

RMIT University Vietnam

TECHNICAL REPORT

COSC2658 Data Structures and Algorithms – Group Project

Group 9

Do Le Long An	S-3963207
Vo Tuong Minh	S-3877562
Nguyen Nguyen Khuong	S-3924577
Tran Ly The Quang	S-3878707

May 2023

TABLE OF CONTENTS

1. INTRODUCTION	2
1.1. Problem Statement	2
1.2. Abstract.....	2
2. OVERVIEW AND HIGH-LEVEL DESIGN.....	3
2.1. Overview.....	3
2.2. High-level Design.....	3
2.2.1. Folder Structure.....	3
2.2.2. Class Structure.....	4
3. DATA STRUCTURES AND ALGORITHMS	6
3.1. Data Structures	6
3.2. Algorithms	6
3.2.1. Initial Guesses (non-optimized)	6
3.2.2. Linear Character Swap – Depth First (non-optimized)	7
3.2.3. Linear Character Swap – Breadth First (non-optimized)	8
3.3. Optimization	9
3.3.1. Frequency Array	9
3.3.2. The Remaining Character	10
3.3.3. Combination of Linear Character Swap algorithms – Depth First vs. Breadth First.....	10
4. COMPLEXITY ANALYSIS.....	12
4.1. Assumptions.....	12
4.2. Initial Guesses.....	12
4.3. General Phase	13
4.3.1. Linear Character Swap – Depth First	13
4.3.2. Linear Character Swap – Breadth First	14
4.4. Overall Complexity	15
5. EVALUATION	16
5.1. Correctness.....	16
5.2. Efficiency	16
5.2.1. Guess Count Efficiency Measurement and Observation.....	17
5.2.2. Correlation Between Character Frequency Metrics and Final Metric of Choice.....	18
5.2.3. Key Length vs. Execution Time vs. Guess Count.....	19
6. CONCLUSION.....	21
REFERENCES	22

1. INTRODUCTION

1.1. PROBLEM STATEMENT

In summary, this is a code-breaking problem that resembles the Mastermind board game invented in 1970 by Mordechai Meirovitz [1].

Initially, there will be a 16-letter secret key containing only the letters 'R', 'M', 'I', and 'T', which the program does not know. To simulate access control, this secret key is implemented as a private string inside of a Java class named `SecretKey` as `SecretKey.correctKey`. The only way to interact with `correctKey` is via the only public method `SecretKey.guess(String guessedKey)` – which accepts a `guessedKey` string and return the number of characters that matched between the `secretKey` and the provided `guessedKey` string. Thus, if `guess()` returns 16, we know that our `guessedKey` is correct.

The class also keeps track of a counter of how many times the program has called `guess()` in `SecretKey.counter` (private integer field). The objective is to guess the correct secret key while keeping `counter` as low as possible.

1.2. ABSTRACT

After iterative research and testing, we came up with a solution inspired by the Linear Search (also known as Sequential Search) algorithm [2], in which each position of the secret key is iterated linearly from left to right to find out to correct guess. This report explains our methodology for developing said solution, shows how our data structures and algorithms design were put into practice, and finally, evaluates their space-time complexity.

2. OVERVIEW AND HIGH-LEVEL DESIGN

2.1. OVERVIEW

Java is chosen as the language of choice to implement and solve this problem, since the provided `SecretKey` is a Java class. The Java toolchain used in this project is Maven 4.0.0, using OpenJDK 17.0.2 (language level 17), and JUnit 5.9.2 for testing and logging (more details below). Additionally, we also utilize IPython and Jupyter to visualize the performance of our solutions (using data logged using JUnit) to aid with further optimization and to verify theoretical space-time complexity calculations with empirical evidence.

2.2. HIGH-LEVEL DESIGN

2.2.1. Folder Structure

```
├── notebooks/
│   ├── images/
│   └── Analysis.ipynb
├── src/
│   ├── main/java/vn/rmit/cosc2658
│   │   ├── development/
│   │   │   ├── InteractiveApp.java
│   │   │   ├── SecretKey.java
│   │   │   └── SecretKeyGuesser.java
│   │   ├── SecretKey.java
│   │   └── SecretKeyGuesser.java
│   └── test/java/vn/rmit/cosc2658/development
│       └── SecretKeyGuesserTest.java
├── test-data/
├── AssessmentDetails.md
├── README.md
├── LICENSE
├── pom.xml
└── requirements.txt
```

This is a typical Maven project structure [3] with some additional tweaks to enable performance test data visualization and analysis using Python Jupyter Notebook:

- `notebooks` folder contains all Python and Jupyter Notebook related files and outputs.
- `requirements.txt` file contains all Python Pip package dependency necessary to setup a Python data-visualization environment.
- `test-data` folder contains all JUnit test performance outputs as .csv files, these are imported into our Jupyter Notebook for performance analysis.
- `AssessmentDetails.md` Markdown file contains the detailed description for our problem.

Finally, in the `src` folder, we have:

2.2.1.1. `main/java.vn.rmit.cosc2658` Java package:

`SecretKey` and `SecretKeyGuesser` are used as submissions for this project: `SecretKey` is kept untouched, while `SecretKeyGuesser` is our final solution coupled with `SecretKey`.

2.2.1.2. `main/java.vn.rmit.cosc2658.development` Java package:

The development package contains classes to aid us in developing our solution:

- `SecretKey` class: a modified version of the provided `SecretKey` class from the parent package. These modifications enable us to choose the secret key or produce a variety of random keys based on our requirements during our solution accuracy testing process.

- **SecretKeyGuesser** class: this class has the above **SecretKey** as a dependency instead of the provided **SecretKey** class from the parent package. This is to aid us in tests to create flexible test cases (variable key length, random keys, etc.) using the modified **SecretKey** class.
- **InteractiveApp** class: an interactive console applet to manually test out individual guesses for a random key of arbitrary length. This helps us to manually test our ideas and eventually come up with the final solution. This class will not be covered in-depth in this report because it only played a minor role in kick-starting our development process. However, it is kept as a way for us to quickly do manual tests.

2.2.1.3. *test/vn.rmit.cosc2658.development* Java package

This is where we store our JUnit tests and driver code for **development.SecretKeyGuesser**, conforming to the Maven project structure [3].

