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

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-character secret key containing only the characters *'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

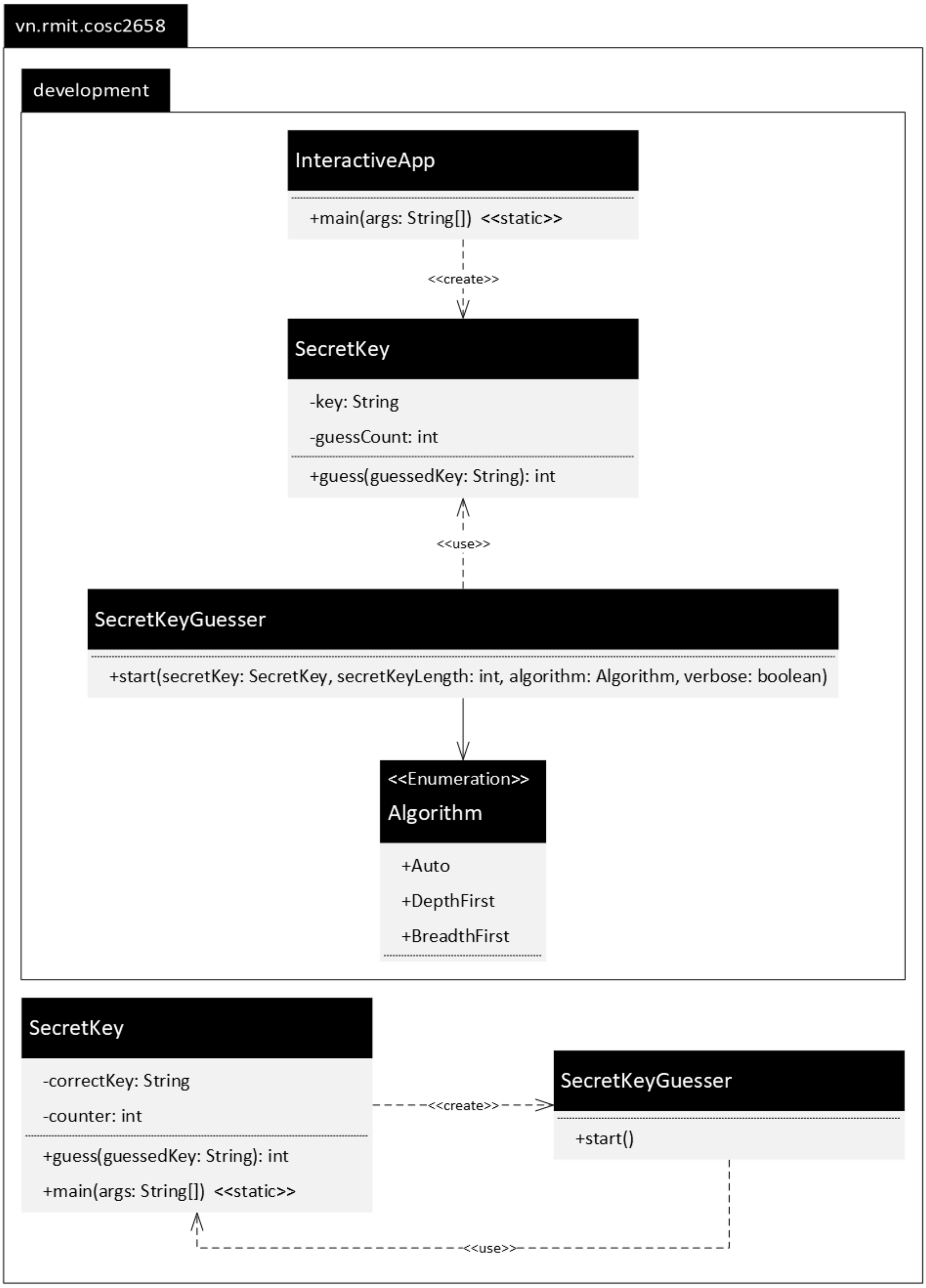


Figure 1: Class Diagram:

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 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 `test-data` 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 `test-data` 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 $\boldsymbol{n = 16}$.
- `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 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 character, we create a consecutive repeated character string with that character.
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 ($\boldsymbol{n = 16}$), 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 of the repeated character *'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 all other possible character (depth-wise). The number of matching positions of this new guess is used to determine which character is correct for that position in Step 3.

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. If it is lower, the previous character is kept. Otherwise, if the number of matching positions remains unchanged, the algorithm tries the next possible character until it finds the correct one.

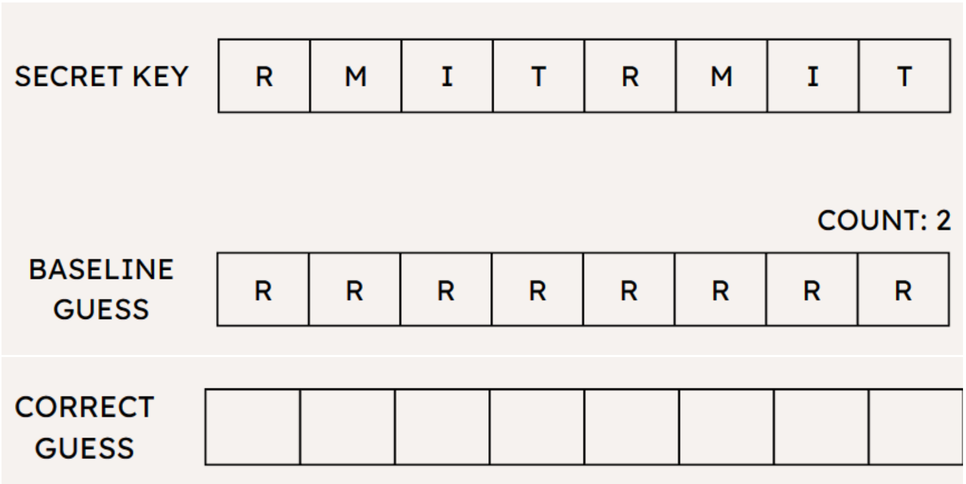
Repeat Step 2 and 3 for all $\boldsymbol{n} = \mathbf{16}$ positions of the guess string, left to right.

3.2.2.2. Visualized Description:

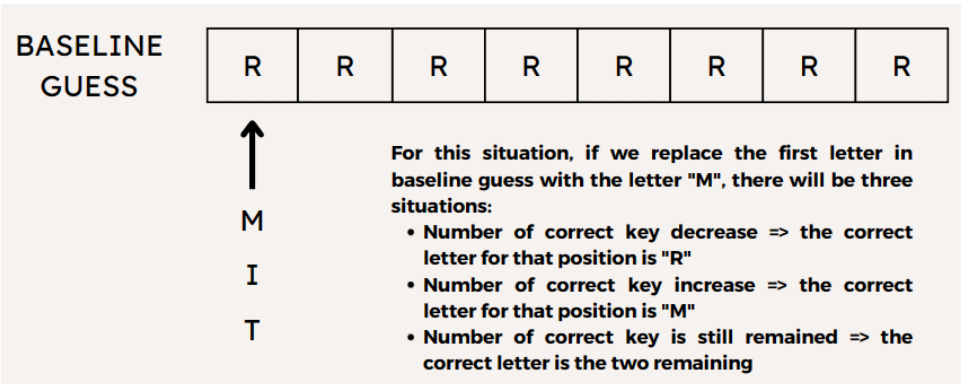
The figures below will describe this algorithm, with `COUNT` accounts for the number of correct matches:

The secret key for this example is "RMITRMIT" with key length $\boldsymbol{n} = \mathbf{8}$ (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 correct guess string with the same length.



