`, `%s` %n",
                anagramPhrase, wordlistLocation, md5hash);

        var before = System.currentTimeMillis();
        var words = FileUtil.readData(wordlistLocation);

        if (words.size() == 0) {
            throw new RuntimeException("No words in list. Impossible to solve.");
        }

        var anagramWords = StringUtils.remove(anagramPhrase, " ");
        var anagramLength = anagramWords.length();
        var sortedWords = getSortedWords(anagramPhrase, words);
        var possibleWordsCombinations =
groupWordsWithPossibleCombinations(anagramPhrase, anagramLength, sortedWords);
        searchAndPrintPhrase(anagramWords, possibleWordsCombinations, md5hash);
    }
    // Other code part omitted.
}
```

Building project

You need to know how gradle works and also need to have java (e.g. OpenJDK Runtime Environment Corretto-11.0.10.9.1). I have used wrapped gradle so just go to root of project and run:

Building project

```
./gradlew build
```

If everything is ok you should get something like that `BUILD SUCCESSFUL in 2s`

Running project

Runing jar after build

```
# Do not forgot to build first
cd app/build/libs
# For easy case
java -jar app.jar --anagram-phrase "poultry outwits ants" --word-list-location
"../../../wordlist" --md5hash e4820b45d2277f3844eac66c903e84be
# For medium case
java -jar app.jar --anagram-phrase "poultry outwits ants" --word-list-location
"./../../wordlist" --md5hash 23170acc097c24edb98fc5488ab033fe
# For hard case
java -jar app.jar --anagram-phrase "poultry outwits ants" --word-list-location
"./../../wordlist" --md5hash 665e5bcb0c20062fe8abaaf4628bb154
```

Execution results

For easy case we have the phrase: **printout stout yawls**

Output for easy case with /usr/bin/time

```
Trying to solve with args `poultry outwits ants`, `../../../wordlist`,
`e4820b45d2277f3844eac66c903e84be`
Found phrase `printout stout yawls` with `e4820b45d2277f3844eac66c903e84be`
Takes 11.705s to solve
    Command being timed: "java -jar app.jar --anagram-phrase poultry outwits ants
--word-list-location ../../../wordlist --md5hash e4820b45d2277f3844eac66c903e84be"
    User time (seconds): 13.66
    System time (seconds): 0.26
    Percent of CPU this job got: 117%
    Elapsed (wall clock) time (h:mm:ss or m:ss): 0:11.84
    Average shared text size (kbytes): 0
    Average unshared data size (kbytes): 0
    Average stack size (kbytes): 0
    Average total size (kbytes): 0
    Maximum resident set size (kbytes): 1125452
    Average resident set size (kbytes): 0
    Major (requiring I/O) page faults: 0
    Minor (reclaiming a frame) page faults: 281592
    Voluntary context switches: 3386
    Involuntary context switches: 239
    Swaps: 0
    File system inputs: 0
    File system outputs: 176
    Socket messages sent: 0
    Socket messages received: 0
    Signals delivered: 0
    Page size (bytes): 4096
    Exit status: 0
```

For medium case we have the phrase: **ty outlaws printouts**

Output for medium case with /usr/bin/time

```
Trying to solve with args `poultry outwits ants`, `../../../wordlist`,
`23170acc097c24edb98fc5488ab033fe`
Found phrase `ty outlaws printouts` with `23170acc097c24edb98fc5488ab033fe`
Takes 5.927s to solve
    Command being timed: "java -jar app.jar --anagram-phrase poultry outwits ants
--word-list-location ../../../wordlist --md5hash 23170acc097c24edb98fc5488ab033fe"
    User time (seconds): 7.80
    System time (seconds): 0.20
    Percent of CPU this job got: 132%
    Elapsed (wall clock) time (h:mm:ss or m:ss): 0:06.03
    Average shared text size (kbytes): 0
    Average unshared data size (kbytes): 0
    Average stack size (kbytes): 0
    Average total size (kbytes): 0
    Maximum resident set size (kbytes): 828356
    Average resident set size (kbytes): 0
    Major (requiring I/O) page faults: 0
    Minor (reclaiming a frame) page faults: 207161
    Voluntary context switches: 2482
    Involuntary context switches: 192
    Swaps: 0
    File system inputs: 0
    File system outputs: 96
    Socket messages sent: 0
    Socket messages received: 0
    Signals delivered: 0
    Page size (bytes): 4096
    Exit status: 0
```

For hard case we have the phrase: **poultry outwits ants**

Output for hard case with /usr/bin/time

```
Trying to solve with args `poultry outwits ants`, `../../../wordlist`,
`665e5bcb0c20062fe8abaaf4628bb154`
Found phrase `wu lisp not statutory` with `665e5bcb0c20062fe8abaaf4628bb154`
Takes 1312.068s to solve
    Command being timed: "java -jar app.jar --anagram-phrase poultry outwits ants
--word-list-location ../../../wordlist --md5hash 665e5bcb0c20062fe8abaaf4628bb154"
    User time (seconds): 1324.51
    System time (seconds): 1.63
    Percent of CPU this job got: 101%
    Elapsed (wall clock) time (h:mm:ss or m:ss): 21:52.20
    Average shared text size (kbytes): 0
    Average unshared data size (kbytes): 0
    Average stack size (kbytes): 0
    Average total size (kbytes): 0
    Maximum resident set size (kbytes): 902196
    Average resident set size (kbytes): 0
    Major (requiring I/O) page faults: 0
    Minor (reclaiming a frame) page faults: 227620
    Voluntary context switches: 260320
    Involuntary context switches: 16761
    Swaps: 0
    File system inputs: 0
    File system outputs: 16888
    Socket messages sent: 0
    Socket messages received: 0
    Signals delivered: 0
    Page size (bytes): 4096
    Exit status: 0
```