Course: COSC2658

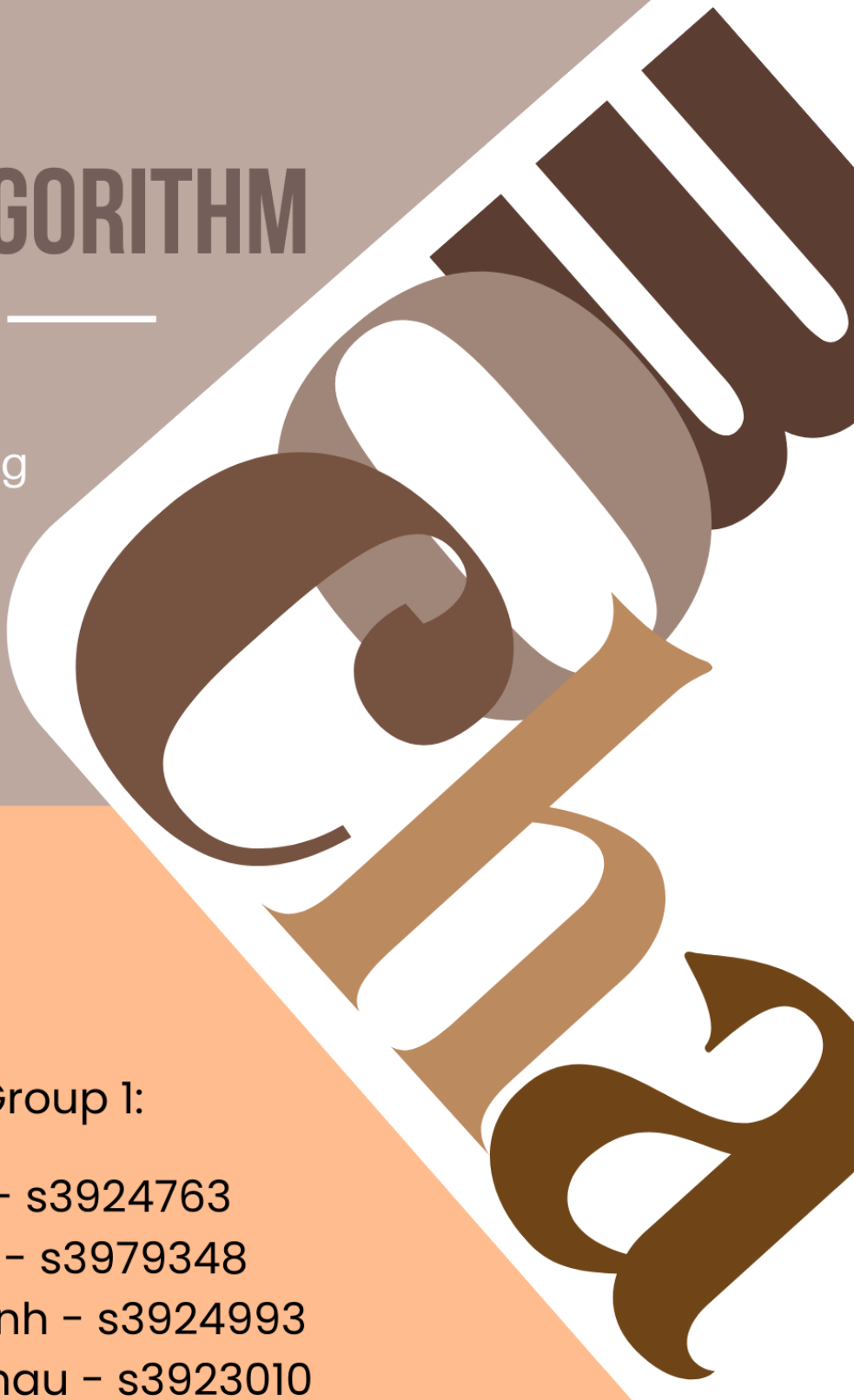
RMIT UNIVERSITY

PASSWORD ALGORITHM

Group Project

Lecturer
Ling Huo Chong

Dec 22nd, 2023



Prepared By Group 1:

Huynh Nhat Anh - s3924763
Nguyen Son Tung - s3979348
Nguyen Le Thuc Quynh - s3924993
Nguyen Dinh Minh Chau - s3923010

Table of Contents

1. Overview and Motivation

Password Algorithm is a program developed to provide a solution for the problem statement in course COSC2856 [1]. Regarding the problem background, the program needs to utilize a magic function “guess” to determine a forgotten password. The constraints are that this password has 12 characters; each of them is made up of M, O, C, H, and A which is the inspiration for that password.

The source code[2] of this problem was given as an abstract program while using brute force technique [3] which is widely used and get well-known in cryptography with the term “brute-force attack”. However, with the above constraints, it would take 5^{12} guesses. It is more than 240 million trials, which means we will get blocked before finding our password if the system locks your IP [4]. Subsequently, you wish someone would give you another magic function but no, the course is over. Therefore, this guess function should be called as small as possible.

Password Algorithm took our team a week to ideate and three weeks to implement. It was encouraged to apply course delivered concepts which enhance our skills in problem solving and practicing new methodologies. Our algorithm not only aims to minimise the number guessing but also the excution time as if the new password could be longer.