Then, we go through each index of the string and try all other possible characters.



For example, we change the first character to *'M'*. As a result, the number of matching positions of the new guess is ***smaller*** than the one of baseline guess.

KEY	R	M	I	T	R	M	I	T
BASELINE GUESS	R	R	R	R	R	R	R	R

COUNT: 2

NEW BASELINE GUESS	M	R	R	R	R	R	R	R

COUNT: 1

The "count" had been decreased => the correct letter for the first position is "R"

Due to that reason, this character for this position should be *'R'*. We add that character to correct guess string at the same position. Then we move to the next position because there is no need to try *'I'* and *'T'*.

CORRECT GUESS	R							

In this case, when we change the second character to *'M'*, the number of matching positions of this new guess will be ***higher*** than the one of baseline guess.

BASELINE GUESS	R	R	R	R	R	R	R	R

↑

M

I

T

KEY	R	M	I	T	R	M	I	T
BASELINE GUESS	R	R	R	R	R	R	R	R

COUNT: 2

NEW BASELINE GUESS	R	M	R	R	R	R	R	R

COUNT: 3

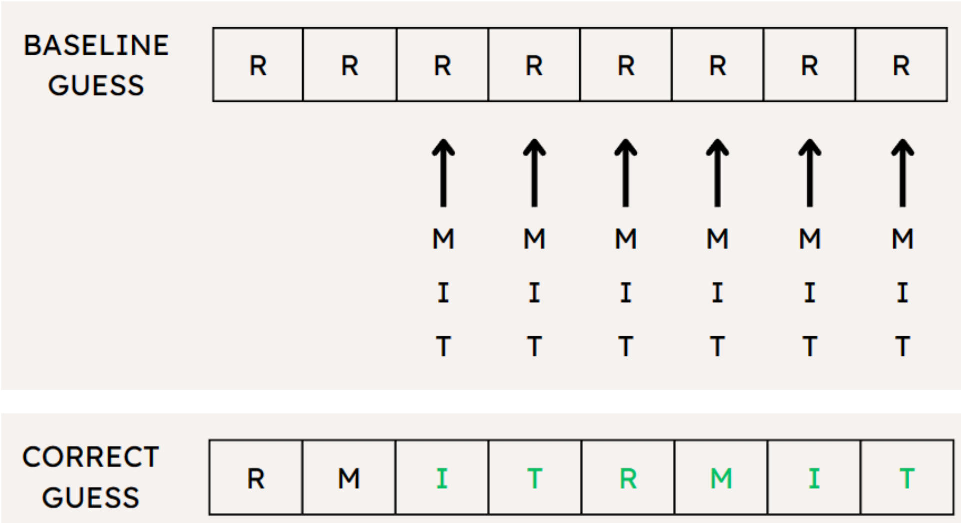
The "count" had been increased => the correct letter for the first position is "M"

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.

CORRECT GUESS	R	M						

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

We repeat these steps for the whole baseline guess.



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 of the repeated character *'R'*. Then, we count the number of matching positions of that guess. 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 that index to `true`. Next, when we iterate through the guess array again, we can bypass positions that have already been marked as `true` and proceed to the next location.

Step 2: We access a specific character in the list of characters and spread it to every position (breadthwise) of the guess string, where is still marked as `false`. The number of matching positions of this new guess is used to determine which character is correct for that position in Step 3.

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 then, we mark that position as `true`. If it is lower, the baseline character is kept, then, we mark that position as `true`. Otherwise, if the number of matching positions remain unchanged, we move to the next position. After reaching the end of the string, we try the next possible character.

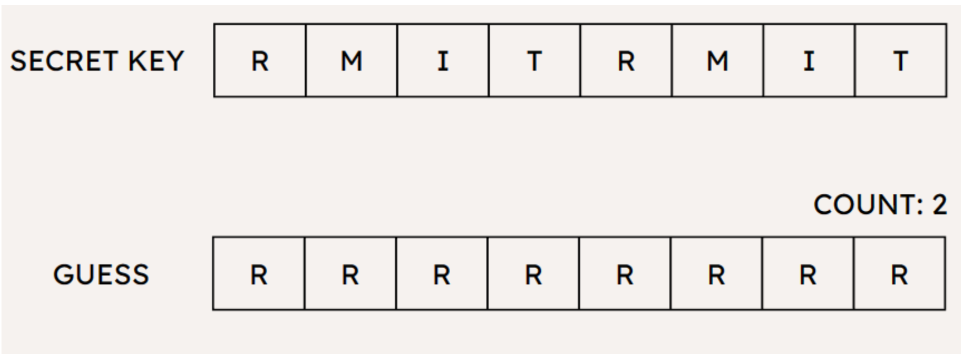
Repeat Step 2 and 3 until we have exhausted all incorrect positions.

3.2.3.2. Visualized Description:

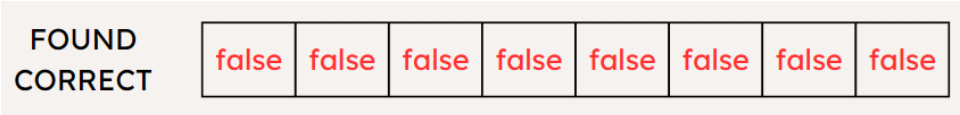
The figures below will describe this algorithm, with `COUNT` accounts for the number of correct matches:

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

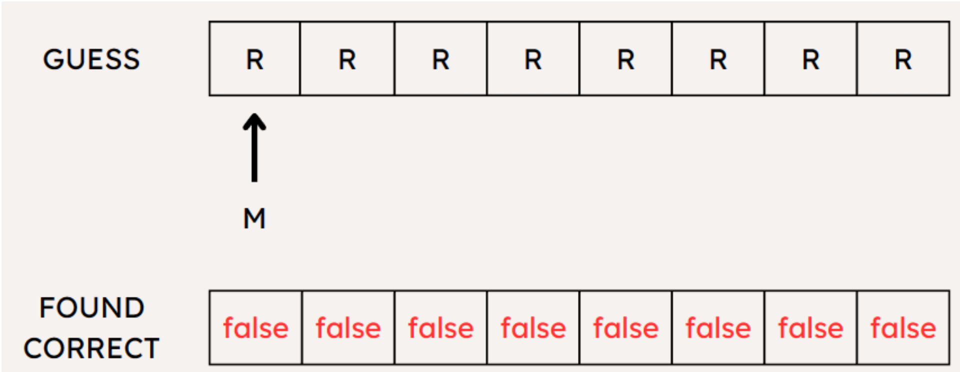
First, we will initialize the base string "RRRRRRRR" and count the number of matching positions of that guess by calling the `guess()` method from `SecretKey` class.



