

RMIT

UNIVERSITY

Term	Semester 3, 2023
Course code	COSC2658
Course name	Data Structures and Algorithms
Title of Assignment	Group Project
Due date	Dec 22 by 23:59
Group	Group 17
Name and student ID	Nguyen Thien Co - s3938338 Vo Minh Thien An - s3916570 Nguyen Mau Bach - s3926937 Nguyen Huu Anh Duy - s3922568
Location	Saigon South Campus
Lecturer	Ling Huo-Chong
Number of Page	23 (excluding Cover Page)

Secret Key Guesser - Enhanced Digit by Digit Modification

Group Project - Group 17

Nguyen Thien Co - s3938338
Nguyen Mau Bach - s3926937

Vo Minh Thien An - s3916570
Nguyen Huu Anh Duy - s3922568

Table of Contents

I. Problem Statement	2
II. System Specification	2
1. Introduction	2
2. High-level Design	2
2.1. Folder Structure	2
2.2. Program Structure (Source Code)	4
A. Relationship	4
B. Component	4
III. Development Process	7
IV. Data Structures and Algorithms	8
1. Data Structure - Frequency Dictionary	8
2. Algorithms	11
2.1. Main Algorithm - Digit By Digit Optimization	11
2.2. Sorting Algorithm	14
2.3. Optimization Algorithms	15
A. Efficient Base GuessKey Selection	15
B. Guessing Sequence Enhancement	15
C. Early Termination	17
V. Complexity Analysis	19
1. Abbreviations	19
2. Workflow	19
3. Analysis	20
VI. Evaluation	21
1. Testing Approach	21
2. Correctness Evaluation	21
3. Efficiency Evaluation	22
References	23

I. Problem Statement

"You found an important file encrypted with a 12-character secret key, you don't remember the password but you know that the key is made of five characters 'M', 'O', 'C', 'H' and 'A' which is inspired by your favourite coffee 'mocha'. You created a function that uses brute force to find out the key, but it will take a massive number of 5^{12} guesses to find the key. Now you need to modify it to enhance the code so that it takes as few guesses as possible." [1]

II. System Specification

1. Introduction

Our "OptimizedDigitByDigitMod" is a standalone application that guesses fixed-length string keys by employing strategic algorithms.

This system is written in Java, Oracle OpenJDK version 21.0.1 and its main goal is to overcome the challenge of finding secret string keys. Instead of applying a brute force method, the program's problem-solving strategy involves reducing the number of potential keys through queries that focus on accurate positions and optimising subsequent guesses via frequency analysis of characters in the guessed keys.

The components include a secret key representation, a guessing driver, and character frequency storing and sorting. In order to find the target key, the best next guess orders are identified by carefully tracking and evaluating these frequencies.

- Github Repo URL:

https://github.com/tonydiggory/RMIT_GroupProject_AlgosStructure_Group17

2. High-level Design

2.1. Folder Structure

This section is intentionally blank.
The content is on the next pages.

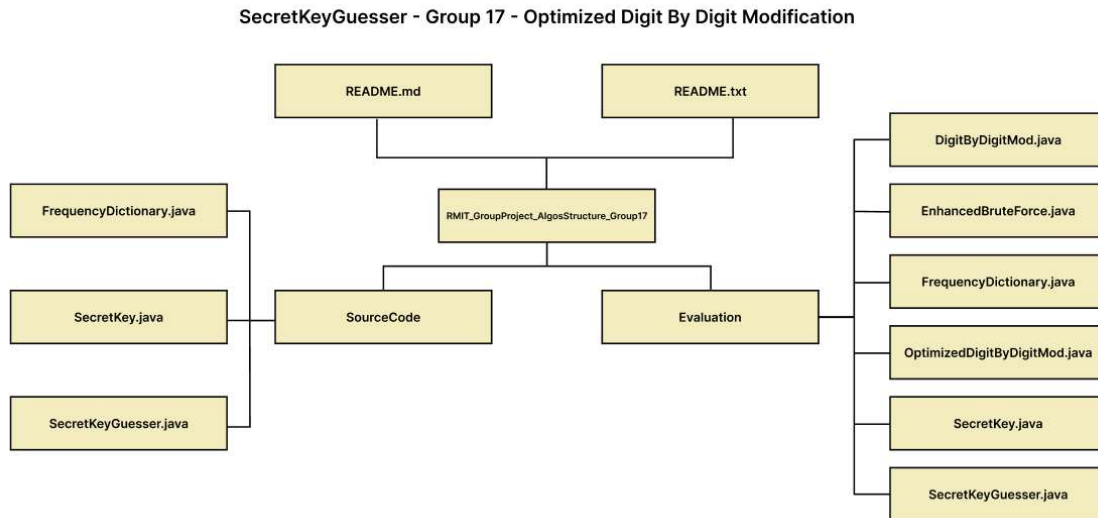


Figure 1: Folder Structure

RMIT_GroupProject_AlgosStructure_Group17

- README.md, README.txt: describes the contribution score of all members, folder structure and contains the link to the project video

Note: The program in the evaluation process will be slightly different from the one from the "SourceCode" folder to reduce complexity and enhance convenience while calculating the metrics.

- SourceCode: stores Group 17's final solution
 - + FrequencyDictionary.java
 - + SecretKey.java
 - + SecretKeyGuesser
- Evaluation: stores different approaches and testing program
 - + DigitByDigitMod.java
 - + EnhancedBruteForce.java
 - + FrequencyDictionary.java
 - + OptimizedDigitByDigitMod.java
 - + SecretKey.java
 - + SecretKeyGuesser.java

2.2. Program Structure (Source Code)

- Github Repo URL:

https://github.com/tonydiggory/RMIT_GroupProject_AlgosStructure_Group17/tree/master/SourceCode

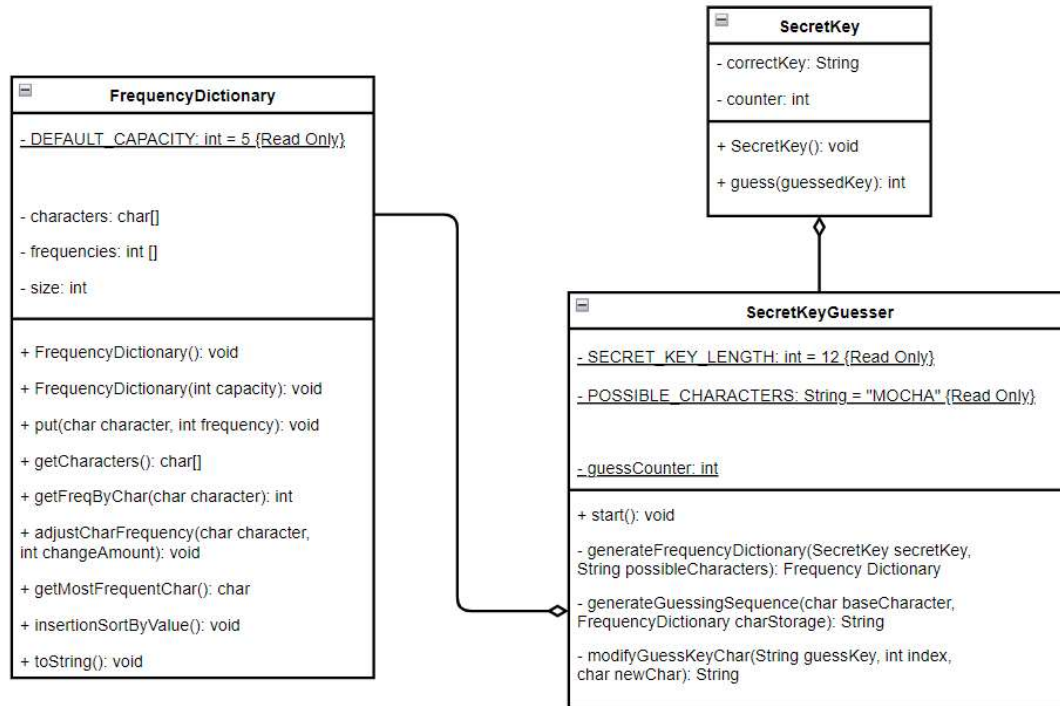


Figure 2: Class Structure

A. Relationship

- FrequencyDictionary aggregates SecretKeyGuesser
- SecretKeyGuesser aggregates SecretKey

B. Component

- **FrequencyDictionary**: This class provides storage and analysis of input character frequencies. It supports building up and querying this frequency data to help inform smarter guesses.