2. Software Design

2.1. Directory Tree

```
└──src
    └──Password
            │       SecretKey.java
            │       SecretKeyGuesser.java
            │
            └──MyADT
                    │       CustomHashMap.java
                    │
                    ├──Element
                    │           Chain.java
                    │           Node.java
                    │           Pair.java
                    │
                    └──Key
                                KeyList.java
                                KeyNode.java
                                KeySet.java
```

<3

`SecretKey.java` holds the password and the magic hacking function.

`SecretKeyGuesser.java` holds the algorithm to solve the password.

Java files in folder MyADT and its sub-folders `Element` and `Key` are all self-built ADTs.

2.2. UML Diagram

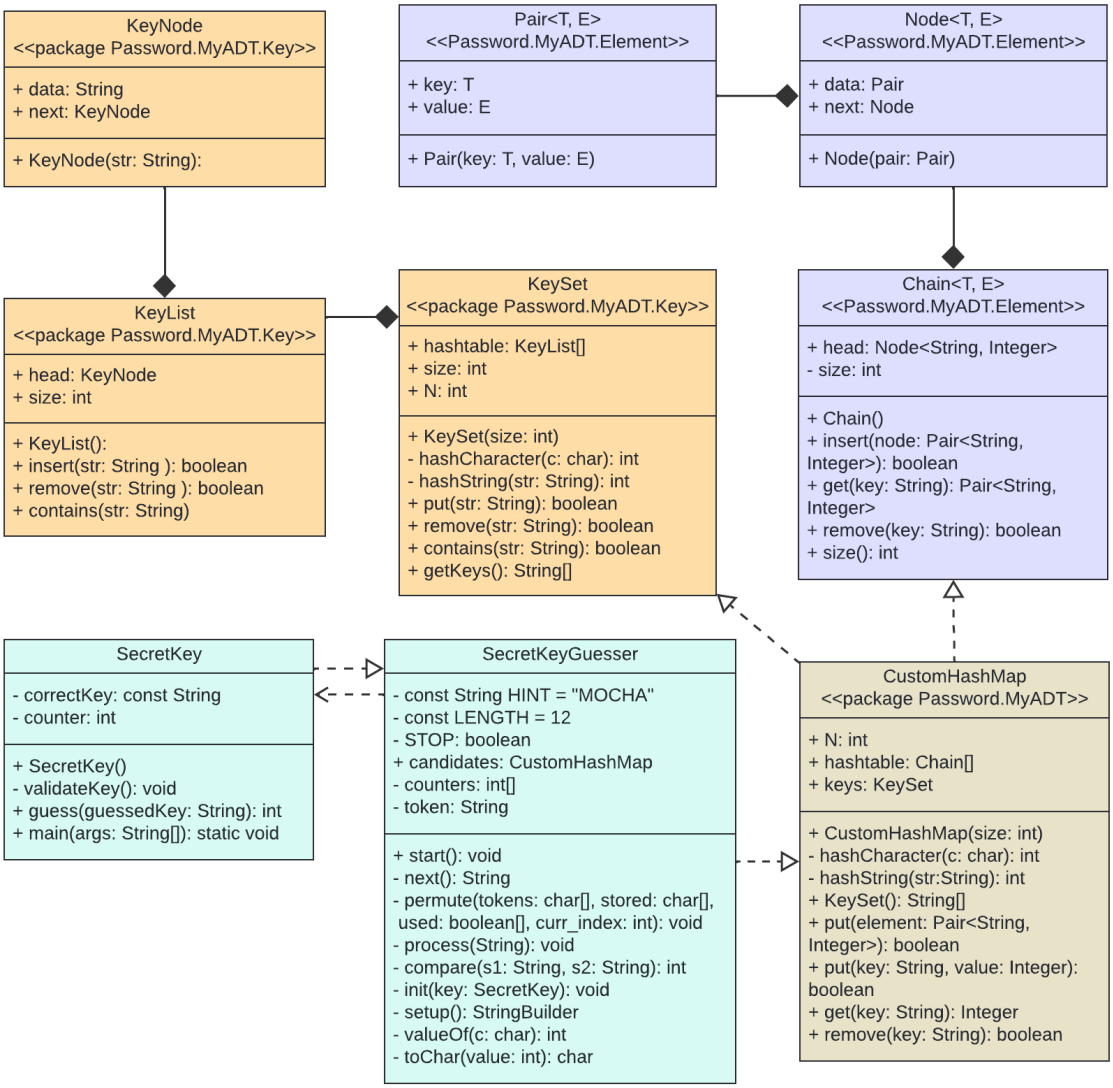


Figure 1: Password Algorithm UML Diagram

3. Abstract Data Type(s) Implementation

Due to the limitation of using concrete data structures, we have implemented our own ADTs as shown in Figure 2.

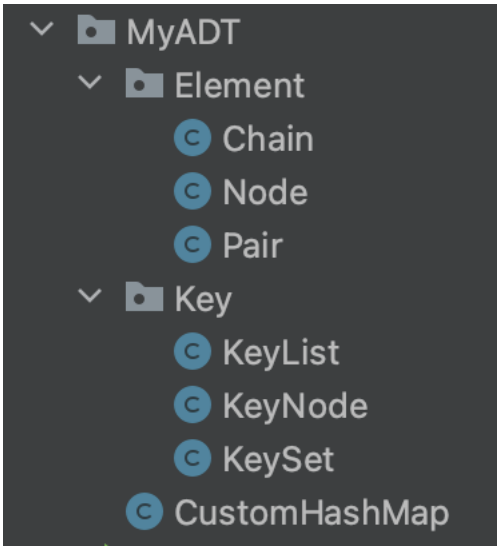


Figure 2: Password Algorithm ADTs

3.1. CustomHashMap

For the best performance of the SecretKeyGuesser, we decided that HashMap is the primary abstract data type of our algorithm. CustomHashMap.java is where the data type and its operations are designed. This blueprint is aimed to store guessed passwords (String) that are generated by the SecretKeyGuesser. Anytime a new object of CustomHashMap is created, CustomHashMap must establish the demanded size in the hashmap. Behind the scenes, the operation is dependent on the two classes Chain.java and KeySet.java. These two classes are needed to prevent hashing collisions (refer to Section 3.2, 3.3).

```
public CustomHashMap(int size){
    N = size;
    hashtable = new Chain[size];
    keys = new KeySet(size);
}
```

a. hashCharacter()