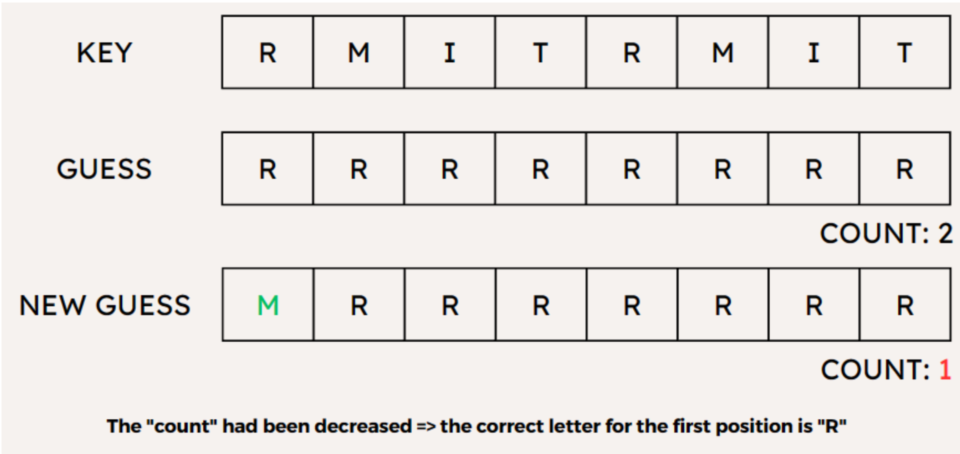
We also initialized a Boolean array to mark the positions that the algorithm has already found the correct character for. It its filled with `false` at the start of the algorithm.

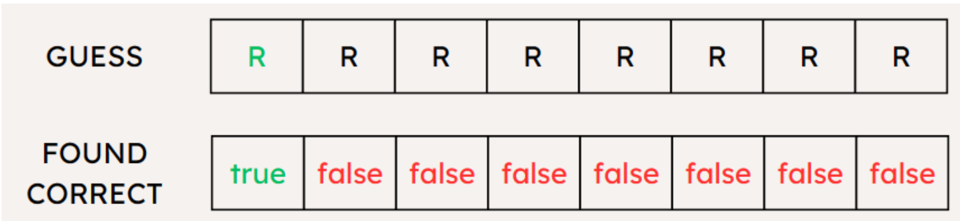


Then we spread the remaining possible characters through the guess string, left to right. For example, we spread the next possible character *'M'*.

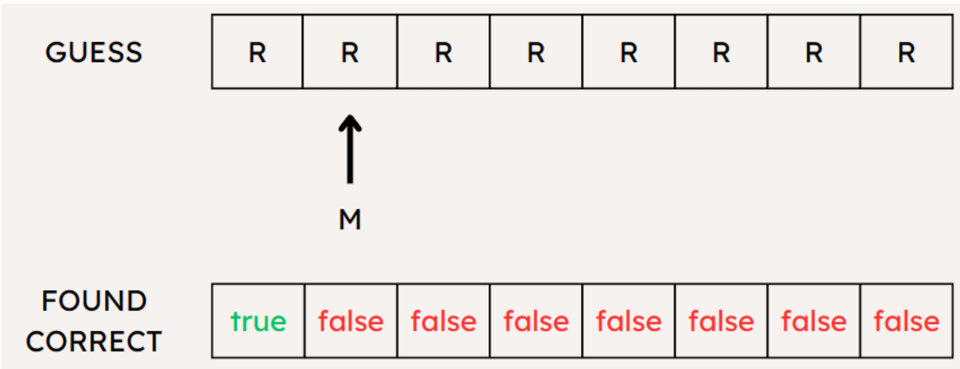


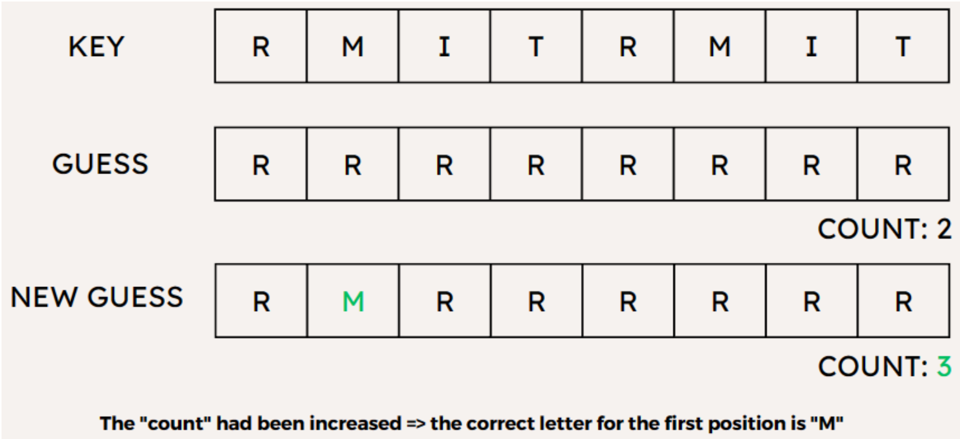
When we change the first index of the guess string to *'M'*, the count reduced. Thus, *'R'* is the correct character for this position, which will be marked as `true`. Then, we move to the next index.

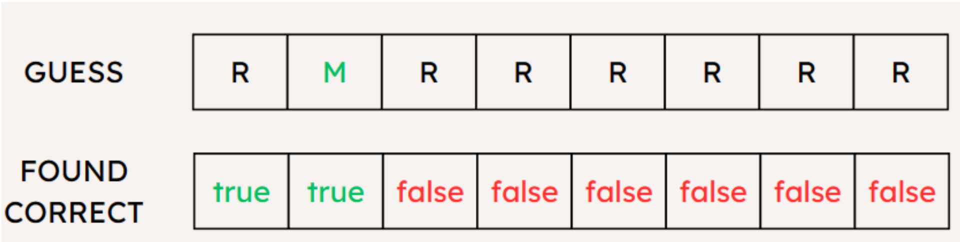




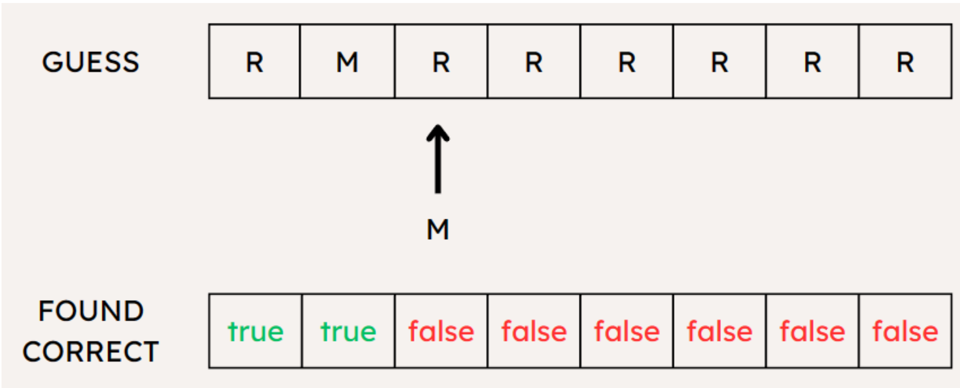
At the next position, after changing it to *'M'*, the number of matches increased, which means that *'M'* is the correct character at this index. The guess string now changed to "RMRRRRRR":