- Attributes:
 - **characters**: Array of char primitives to hold possible chars.
 - **frequencies**: Parallel array to store frequency count of chars.
 - **size**: Integer variable to track number of chars stored and initialised to 0 when instantiated.
- Methods:

- **FrequencyDictionary()**: to create a FrequencyDictionary with the default capacity.
- **FrequencyDictionary(int capacity)**: to create a Frequency with the predefined capacity.
- **put(char character, int frequency)**: to insert passed character and associated frequency count into storage arrays.
- **getCharacters()**: to return a char array containing all stored characters.
- **getFreqByChar(char Character)**: to return associated frequency if character found.
- **adjustCharFrequency(char character, int changeAmount)**: to increase associated frequency by changeAmount
- **getMostFrequentChar()**: to identify maximum frequency count
- **insertionSortByValue()**: to sort the FrequencyDictionary in descending order based on frequency using Insertion Sort algorithm.
- **toString()**: to return a String that represents the FrequencyDictionary.

- **SecretKey**: This class represents the secret key being guessed. It can only be queried to check guesses but otherwise does not reveal any information about the actual key.

- Attributes:
 - **correctKey**: a 12-character String variable to store actual secret key and populated with real secret key at runtime.
 - **counter**: Integer variable to store number of guesses and set to 0 on start.
- Methods:
 - **guess()**: to return the count of matched positions between the guessed key and the correct key.
 - Scan through characters checking for matches.
 - Counter tracking guess count.
 - Returns number of matching characters (0-12).

- **SecretKeyGuesser**: This class is used to guess the secret key

- Attributes:
 - **SECRET_KEY_LENGTH**:
 - Constant integer set to 12
 - Determines length of the secret key
 - **POSSIBLE_CHARACTERS**:
 - String constant with value "MOCHA"
 - Holds pool of 5 possible characters

- **guessCounter:**
 - Integer variable to store number of guesses used
 - Initialised to 0 when class start
- **Methods:**
 - **start():** to start the process of finding the secret key
 - Instantiates a **"SecretKey"** object to represent the unknown key.
 - The function then calls **"generateCharStorage()"**, using the **"SecretKey"** and the possible characters as input.
 - After **"generateCharStorage()"** using **"FrequencyDictionary"**, the dictionary is then analysed to find the most used character for the **"Base GuessKey Generating"** process.
 - The **"start()"** method then enters a loop where it continuously tries to guess the key.
 - On each loop, **"generateGuessingSequence()"** is called to get the best order for the examining position based on characters' remaining usages and impacts.
 - Inputs characters into the guess string using **"modifyGuessKeyChar()"**.
 - Call the **"guess()"** method to check the updated key with the secret key.
 - Repeat this until the key is fully matched.
 - **generateCharStorage(SecretKey secretKey, String possibleCharacters):** to build the character storage.
 - It take the secret key and possible character as input
 - Loops through each possible character repeating that single char throughout the length of the actual key.
 - These guess keys are passed to the secret key's **"guess()"** method and record frequency
 - These frequency counts for each char are stored into a **"FrequencyDictionary"**, which is used in the future.
 - **generateGuessingSequence(char baseCharacter, FrequencyDictionary charStorage):** to determine the best guessing order for current position.
 - This takes the base character and FrequencyDictionary as input.
 - It gets the sorted characters from the dictionary.
 - It returns a string with characters, which are not the base one and do not run out of usage, in descending order.

- **modifyGuessKeyChar(String guessKey, int index, char newChar):** to replace a character at a specified position in a guessKey with a new character.

III. Development Process

In our current group project for the Algorithms and Analysis course, the objective is to develop a program that unveils the secret key, which is a 12-character combination consisting of the letters 'M', 'O', 'C', 'H', and 'A'. At the same time, we are provided with a guessing system that can return the number of correct values in the hidden sequence. Although the deadline is approximately a month, our group delved into the problem as soon as the countdown clock began. Following a thorough examination and discussion of the assignment specifics within the team, each member of the Group 17 had been allocated a week to formulate solutions, write the corresponding pseudocodes, as well as explore potential insights embedded in the exercise.

During this timeframe, numerous approaches and algorithms were proposed. Unfortunately, a majority of them were extremely sophisticated for practical implementation, which consumed significant spaces and time for execution while providing no substantial enhancement in the guess usage over standard Brute Force. However, there were some notable and highly effective ideas emerged from our efforts, including:

- **Enhanced Brute Force using Character Frequencies:** Instead of trying all combinations of the 5 letters 'M', 'O', 'C', 'H' and 'A', this method reduces a sizable amount of wrong test cases by utilising the first 5 guesses to calculate the frequency of each letter in the secret sequence and then making guesses based on potential permutations.
 - + This technique provides us with an opportunity to be familiar with tracking letter frequencies and selecting an effective base guess key.
 - + URL:
https://github.com/tonydiggory/RMIT_GroupProject_AlgosStructure_Group17/blob/master/Evaluation/EnhancedBruteForce.java
- **Digit-by-Digit Modification:** This approach is designed by applying the Decrease and Conquer paradigm. Instead of guessing the entire key, this solution leverages the existing guessing mechanism and reinforces each position from left to right. After modifying and correcting each of the 12 characters, the remaining result constitutes the secret key.

+ URL:

https://github.com/tonydiggory/RMIT_GroupProject_AlgosStructure_Group17/blob/master/Evaluation/DigitByDigitMod.java

After the first week, our team members continued to explore fresh options actively. Simultaneously, there was an increased emphasis on collaborative discussions to identify and integrate the most optimal ideas. Along with it, we started the programming phase and then worked on perfecting the final solution.

Lastly, we decided to tackle the problem of discovering the correct key sequence using a depth-first guessing approach, which is modifying each character individually from left to right until all 12 positions are corrected. Alongside with this method, a data structure and several optimization algorithms were employed, including Dictionary, Efficient Base GuessKey Selection, Character Frequency Tracking, Guessing Sequence Enhancement, and Early Termination. The upcoming section (**IV. Data Structures and Algorithms**) will provide a clear explanation of these strategies' details.