2.2.2. Class Structure

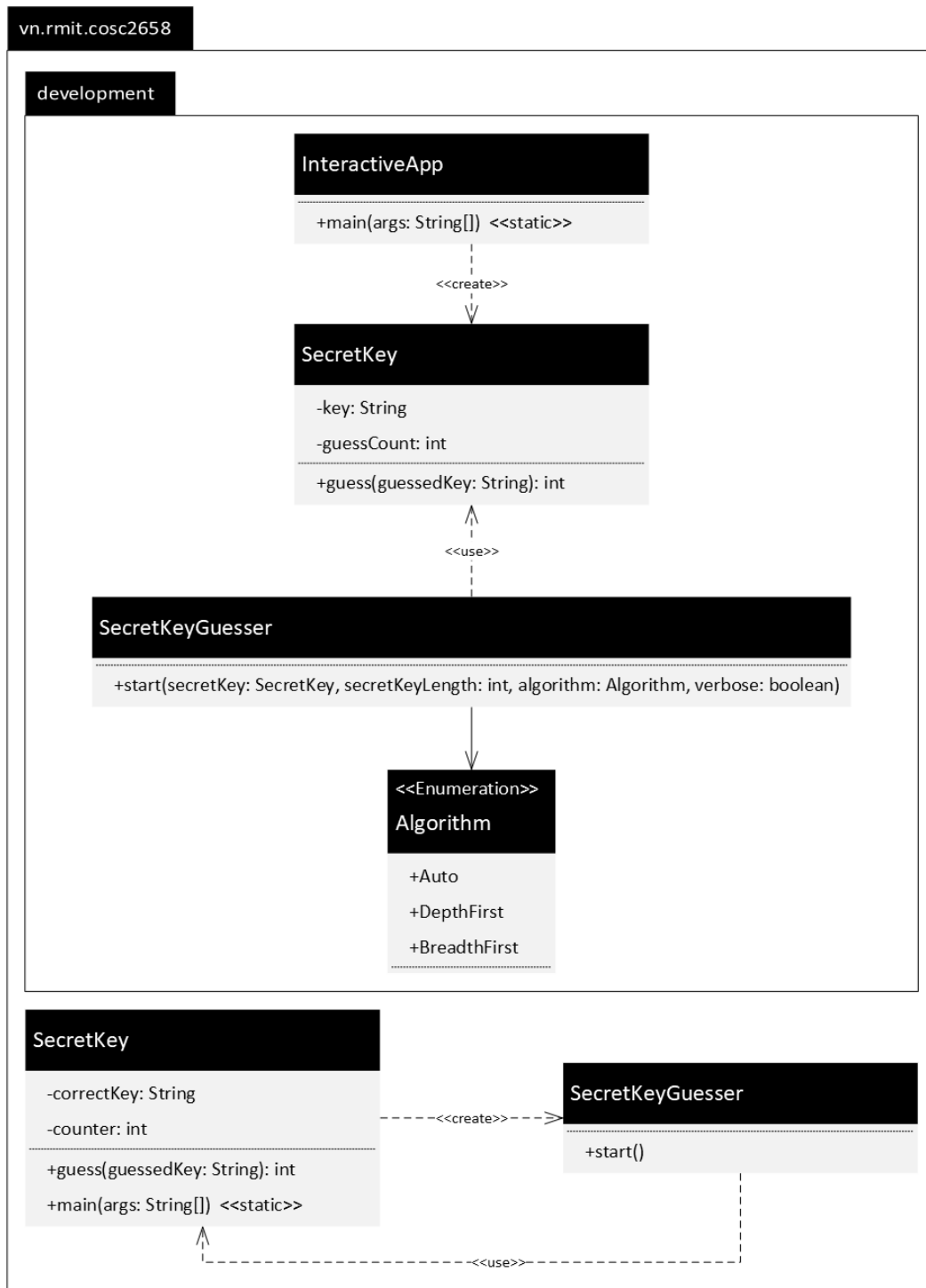


Figure 1: Class Diagram:

We are aware of the circular dependency between `SecretKey` and `SecretKeyGuesser` in the `vn.rmit.cosc2658` package. However, we cannot break this dependency because according to the lecturer, we are not allowed to modify the `SecretKey` class. Thus, we had to keep this circular dependency and work around it.

To avoid modifying the provided `SecretKey` class, all our development and testing are done in isolation within the `development` package. Only after we finalized our solution in `development.SecretKeyGuesser` did we adapt the code in that class into the outer `SecretKeyGuesser` class to work with the unmodified `SecretKey` class.

Within the `development` package, the bulk of our solution development and automated testing focuses on these two classes:

SerecretKeyGuesser: The primary function of the this class is the `start()` method, which calls `linearCharacterSwapDepthFirst()` and `linearCharacterSwapBreadthFirst()` private methods to execute our solutions in the most effective manner (see [3.3.3. Combination of Linear Character Swap algorithms – Depth First vs. Breadth First](#)). Additionally, our algorithms are supported by a wide range of private functions, such as `rankCharByFrequency()`, which sorts a frequency array using the merge sort algorithm, and `hash()`, which converts characters to integers.

SecretKey: This class is derived from the one provided in the parent package with more extension on its functionality to better support automated testing with various types of secret keys to evaluate the efficiency and accuracy of our solutions. These functions are accessed through our modified constructors:

- `SecretKey(String key)`: Creates a `SecretKey` object and assign its key from a specified string. This is useful when we want to write hard-coded test values.
- `SecretKey(int keyLength, int seed)`: Creates a `SecretKey` object and assign to it a pseudo-random key using the provided seed. Calls that provide the same seed will create identical pseudo-random keys. This is useful for when we want to programmatically generate random keys of different length while keeping the reproducibility of our tests.
- `SecretKey(int keyLength)`: Similar to the above constructor, but the seed is automatically generated from system time. This means each call to this constructor creates entirely different unrelated keys. This is useful for when we want to programmatically generate random keys of different length, but test reproducibility is not of concern.

All automated tests are implemented in `development.SecretKeyGuesser`. The tests of focus for our report are:

Table I: Important Tests

Test name	Description
key16TestAlgoAuto	Test the accuracy of our solutions using hard-coded secret keys of length 16. This is to make sure that we will deliver 100% accuracy on subsequent tests for performance (below).
key16TestAlgoDepthFirst	
key16TestAlgoBreadthFirst	
randomKey16TestAuto	Test the accuracy and performance of our solutions using 1,000,000 pseudo-random secret keys. The test results are exported into <code>test-data</code> folder as .csv files.
randomKey16TestDepthFirst	
randomKey16TestBreadthFirst	
randomKeyVariableLengthTestAuto	Test the accuracy and performance of our solutions using pseudo-random secret keys of length ranging from 1 to 512 (inclusive). The test results are exported into <code>test-data</code> folder as .csv files.
randomKeyVariableLengthTestDepthFirst	
randomKeyVariableLengthTestBreadthFirst	

3. DATA STRUCTURES AND ALGORITHMS

3.1. DATA STRUCTURES

Our solution is entirely procedural. Thus, we are only utilizing primitive datatypes such as Boolean, Char, and Integer. We did not construct any abstract data types. The most complex data structure in our solution are simple primitive arrays.

3.2. ALGORITHMS

```
public class SecretKeyGuesser {
    public static final int secretKeyLength = 16;
    public static final char[] CHAR = "RMIT".toCharArray();
    private int guessCount = 0;

    ...
}
```

We begin by declaring three variables:

- `secretKeyLength` (integer): stores the secret key's length. In our problem, it is the constant **n = 16**.
- `CHAR` (array of characters): contains all possible characters in the secret key. In this our problem, they are 'R', 'M', 'I', and 'T'.
- `guessCount` (int): keeps track of the total number of calls to `SecretKey.guess()`. We keep this variable to display to the console verbosely on our guessing progress.

Our solution utilizes a divide-and-conquer approach. Thus, it has two main phases: Initial Guesses and General Phase, of which, it can choose between one of the two algorithms: Linear Character Swap – Depth First and Linear Character Swap – Breadth First, depending on which one would be the most efficient for the given character frequency distribution found in the Initial Guess Phase (see [3.3.3. Combination of Linear Character Swap algorithms – Depth First vs. Breadth First](#)):

3.2.1. Initial Guesses (non-optimized)

3.2.1.1. Overview:

We tested every string containing a single possible character, in this case 'R', 'M', 'I', and then finally 'T'. This step requires four guesses, unless one of the strings is the secret key, in which case the program will terminate early and return the correct guess.

```
for (int charHash = 0; charHash < CHAR.length; charHash++) {
    String guess = Character.toString(CHAR[charHash]).repeat(secretKeyLength);

    int matchCount = secretKey.guess(guess);
    guessCount++;
    System.out.printf("Guess \"%d\" (match: %d)\n", ...)
    if (matchCount == secretKeyLength) {
        System.out.printf("\nI found the secret key after %d guess(es). It is \"%s\"\n", ...);
        return;
    }
}
```

1. In this for loop, for each possible value, we create a consecutive repeated character string with that value.
2. Then, we call the `guess()` method on these strings to get the number of matching positions and store it in the local variable `matchCount`.
3. If `matchCount` equals secret key length (**n = 16**), return that string as the correct guess.
4. Move on to the next stage if there is no secret key found at the end of the loop.