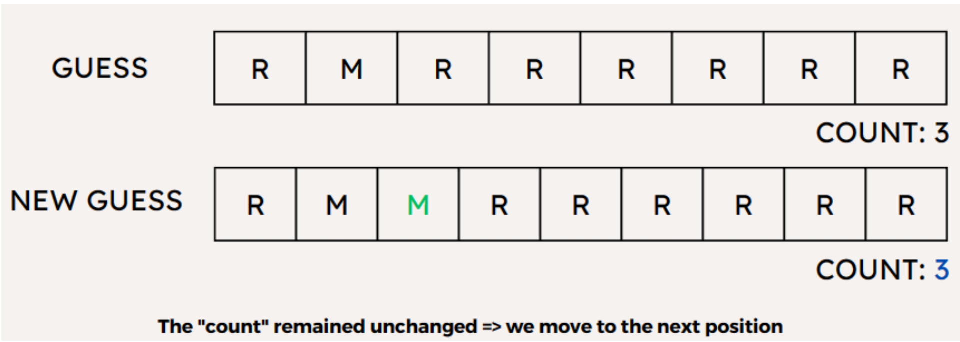




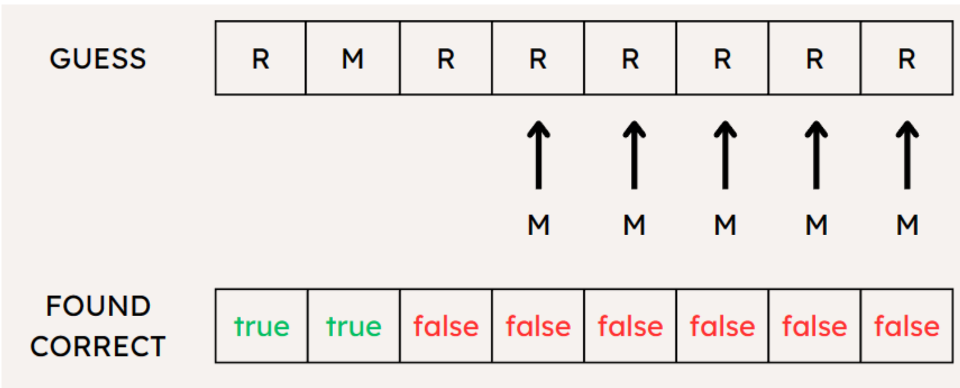


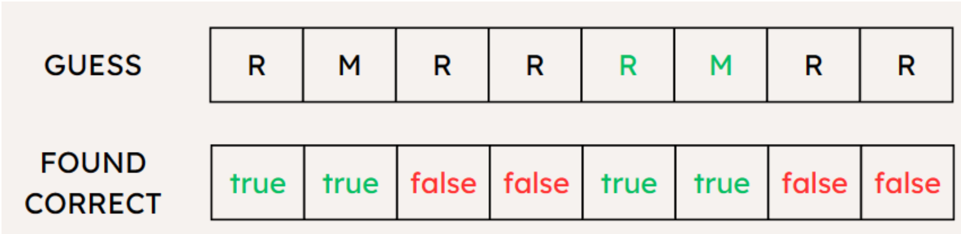
At the next position, after changing it to *'M'*, the counter remained the same, thus, *'R'* and *'M'* are both wrong.



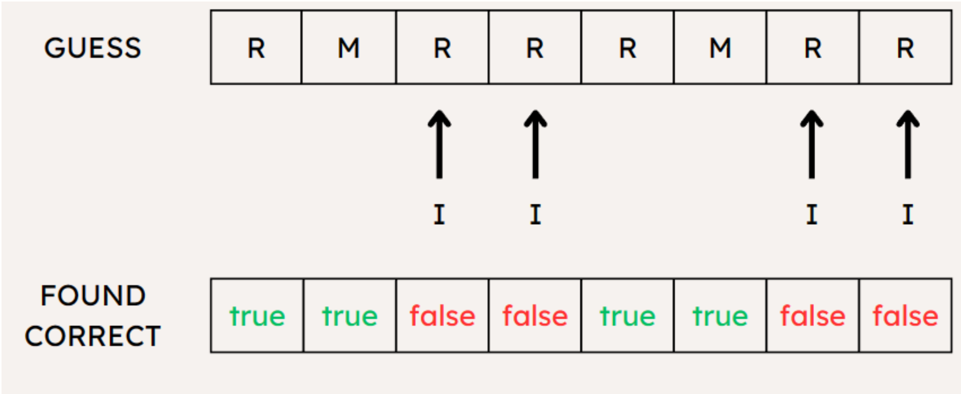


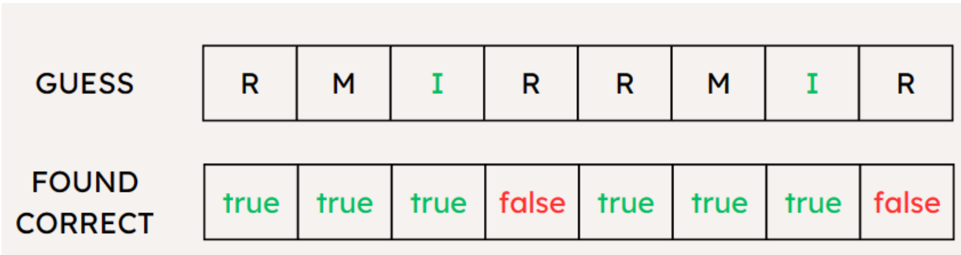
We move to the next position and repeat these steps to the end of the guess string.



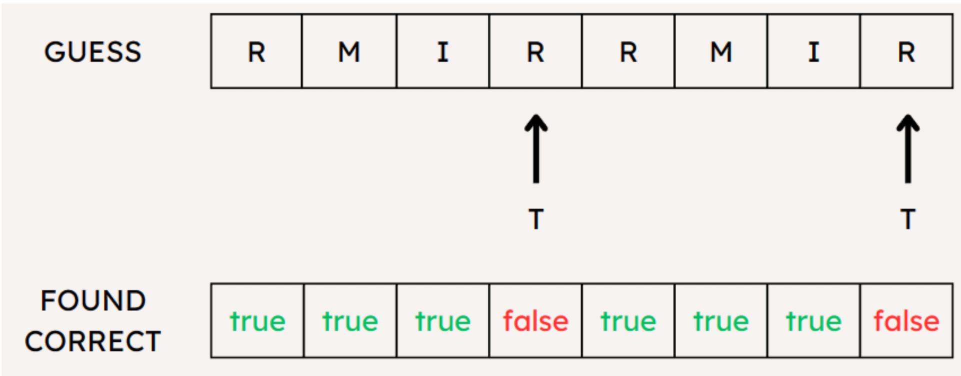


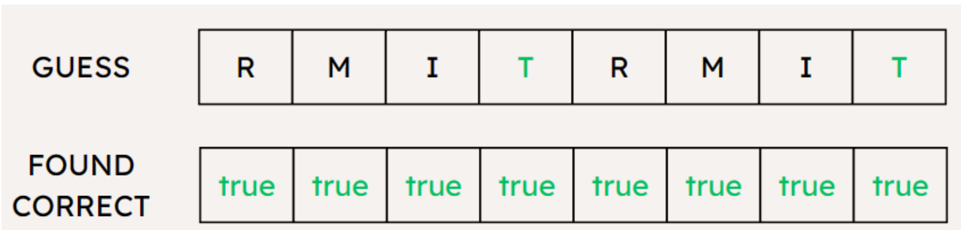
Once *'M'* has reached the end of the guess string, we try the next possible character *'I'* and spread it to all other positions that are still marked as `false`. We repeat the same process as the previous character *'M'*.