This method transforms one single character (char) to a hash value (int) at once. Then it will return that value.

b. hashString()

This method converts a string (String) into a hash value (int). The concept is that the string is deconstructed into an array of characters. After that, we use the method hashCharacter() to convert each of the characters to its hash value. Eventually, the method will return the hash value of this string.

c. keySet()

To get all the element of keys (in a String array) using the getKeys() method (refer to Section 3.3.c) and

.

d. put(Pair<String, Integer> element)

The purpose of put() is to put new elements of the Pair<String, Integer> type to the hashmap. First of all, it must establish the index of a given element by hashing the **key** of that element. Obtaining that index is essential for checking if the room of that index is empty to store a new element. If it is empty, a new chain (an object of the Chain class (refer to Section 3.2)) is created. The new element is now being processed to add to the hashtable at the index that the function has just examined. If the insertion of the element is successful, its key will be inserted into the key set also using the put(). Finally, the whole process will return a boolean value.

e. put(String key, Integer value)

This is an alternative way to insert a new element into the hashmap. The method gains a string as a key and an integer as a value of the element with the aim of forming that element. Then it passes this element into the put(Pair<String, Integer> element).

f. get(String key)

The method is utilized for getting an element's value (Integer) with a specific key.

3.2. Element (package Password.MyADT.Element)

a. The Node class

Node.java is the blueprint for producing the unit that is able to form a chain (refer to Section 3.2.c). It includes 2 member attributes: data (Pair<T, E>) and next (Node<T, E>). Later on, the **data** part will carry both the strings produced by SecretKeyGuesser and their keys. Meanwhile, the **next** part is set to point to the next node in a linked list **Chain**.

b. The Pair class

Objects of the **Pair** type are a mean to store passwords (String) that are used to evaluate the true secret key. Those String values are always accompanied by a key (Integer) (refer to CustomHashMap.java for the method of forming a key). Keys are crucial to insert a new password to or read a saved password from the hashmap.

c. The Chain class

Chains are linked lists that help to avoid hash collisions since the procedure relies on hashmap. Chains are created from nodes (Node<String, Integer>).

3.3 Key (package Password.MyADT.Element)

a. The KeyNode class

KeyNode objects are the unit to construct a KeyList. One of its attributes, named data (String), stores a password that we generated. Another one is next (KeyNode), which is used for pointing to the next KeyNode in a linkedList of type KeyList.

b. The KeyList class

A KeyList is actually a modified version of a linkedList that we apply to prevent collisions when using hashmaps. The head property (KeyNode) is a reference to the first KeyNode of the linkedList. The size (int) is the total count of nodes in the linkedList.

c. The KeySet class

A KeySet instance is a collection of linkedList (KeyList). The class covers all hashmap's fundamental functions, including the method of hashing a given string, and the insertion and removal of new items. The CustomHashMap class relies on those functions to store a newly yielded password at the proper location int the hashmap, and to retrieve the right password with the corresponding hashed key.

4. Algorithm Analysis and Time Complexity

4.1. Attributes

Table 1: Attributes in SecretKeyGuesser class

Name	Data Type	Modifier(s)	Description
HINT	String	private final	This string carries all the character set exists in the secret key. In this assessment, these characters are M, O, C, H and A.
LENGTH	int	private final	This is the number of characters in the secret key. In this assessment, LENGTH equals 12.
STOP	boolean	private	STOP is a break for permute() method. Whenever a token is generated and taken as a valid guess in process(), set the STOP for the permute().
candidates	CustomHashMap	private	This is a HashMap to store all potential tokens that are generated with its Key is the token itself and its Value is the matched characters with the correct one.
counters	int[]	private	It is used to remember matches for each character in HINT
token	String	private	Tokens are generated strings that are compared with the secret key.

4.2. Functions and Workflow

Main function and Workflow:

```
public void start(){
    SecretKey key = new SecretKey();
    candidates = new CustomHashMap( size: 20);
    counters = new int[HINT.length()];
    STOP = false;

    init(key);
    token = String.valueOf(setup());
```

This is the method to start the program. At first, initialize the counters[] in init() method as the first component, which helps in the generation of the first potential token.

```
private void init(SecretKey key){
    int count = 0;
    String chars = "MOCHA";
    for (int i = 0; i < HINT.length() - 1; i++) {
        if (count == LENGTH) return;
        String token = Character.toString(chars.charAt(i)).repeat(LENGTH);

        System.out.println("Guessing....." + token);
        counters[i] = key.guess(token);

        if(counters[i] == LENGTH){
            System.out.println("I found the secret key. It is " + token);
            System.exit( status: 0);
        }

        count += counters[i];
    }
    counters[4] = LENGTH - count;
}
```

The first potential token is set up based on the number of each character appearances of the **CORRECT KEY**. The token is the String returning from setup().

```
private StringBuilder setup(){
    StringBuilder str = new StringBuilder();
    for(int i = 0; i < counters.length; i++){
        str.append(Character.toString(toChar(i)).repeat(counters[i]));
    }
    return str;
}
```

After determining each character's frequency, formed the token. Instead of using String variable then add them up with the +=, we decide to use the StringBuilder type.