3.2.2. Linear Character Swap – Depth First (non-optimized)

3.2.2.1. Overview:

Step 1: We initialize a baseline guess which is a string with repetitive 'R'. Then, we count the number of matching positions of that guess.

Step 2: We access a specific index of the baseline guess string and change it to another possible character (depth-wise). The number of matching positions of this new guess is also counted by the guess method.

Step 3: We compare the number of matching positions of the new guess from Step 2 to the baseline guess. If it is higher, that character is accepted as the correct solution for that position. Otherwise, the previous character is kept. If the number of matching positions remains unchanged, the algorithm tries the next possible character until it finds the exact one.

Repeat Step 2 and 3 for all **n = 16** positions of the guess string, left to right.

3.2.2.2. Visualized Description:

The figures below will describe this algorithm:

The secret key for this example is "RMITRMIT" with key length **n = 8** (any other arbitrary key length works the same way).

First, we will initialize the base string "RRRRRRRR" and then use the `guess()` method of the `SecretKey` class to calculate the number of matching positions. We also create a current guess string with the same length.

	1	2	3	4	5	6	7	8	
KEY:	R	M	I	T	R	M	I	T	count:8
BASLINE GUESS:	R	R	R	R	R	R	R	R	count:2
CURRENT GUESS:									

Then, we go through every index of the string and change it to other possible values.

	1	2	3	4	5	6	7	8	
KEY:	R	M	I	T	R	M	I	T	count:8
BASLINE GUESS:	R	R	R	R	R	R	R	R	count:2
	M								
	I								
	T								


For example, we change the first letter to M. As a result, the number of matching positions of the new guess is SMALLER than the one of baseline guess, which will be the same with 'I' and 'T'.

	1	2	3	4	5	6	7	8	
KEY:	R	M	I	T	R	M	I	T	count:8
BASLINE GUESS:	M	R	R	R	R	R	R	R	count:1
	I								
	T								

Due to that reason, this letter for this position should be 'R'. Then we move to the next position.

	1	2	3	4	5	6	7	8
CURRENT GUESS:	R							

In this case, when we change the second letter to M, the number of matching positions of this new guess will be HIGHER than the one of baseline guess.



















	1	2	3	4	5	6	7	8	
KEY:	R	M	I	T	R	M	I	T	count:8
BASLINE GUESS:	R		R	R	R	R	R	R	count:3
		M							
		I							
		T							

Hence, the right option for this index is 'M', we do not need to try 'I' and 'T'. 'M' will be added to the current guess.

	1	2	3	4	5	6	7	8	
CURRENT GUESS:	R	M							

Furthermore, if the total number of matching counts remains constant, we continue to try alternative values until the number of matching positions of that string is greater than the previous one.

We repeat these steps for the whole baseline guess.

	1	2	3	4	5	6	7	8	
KEY:	R	M	I	T	R	M	I	T	count:8
BASLINE GUESS:	R	R			R				count:8
						M			
			I				I		
				T				T	
	1	2	3	4	5	6	7	8	
CURRENT GUESS:	R	M	I	T	R	M	I	T	

3.2.3. Linear Character Swap – Breadth First (non-optimized)

3.2.3.1. Overview:

Step 1: We initialize a baseline guess which is a string with repetitive 'R'. Then, we count the number of matching positions of that guess.

Step 2: We access a specific value in the list of possible characters and spread it to every position (breadthwise) of the guess string. The number of matching positions of this new guess is also counted by the guess method.

Step 3: We compare the number of matching positions of the new guess from Step 2 to the previous one. If it is higher, that character is accepted as the correct solution for that position. Otherwise, the previous character is kept. After reaching the end of the string, we try the next possible value.

Repeat Step 2 and 3 until we have exhausted all incorrect positions.

3.2.3.2. Visualized Description:

The secret key for this example is "RMITRMIT" with key length $n = 8$ (any other arbitrary key length works the same way).

First, we will initialize the base string "RRRRRRRR" and count the number of matching positions of that guess by calling the `guess()` method from `SecretKey` class.

	1	2	3	4	5	6	7	8	
KEY:	R	M	I	T	R	M	I	T	count:8
GUESS:	R	R	R	R	R	R	R	R	count: 2

Then we spread the possible value throughout the guess string. For example, we spread the first possible value 'M'.

	1	2	3	4	5	6	7	8		
KEY:	R	M	I	T	R	M	I	T		count:8
GUESS:	R	R	R	R	R	R	R	R		count: 1
	M	M	M	M	M	M	M	M		

When we change the first index of the guess string to 'M', the count would be reduced. Hence, 'R' is the correct value for this position. Then we move to the next index.

	1	2	3	4	5	6	7	8		
KEY:	R	M	I	T	R	M	I	T		count:8
GUESS:	R	R	R	R	R	R	R	R		count: 3
	M	M								

At this position, the number of matching positions is higher, which means that 'M' is the correct letter in this index. The guess string changed to "RMRRRRRR":

	1	2	3	4	5	6	7	8		
KEY:	R	M	I	T	R	M	I	T		count:8
GUESS:	R	R	R	R	R	R	R	R		count: 3
	M	M	M							

In this case, the counter is unchanged, thus 'R' and 'M' are both wrong. We move to the next position and repeat these steps to the end of the guess string.

	1	2	3	4	5	6	7	8		
KEY:	R	M	I	T	R	M	I	T		count:8
GUESS:	R	R	R	R	R	R	R	R		count: 4
	M	M	M	M	M	M	M	M		

After that, we move to the next possible value and spread it to the whole string just the same as the previous one.

	1	2	3	4	5	6	7	8		
KEY:	R	M	I	T	R	M	I	T		count:8
GUESS:	R	M	R	R	R	M	R	R		count: 6
	I	I	I	I	I	I	I	I		

With the possible value 'I'.

	1	2	3	4	5	6	7	8		
KEY:	R	M	I	T	R	M	I	T		count:8
GUESS:	R	M	I	R	R	M	I	R		count: 8
	T	T	T	T	T	T	T	T		

With the possible value 'T'.

3.3. OPTIMIZATION

Following the implementation of all the algorithms, we realize that certain portions of the algorithms can be improved to decrease the guess counter.

3.3.1. Frequency Array

3.3.1.1. Overview:

While attempting all possibilities in Initial Guess, as described previously, we discover that we can store all guess results in an array called frequency array. In other words, this array can store the number of appearances of each possible value.

Moreover, this array is also sorted in descending order.

3.3.1.2. Usage:

By utilizing the frequency array, the number of estimate counters can be reduced.

First, in both algorithms, rather than beginning with a baseline consisting of repeated 'R', we construct the string using the most frequent letter, and the order of the possible values that we use to change at each position is determined by the order of the sorted frequency array. For example, if the frequency array's order is 'M', 'T', 'I', and 'R'. In this instance, M is used as the first guess, followed by 'T', 'I', and 'R' in that order. This modification will reduce the number of guesses because the proportion of that possible value appearing in the secret key will increase, as will the possibility that it is the correct answer for the present position.

Second, each time we effectively alter the specific value from the initial guess, we will decrease the frequency of it. If its frequency reaches zero, we stop and move to the next possible value in the Linear Character Swap – Breadth First algorithm, whereas in the Linear Character Swap – Depth First algorithm, we do not need to check that possible value in all following positions, thereby reducing the number of guesses compared to the original algorithm, which required us to check every possible circumstance on the entire set of values.