With the next possible character *'T'*.





3.3. Optimization

Following the implementation of all the algorithms, we realize that they can be improved to decrease the guess count even further using the following techniques:

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 a frequency array `charFreq[]`, which stores the number of appearances of each possible character.

Moreover, this array is then also sorted in descending order to guess the most common character in the secret key first, thus reduce the number of incorrect guesses.

3.3.1.2. Usage:

By utilizing this frequency array in <u>both</u> Linear Character Swap algorithms, the number of guesses can be reduced:

First, in both algorithms, rather than beginning with a baseline consisting of repeated *'R'*, we construct the baseline guess string using the most frequent character, and the order of the possible characters that we use to change at each position is determined by the order of the sorted frequency array. For example, if the frequency array is {'*M*', '*T*', '*I*', '*R*'}. In this instance, *'M'* is used as the baseline guess "MMMMMMMM", followed by *'T'*, *'I'*, and *'R'* in that order. This optimization will reduce the number of guesses because the probability of that possible character appearing in the secret key is higher.

Second, each time we successfully swap to the correct character in the baseline guess, we will decrease the frequency of it. If its frequency reaches zero, we stop guessing for that character, thus, reducing the number of guesses compared to the original algorithm, which required us to check every possible circumstance on the entire set of characters.

3.3.2. The Last Remaining Character

Using this technique, we eliminate the need to call `guess()` for the last remaining character after all other possible characters have been exhausted.

- **For Linear Character Swap – Depth First:** In this algorithm, if the number of matching positions is not higher after trying the first two probable characters 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 calls to the `guess()` method.
- **For Linear Character Swap – Breadth First:** In this algorithm, after exhausting all possible characters except for the last remaining one, and some positions are still marked as `false`, we can infer that the remaining character 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).

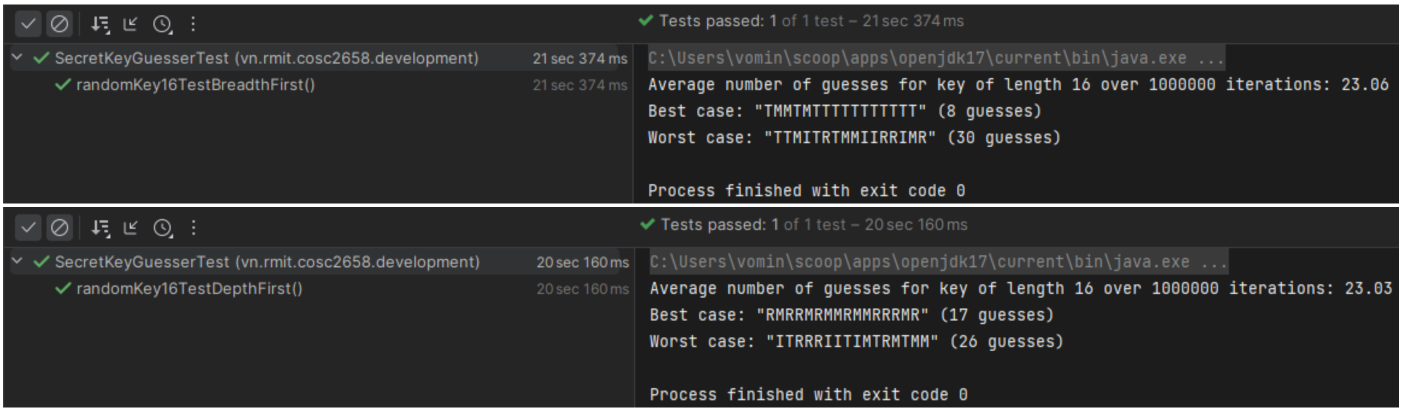


Figure 2. Test Results for Linear Character Swap - Breadth First and Linear Character Swap - Depth First over 1,000,000 pseudo-random 16-character keys

We observed that in some test cases, the Linear Character Swap – Depth First algorithm and the Linear Character Swap – Breadth First algorithm 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 analysis using our Python Jupyter Notebook, we determined that the difference in character frequency distribution is the factor that affects performance between our two algorithms. Consequently, based on descriptive statistics [4], we further investigated character frequency distribution as follow:

Using Junit to ensure solution accuracy (the test would fail if only one guess is wrong), we tested the same 1,000,000 pseudo-random secret keys for both Linear Character Swap – Depth First and Linear Character Swap – Breadth First algorithms to guess. In these two runs, we store these metrics calculated from the character distribution frequency of each test case:

- **Character Frequency Range:** The character frequency range R is the numerical distance between the highest character frequency and the lowest character frequency, calculated by this formula:

$$R = x_{max} - x_{min}$$

$$\textit{where } x \textit{ is the frequency of a character}$$

- **Character Frequency Median:** The median character frequency $\tilde{x}$ is the numerical value between the lower half and the upper half of an unsorted character frequency vector. It is calculated using the following formula for k possible characters:

$$\tilde{x} = x_p$$

$$\textit{where index } p = \frac{k}{2} \textit{ for 0-based indices}$$

- **Character Frequency Variance:** Character frequency variance σ^2 measures the dispersion of k possible characters' frequencies from their mean μ:

$$\mu = \frac{\sum_{i=0}^{k-1} x_i}{k}$$

  It is calculated using the following formula:

$$\sigma^2 = \frac{\sum_{i=0}^{k-1} (x_i - \mu)^2}{k}$$

For each of the above three metrics, we visualized the output test data using Python and Jupyter Notebook in the `notebooks` folder (See the visualizations in 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 $\boldsymbol{k}$ be the number of possible characters. In this problem, it is a constant $\boldsymbol{k} = \mathbf{4}$ for the possible characters *'R'*, *'M'*, *'I'*, and *'T'*.
- Let $\boldsymbol{n}$ be the secret key length.
- Let $\boldsymbol{C}$ 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 these 3 aspects simultaneously: **Space**, **Time, and Guess Count Complexity.**

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: $\boldsymbol{n} \ = \mathbf{16}$
- 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
GUESS:	R	R	R	R	R	R	R	R

Figure 4: Initial Phase – Worst case scenario

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

- Match count $< \boldsymbol{n} = \mathbf{16}$
- The algorithm continues to guess each possible character in the secret key until the cumulative frequency count reaches the secret key length.