The reason is that in Java, the String class is immutable, meaning once a string is created, it cannot be changed. When using the '+=' operator to add to a string, a new string is created every time, which can consume a lot of memory if done repeatedly. But the StringBuilder class is mutable. It allows us to modify the string without creating a new one. This makes it more efficient for operations that need to modify a string repeatedly, such as in a for – loop.

```
    int matched = 0;
    while(true){
        System.out.println("Guessing....." + token);
        matched = key.guess(token);

        if(matched == LENGTH){
            break;
        }
        candidates.put(token, matched);

        token = next();
    }
    System.out.println("I found the secret key. It is " + token);
}
```

Up to now, we have finished the set – up process for this problem. Let's the token being formed recently be our first token to guess. From now on in **start()**, this is the core process of this assignment.

1. Iterate the loop with functions guess(token) to return the matches, the put(token, matches) to store the token in the HashMap and next() to looking for the next potential token.
2. Update the 'matched' variable with the value returned by guess(token). If the matched = 12 (The token has 12 matches with the **CORRECT KEY**), then break the iteration and print out the result is that token.
3. For every guess(token) called, the token will be stored in the HashMap. The Key of the element is the token itself (a string), and the Value is the number which SecretKey.guess(token) returns, the number of matches between the token and the **CORRECT KEY**.
4. After storing data of the recent token, run the next() method to choose out a next potential token among all permutations.

```
private String next(){
    char[] tokens = token.toCharArray();
    char[] stored = new char[tokens.length];
    boolean[] used = new boolean[tokens.length];

    permute(tokens, stored, used, curr_index: 0);

    STOP = false;
    return token;
}
```

5. Initialize all necessary arrays to ready for the permute(). Run the permute() method to generate the next token.

```
private void permute(char[] tokens, char[] stored, boolean[] used, int curr_index){
    if(STOP) return;

    if(curr_index == tokens.length){
        process(String.valueOf(stored));
        return;
    }

    boolean[] duplicated = new boolean[HINT.length()];
    for(int i = 0; i < tokens.length; i++){
        if(used[i] || duplicated[valueOf(tokens[i])]) continue;

        duplicated[valueOf(tokens[i])] = true;

        stored[curr_index] = tokens[i];
        used[i] = true;

        permute(tokens, stored, used, curr_index: curr_index + 1);
        used[i] = false;
    }
}
```

6. To stop the recursion, either STOP or process() condition has to be satisfied. The STOP will stop the permute() from continuing to recurse as a new potential token is found. The second is used to determined whenever there is 1 available permutation. Call the process() with the String.valueOf(stored) to convert char[] into String.

```
private void process(String stored){
    for(String key : candidates.keySet()){
        if(compare(stored, key) != candidates.get(key)) return;
    }
    STOP = true;
    token = stored;
}
```

7. The process(String) will make a check if the String input is a new potential token. To accomplish this, the compare(String, String) is made with the same idea as the guess() function in SecretKey. We compare the recent permutation (stored) with all the previous potential guesses which is stored in the HashMap. If the number of matching characters does not match with the corresponding value in the map (which represents the number of **CORRECT** characters in that guess), the function immediately returns. This means that the current permutation of characters is not a better guess than all the previous ones.

```java
private int compare(String s1, String s2){
    if(s1.length() != s2.length())
        throw new IllegalArgumentException("Two strings should have the same length");

    int match = 0;
    for(int i = 0; i < s1.length(); i++){
        char c = s1.charAt(i);

        if(valueOf(c) == -1)
            throw new IllegalArgumentException("Unexpected char: " + c);

        if(c == s2.charAt(i)){
            match++;
        }
    }
    return match;
}
```

8. If that permutation (stored) satisfies all the comparisons, which means it is a better guess, more potential matching with the **CORRECT KEY**, set a STOP to stop the recursion in the permute() and update the token value to stored.
9. Everything is ready for the next guess().

Ideas and estimation will be cover later in Performance and Probability.

Support Functions:

```java
private int valueOf(char c) {
    return switch (c) {
        case 'M' -> 0;
        case 'O' -> 1;
        case 'C' -> 2;
        case 'H' -> 3;
        case 'A' -> 4;
        default -> -1;
    };
}
```

The function is used to convert the character M, O, C, H, A to the corresponding values 0, 1, 2, 3, 4. If there are other characters apart from these 5, return -1.

```
private char toChar(int value){
    return switch (value){
        case 0 -> 'M';
        case 1 -> 'O';
        case 2 -> 'C';
        case 3 -> 'H';
        case 4 -> 'A';
        default -> throw new IllegalStateException("Unexpected value: " + value);
    };
}
```

The function is opposite with the valueOf(char). It converts the value into the corresponding character. If the value is other than 0, 1, 2, 3, 4, an exception will be thrown to make a notice of an unexpected value.

4.3. Performance and Probability

Now, the key problem is how to define the valid token in process() as the input for every next guess() function. Our approach is to select the next guess based on its matched positions with all the previous candidates. Let's break down and analyze this problem.