IV. Data Structures and Algorithms

1. Data Structure - Frequency Dictionary

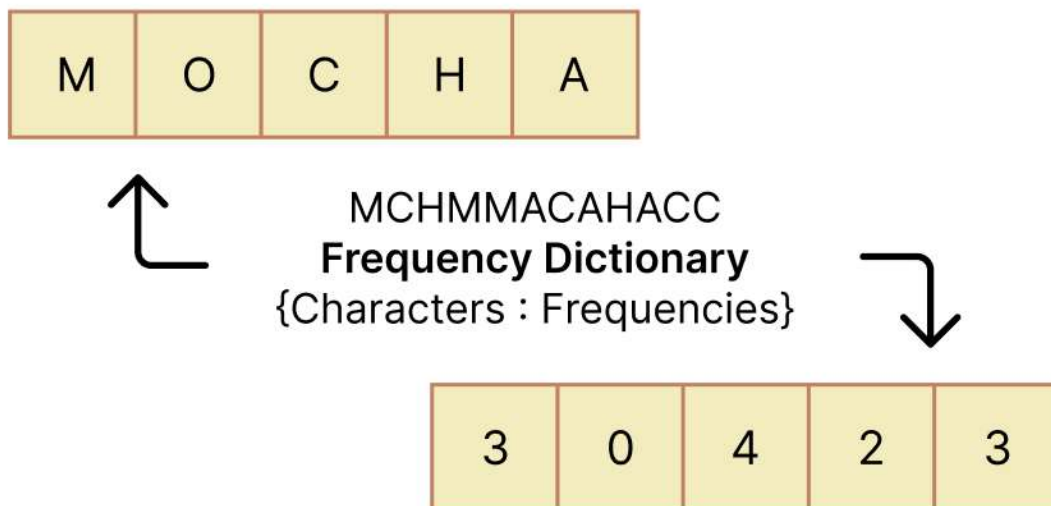
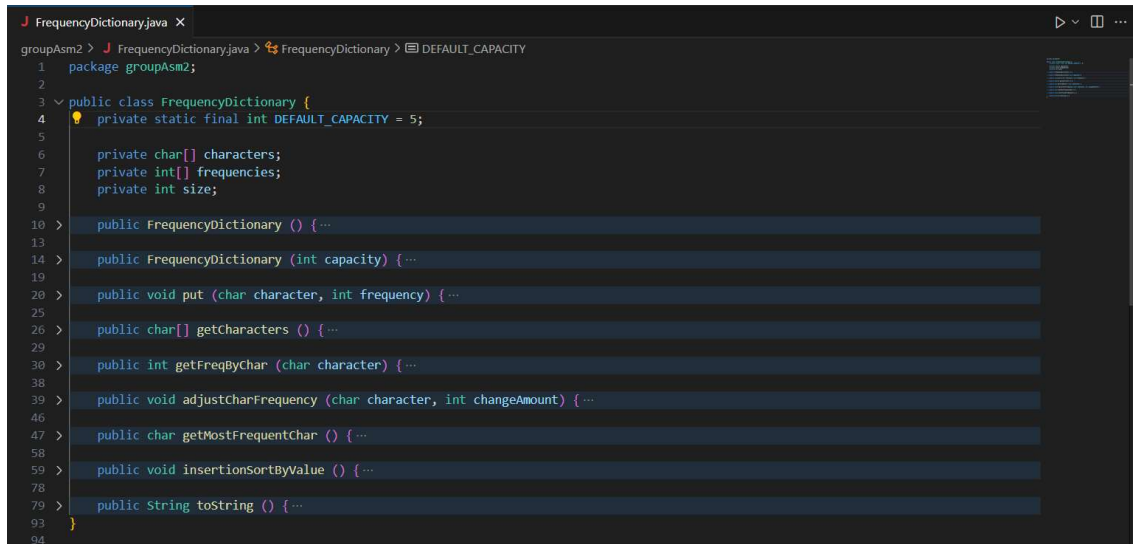


Figure 3: Frequency Dictionary

In our Secret Key Guesser program, we have implemented a Frequency Dictionary using an array-based structure. This design allows us to conveniently record

potential characters along with their corresponding frequencies within the hidden sequence.



```
FrequencyDictionary.java
groupAsm2 > J FrequencyDictionary.java > FrequencyDictionary > DEFAULT_CAPACITY
1 package groupAsm2;
2
3 public class FrequencyDictionary {
4     private static final int DEFAULT_CAPACITY = 5;
5
6     private char[] characters;
7     private int[] frequencies;
8     private int size;
9
10    public FrequencyDictionary () { ...
13
14    public FrequencyDictionary (int capacity) { ...
19
20    public void put (char character, int frequency) { ...
25
26    public char[] getCharacters () { ...
29
30    public int getFreqByChar (char character) { ...
38
39    public void adjustCharFrequency (char character, int changeAmount) { ...
46
47    public char getMostFrequentChar () { ...
58
59    public void insertionSortByValue () { ...
78
79    public String toString () { ...
93
94 }
```

Figure 4: FrequencyDictionary Implementation

Specifically, our Dictionary stores information in pairs utilising two arrays, one containing characters (char) while the other holding integers (int). In this data structure, each character and its associated frequency have the same index in their respective arrays. This idea not only simplifies the implementation but also enhances accessibility as well as overall code readability. However, there exists a limitation as we need to predefine the capability of the Frequency Dictionary before use. Therefore, in our efforts to minimise errors during code execution, we have added two additional attributes to the class: DEFAULT_CAPACITY (set to 5, equivalent to the number of unique letters in “MOCHA”) and size (to keep track of the current Dictionary size), as well as, employed function overloading in the constructor to handle various scenarios effectively. Furthermore, there are additional methods to assist in handling data within the Dictionary as well as to support the secret key search process, including put(char character, int frequency), getCharacters(), getFreqByChar(char character), adjustCharFrequency(char character, int changeAmount), getMostFrequentChar(), insertionSortByValue(), and toString() (detailed content in section II.2.2).

In our SecretKeyGuesser program, Frequency Dictionary is used to initialise and manage the Character Storage (charStorage), where the quantity of each letter yet to appear for the completion of the Secret Key is stored. To accomplish this, we sequentially employ 4 Guess Keys, each containing 12 identical characters, namely ‘M’, ‘O’, ‘C’, and ‘H’. After using each Guess Keys with the provided guessing system, the returned value indicates the frequency of the corresponding letter. Besides, the “frequencyCounter” variable is used to reduce the number of guesses when

calculating the frequency of the last character (letter 'A') and to enhance convenience in managing the total character count. Specifically, after each iteration, the program immediately calls the "put(character, frequency)" method to store the current pair value in the "charStorage" object. Simultaneously, the "frequencyCounter" variable is updated with the frequency of the current character. Finally, the charStorage initialization process ends after all possible letters have been completed or 12 characters have been counted (frequencyCounter = 12).

generateCharStorage (Frequency Dictionary)

SecretKey: MCHMMACAHACC

M	M	M	M	M	M	M	M	M	M	M	M
---	---	---	---	---	---	---	---	---	---	---	---

GuessKey - M CorrectPositions: 3 ⇒ Frequency of letter 'M' = 3
 ⇒ FrequencyCounter = 3

⇒ charStorage.put(M, 3)

O	O	O	O	O	O	O	O	O	O	O	O
---	---	---	---	---	---	---	---	---	---	---	---

GuessKey - O CorrectPositions: 0 ⇒ Frequency of letter 'O' = 0
 ⇒ FrequencyCounter = 3

⇒ charStorage.put(O, 0)

C	C	C	C	C	C	C	C	C	C	C	C
---	---	---	---	---	---	---	---	---	---	---	---

GuessKey - C CorrectPositions: 4 ⇒ Frequency of letter 'C' = 4
 ⇒ FrequencyCounter = 7

⇒ charStorage.put(C, 4)

H	H	H	H	H	H	H	H	H	H	H	H
---	---	---	---	---	---	---	---	---	---	---	---

GuessKey - H CorrectPositions: 2 ⇒ Frequency of letter 'H' = 2
 ⇒ FrequencyCounter = 9

⇒ charStorage.put(H, 2)

⇒ Frequency of letter 'A' = SecretKey.length - frequencies of other letters
 = 12 - FrequencyCounter = 12 - 9 = 3

⇒ charStorage.put(A, 3)

Figure 5: Frequency Calculation

2. Algorithms

2.1. Main Algorithm - Digit By Digit Optimization

The Digit-by-Digit Modification (or Depth-first Guessing procedure) is an algorithm that systematically verifies the accuracy for each position in the Base GuessKey, proceeding sequentially from left to right until the Secret Key is found. This process carefully examines and refines each character, fostering a thorough exploration of options in the pursuit of revealing the hidden secret.