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

→ The operation cost can be to be linear for **Time**, **Space** and **Guess** complexity regarding the $\boldsymbol{k} = \mathbf{4}$ 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 keeps track of each possible character remaining frequency, so that if a character frequency is already exhausted ($x = 0$), it no longer guesses 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 $\boldsymbol{n-1}$ 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 $\boldsymbol{k-2}$, 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:

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

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 $\boldsymbol{n-2}$. In the worst case, it iterates $\boldsymbol{n-1}$ times.
- The inner loop iterates over `nextCommonCharIndex` from 1 to $\boldsymbol{k-2}$. In the worst case, it iterates $\boldsymbol{k-1}$ 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 keeps track of each possible character remaining frequency, so that if a character frequency is already exhausted ($x = 0$), it no longer guesses 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 correc
        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
`for nextCommonCharIndex from 1 to CHAR.length – 1 do:` `    ...`	The outer loop performs iterations equivalent to the constant value of $\boldsymbol{k-2}$, without considering input size. Thus, the inner loop's time complexity can be deemed as $O(1)$.

`for i from 0 to secretKeyLength – 1 do:`
`    if correctCount >= secretKeyLength – 1 or charFreq[nextCommonCharHash] <= 0 or i >= secretKeyLength:`
`        break`
`    ...`
	The inner loop iterates $\boldsymbol{n-1}$ times. Hence, it has a time complexity of $O(n)$.

We can see the complexity of outer and inner loop is just opposite to those of the Linear Character Swap – Depth First algorithm's outer and inner loop. As a result, the overall complexity of Linear Character Swap – Breadth First the same as that of 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: best case of initial guesses phase – $O(1)$
- Worst case: worst case of general phase – $O(n)$
- Average case: average case of general phase – $O(n)$

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).

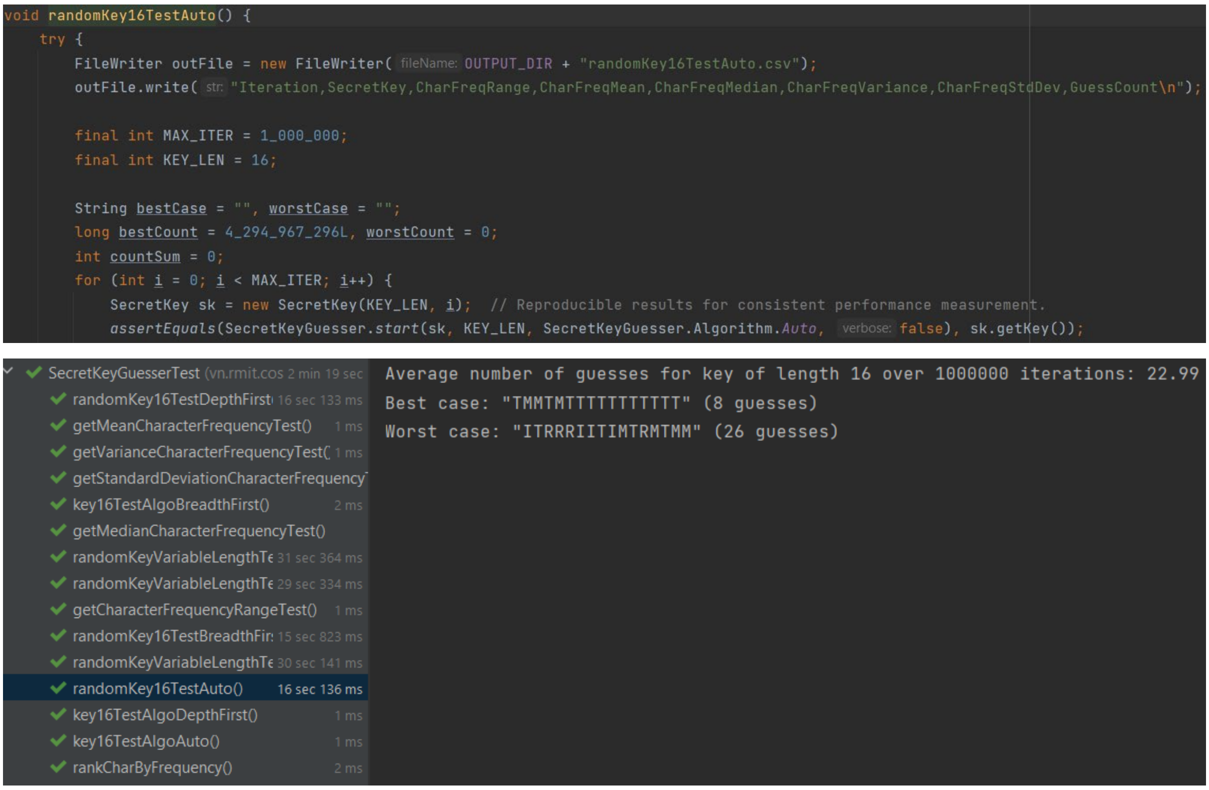


Figure 7: Test Code and Output

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 R,
- Character Frequency Median $\bar{x}$,
- Character Frequency Variance σ^2.

5.2.1. Guess Count Efficiency Measurement and Observation

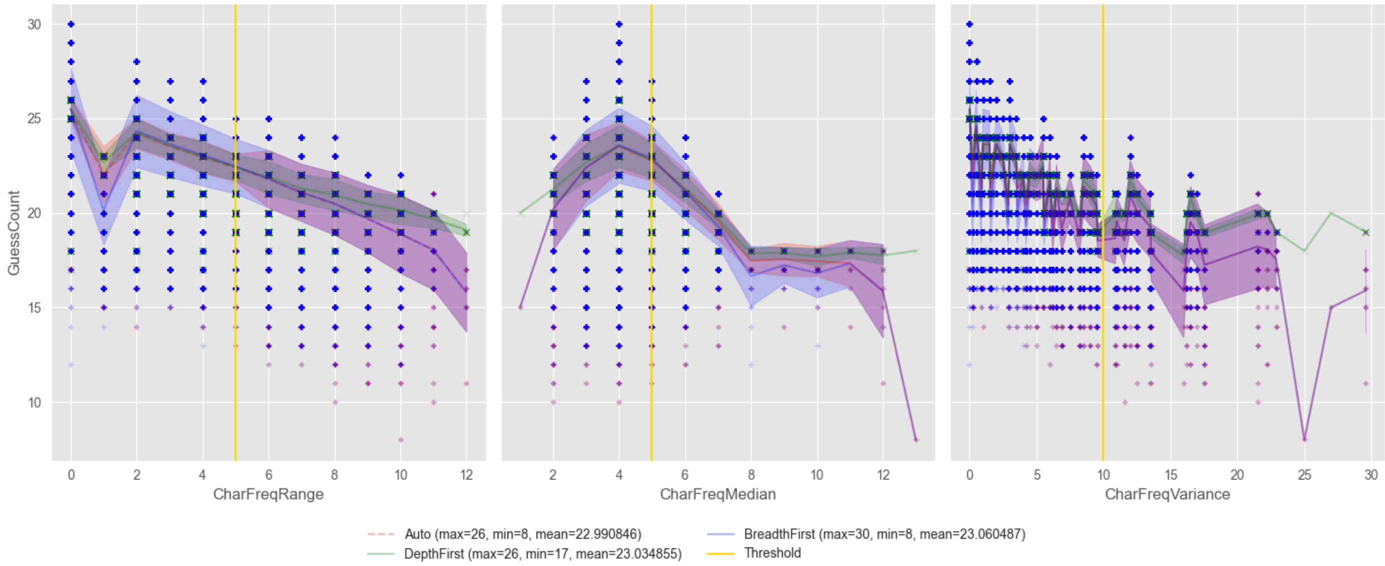


Figure 8: Guess Count by Character Frequency Metrics with Threshold marker (in gold)