Because we're using permutation, the first thing we need to figure out is the total number of outcomes satisfy these following:

1. It has n characters (n = 12).
2. It has exactly k positions matched with the **CORRECT KEY**.

$$T(k,n) = \binom{n}{k} = C_n^k$$

Where, $T(k,n)$ is the total number of n – digit outcomes with each outcome having exactly k matching position with the **CORRECT KEY**. This is also known as "choosing k positions out of n positions.".

Next, we determine the odds of selecting the correct value at each position. With M O C H A, the probability of choosing the correct character is 1 out of 5 $\left(\frac{1}{5}\right)^k$ Consequently, the probability of choosing an incorrect character from the given five options is 4 out of 5 $\left(1 - \frac{1}{5} = \frac{4}{5}\right)$. Therefore, the probability of getting k correct positions out of n positions would be:

$$p(k,n) = \left(\frac{1}{5}\right)^k \times \left(\frac{4}{5}\right)^{n-k}$$

Where, $p(k\ ,n)$ is the probability of each case to get an n – digit outcome which matched exactly k chosen positions.

However, we must consider that each scenario has $T(k,n)$ possible outcomes. So, the probability of having exactly k matched positions compared to the **CORRECT KEY**, also known as the probability for SecretKey.guess(outcomes) to return k [5], is as follows:

$$P(k) = T(k,n) \times p(k,n) = \binom{n}{k} \times \left(\frac{1}{5}\right)^k \times \left(\frac{4}{5}\right)^{n-k}$$

Our strategy is to keep all the tokens that could still potentially be the **CORRECT KEY** after each SecretKey.guess(token). In this scenario, for every guess(valid token), the token will be stored in the HashMap. If the guess(token) != 12 → Continue to find the next potential.

After finding a permutation with permute(), run the process() to check if that permutation is potential. To accomplish this, we iterate through the HashMap and compare these two values:

1. The number of matched characters between the current one with all previous potential tokens → compare(stored, key).
2. The Value of all previous guesses, which is also known as the number of matched characters between themselves with the **CORRECT KEY**.

At this point, the current permutation is already known. Speaking of all previous guesses, they are also generated randomly. So the probability of tokens that have k matched positions with the current one is P(k).

And also, the probability of SecretKey.guess(current) returning k is P(k). Therefore, the average chance for both events happening simultaneously is P(k) x P(k), which simplifies to $P(k)^2$, which is also known as the percentage of permutations we keep after each SecretKey.guess(current) to choose a next guess token among them. So,

- If k = 0, we will keep $P(0)^2$ permutations.
- If k = 1, we will keep $P(1)^2$ permutations.
- If k = 2, we will keep $P(2)^2$ permutations.
- …
- If k = 12, we will keep $P(12)^2$ permutations.

So, the probability of the remaining permutations after each guess() function is the total of all average percentages above [6]

$$P(0)^2 + P(1)^2 + \cdots + P(12)^2 = \sum_{k=0}^{12} [P(k)^2] = \sum_{k=0}^{12} \left[\binom{12}{k} \times \left(\frac{1}{5}\right)^k \times \left(\frac{4}{5}\right)^{12-k} \right]^2 \approx 20.4111\%$$

After each guess() is called, 20.4111% of the permutaions is kept. The total number of permutations is defined above (5^{12}). After all the guess(), there will be approximately one

permutation left which is the **CORRECT KEY**. As a result, the largest number of guess() our program might run (in the worst case) is X in which:

$$5^{12} \times (0.204111)^X \approx 1$$

$$\rightarrow X = \log_{0.204111}\left(\frac{1}{5^{12}}\right) \approx 12$$

To summarize, the number of guess() function that our program has to call is between the best case: 1 and the worst case: 12 + 4 = 16 (refer to Appendix A).

4.4. Algorithm Time Complexity

The specification:

- The Length of the Key: n
- The Length of the Hint: h
- The Length of counters: k
- The number of Key in HashMap: a (for process(String))

```
private void validateKey(){                                                    Cost  Occurences
    for(int i = 0; i < correctKey.length(); i++){                               c1       n
        char check = correctKey.charAt(i);                                      c2       n
        if (check != 'M' && check != 'O' && check != 'C'
                        && check != 'H' && check != 'A') {                      c3       n
            System.out.println("Unexpected tokens in KEY: " + check +
                               " at [" + i + "].");                             c4       1
            System.exit(0);                                                     c5       1
        }
    }
}
```

Time Complexity = n * (c1 + c2 + c3) + c4 + c5 ⇨ O(n)

```
public int guess(String guessedKey) {                                   Cost   Occurences
    counter++;                                                           c1        1
    if (guessedKey.length() != correctKey.length()) {                    c2        1
        System.out.println("Unexpected Length in KEY:  "                 c3        1
                           + correctKey.length());
        System.exit(0);                                                  c4        1
    }
    int matched = 0;                                                     c5        1
    for (int i = 0; i < guessedKey.length(); i++) {                      c6        n
        char c = guessedKey.charAt(i);                                   c7        n
        if (c != 'M' && c != 'O' && c != 'C' && c != 'H' && c != 'A') {   c8        n
            return -1;                                                   c9        1
        }

        if (c == correctKey.charAt(i)) {                                 c10       n
            matched++;                                                   c11       n
        }
    }
    if (matched == correctKey.length()) {                                c12       1
        System.out.println("Number of guesses: " + counter);             c13       1
    }
    return matched;                                                      c14       1
}
```

