Assignment 2

Part 1 A:

Approaches:

Linear:

In this approach, we store dictionary in List object. Then we check if List contains the current word in text. Considering checking from List takes O(n) time, where n is size of dictionary, and we have to repeat operation for each m words, where m is the size of text, time complexity of Linear approach is O(n*m), which makes it almost unusable for large texts.

BBST:

In this approach, I stored the dictionary in String array and checked the words using binary search. Considering dictionary is already sorted file, we do not need to check if it is sorted, which would take O(n*logn) time. Using binary search takes O(logn) time, using it for m words, where m is the length of the text we need to check, takes O(m*logn) time.

Trie:

In this approach I stored the dictionary in Trie object which was implemented by me. Here, checking certain String from dictionary takes O(n) time, where n is the length of the String. Considering length of the String we want to check is usually very short, we may as well say it takes O(1) time.

HashSet:

In this approach, we store dictionary in HashSet object. Then we check if m words, where m is the size of text, is contained within the HashSet. Checking certain object from HashSet takes O (1) time, but if we have too many objects stored inside HashSet, in worst case, it can even take O(n) time, where n is the dictionary size. So average time complexity for this approach is O(m) while worst case being O(n*m).

Output:

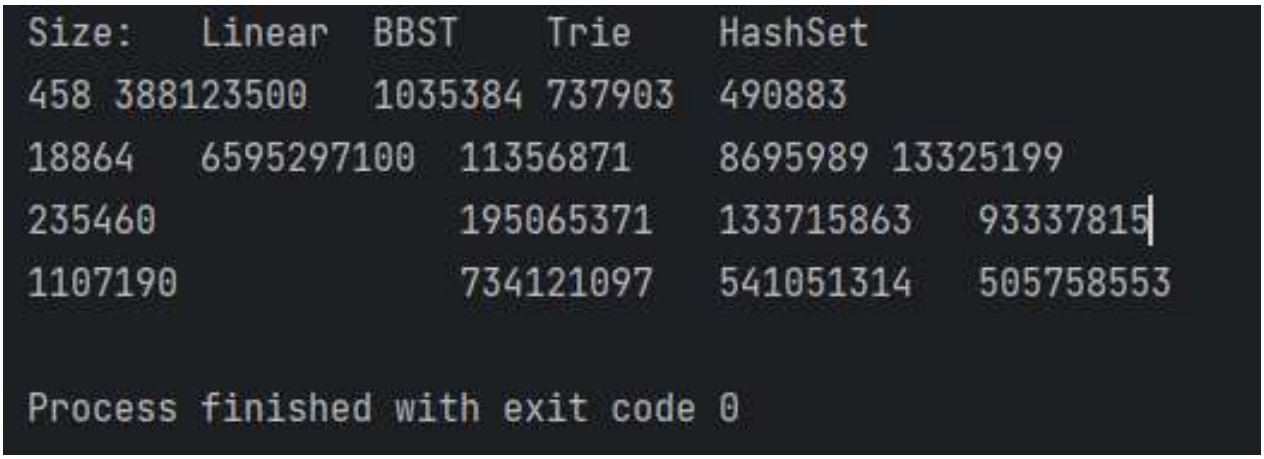
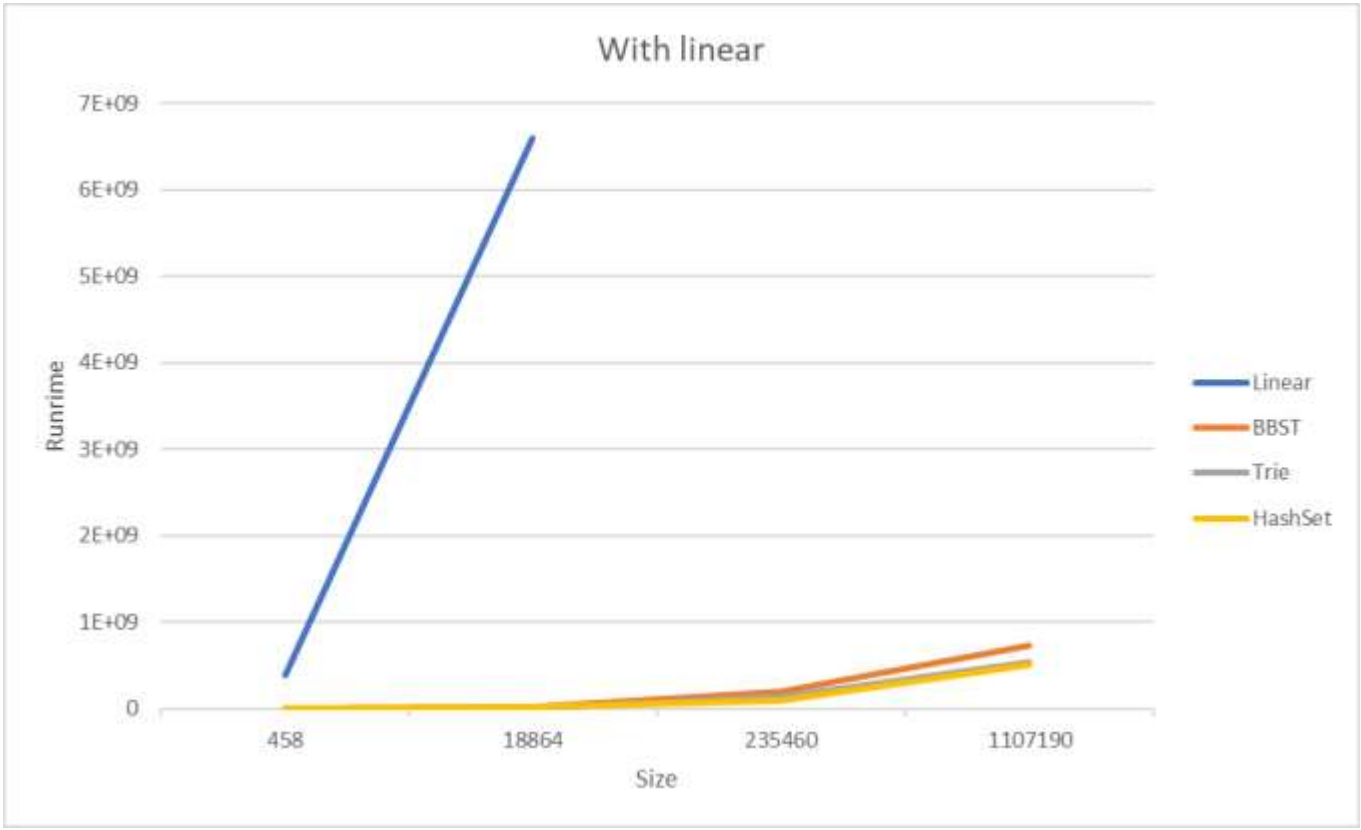
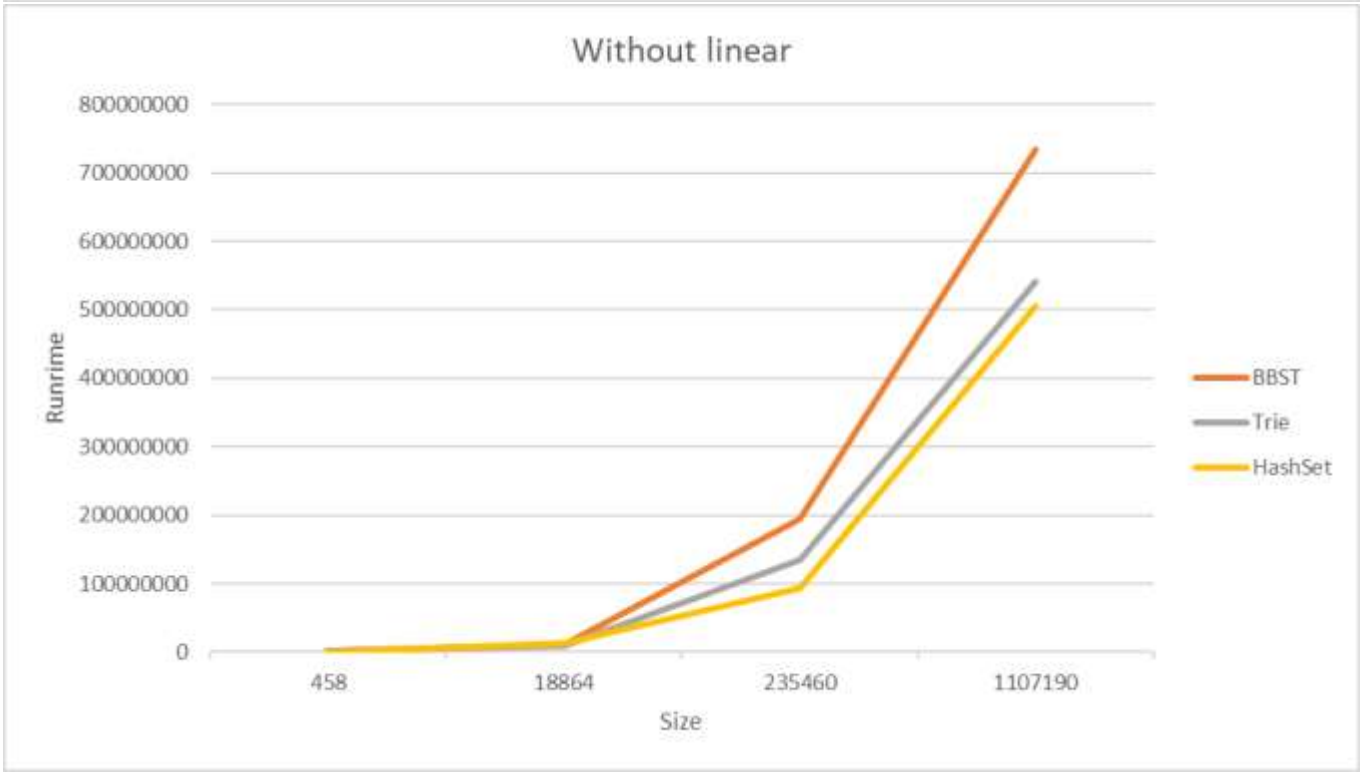