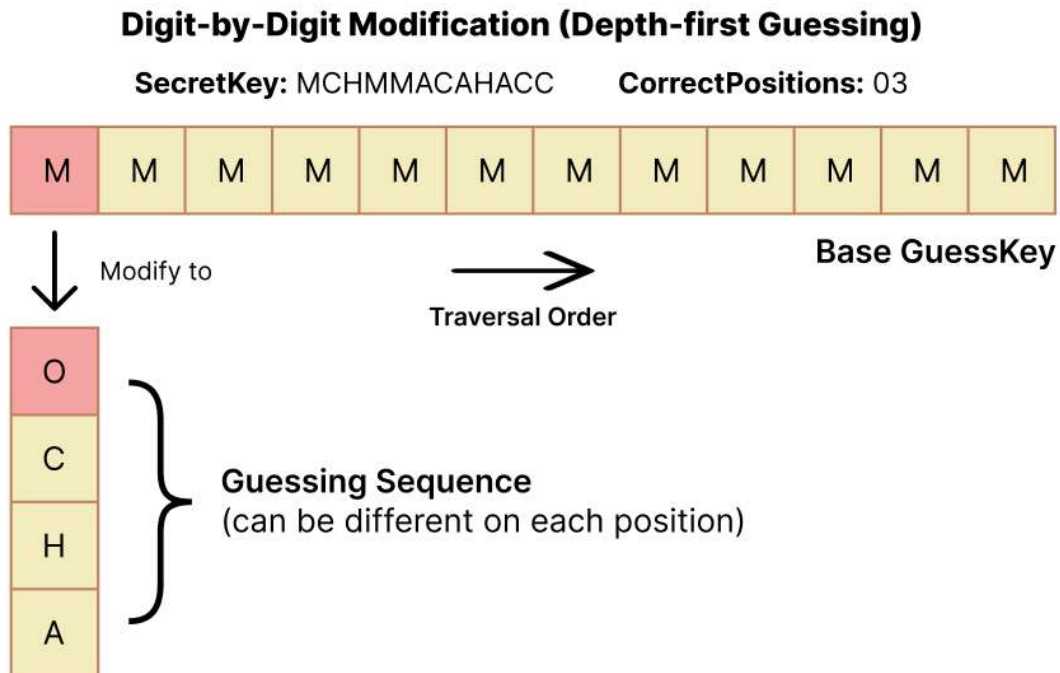


Figure 6: Depth-first Guessing Procedure (Sample) (Step 1)

Initially, the program generates a Base GuessKey and determines the number of correct positions using the given guessing system. Subsequently, before visiting each point, a guessing sequence is produced to facilitate adjustments more conveniently. Then, the current examining character is replaced by elements in the guessing sequence until a correct result is found. However, in our effort to optimise the program and reduce the number of guesses, various algorithms are applied to the guessing sequence; therefore, it may be different at each position. Besides, there are 3 scenarios when modifying a character: the number of correct positions decreases, remains unchanged, or increases.

Digit-by-Digit Modification (Depth-first Guessing)

SecretKey: MCHMMACAHACC

CorrectPositions: 02

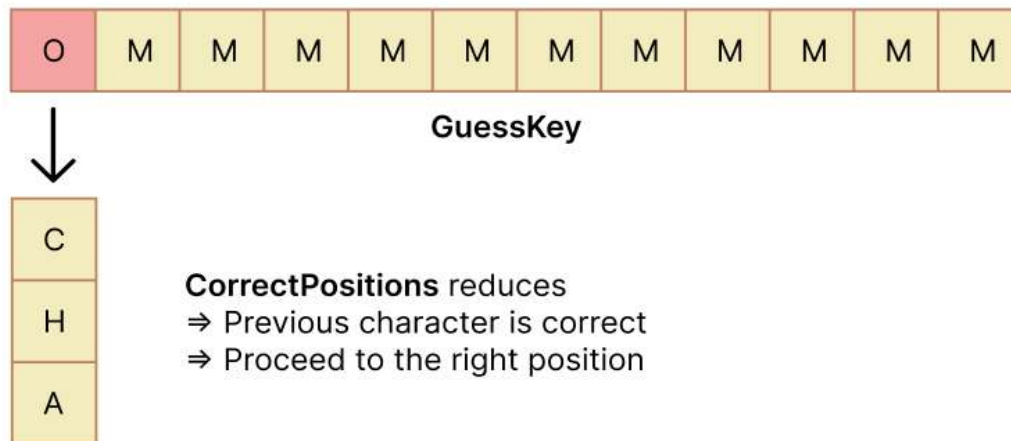


Figure 7: Depth-first Guessing Procedure (Sample) (Step 2)

In the first case, after updating the current character with a value from the guessingSequence, the count of accurate positions in the current GuessKey decreases by 1. This indicates that the character before change was correct, leading to the restoration of the current position to its previous one. Then, the program will advance to the next slot, located to the right following the designated traversal order.

Digit-by-Digit Modification (Depth-first Guessing)

SecretKey: MCHMMACAHACC

CorrectPositions: 03

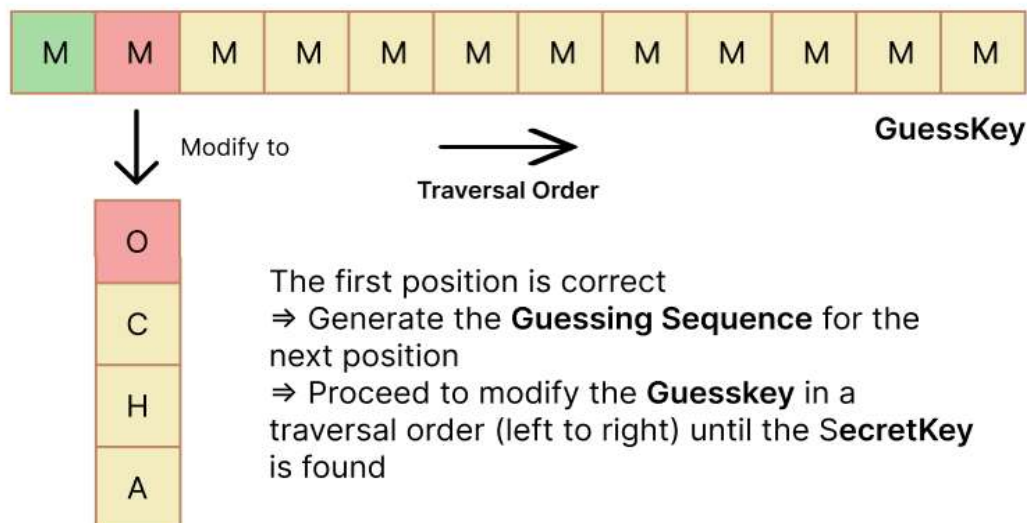


Figure 8: Depth-first Guessing Procedure (Sample) (Step 3)

Additionally, another Guessing Sequence is generated for the next position before any new modifications are made.