3.3.2. The Remaining Character

Specifically, when we try the first two possible values and the condition is not met, we can assume that the character with the lowest frequency matches to the position.

- **For Linear Character Swap – Depth First:** In this algorithm, if the number of matching positions is not higher after trying the first two probable values for each position, we can conclude that the last character, which also has the lowest frequency, is the correct solution. This will result in fewer queries to the `guess()` method.
- **For Linear Character Swap – Breadth First:** In contrast to the Depth First algorithm, this one requires the declaration of a Boolean array that marks the positions where the solution has already been discovered by setting the same index to true. Next, when we iterate through the guess array again, we can bypass positions that have already been modified or marked as true and proceed to the next location. If, after exhausting the first two possible values, some positions are still marked as false, we can infer that the final value is the correct solution for these positions. This saves us additional `guess()` method calls.

3.3.3. Combination of Linear Character Swap algorithms – Depth First vs. Breadth First

While testing the Linear Character Swap – Breadth-First and Linear Character Swap – Depth-First algorithms using one million random 16-character keys, we discovered that their best-case, worst-case, and average case guess counts were different. While Breadth-First has a maximum guess count of 30, a minimum guess count of 8, and an average guess count of 23.06. In contrast, Depth-First has a maximum guess count of 26, a minimum guess count of 17, and an average guess count of 23.03.

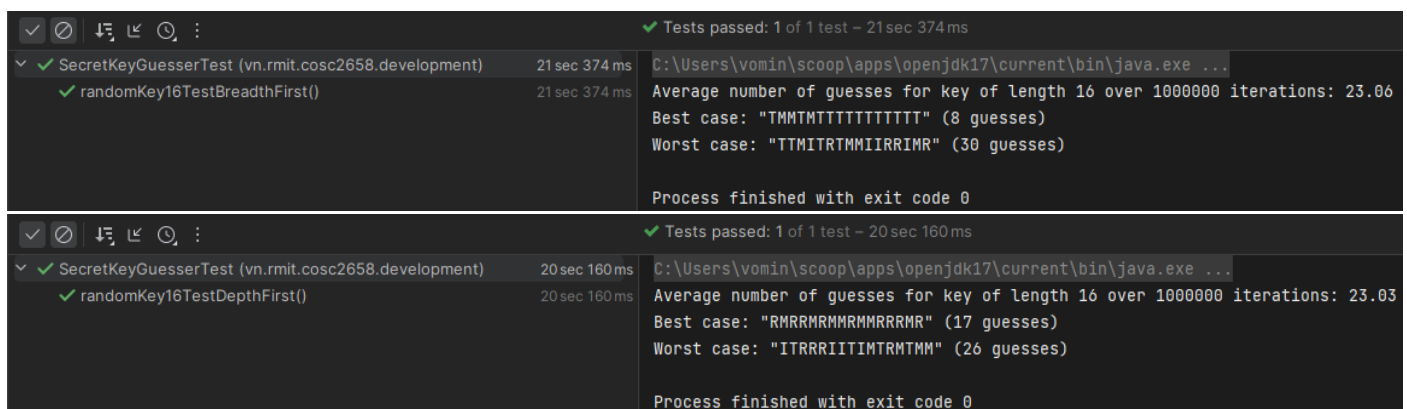


Figure 2. Test Results for Linear Character Swap - Breadth First and Linear Character Swap - Depth First over 1,000,000 pseudo-random 16-character keys

In specific circumstances, Depth First and Breadth First may perform better than the other. Therefore, we decided to find a method to automate algorithm selection to obtain the most efficient solution. Through numerous test cases and investigation using our Python Jupyter Notebook, we determined that the difference in

character frequency is the factor that affects performance between our two algorithms. Consequently, based on descriptive statistics [4], we further investigated character frequency using three metrics:

- **Character Frequency Range:** First, we calculate the range between the character with the highest frequency and the character with the lowest frequency. The number of guesses is then recorded for each case.
- **Character Frequency Median:** The median is the numerical value between the lower and upper halves of a data set. Because the length of our frequency array is an even number, its median is the mean of its two center values. Then, we also do the same with the character frequency range.
- **Character Frequency Variance:** Variance measures the extent to which the data deviate from the standard deviation. The variance formula is the sum of squared deviations from the mean divided by the size of the data set. We also computed the change in the number of guesses based on the variances.

For each of the above three metrics, we illustrated all graphed data using Python and Jupyter Notebook (See [5.2. Efficiency](#)).

This results in the development of a third “Auto” algorithm that works based on our research into the correlation between character frequency distribution and guess count to enhance our solution’s efficiency by choosing the best algorithm for certain character frequency distribution based on a numerical threshold of the Character Frequency Range ([See 5.2.2. Correlation Between Character Frequency Metrics and Final Metric of Choice](#)).

4. COMPLEXITY ANALYSIS

The complexity analysis will take two important cases of our key guessing algorithm into account: Initial Case and General Case, in which we have two important algorithms: Linear Character Swap – Depth First and Linear Character Swap – Breadth First (see [3.2. Algorithms](#)).

4.1. ASSUMPTIONS

- Let **k** be the number of possible characters. In this problem, it is a constant **k = 4** for the possible characters 'R', 'M', 'I', and 'T'.
- Let **n** be the secret key length.
- Let **C** be the constant computational cost.

Having mentioned in the above Data Structure and Algorithms, our analysis will first summarize the key concepts of the algorithm to provide a parallel and thorough complexity analysis on two different aspects: **Time Complexity and Guess Count Complexity**.

Finally, our program will always have Space Complexity of $O(n)$ regarding our Data Structure.

4.2. INITIAL GUESSES

- Performs iteration on the 'R', 'M', 'I', and creates a string of guesses by repeating same character for a secret key length.
- A 1D array named `charFreq[]` is used to store the frequency (distribution) of each character 'R', 'M', 'I', 'T' in the secret key.

→ The program algorithm is responsible for tracking the cumulative frequency distribution of characters in the `charFreq[]` array. The program will end once the guess matches the secret key.

With that having considered, let's visualize the average case, best-case, and worst case for this phase. Similar to the above section, I will provide a secret key of 8 characters:

a) Average case:

$$F_{average} = C \cdot k = O(k) \in O(1)$$

→ As the loop iterates through **k** possible character, the program performs an average complexity of linear for **Time, Space and Guess**.

b) Best case:

	1	2	3	4	5	6	7	8
KEY:	R	R	R	R	R	R	R	R
GUESS:	R	R	R	R	R	R	R	R

Figure 3: Initial Phase – Best case scenario

In the best-case scenario, the secret key consists of entirely character 'R', which matches with the first guess of this phase. As a result:

- Match count: **n = 16**
- The program stops as a secret key is found during the first run.

$$F_{best} = C \cdot 1 \in O(1)$$

→ The operation cost can be estimated to be constant for **Time, Space and Guess** complexity without regarding the number of possible characters in the secret key and the key length.

c) Worst case:

	1	2	3	4	5	6	7	8	
KEY:	R	M	I	T	R	M	I	T	count:8
GUESS:	R	R	R	R	R	R	R	R	count: 2

Figure 4: Initial Phase – Worst case scenario

Assuming the secret key only contains the possible characters stated in the project description, in worst case scenario, the secret key contains multiple characters instead of a repetition of a character. As a result:

- Match count < **n = 16**
- The algorithm continues to guess each possible character in the secret key until the cumulative frequency count reaches the secret key length.

$$F_{\text{worst}} = C \cdot k = O(k) \in O(1)$$

→ The operation cost can be to be linear for **Time**, **Space** and **Guess** complexity regarding the **k = 4** possible characters.