Time Complexity = n * (c6 + c7 + c8 + c10 + c11) + (c1 + c2 + c3 + c4 + c5 + c9 + c12 + c13 + c14) ⇨ O (n)

```
private void init(SecretKey key){                                       Cost   Occurences
    int count = 0;                                                       c1        1
    String chars = "MOCHA";                                              c2        1
    for (int i = 0; i < HINT.length() - 1; i++) {                        c3        h
        if (count == LENGTH) return;                                     c4        h
        String token = Character.toString(chars.charAt(i)).repeat(LENGTH); c5      h

        System.out.println("Guessing....." + token);                     c6        h
        counters[i] = key.guess(token);                                  n         h
```

```
        if(counters[i] == LENGTH){                                      c7        h
            System.out.println("I found the secret key. It is " + token);  c8        1
            System.exit(0);                                             c9        1
        }

        count += counters[i];                                           c10       h
    }
    counters[4] = LENGTH - count;                                       c11       1
}
```

Time Complexity = h * (n + c3 + c4 + c5 + c6 + c7 + c10) + (c1 + c2 + c8 + c9 + c11)⇨ O (n * h)

```
private StringBuilder setup(){                                          Cost   Occurences
    StringBuilder str = new StringBuilder();                             c1        1
    for(int i = 0; i < counters.length; i++){                            c2        k
        str.append(Character.toString(toChar(i)).repeat(counters[i]));   n - 1     k
    }
    return str;                                                          c3        k
}
```

Time Complexity = k * (n – 1) + k * (c2 + c3) + c2 ⇨ O(n * k)

The reason for the n – 1 is that, if the worst case occurs, that means counters[] can count for max length of the key (n). If that case happens, the init(key) method is sooner return the **CORRECT KEY** and exit() the program.

```
private int compare(String s1, String s2){                              Cost   Occurences
    if(s1.length() != s2.length())                                       c1        1
        throw new
              IllegalArgumentException("Two strings should
                                    have the same length");              c2        1

    int match = 0;                                                       c3        1
    for(int i = 0; i < s1.length(); i++){                                c4        n
        char c = s1.charAt(i);                                           c5        n
```

```
        if(valueOf(c) == -1)                                                    c6      n
            throw new
               IllegalArgumentException("Unexpected char: " + c);               c7      1

        if(c == s2.charAt(i)){                                                  c8      n
            match++;                                                            c9      n
        }
    }
    return match;                                                               c10     1
}
```

Time Complexity = n * (c4 + c5 + c6 + c8 + c9) + c1 + c2 + c3 + c7 + c10 ⇨ O(n)

```
private void process(String stored){                                    Cost  Occurences
    for(String key : candidates.keySet()){                               c1       a
        if(compare(stored, key) != candidates.get(key)) return;          n        a
    }
    STOP = true;                                                         c2       1
    token = stored;                                                      c3       1
}
```

Time Complexity = a * (n + c1) + c2 + c3 ⇨ O(n * a)

```
private void permute(char[] tokens, char[] stored,
                     boolean[] used, int curr_index){
    if(STOP) return;

    if(curr_index == tokens.length){
        process(String.valueOf(stored));
        return;
    }
    boolean[] duplicated = new boolean[HINT.length()];
    for(int i = 0; i < tokens.length; i++){
        if(used[i] || duplicated[valueOf(tokens[i])]) continue;
```

```
        duplicated[valueOf(tokens[i])] = true;

        stored[curr_index] = tokens[i];
        used[i] = true;

        permute(tokens, stored, used, curr_index + 1);
        used[i] = false;
    }
}
```

Consider the permutation generation process. We have n choices for the first position. Following the placement of the first element, we have n – 1 possibility for the second position, N – 2 choices for the third position, and so on until we have just 1 choice for the final location.

The recursive relation for the time complexity: $T(n) = n * T(n - 1) + O(1)$

$T(n) = n * T(n - 1) + 1$

$= n * [(n - 1) * T(n - 2) + 1] + 1$

$= . . .$

$= n * [(n - 1) * [(n - 2) * ... * T(1) + 1] + ...] + 1$, which $T(1) = 1$

Time Complexity ⇨ O(n!)

```
private String next(){                                                  Cost  Occurences
    char[] tokens = token.toCharArray();                                 n        1
    char[] stored = new char[tokens.length];                             c1       1
    boolean[] used = new boolean[tokens.length];                         c2       1

    permute(tokens, stored, used, 0);                                    n!       1

    STOP = false;                                                        c3       1
    return token;                                                        c4       1
}
```

Time Complexity = n! + n + (c1 + c2 + c3 + c4) ⇨ O(n!)

```
public void start(){                                                    Cost  Occurences
    SecretKey key = new SecretKey();                                     n        1
    candidates = new CustomHashMap(20);                                  c1       1
    counters = new int[HINT.length()];                                   c2       1
    STOP = false;                                                        c3       1

    init(key);                                                           n * h    1
    token = String.valueOf(setup());                                     n * k    1

    int matched = 0;                                                     c4       1
    while(true){                                                         c5       n
        System.out.println("Guessing....." + token);                     c6       n
        matched = key.guess(token);                                      n        n

        if(matched == LENGTH){                                           c7       n
            break;                                                       c8       n
        }
        candidates.put(token, matched);                                  c9       n

        token = next();                                                  n!       n
    }
    System.out.println("I found the secret key. It is " + token);       c10      1
}
```

Time Complexity = n * (1 + c7 + c8 + c9 + n!) + (n * h + n * k) + (c1 +c2 + c3 + c4 + c10) ⇨ **O(n * n!)**

In conclusion, after several calculations of time complexity O of all method, the complexity of our algorithm is **O(n * n!)**.