Chart:


With linear
7E+09
6E+09
5E+09
4E+09
3E+09
2E+09
1E+09
0
Runrime
458
18864
235460
1107190
Size
Linear
BBST
Trie
HashSet


Without linear
800000000
700000000
600000000
500000000
400000000
300000000
200000000
100000000
0
Runrime
458
18864
235460
1107190
Size
BBST
Trie
HashSet

In this code, I testes 4 files, each containing different number of words (458, 18864, 235460 and 1107190). Since it's impractical to use Linear approach for large text sizes, I calculated runtime of Linear approach only for small and medium sized texts. For other approaches, I calculated runtime of each of them 100 times and calculated average to get more stable answer.

Findings:

As seen in Chart 1, Linear approach takes too much time to calculate even for smaller size of texts, so this approach is not ever suggested.

As in Chart 1, Trie and HashSet is non-distinguishable from each other, I provided second chart where difference is clearly seen.

At smaller sizes of texts, all three approaches takes almost the same time, but as size grows, it is clear that BBST takes more time than both Trie and HashSet. But considering it should take O(logn) time to search for the word in BBST approach, it is very impressive that, BBST is still usable even for large text sizes.

Trie and HashSet are equally good for large text sizes, Trie being a little bit behind. Trie even showed better performance in middle sized text. Considering calculations also depend on other factors as CPU, RAM, Threads and others, we may conclude HashSet and Trie are equally good for spell checking.

Part 1 B:

Code explanation:

class Position – represents the position in the labyrinth.

class Wizard – represents the individual wizard.

class TriwizardTournament – contains main() method and other methods we need

isValidPosition() – to check if the position is bound within the labyrinth.

findShortestPath() – to find shortest path to exit using breadth-first search

predictWinner() – to determine the winner of the Tournament

main() – to implement all classes and methods

First I created labyrinth from 2D matrix using List object. Then I created wizards List to contain all created wizards and added 3 wizards, giving them initial positions and speeds. Then I called the predictWinner() method and provided labyrinth and wizards as an argument. This method calls findShortestPath() method for each wizard so they can get to exit as fast as possible. At the same time, method calculates the time it took them to reach the end and returns the one with shortest time as a winner.

Input:

```
List<List<Character>> labyrinth = new ArrayList<>();
labyrinth.add(List.of('.', '.', '.', '.', '#', '.', '.'));
labyrinth.add(List.of('.', '#', '#', '.', '#', '#', '.'));
labyrinth.add(List.of('.', '.', '#', '.', '.', '.', '.'));
labyrinth.add(List.of('.', '#', '#', '#', '#', '#', '.'));
labyrinth.add(List.of('.', '.', '.', '.', '.', '#', 'E'));

List<Wizard> wizards = new ArrayList<>();
wizards.add(new Wizard(new Position( r: 0,  c: 1),  spd: 1));
wizards.add(new Wizard(new Position( r: 1,  c: 1),  spd: 1));
wizards.add(new Wizard(new Position( r: 4,  c: 4),  spd: 2));
```

Output:

```
"C:\Program Files\Java\jdk-19\bin\java.exe"
Wizard 1 wins!

Process finished with exit code 0
```

Here, 3rd wizard has 2 speed, but he is the furthest away from the Exit, 1st wizard is the closest one, that's why he reach the end first.

Input:

```
List<List<Character>> labyrinth = new ArrayList<>();
labyrinth.add(List.of('.', '.', '.', '.', '#', '.', '.'));
labyrinth.add(List.of('.', '#', '#', '.', '#', '#', '.'));
labyrinth.add(List.of('.', '.', '#', '.', '.', '.', '.'));
labyrinth.add(List.of('.', '#', '#', '#', '#', '#', '.'));
labyrinth.add(List.of('.', '.', '.', '.', '.', '#', 'E'));

List<Wizard> wizards = new ArrayList<>();
wizards.add(new Wizard(new Position( r: 0,  c: 1),  spd: 1));
wizards.add(new Wizard(new Position( r: 1,  c: 1),  spd: 1));
wizards.add(new Wizard(new Position( r: 4,  c: 4),  spd: 3));
```

Output:

```
"C:\Program Files\Java\jdk-19\bin\java.exe"
Wizard 3 wins!

Process finished with exit code 0
```

In this example, 3rd wizard is the winner, because even though he is furthest away from the exit, he is thrice as fast as others.