RMIT UNIVERSITY

COSC2658 – Data Structures and algorithms

GROUP PROJECT

Subject code	COSC2658
Subject name	Data Structures and Algorithms
Location & Campus (SGS or HN) where you study:	SGS Campus
Title of Assignment:	Group Assignment
Lecture name:	Ling Huo Chong
Assignment due date:	December 22nd , 2023
Number of pages include in this one:	21
Word Count:	2411
Student name:	Ngo Lam Bao Trung (s3938154) Bui Sy Quang (s3958668) Vo Nguyen Kien (s3938016) Vo Nguyen Don Vinh (s3880306)

RMIT UNIVERSITY

Table of Contents

I. Overview and high-level Design:

1. SecretKey Class

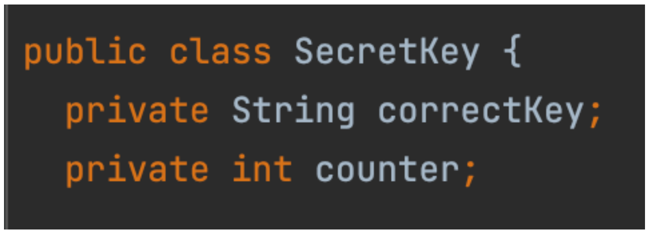
a. Overview:

The ‘SecretKey’ class is a class which is used to generate and store a secret key to use in the ‘SecretKeyGuesser’ class. The secret key is a string of characters that has 12 characters where each character can only be ‘M’, ‘O’, ‘C’, ‘H’, ‘A’. The `guess` method takes the string as the input and compares each character in the input string to the corresponding character in the correct key while the `matched` counter is incremented. If all the characters in the input string match the correct key, the `guess` method will prints out how many guesses were made before this happened. The `main` method serves as an entry point for the program.

b. High-level Design:

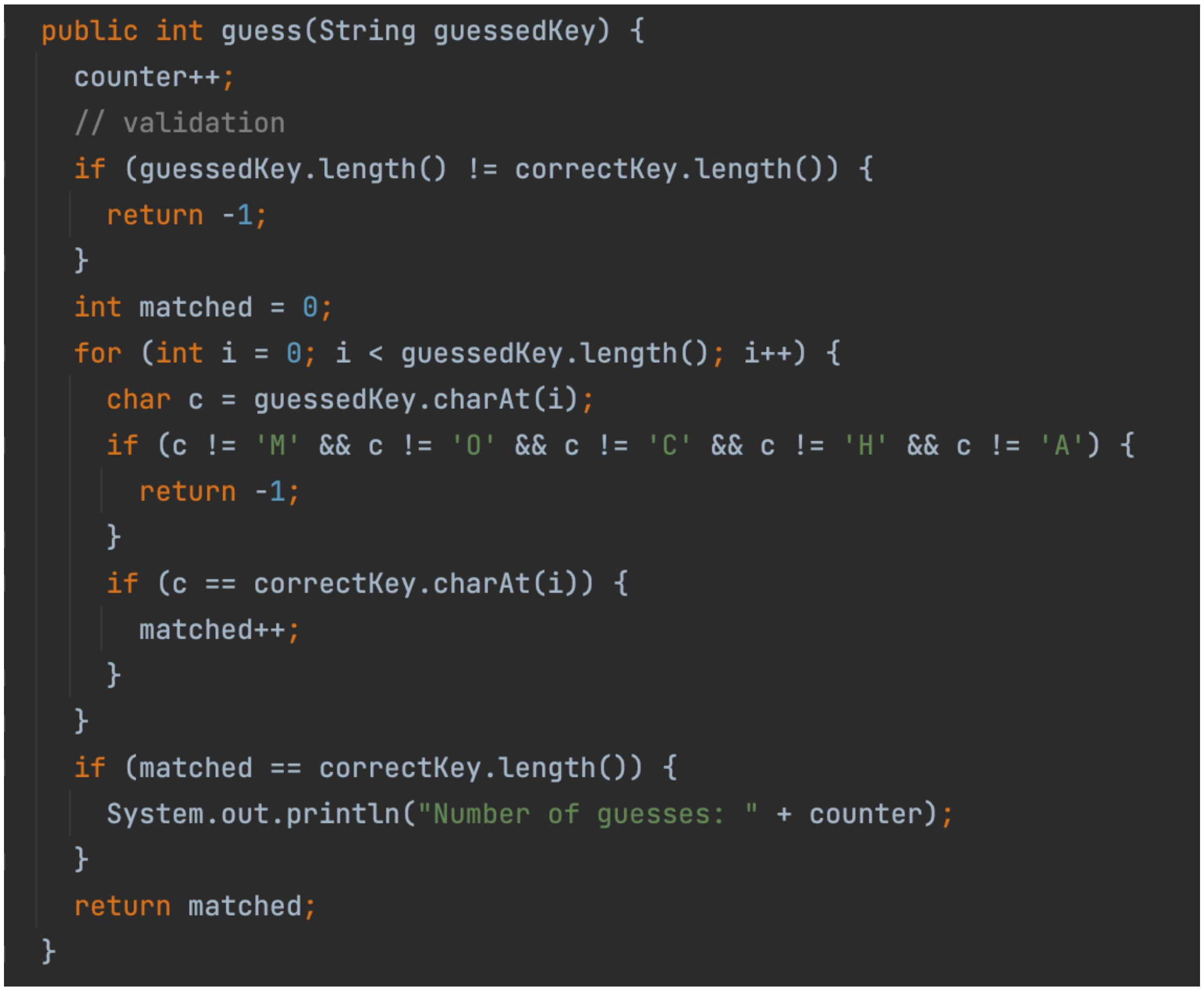
The ‘SecretKey’ class has two private instance variables: ‘correctKey’ and ’counter’. The ‘correctKey’ variable stores the correct key, and the `counter` variable counts the number of guesses made by the program. The class has three methods.