5. System Evaluation

Our custom Abstract Data Type (ADT) serves as the core data structure for storing and retrieving Key – Value pairs. The effectiveness of this ADT is defined by its ability to handle collisions efficiently. For collision management, we've adopted the method of separate chaining. This

technique uses linked lists to store elements that have the same index, ensuring error – free operation.

Our ideas were founded on empirical experiments, including several attempts at first due to the varying change of the returning counter when guessing. Because implementing all of them would be impracticable and time-consuming, we decide to spend the first week just brainstorming.

We considered several ways, the simplest of which was inherited from the source code. The first thing that came to our mind was brute – force. With 5 characters from the HINT (M O C H A) and the LENGTH of the correct key being 12, the total number of possible guesses is $5^{12} > 240$ million. It is a TERRIBLE choice. Then, we come up with a better notion of using pruning to eliminate all impossible scenarios. If we just guess one character's placement, the first big O will be 4 and the remainder will be 3. To summarize, it would require 4 + 3 x 11 = 37 guesses. It was a large figure, but it was better than the brute – force one.

After that, we try to minimize this number by counting the number of times each letter M, O, C, H, and A appears. This requires 4 guesses at first; however, we will not need to estimate the depleted number, which does not appear in the very last characters. So, in the worst – case scenario, all characters appear at the very end. In particular, AAAAAAAAHCOM, if we guess the password in the following order: M → O → C → H → A, the pre–identification will take 4 attempts. The first character slot was also taken 4 times, then 3 times up to index 7^{th}, 2 times for index 8^{th}, and 1 for both 9^{th} and 10^{th}. We don't need to estimate the last index because it was pre-calculated. As a result, this method lowered the guessing times from 37 to 33, which is not very good. We attempted trimming for pre-identification at this point, but it did not help to lessen the worst scenario.

Finally, we decided to use the permutation with pruning method to optimize the guesses required to find the correct key. We generate all possible permutations without any duplication [7]. Instead of having to compare and estimate each permutation, we believe it is preferable to locate the spots that match the correct key. Following guesses, the values considered possible (or potentially true) are those that at least contain the positions identified in previous guesses. This can also be understood as comparing the number of characters considered identical in the correct key and the next possible values with the values that have been used. To accomplish this, we need an ADT that can include both GuessKey (a guessed value) and Matched Value (a value identified as

matching the proper key). Instead of applying two arrays to achieve this, a hash map is a far better option. So, we decided to make our own abstract data structures in order to support this assessment.

In our custom ADT that we have implemented, a hash map forms the core data structure for storing and retrieving Key – Value pairs. The efficiency of this ADT is determined by how well it manages collisions. We have used the separate chaining for collision handling. Separate chaining uses linked lists to store the element that hash to the same index.

In the case of two parallel arrays – one for keys and one for values – the time complexity for search, insert and delete operations is O(n). However, a custom HashMap ADT has an average time complexity of O(1) for insert and search. Even in worst – case scenarios, this ADT's time complexity is as efficient as the two arrays.

About our custom ADT, it forms the core data structure for storing and retrieving Key – Value pairs. The efficiency of this ADT is determined by how well it manages collisions. We have used the separate chaining for collision handling. Separate chaining uses linked lists to store the element that hash to the same index.

6. Testing

6.1. Valid Cases

Case 1: MMMMMMMMMMMM (Best case)

```
Guessing.....MMMMMMMMMMM
Number of guesses: 1
I found the secret key. It is MMMMMMMMMMM
```

Based on our workflow and algorithm when generating the first potential token, it is clear that this case has to be our best scenario.

Case 2: MMOOOCCCHHAA

```
Guessing.....MMMMMMMMMMM
Guessing.....OOOOOOOOOOO
Guessing.....CCCCCCCCCCC
Guessing.....HHHHHHHHHHH
Guessing.....MMOOOCCCHHAA
Number of guesses: 5
I found the secret key. It is MMOOOCCCHHAA
```

Case 3: MMMMMMMOMOCHA

```
Guessing.....MMMMMMMMMMM
Guessing.....OOOOOOOOOOO
Guessing.....CCCCCCCCCCC
Guessing.....HHHHHHHHHHH
Guessing.....MMMMMMMOOCHA
Guessing.....MMMMMMMOOCAH
Guessing.....MMMMMMMOCOHA
Guessing.....MMMMMMMOMOCHA
Number of guesses: 8
I found the secret key. It is MMMMMMMOMOCHA
```

Case 4: MMOCCHAMCOAH

```
Guessing.....MMMMMMMMMMM
Guessing.....OOOOOOOOOOO
Guessing.....CCCCCCCCCCC
Guessing.....HHHHHHHHHHH
Guessing.....MMMOOCCCHHAA
Guessing.....MMMCCOOHAACH
Guessing.....MMMCCHAAOOHC
Guessing.....MMMCHAAHCOOC
Guessing.....MMMCAHHACOCO
Guessing.....MMCCMHAHOOCA
Guessing.....MMOCCHAMCOAH
Number of guesses: 11
I found the secret key. It is MMOCCHAMCOAH
```