Observation: If our program stops early at this phase, then the **Time**, **Space** and **Guess** complexity for the whole solution will be $O(1)$.

4.3. GENERAL PHASE

As mentioned in [3.2. Algorithms](#), the general phase consists of two key algorithms: Linear Character Swap – Depth First and Linear Character Swap – Breadth First.

4.3.1. Linear Character Swap – Depth First

- The algorithm uses a depth-wise approach to progressively enhance the guessing key.
- It iteratively substitutes all possible characters, **one position at a time**, until it has reached the end of the secret key.
- It is optimized by going from the most common character to the least common character, using the character frequency information obtained in the initial guesses. It also keep track of how many character for each character class is remaining, so that if a character is already exhausted, it no longer guess that character.

```

if newMatchCount < charFreq[hash(mostCommonChar)]:
    // Most common character is correct for this position
    correctKey[charPos] = mostCommonChar
    foundCorrect = true
    charFreqPool[hash(mostCommonChar)]--

if newMatchCount > charFreq[hash(mostCommonChar)]:
    // Next most common character is correct for this position
    correctKey[charPos] = CHAR[nextCommonCharHash]
    foundCorrect = true
    charFreqPool[nextCommonCharHash]--

if not foundCorrect:
    // Least common character is correct for this position
    correctKey[charPos] = leastCommonChar
    charFreqPool[hash(leastCommonChar)]--

```

Figure 5: Linear Character Swap – Depth First key concept Pseudo-code

`hash()` function facilitates the mapping of characters to their frequency count by providing a consistent index value for each character in the `charFreq[]` array.

In addition, we can see the program has a nested *for* loop:

Pseudo-code	Operation Cost
for charPos from 0 to secretKeyLength - 2 do: ...	The outer loop iterates n - 1 times. Hence, it has a time complexity of $O(n)$.
for nextCommonCharIndex from 1 to CHAR.length - 2 do: ...	The inner loop performs iterations equivalent to the constant value of k - 2 , without considering input size. Thus, the inner loop's time complexity can be deemed as $O(1)$.

With that having considered, let's visualize the average case, best case, and worst case for this algorithm. Like the above section, I will provide a secret key of 8 characters:

a) Average case:

$$F_{average}(k, n) = C \cdot (n - 1) \in O(n)$$

→ The program performs with a linear complexity on **Time** and **Guess** and **Space**.

b) Best case:

$$F_{best}(k, n) = C \cdot (n - 1) \in O(n)$$

The secret key permutations always match the character frequency rank (most common character):

The algorithm immediately identifies the correct character for each position and updates the `correctKey[]`.

- The time complexity of the inner loop is $O(1)$ since it iterates a constant number of times.

→ The program's performance is dominated by the linear complexity for **Time** and **Guess** and **Space** complexity of the outer loop.

c) Worst case:

$$F_{worst}(k, n) = C \cdot (k \cdot 2) \cdot (n - 1) \in O(k \cdot n) \in O(n)$$

The secret key permutations never match the character frequency rank (least common character):

- The outer loop iterates over `charPos` 0-based indexes from 0 to **n - 2**. In the worst case, it iterates **n - 1** times.
- The inner loop iterates over `nextCommonCharIndex` from 1 to **k - 2**. In the worst case, it iterates **k - 1** times.

4.3.2. Linear Character Swap – Breadth First

- The algorithm uses a breadthwise approach to progressively enhance the guessing key.
- It iteratively replaces all positions in the first guess with the next most common character and confirms if the replacement changes the match count or not (in that case, we have found the correct solution for that position). In contrast, the Linear Character Swap – Depth First algorithm loops through all possible characters for a position before moving on to the next one.
- It is optimized by going from the most common character to the least common character, using the character frequency information obtained in the initial guesses. It also keep track of how many character for each character class is remaining, so that if a character is already exhausted, it no longer guess that character.
- It terminates when all positions in the secret key except the last one is correctly guessed, or when all but the least common character is left. Then it will just fill in the remaining position still marked incorrect with the remaining least common character.

```

switch (newMatchCount - cumulativeMatchCount) {
    case 1:
        // New replacement character is correct for this position
        correct[i] = true;
        charFreq[nextCommonCharHash]--;
        totalCharFreq--;
        correctCount++;
        cumulativeMatchCount = newMatchCount;
        break;
    case -1:
        // Original baseline most common character guess is correct
        correct[i] = true;
        charFreq[mostCommonCharHash]--;
        totalCharFreq--;
        correctCount++;
        guess[i] = CHAR[mostCommonCharHash];
        break;
}

if (totalCharFreq == charFreq[leastCommonCharHash]) {
    // Only least common character left. No need to guess anymore.
    for (int j = 0; j < secretKeyLength; j++) {
        if (!correct[j]) guess[j] = CHAR[leastCommonCharHash];
    }
}

```

Figure 6: Linear Character Swap – Breadth First key concept Pseudo-code

The variable `cumulativeMatchCount` stores the total number of matches that have been found during the guessing procedure. The program maintains a count of the total correct character positions found in the secret key.

In addition, we can see the program has a nested for – loop:

Pseudo-code	Operation Cost
<pre> for nextCommonCharIndex from 1 to CHAR.length - 1 do: ... </pre>	<p>The outer loop performs iterations equivalent to the constant value of $k - 1$, without considering input size. Thus, the inner loop's time complexity can be deemed as $O(1)$.</p>
<pre> for i from 0 to secretKeyLength - 1 do: if correctCount >= secretKeyLength - 1 or charFreq[nextCommonCharHash] <= 0 or i >= secretKeyLength: break ... </pre>	<p>The inner loop iterates $n - 1$ times. Hence, it has a time complexity of $O(n)$.</p>

We can see the complexity of outer and inner loop is opposite when compared to above Linear Character Swap – Depth First. As a result, the Big O Notation for the three cases of Linear Character Swap – Breadth First is similar to Linear Character Swap – Depth First, which is $O(n)$.

4.4. OVERALL COMPLEXITY

Considering all the above analysis, we can see that our solution has an overall **Time** and **Guess** and **Space** complexity of:

- Best case: $O(1)$
- Worst case: $O(n)$
- Average case: $O(n)$

5. EVALUATION

5.1. CORRECTNESS

Our Java app and algorithm were thoroughly tested to guarantee their accuracy. With the help of code coverage tools, we developed an extensive set of JUnit tests that exercise all major features and functions. We also did integration testing to ensure that all the parts of the program were compatible with one another. The Maven build tool was used to automate testing and guarantee dependability. Finally, we used input data with known output values to assess the algorithm's performance (See [2.2.2. Class Structure – Table I](#)).

```
void randomKey16TestAuto() {
    try {
        FileWriter outFile = new FileWriter( fileName: OUTPUT_DIR + "randomKey16TestAuto.csv");
        outFile.write( str: "Iteration,SecretKey,CharFreqRange,CharFreqMean,CharFreqMedian,CharFreqVariance,CharFreqStdDev,GuessCount\n");

        final int MAX_ITER = 1_000_000;
        final int KEY_LEN = 16;

        String bestCase = "", worstCase = "";
        long bestCount = 4_294_967_296L, worstCount = 0;
        int countSum = 0;
        for (int i = 0; i < MAX_ITER; i++) {
            SecretKey sk = new SecretKey(KEY_LEN, i); // Reproducible results for consistent performance measurement.
            assertEquals(SecretKeyGuesser.start(sk, KEY_LEN, SecretKeyGuesser.Algorithm.Auto, verbose: false), sk.getKey());
        }
    }
}
```

