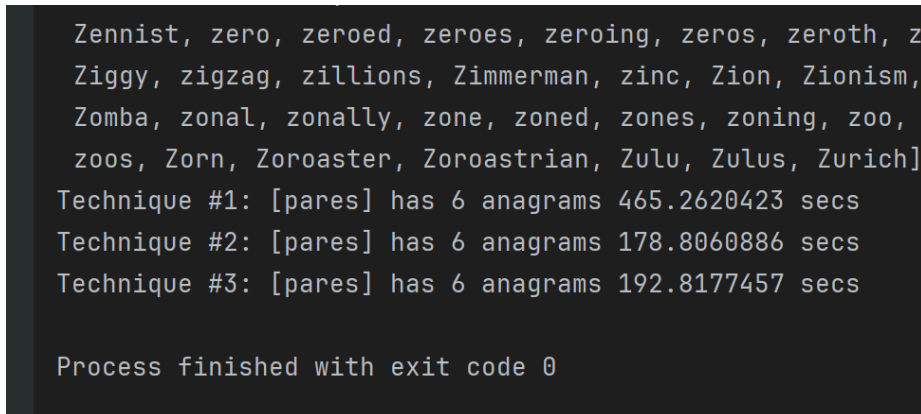


Lab 2

Techniques Screenshot



```
Zennist, zero, zeroed, zeroes, zeroing, zeros, zeroth, z  
Ziggy, zigzag, zillions, Zimmerman, zinc, Zion, Zionism,  
Zomba, zonal, zonally, zone, zoned, zones, zoning, zoo,  
zoos, Zorn, Zoroaster, Zoroastrian, Zulu, Zulus, Zurich]  
Technique #1: [pares] has 6 anagrams 465.2620423 secs  
Technique #2: [pares] has 6 anagrams 178.8060886 secs  
Technique #3: [pares] has 6 anagrams 192.8177457 secs  
  
Process finished with exit code 0
```

Here is a screenshot of the techniques #1, #2, and #3 after running my Java code.

AnagramLetterMatcher

```
for (int i = 0; i < word1.length(); i++)
```

It will run n times

```
if (word2.contains(word1.charAt(i) + ""))
```

This one runs n times as well

Makes the overall efficiency of the program n^2 which is big $O(n^2)$

AnagramWordSorter

<https://learn.microsoft.com/en-us/dotnet/api/java.util.arrays.sort?view=net-android-34.0>

The string sorting method we used, is implemented using merge sort. Which has the time complexity of $O(n \log n)$. Therefore, both the sorting calls have the complexity of $O(n \log n)$.

AnagramHashCounter

```
// O(n)
for (int i = 0; i < word1.length(); i++) {
    buckets[word1.charAt(i)]++;
}
// O(n)
// .length can change
for (int i = 0; i < word2.length(); i++) {
    buckets[word2.charAt(i)]--;
}

// ASCII_SIZE does not change
// O(128) == O(1) we dont care about constants
for (int i = 0; i < ASCII_SIZE; i++) {
    if (buckets[i] != 0) {
        return false;
    }
}
```

Overall, the runtime is $O(2n) = O(n)$ we don't care about the constant

Which one is better?

T1: Letter Anagram Matcher $O(n^2)$	WORST
T2: AnagramWordSorter $O(n \log n)$	
T3: AnagramHashCounter $O(n)$	BETTER

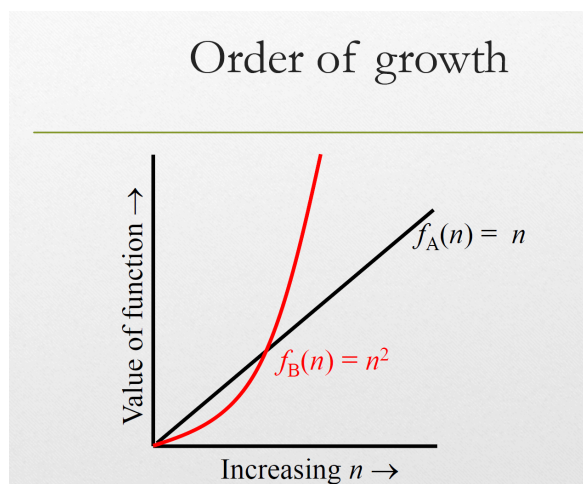
The AnagramHashCounter has the $O(n)$ which is better because it is more efficient. Which is Techniques #3.

Even though hashing using ascii characters makes the algorithm time efficient, it comes with a cost of space efficiency. We had to declare an array to store all the counts (as indexing itself is a cheap operation). Therefore it's only better if we do not care about the space.

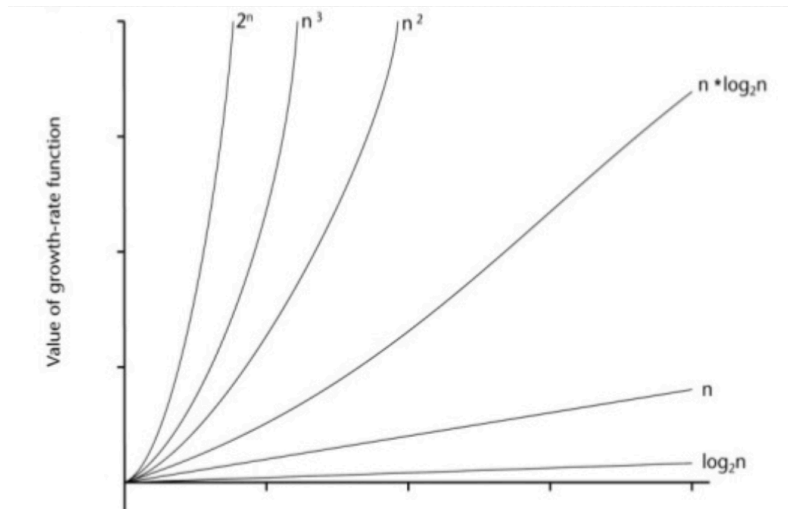
The length of an average English word is 5, $n \log(n)$ with $n=5$ is 8.04718956217, which is way less than 128. Even though 128 is constant, it's way longer than the length of any English words we'll encounter hence it ends up being faster. If our words were longer than 128 characters then we'd see the technique 3's speed.

If we look at time complexity it might look like #2 is better. However, just for our use case where $n = 5$ (on average) technique 2 does better but for longer n , then technique 3 is always gonna do better. Also, time complexity is usually considered the worst case so n is supposed to be extra long in the worst case.

T1 is still the worst. T3 is the best. Even though t3 is the most efficient, the length of strings in the test file are usually short and t3 has a loop that runs 128 times, which (even though it's constant) is way longer than what sorting operation costs on the strings that are provided in the file. So length 36 is the length after which it starts showing the difference $n \log n$ for 36 is 129 which is more than 128.



We are looking at values that are below the crosspoint which doesn't give an accurate representation by just running it with a word that is around 5 characters. But if we got words longer than 128 then it should pass that crosspoint and give us a better representation of which one is the best and worst case.



We can also see based on this chart that we want a line that is closer along the x axis. This concludes that T3 with $O(n)$ is the best case.