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:
  - $R < 5$
  - $\bar{x} < 5$
  - $\sigma^2 < 10$
- Breadth First Algorithm's best performance range:
  - $R \geq 5$
  - $\bar{x} \geq 5$
  - $\sigma^2 \geq 10$

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

- $T_R = {}^{n}/_{3.2}$
- $T_{\bar{x}} = {}^{n}/_{3.2}$
- $T_{\sigma^2} = ({}^{n}/_{5})^2$

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

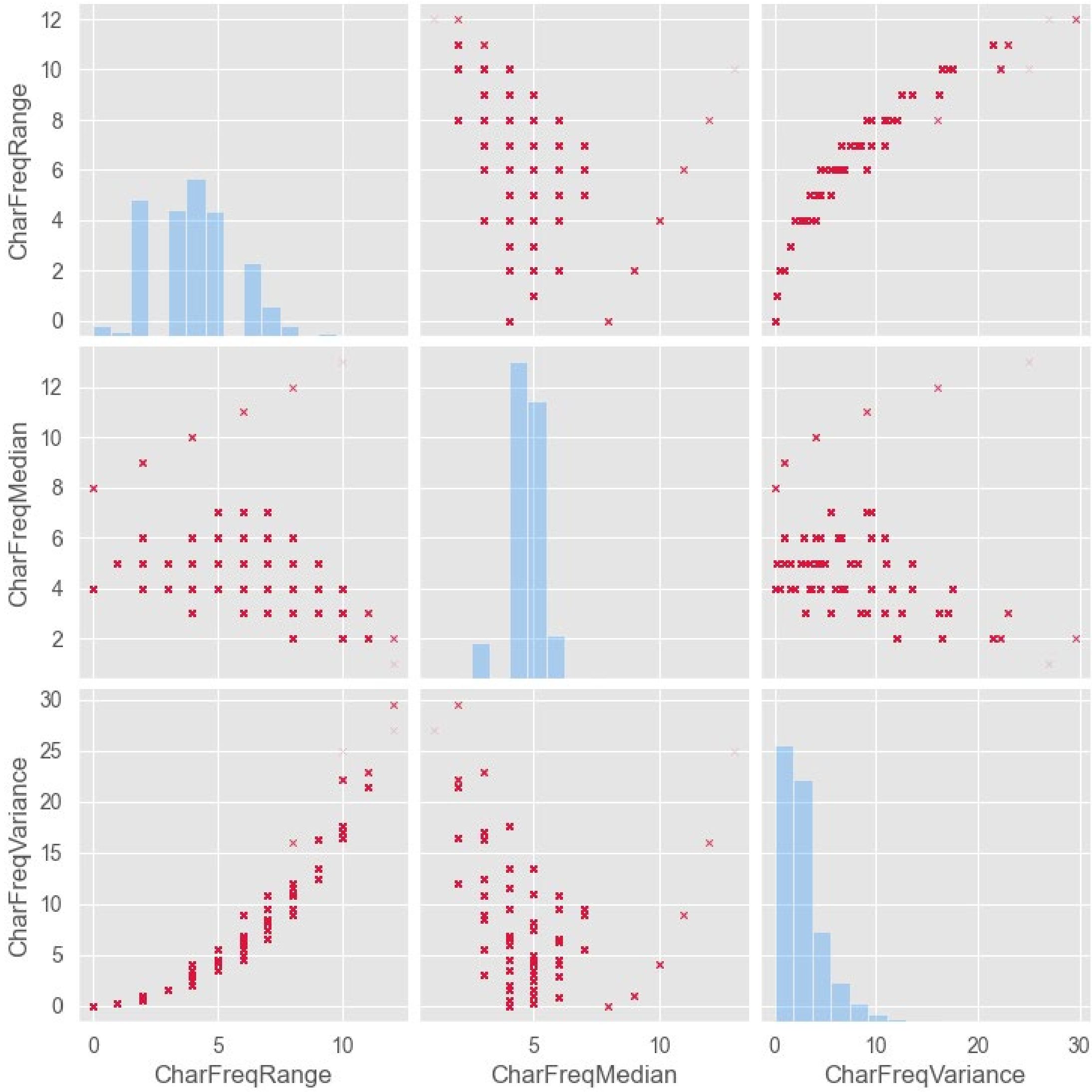


Figure 9: Correlation map between Character Frequency Range, Character Frequency Median, and Character Frequency Variance

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:

$$Range = x_{max} - x_{min} \text{ (See 3.3.3)}$$

Thus, our threshold is $\boldsymbol{n}/\mathbf{3.2} = \mathbf{16}/\mathbf{3.2} = \mathbf{5}$. Implemented in our class `SecretKeyGuesser`, it is:

```
...
double algoThreshold = secretKeyLength / 3.2;
int characterFrequencyRange = getCharacterFrequencyRange(charFreq);
if (characterFrequencyRange <= algoThreshold) {
     ...
     linearCharacterSwapDepthFirst(...);
} else {
     ...
     linearCharacterSwapBreadthFirst(...);
}
```