✓ SecretKeyGuesserTest (vnrmitcos 2 min 19 sec) Average number of guesses for key of length 16 over 1000000 iterations: 22.99
✓ randomKey16TestDepthFirst 16 sec 133 ms Best case: "TMMTMTTTTTTTTT" (8 guesses)
✓ getMeanCharacterFrequencyTest() 1 ms Worst case: "ITRRRIITIMTRMTMM" (26 guesses)
✓ getVarianceCharacterFrequencyTest() 1 ms
✓ getStandardDeviationCharacterFrequencyTest() 1 ms
✓ key16TestAlgoBreadthFirst() 2 ms
✓ getMedianCharacterFrequencyTest() 2 ms
✓ randomKeyVariableLengthTest 31 sec 364 ms
✓ randomKeyVariableLengthTest 29 sec 334 ms
✓ getCharacterFrequencyRangeTest() 1 ms
✓ randomKey16TestBreadthFirst 15 sec 823 ms
✓ randomKeyVariableLengthTest 30 sec 141 ms
✓ randomKey16TestAuto() 16 sec 136 ms
✓ key16TestAlgoDepthFirst() 1 ms
✓ key16TestAlgoAuto() 1 ms
✓ rankCharByFrequency() 2 ms

Figure 7: Test Code and Output

[Figure 7](#) above demonstrates how we check if the guessed key is the same as the secret key by using the `assertEquals` function. All tests were completed successfully, and the guessed key always correlated with the secret keys, resulting in a 100% success rate.

Conclusion: Our Java application and algorithm for generating and validating secret keys were rigorously evaluated and found to be dependable and accurate. Using unit tests, integration testing, and input data analysis to evaluate the program's performance, we found a secret key length of 16 over 1,000,000 random iterations and 512 random keys with length of 1 to 512 both returned a 100% success rate. The utilization of code coverage tools and the Maven build toolchain with JUnit guaranteed automated testing dependability.

5.2. EFFICIENCY

It has been observed that there are certain situations where the Linear Character Swap – Depth First algorithm is more efficient, while in other situations, the Linear Character Swap – Breadth First algorithm outperforms it. Through experimentation, it has been discovered that this is closely linked to the frequency distribution of characters in the secret key. Therefore, we have put efforts to illustrate this relationship using these three metrics:

- Character Frequency Range,
- Character Frequency Median,
- Character Frequency Variance.

5.2.1. Guess Count Efficiency Measurement and Observation

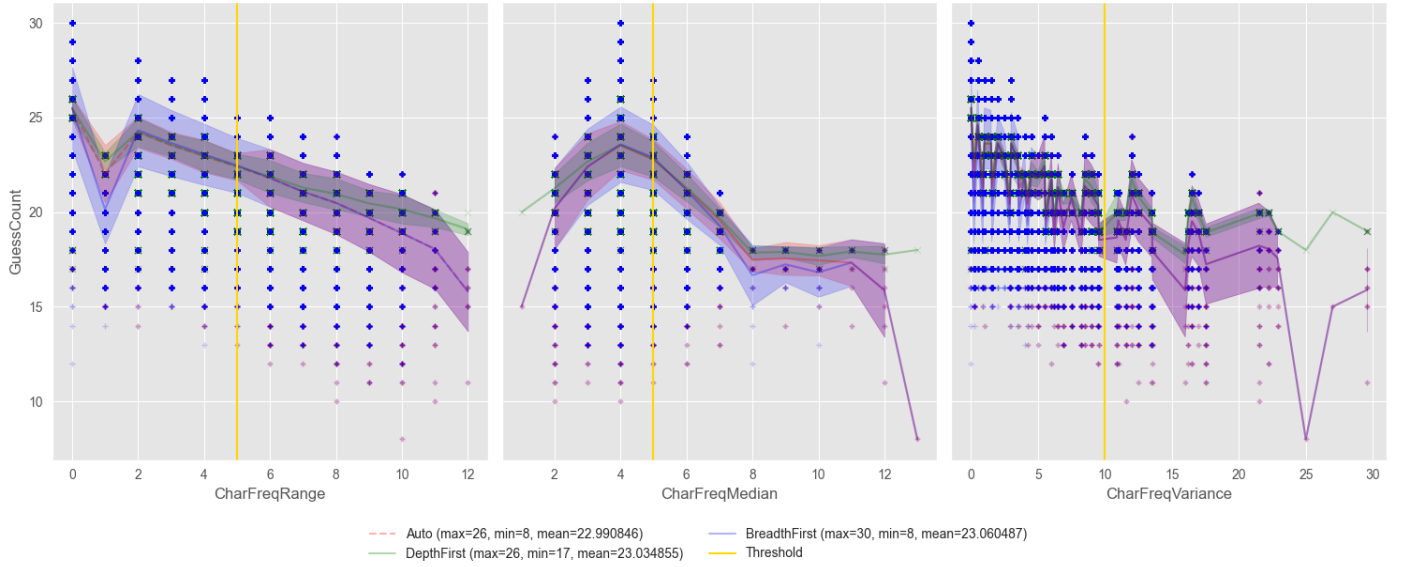


Figure 8: Guess Count by Character Frequency Metrics with Threshold marker (in gold)

Plotted above in [Figure 8](#) is a comparison of Guess Count over the three metrics above (data taken from 1,000,000 random test keys of length $n = 16$).

It is apparent that the Depth First algorithm exhibits superior performance prior to the threshold (indicated by the gold line on the graph), while the Breadth First algorithm demonstrates better performance beyond the threshold. Therefore, we can determine the practical processing phase threshold as follows:

- Depth First Algorithm's best performance range:
 - $CharFreqRange < 5$
 - $CharFreqMedian < 5$
 - $CharFreqVariance < 5$
- Breadth First Algorithm's best performance range:
 - $CharFreqRange \geq 5$
 - $CharFreqMedian \geq 5$
 - $CharFreqVariance \geq 5$

Given the length of the secret key is $n = 16$, we can extrapolate that the best threshold to switch between our algorithms are:

- $T_{range} = n/3.2$
- $T_{median} = n/3.2$
- $T_{variance} = (n/5)^2$

In the next part, we will discuss and choose only one of the above metrics for implementation into our solution.