Digit-by-Digit Modification (Depth-first Guessing)

SecretKey: MCHMMACAHACC

CorrectPositions: 03

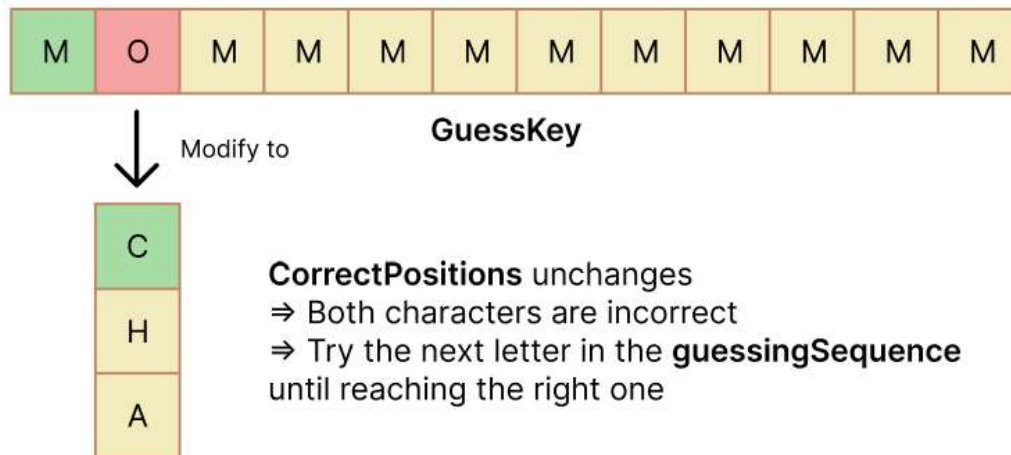


Figure 9: Depth-first Guessing Procedure (Sample) (Step 4)

The next situation happens when the CorrectPositions variable remains unchanged after a change. This suggests that both characters are incorrect so the program will keep trying next elements in the guessingSequence until it reaches the right one.

Digit-by-Digit Modification (Depth-first Guessing)

SecretKey: MCHMMACAHACC

CorrectPositions: 04

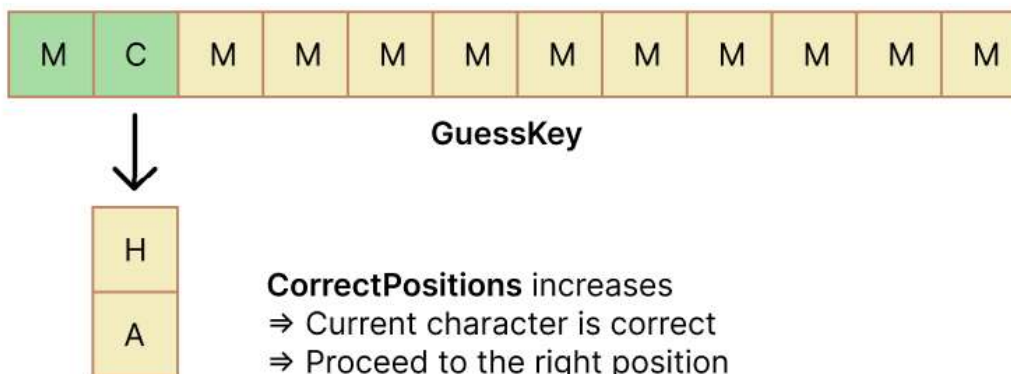


Figure 10: Depth-first Guessing Procedure (Sample) (Step 5)

The last scenario exists when the program witnesses an increase in the number of correct positions in the GuessKey. This means that the new character now matches with the current position, and the program will skip the next values in the guessingSequence to proceed to the subsequent position on the right.

Digit-by-Digit Modification (Depth-first Guessing)

SecretKey: MCHMMACAHACC

CorrectPositions: 12

M	C	H	M	M	A	C	A	H	A	C	C
---	---	---	---	---	---	---	---	---	---	---	---

SecretKey Found



After modifying all 12 positions or
achieving 12 **CorrectPositions**:
⇒ **SecretKey** found!

C
H
A

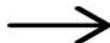
Figure 11: Depth-first Guessing Procedure (Sample) (Last Step)

2.2. Sorting Algorithm

InsertionSortByFreq (Frequency Dictionary)

SecretKey: MCHMMACAHACC

M	O	C	H	A
3	0	4	2	3



C	M	A	H	O
4	3	3	2	0

CharStorage Sorted CharStorage

Figure 12: Insertion Sort

An Insertion Sort algorithm is implemented in our program to sort the Frequency Dictionary in descending order. Specifically, it rearranges the values in the “frequencies” array but with each order swap, the corresponding positions in the “characters” array will also be updated. Considering the current problem’s constraint of only 5 unique letters, Insertion Sort appears to be one of the most effective approaches for handling small datasets. Besides, since it simulates natural behaviours, it simplifies the complexity of implementation and enhances readability. Finally, we choose to use Insertion Sort because CharStorage is frequently sorted, and this algorithm does not consume much time for arrays that are partially ordered.

2.3. Optimization Algorithms

A. Efficient Base GuessKey Selection

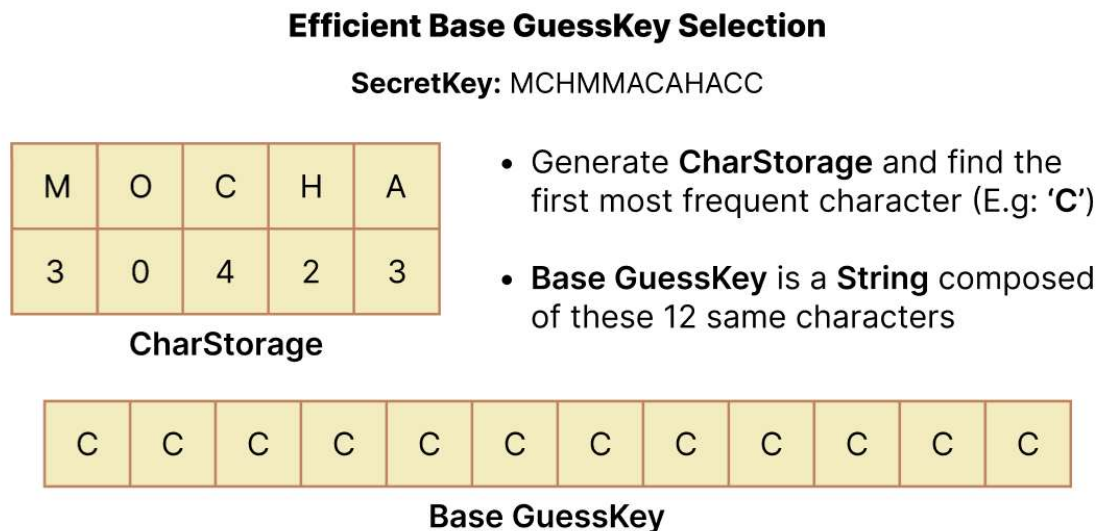


Figure 13: Efficient Base GuessKey Selection

Efficient Base GuessKey Selection is an algorithm designed to minimise the number of modifications because checking for a correct position consumes 1 guess while updating until correct is up to 4. Initially, after invoking the "generateCharStorage()" function, the program identifies the first most frequent character, such as the letter 'C' in the above example. Then, a Base GuessKey is created by repeating this character 12 times (based on the length of the Secret Key).