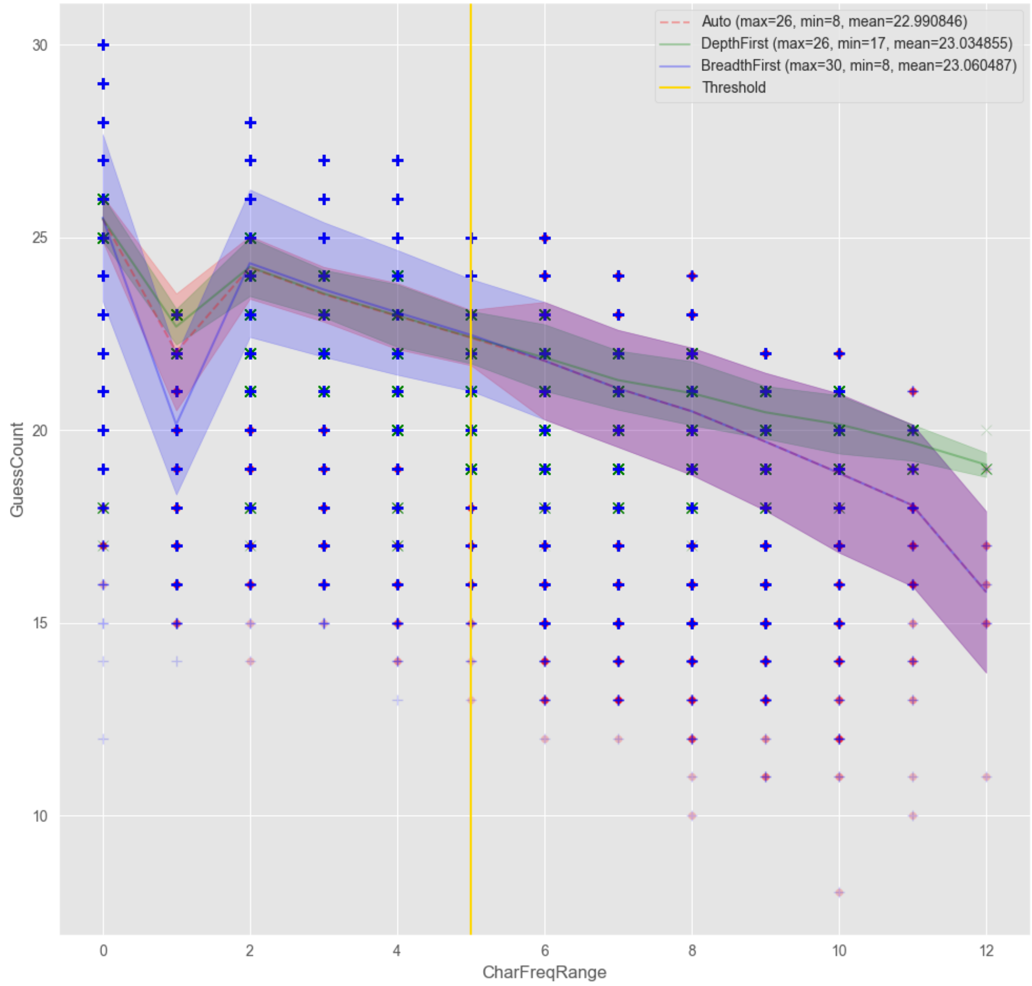


Figure 10: Guess Count over Character Frequency Range with Threshold marker (in gold)

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 $O(n)$ as analysed in 4. Complexity Analysis. 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 algortihm 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.

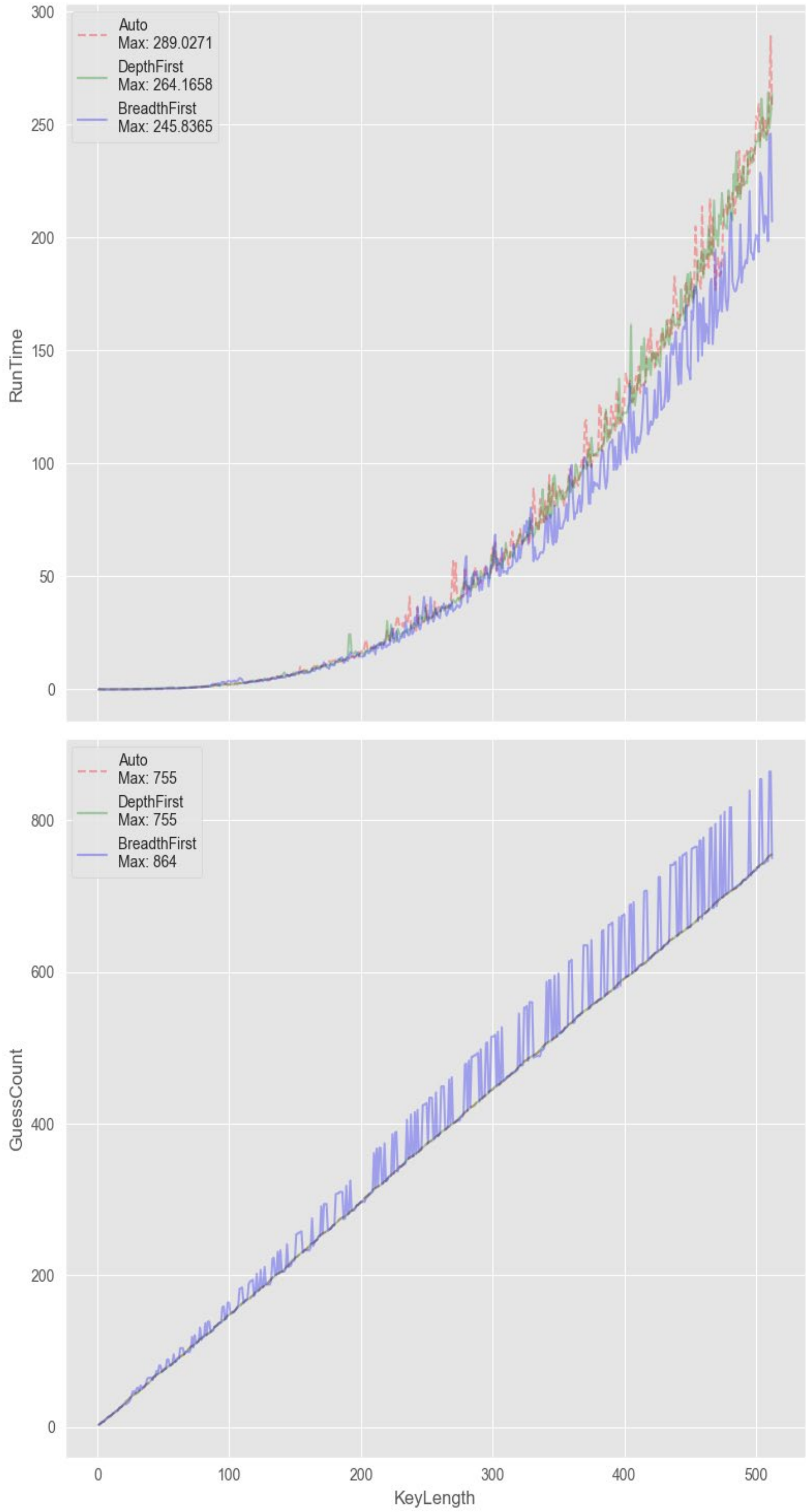


Figure 11: Execution Time and Guess Count by Secret Key Length

6. CONCLUSION

Through test-driven iterative research & development, 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 $\boldsymbol{n} = \mathbf{16}$)**. The **space**, **time**, and **guess count complexity** of our solution are all 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 $\boldsymbol{n}$ that consists of $\boldsymbol{k}$ possible values, the **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 efficiency keep track and store the frequencies of possible characters in the secret key, along side with a binary-tree structure to keep track of how common these characters are in the secret key. This will enable constant time retrieval of character frequency and logarithmic time retrieval of character commonality rank, thus cutting down on our execution time (but not the number of guesses, this is purely a time complexity optimization). However, this is not in the scope of the original problem, as we only have $\boldsymbol{k} = \mathbf{4}$ possible characters. Thus, we did not pursue this option for this problem, as the time complexity increase would be negligible comparing to the amount of effort that we put in.

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