Case 5: CHAMOMOMOCHA

```
Guessing.....MMMMMMMMMMM
Guessing.....OOOOOOOOOOO
Guessing.....CCCCCCCCCCC
Guessing.....HHHHHHHHHHH
Guessing.....MMMOOOCCHHAA
Guessing.....MMOMCCOOAAHH
Guessing.....MMOCAAHHMCOO
Guessing.....MOMACHAMOOHC
Guessing.....MOAMHOMACOCH
Guessing.....MAHHOCOAOMMC
Guessing.....AMHOMHMOOACC
Guessing.....AOOHCMOMCHMA
Guessing.....AOCAOMOCMMHH
Guessing.....OOACMCOMHMHA
Guessing.....CHAMOMOMOCHA
Number of guesses: 15
I found the secret key. It is CHAMOMOMOCHA
```

6.2. Error Cases

1. What if the CORRECT KEY has a length smaller than what we expected?

Length – Error case 1: MMOOOCACCH (LENGTH = 10)

```
Guessing.....MMMMMMMMMMM
Unexpected Length in KEY: 10
```

In the guess() function, with the initial implementation of the sub token MM...M (Length = 12), in the first guess, there is a condition to terminate the program if the length of **CORRECT KEY** is different from the guessed token.

Length – Error case 2: MMOOOCACCHHAA (LENGTH = 13)

```
Guessing.....MMMMMMMMMMM
Unexpected Length in KEY: 13
```

2. Unexpected characters in **CORRECT KEY.**

Character – Error case 1: MMMOOCCHHAA**K**

The given hint is MOCHA. What if the **CORRECT KEY** contains the characters other than MOCHA?

```
Unexpected tokens in KEY: K at [11].

Process finished with exit code 0
```

As we have implemented the validateKey() function and run it in the start() method, the program will be terminated if there are any characters other than the given hint.

7. Conclusion and Further Improvements

To conclude, Password Algorithm successfully solves the mysterious password problem using the core concepts throughout the course. By analyzing and testing, it handles a lot of unexpected error cases different from the project constraints. As mentioned in Section 1, this program was design to solve the password as if it is longer. In scenario we realize the password is much longer, we just need to modify the LENGTH in our algorithm to the new one, and it would perform as expectation. The next step to improve this program is that it can handles multiple character sets to make it applicable in many cases.

Reference List

[1] RMIT University Online. (2023). Data Structures & Algorithms. [Online]. Available: https://rmit.instructure.com/courses/131308/assignments/883729

[2] Huo Chong Ling. (2023). Assessment Note. [Online]. Available: https://rmiteduau.sharepoint.com/:f:/r/sites/COSC2658_COSC2469_COSC2203_DSA_AA_Sem3_2023/Shared Documents/Lecture_sessions/Assessment/Group Project

[3] GeeksforGeeks. “Brute Force Approach and its pros and cons”. geeksforgeeks.org. Accessed: Dec. 8, 2023. [Online]. Available: https://www.geeksforgeeks.org/brute-force-approach-and-its-pros-and-cons/

[4] Fortinet. “What Is A Brute Force Attack?”. fortinet.com. Accessed: Dec. 7, 2023. [Online]. Available: https://www.fortinet.com/resources/cyberglossary/brute-force-attack

[5] Nikhil Lohia. “Permutations 2 (LeetCode 47) | Full solution with backtracking examples | Study Algorithms”. (May. 1, 2022). Accessed: Dec. 3, 2023. [Online Video]. Available: https://www.youtube.com/watch?v=YW5F0WqBBWY&pp=ygUSdW5pcXVlIHBlcm11dGF0aW9u

[6] qy9Mg. “How to explain to interviewer - 843. Guess the Word”. leetcode.com. Accessed: Dec. 4, 2023. [Online]. Available: https://leetcode.com/problems/guess-the-word/solutions/556075/how-to-explain-to-interviewer-843-guess-the-word/

[7] “The Binomial Distribution”. mathsisfun.com. Accessed: Dec. 4, 2023. [Online]. Available: https://www.mathsisfun.com/data/binomial-distribution.html

Appendices

Appendix A - Specification

To summarize, the number of guess() function that our program has to call is between the best case 1 and the worst case 12 + 4 = 16.

```
private void init(SecretKey key){
    int count = 0;
    String chars = "MOCHA";
    for (int i = 0; i < HINT.length() - 1; i++) {
        if (count == LENGTH) return;
        String token = Character.toString(chars.charAt(i)).repeat(LENGTH);

        System.out.println("Guessing....." + token);
        counters[i] = key.guess(token);

        if(counters[i] == LENGTH){
            System.out.println("I found the secret key. It is " + token);
            System.exit( status: 0);
        }

        count += counters[i];
    }
    counters[4] = LENGTH - count;
}
```

In the init() function, when we initialize the counters[] array to do the setup() for the first potential token, we have made a condition that during the guess(token) process, if the counters[i] = 12, which means that all the characters in the token are matched with the **CORRECT KEY**, stop the program because we found the key. So for our best case, considered “MMMMMMMMMMMM”, the number of guess() is 1.

About the worst case, based on the init() and setup(), the following tokens are generated to store the appearances of characters in the **CORRECT KEY** into the counters[] array. Those are:

1. MMMMMMMMMMMM

2. OOOOOOOOOOO
3. CCCCCCCCCCC
4. HHHHHHHHHHH
5. AAAAAAAAAAA

If the **CORRECT KEY** is none of these, which coresponding guess() cases are 1, 2, 3, 4, and 4, we will start the guessing process with the first key generated based on counters[]. That is why, in the worst case, the largest guess() function called has an additional 4 calls, resulting in 12 + 4 = 16 calls.