B. Guessing Sequence Enhancement

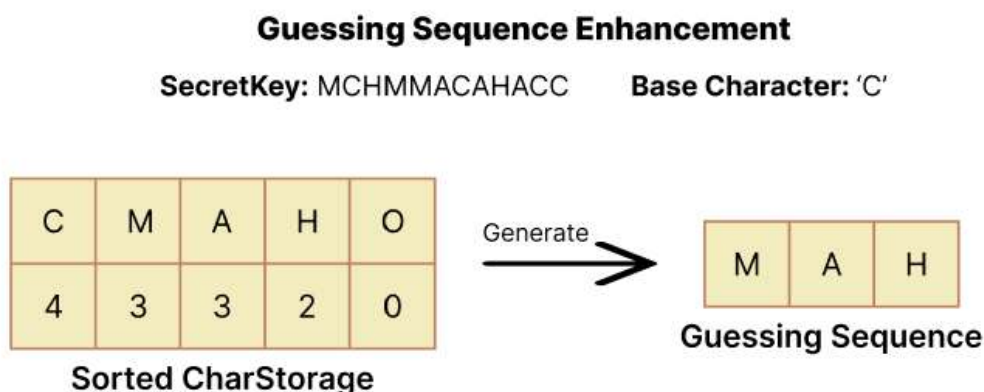


Figure 14: Guessing Sequence Enhancement

Guessing Sequence Enhancement is an algorithm that adapts Character Frequency Tracking technique to reduce the guess usage in our SecretKeyGuesser program. To be more precise, letters with higher remaining uses are more likely to appear at the next position. Therefore, before generating a guessingSequence, the system

rearranges CharStorage by value in descending order and then obtains the “characters” array, which closely resembles the GuessingSequence for the current position. Additionally, we can avoid unnecessary guesses by excluding the base character and letters with a value of 0. After a successful modification, the corresponding character’s value updates, triggering a sort of charStorage. The images below will vividly illustrate the process of optimising the guessingSequence.

Initially, the program generates the GuessingSequence based on the current order of each character in the sorted charStorage. Then, the base character (letter ‘C’) and any characters with 0 remaining uses (letter ‘O’) will be eliminated. As a result, in the example above, the guessingSequence for the first position would contain ‘M’, ‘A’, and ‘H’.

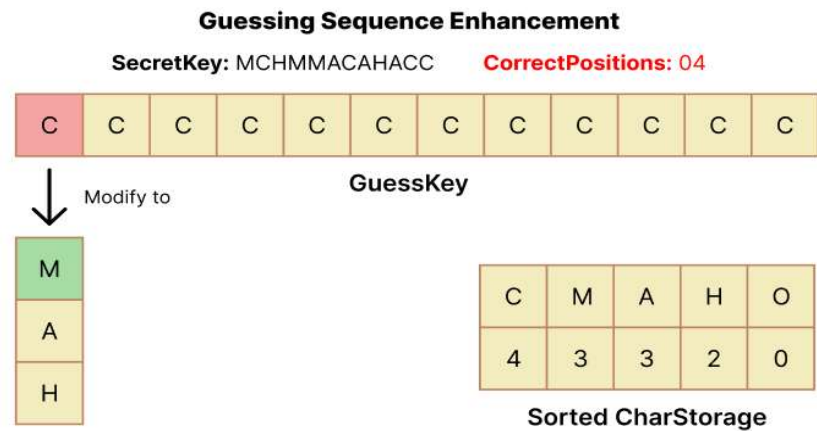


Figure 15: Guessing Sequence Enhancement Procedure (Sample) (Step 1)

Next, the first character will be replaced in turn until the program detects a change in the “CorrectPositions” variable. Fortunately, in this instance, right after the transformation with the letter ‘M’, the first position has been accurately identified; therefore, the system will move on to the next one.

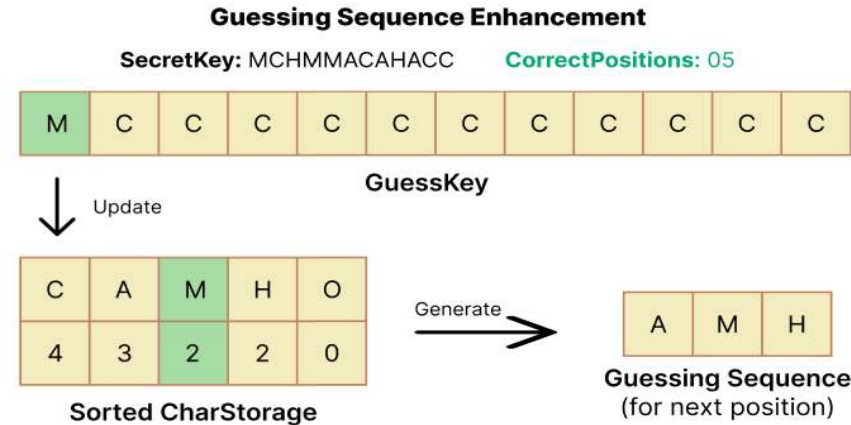


Figure 16: Guessing Sequence Enhancement Procedure (Sample) (Step 2)

Before starting to modify the next character, the charStorage gets updated. In the given example, the value of letter 'M' decreases by 1, leaving 2 uses. Subsequently, a new GuessingSequence is generated - this time 'A', 'M', and 'H' for the second position. The program will iterate through this process for all locations until the SecretKey is found.

C. Early Termination

“Early Termination” method is designed to end the current function as soon as its objective has been reached. This helps eliminate unnecessary procedures, improving performance in terms of both time and the number of guesses. There are 3 cases utilising this approach and this will be discussed in depth below, along with visualisations.