5.2.2. Correlation Between Character Frequency Metrics and Final Metric of Choice

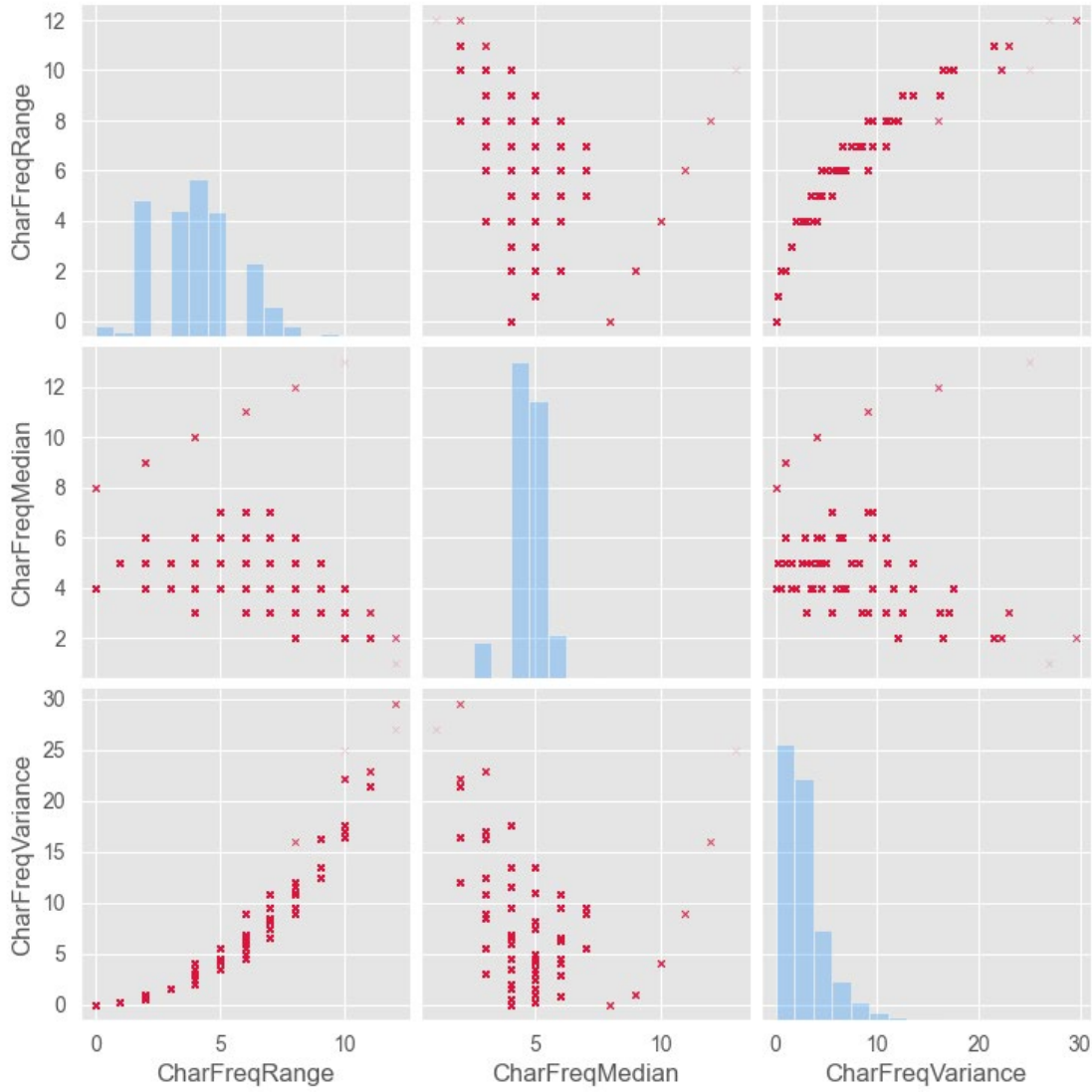


Figure 9: Correlation map between Character Frequency Range, Character Frequency Median, and Character Frequency Variance

From [Figure 9](#) plotted from our test data (1,000,000 random keys of length 16), we can see that these 3 metrics are very closely correlated – which is to be expected considering their definition and formulae [4] – we can just use 1 of the 3 to find the optimal policy for our Auto algorithm for when to use Linear Character Swap – Breadth First and when to use Linear Character Swap – Depth First. To maximize our solution performance, we are using Character Frequency Range because it is the most computationally efficient metric to calculate:

$$T_{range} = f_{max} - f_{min}$$

where f is the character frequency for a character.

Thus, our threshold is $n/3.2 = 5$. Implemented in our class `SecretKeyGuesser`, it is:

```
final char[] charCommonalityRank = rankCharByFrequency(charFreq);
double algoThreshold = secretKeyLength / 3.2;
int characterFrequencyRange = getCharacterFrequencyRange(charFreq);
if (characterFrequencyRange <= algoThreshold) {
    ...
    linearCharacterSwapDepthFirst(secretKey, charFreq, charCommonalityRank);
} else {
    ...
    linearCharacterSwapBreadthFirst(secretKey, charFreq, charCommonalityRank);
}
```

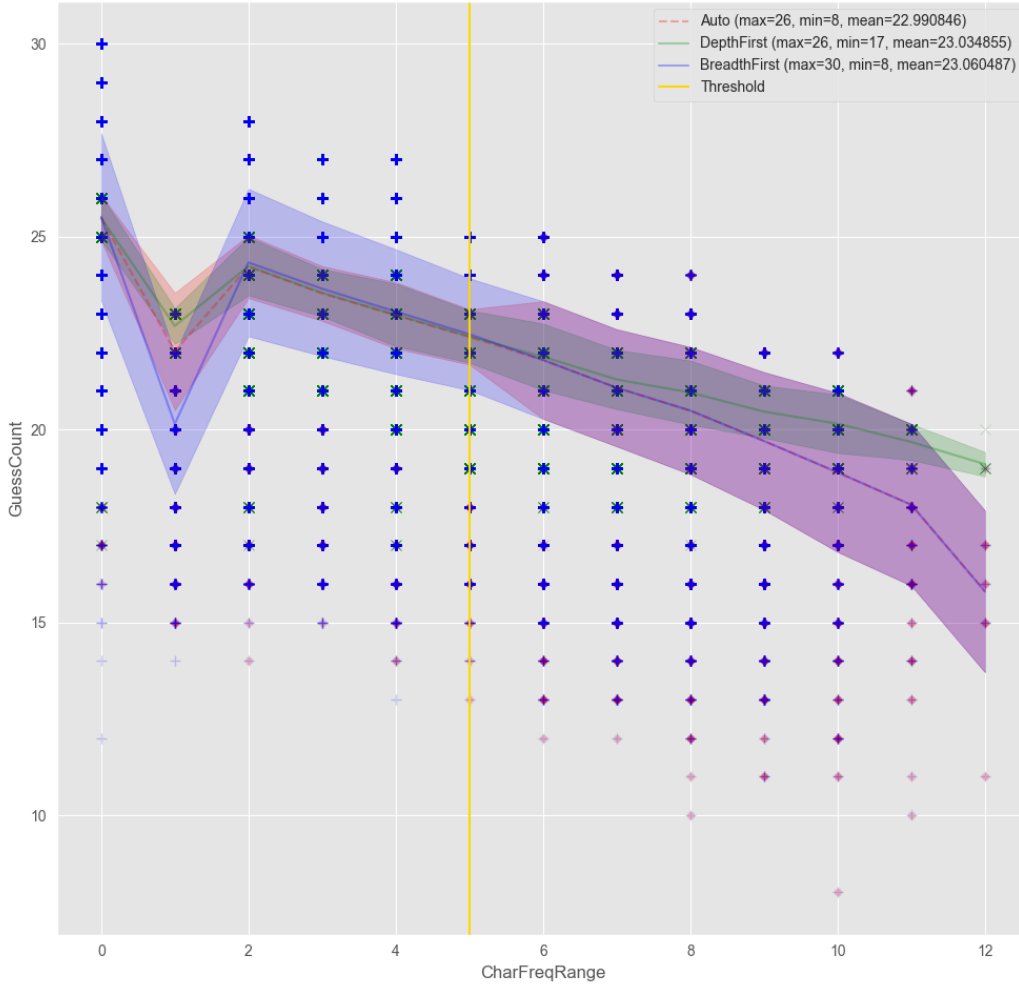


Figure 10: Guess Count over Character Frequency Range with Threshold marker (in gold)

From [Figure 10](#), we can clearly see that the implemented “Auto” algorithm has captured well the best guess count performance between the Linear Character Swap – Depth First and the Linear Character Swap – Breadth First algorithms: Its mean (average) test guess count over 1,000,000 random test cases is 22.99, lower than that of both Depth First (23.03) and Breadth First (23.06), while still keeping the guess count complexity at $O(n)$. Over those 1,000,000 test cases, “Auto” algorithm’s minimum guess count and maximum guess count (8 and 26) is also a combination of the best guess count performance of Depth First (17 and 26) and Breadth First (8 and 30).