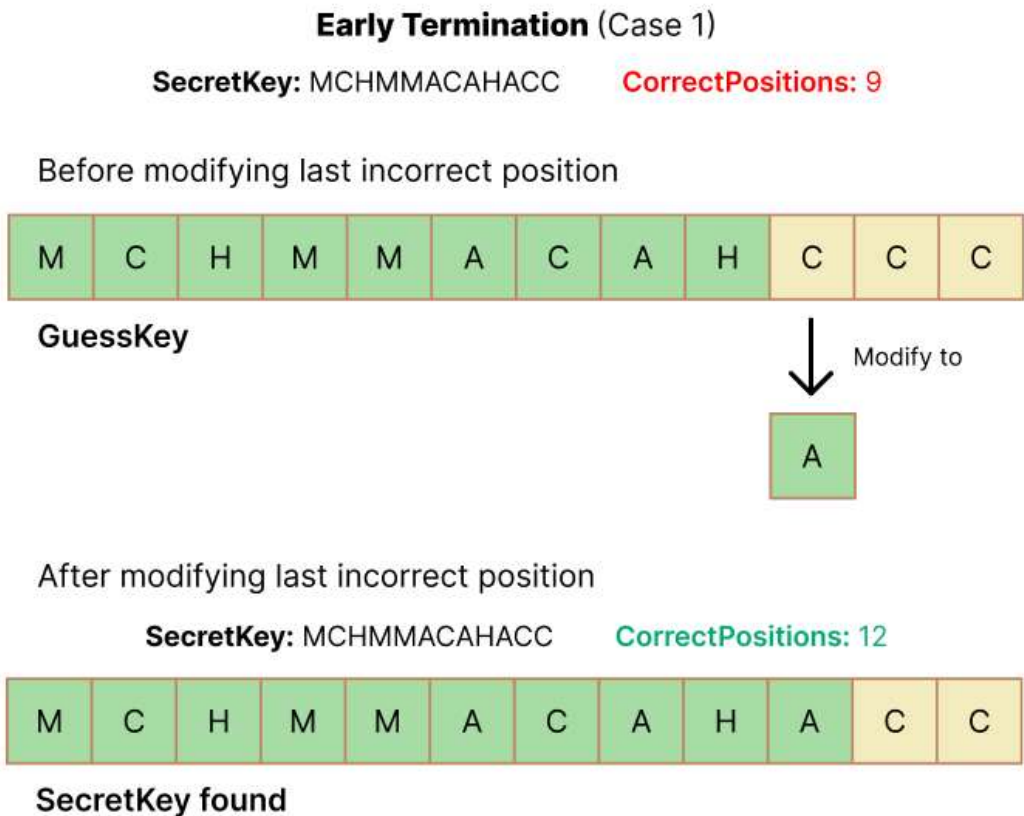
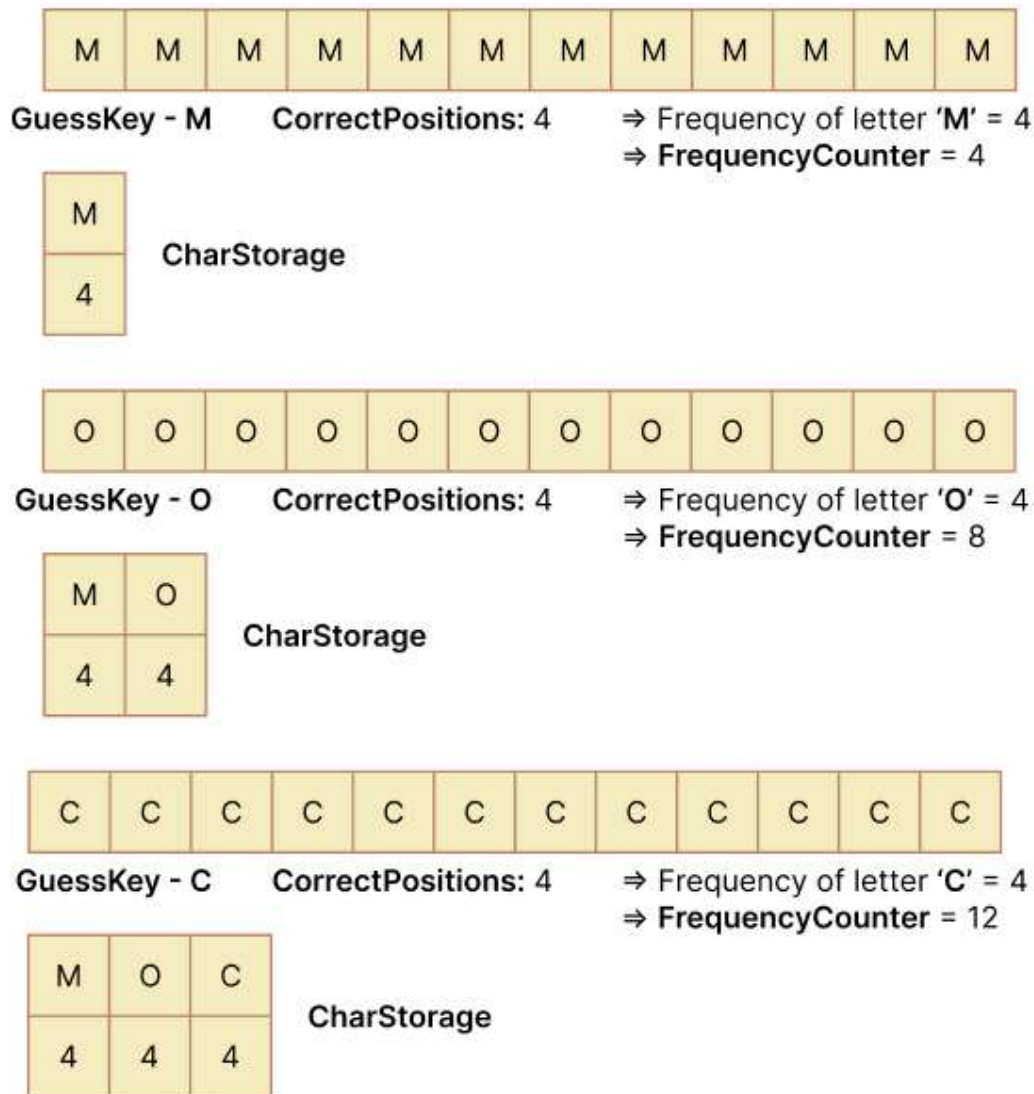


Figure 17: Early Termination Case 1 (Modifying the last incorrect position)

The first scenario happens when the application is modifying the GuessKey. Instead of traversing the entire key, the program will instantly terminate after fixing the final invalid position by checking the “correctPositions” variable (correctPosition = 12).

Early Termination (Case 2)
SecretKey: MOCMOCMOCMOC



⇒ The function **generateCharStorage()** terminated early since we have found all 12 characters

Figure 18: Early Termination Case 2 (Early verify all 12 characters in secretKey)

Meanwhile, the second condition occurs during the generation of character storage. Rather than attempting all letters, the application counts the number of potential characters using the "FrequencyCounter" variable. The function will return the charStorage once it reaches a count of 12, which is the length of the SecretKey. Therefore, the number of guesses can be reduced by up to 3.

Early Termination (Case 3)

SecretKey: 000000000000

O	M
12	0

CharStorage

⇒ SecretKey: 000000000000

Figure 19: Early Termination Case 3 (secretKey consisting of 12 identical characters)

Finally, in the third situation, the application ends right after generating the initial character storage, as all 12 characters of the SecretKey have been discovered and are identical. In the example above, after 2 guesses, the system found the secretKey, which consists of 12 letters 'O' so it directly printed out the result.

V. Complexity Analysis

1. Abbreviations

- Let **m** be the number of possible characters (In this problem, $m = 4$).
- Let **n** be the length of the secret key (In this problem, $n = 12$).
- Let **c** be the constant computational cost.

2. Workflow

01. Startup

- SecretKeyGuesser instantiates SecretKey
- Initialises constants

02. Generate charStorage

- Initiates a new FrequencyDictionary
- Iterates over POSSIBLE_CHARS
- Constructs guess keys using each char
- Passes to SecretKey, stores frequencies

03. Determine Base Char

- Gets most frequent char from dictionary

04. Generate Initial Guess

- Constructs string with base char repeated

05. Main Guessing Loop

- a. For each index in key length
 - i. Get the guessing sequence based on the character's usage and frequency.
 - ii. Modify guess string with the guessing sequence
 - iii. Check against SecretKey, store the one that has highest result

06. Finalise

- a. Output the final guessKey after successfully finding all 12 characters or exhaustion

3. Analysis

- Space Complexity: grows linearly with both m and $n \Rightarrow O(m + n)$
 - Time Complexity:
 - generateCharStorage():
 - Using a for-loop iterating through $m - 1$ characters to try the guessKey:
 - `"".repeat(SECRET_KEY_LENGTH): O(n)`
 - $\Rightarrow O(n \times (m - 1))$
 - generateGuessingSequence()
 - Using Insertion Sort to rearrange the charStorage: $O(m^2)$
 - Using a for-loop iterating through m characters to keep the suitable characters: $O(m)$
 $\Rightarrow O(m^2)$
 - modifyGuessKeyChar(): $O(1)$
 - start():
 - generateCharStorage(): $O(n \times (m - 1))$
 - getMostFrequentChar(): $O(m)$
 - getFreqByChar(character): $O(m)$
 - Outer loop (`int i = 0; i < SECRET_KEY_LENGTH; i++`):
 - Inner loop (`int j = 0; j < guessingSequence.length(); j++`):
 - + `secretKey.guess(guessKey): O(n)`
 - $\Rightarrow O(n \times m \times n) = O(m \times n^2)$
- \Rightarrow Time Complexity: $O(m \times n^2)$

- Guess Count Complexity:

Note: Verifying a position consumes 1 guess while correction that position can cost up to 4 guesses. => checkGuess = $O(1)$ and modifyGuess = $O(m-1)$

- Best Case: Given the secretKey which is "MMMMMMMMMMMM", the program takes only 1 guess to both generate the character storage as well as find the secret key. This is the combination of Early Termination case 1 and 3. => $O(1)$
- Worst Case: The worst case happens when:
 - The number of modifications is max. Therefore, the most frequent character has to be smallest in comparison with other test cases
=> frequencies of most frequent characters = $\text{ceiling}(n / m) = 3$
 - The last 3 position requires only $(1 + 2 + 3)$ guesses because of character frequency tracking
=> Guess Complexity: $\text{max}(\text{generateCharStorage}()) + (3 \times \text{checkGuess}) + 8 \times (\text{modifyGuess}) + 3 + 2 + 1 = O(m - 1) + O(n/m) \times O(1) + O(n - n/m - m - 1) \times O(m - 1) + O((m-1)m/2) = 4 + 3 + 6 \times 4 + 4 + 3 + 2 + 1 = 41$

VI. Evaluation

1. Testing Approach

The testing approach involves the creation of a dataset containing 2007 random secret keys. The procedure includes systematically applying various algorithms to each secret key in the dataset, aiming to unveil the hidden sequence within the current test. At the same time, by employing different algorithms, the approach seeks to evaluate and compare their effectiveness in uncovering hidden sequences.