5.2.3. Key Length vs. Execution Time vs. Guess Count

After implementation of the “Auto” algorithm, we now have 3 solutions:

- Linear Character Swap – Depth First only
- Linear Character Swap – Breadth First only
- “Auto” Algorithm that smartly switches between Linear Character Swap – Depth First and Linear Character Swap – Breadth First based on Character Frequency Range for best resulting performance.

Our team has produced a graphical representation to illustrate the relationship between the duration of execution and the key length for all three solutions ([Figure 11](#)).

From this, it is evident that although the “Auto” algorithm has the same theoretical **time complexity** with Linear Character Swap – Depth First and Linear Character Swap – Breadth First algorithms ($O(n)$), its empirical time efficiency is slightly lower than that of the two algorithms. This is due to the algorithm’s use of a threshold and the need to switch between the Depth First and Breadth First algorithms, which results in a slight execution time increase. Furthermore, upon observation of the empirical runtime graphs of all three algorithms, it appears that their time complexities may be $O(n^2)$ or $O(n \cdot \log(n))$, in contrast to the theoretical time complexity of $O(n)$. This discrepancy could be attributed to costly operations within loops, such as when we need to repeatedly loop through possible characters in the general phase of our solution.

The **guess count complexity** of all three algorithms is $O(n)$ just as anticipated. Although the average guess count for the Linear Character Swap – Breadth First algorithm seem to increase faster than that of “Auto” and Linear Character Swap – Depth First – because of how frequently the Breadth First algorithm encounters its worst case as the secret key length increases (evenly distributed character frequency has more chance to happen the more character we have, as expected in basic probability math) – the “Auto” algorithm seem to have avoided all of those worst cases by correctly choosing the Depth First algorithm for them instead of the Breadth First algorithm, using the threshold mentioned in [5.2.2. Correlation Between Character Frequency Metrics and Final Metric of Choice](#).

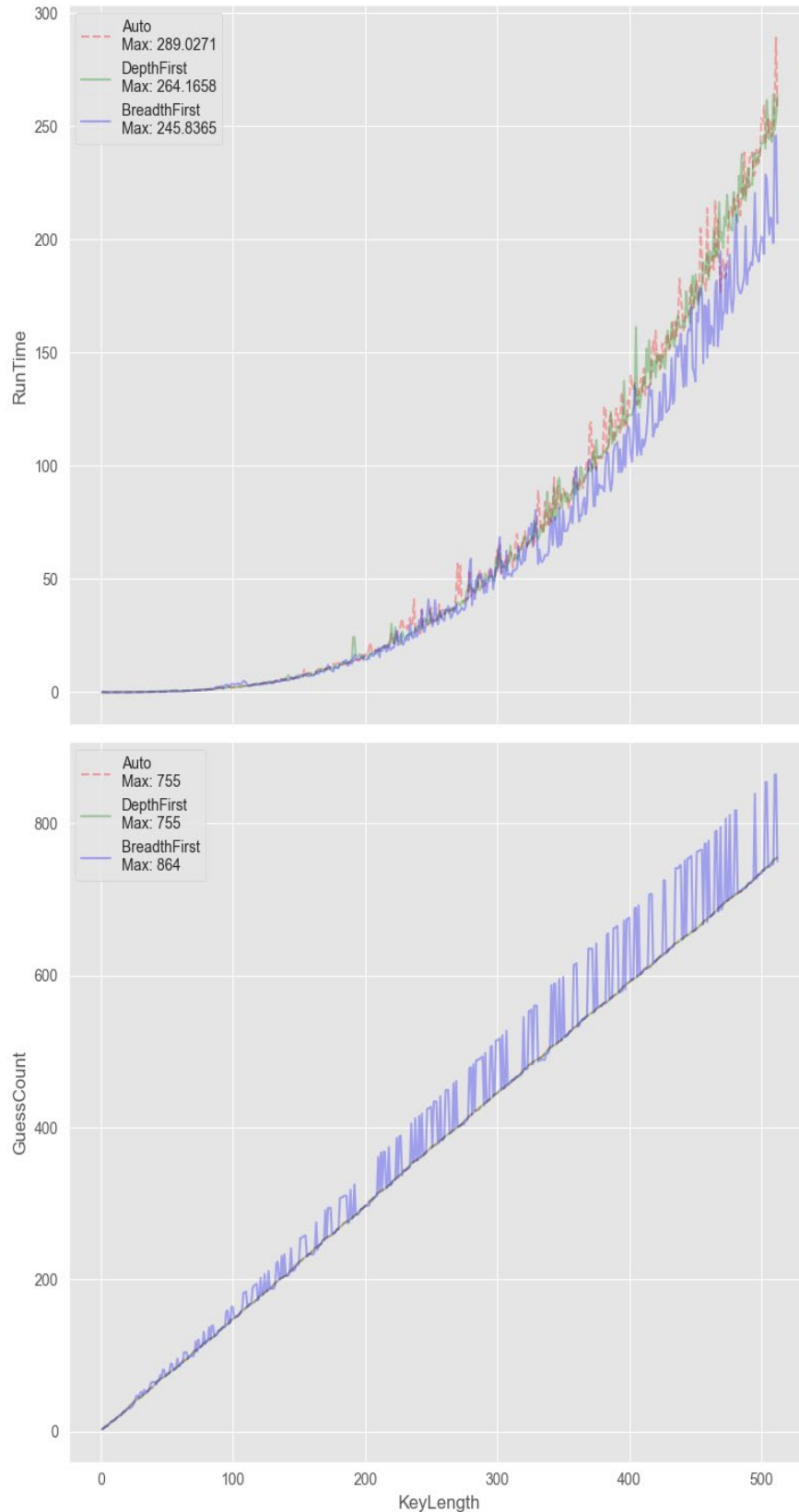


Figure 11: Execution Time and Guess Count by Secret Key Length

6. CONCLUSION

Through test-driven iterative research & development, done in a divide-and-conquer approach, aided by data visualization techniques using Python, we have developed a successful algorithm to consistently (100% accuracy) solve the problem with the lowest guess count possible (**average of 22.99 guesses, maximum of 26 guesses, minimum of 1 guess, for a random secret key of length $n = 16$**). All **space, time, and guess count complexity** of our solution is constant at best and linear at worst, as follow:

- **Best case:** $O(1)$
- **Worst case:** $O(n)$
- **Average case:** $O(n)$

Where **n** is the length of the secret key with 4 possible characters 'R', 'M', 'I', and 'T'.

Generalized problem: If our solution is adapted for the same problem but with a secret key of length **n** that consists of **k** possible values, all **space, time, and guess count complexity** of our solution would be:

- **Best case:** $O(1)$
- **Worst case:** $O(k \cdot n)$
- **Average case:** $O(k \cdot n)$

If we are to further develop our solution to also be able to solve this generalized problem, we will utilize a hash map abstract data structure to efficiently keep track and store the frequencies of possible characters in the secret key, along side with a rank of how common it is in the secret key. This will enable constant time retrieval of character frequency and character commonality rank, thus cutting down on our execution time. However, this is not in the scope of the original problem. Thus, we did not pursue it.

REFERENCES

- [1] E. W. Weisstein, "Mastermind," *MathWorld - A Wolfram Web Resource*. [Online]. Available: <https://mathworld.wolfram.com/Mastermind.html>.
- [2] D. E. Knuth, *The art of computer programming: Volume 3: Sorting and Searching*. Addison-Wesley Professional, 1998.
- [3] Apache Maven Project, "Introduction to the Standard Directory Layout," *Apache Maven Project*, 2023. [Online]. Available: <https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html>
- [4] E. Furey, "Descriptive Statistics Calculator," *CalculatorSoup*. [Online]. Available: <https://www.calculatorsoup.com/calculators/statistics/descriptivestatistics.php>