2. Correctness Evaluation

Our correctness is 100% due to two reasons:

- By theory: Based on our design, the code solves the problem by running through it from left to right and through each character each. It can't miss or skip any character so that the chance for it to fail is 0%.
- By real test: These approach has been tested through a dataset that is about 2307 random secret keys, and the result is 100% correct, which also proves that our theory is correct.

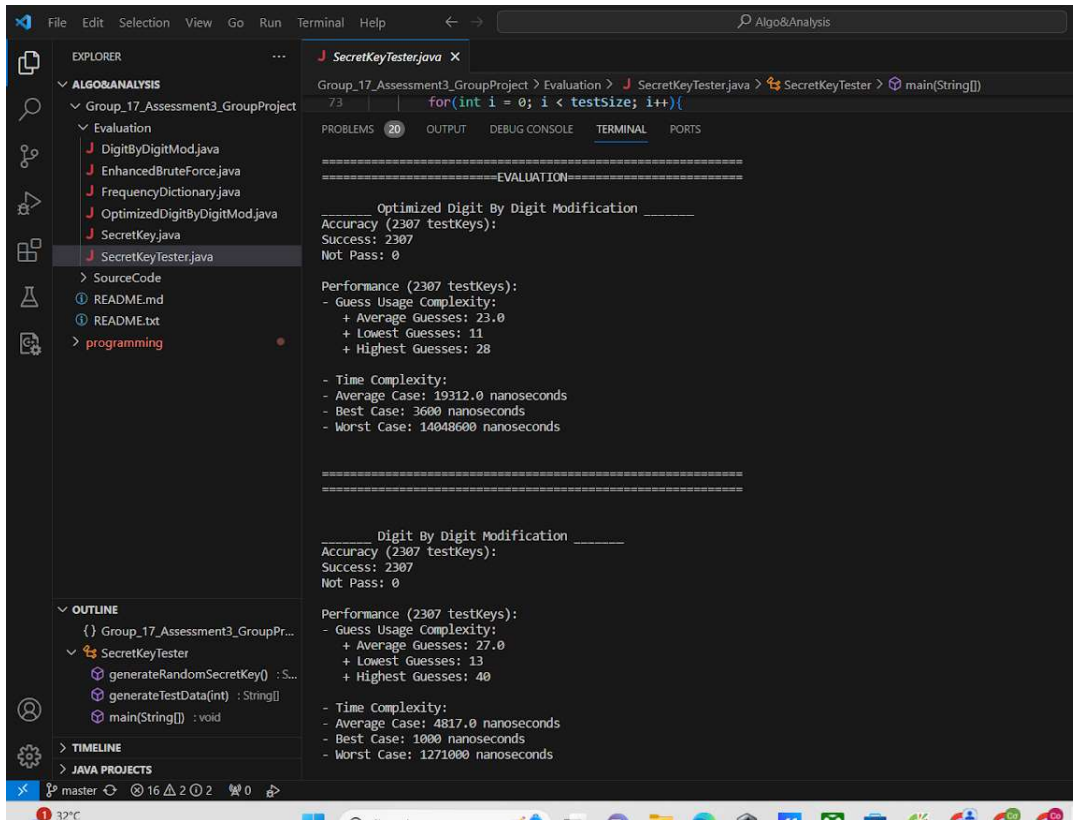


Figure 20: Evaluation

3. Efficiency Evaluation

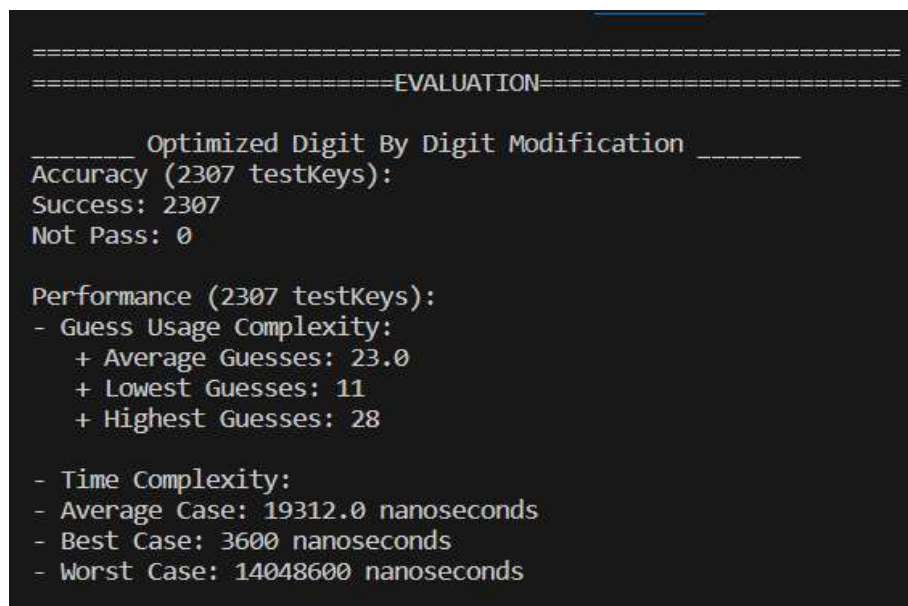


Figure 21: Optimised Digit By Digit Modification (Evaluation)


```
----- Digit By Digit Modification -----
Accuracy (2307 testKeys):
Success: 2307
Not Pass: 0

Performance (2307 testKeys):
- Guess Usage Complexity:
  + Average Guesses: 27.0
  + Lowest Guesses: 13
  + Highest Guesses: 40

- Time Complexity:
- Average Case: 4817.0 nanoseconds
- Best Case: 1000 nanoseconds
- Worst Case: 1271000 nanoseconds
```

Figure 22: Digit By Digit Modification (Evaluation)

Brute force: Even though we tried many ideas to enchant this approach, brute force seems unable to meet our requirement. By skipping those String keys that do not meet the standar and minimar the guess space by splitting the key into smaller parts, we reduced the number of guesses needed, but it still be out weighted by the other two.

Digit by Digit Modification (DbDM): Our theory points out that it will need around 48 guesses in the worst case in this approach, it is way better than using brute force. The real test outcome also proves that with highest guess stay at 48, average at 23 guesses.

Optimised Digit by Digit Modification: This approach is based on the DbDM, it has a slightly better outcome but bad in time complexity, as we can see in the test even though it not much of the change in average and lowest guess, but the highest guess make a significant decrease which is the target of this project.

In conclusion, we choose the Optimised Digit by Digit Modification approach to tackle the secretKey problem, because the requirement here is to have the guess count as low as possible.

References

[1] "Group Project (REAL)." rmit.instructure.com. Accessed 13/12/2003. [Online]. Available: <https://rmit.instructure.com/courses/131308/assignments/883729>