A constructor ‘SecretKey’ which generates a random key, a ‘counter’ and sets its initial value to 0.



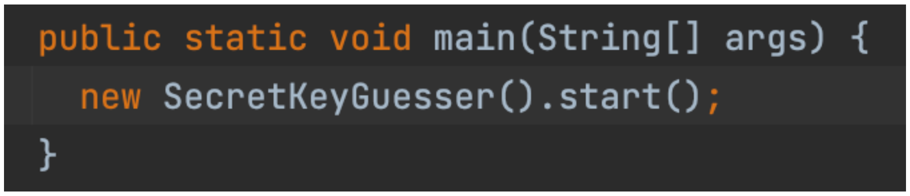
```
public class SecretKey {
  private String correctKey;
  private int counter;
```

A ‘guess’ method takes a string as the input and compares it to the correct key. The method returns the number of characters in the guessed string that match the correct key and increments the `counter` variable.



```java
public int guess(String guessedKey) {
  counter++;
  // validation
  if (guessedKey.length() != correctKey.length()) {
    return -1;
  }
  int matched = 0;
  for (int i = 0; i < guessedKey.length(); i++) {
    char c = guessedKey.charAt(i);
    if (c != 'M' && c != 'O' && c != 'C' && c != 'H' && c != 'A') {
      return -1;
    }
    if (c == correctKey.charAt(i)) {
      matched++;
    }
  }
  if (matched == correctKey.length()) {
    System.out.println("Number of guesses: " + counter);
  }
  return matched;
}
```

A `main` method that creates an instance of 'SecretKeyguesser' class and call the `start` method.



```java
public static void main(String[] args) {
  new SecretKeyGuesser().start();
}
```

2. SecretKeyGuesser Class

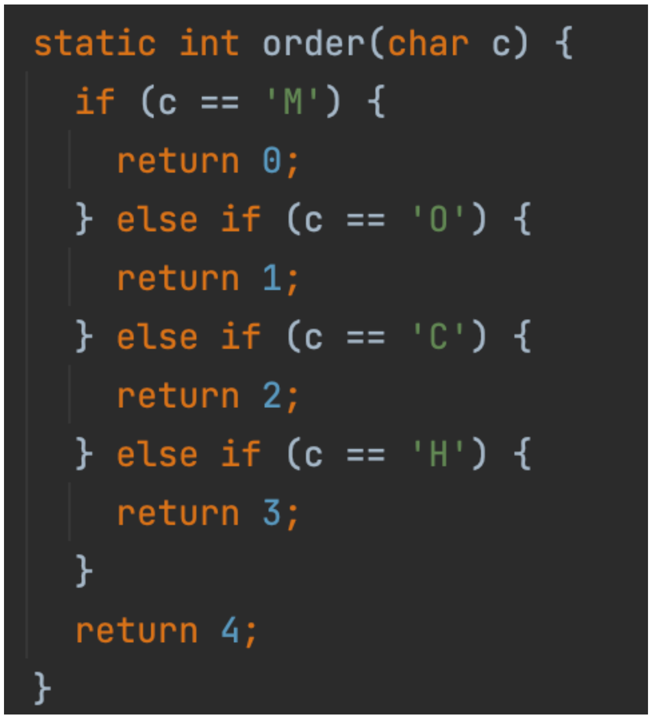
a. Overview

The 'SecretKeyGuesser' class is a multiway tree data structure which stores all the possible guesses for the secret key, where each node in the tree is a guess generated based on its parent node. The class has 6 methods: ‘order()’, ‘charOf()’, ‘moveForward()’, ‘trackBack()’, ‘solve()’ and ‘start()’ to start break down the secret key provided from the ‘SecretKey’ class.

b. High – level design:

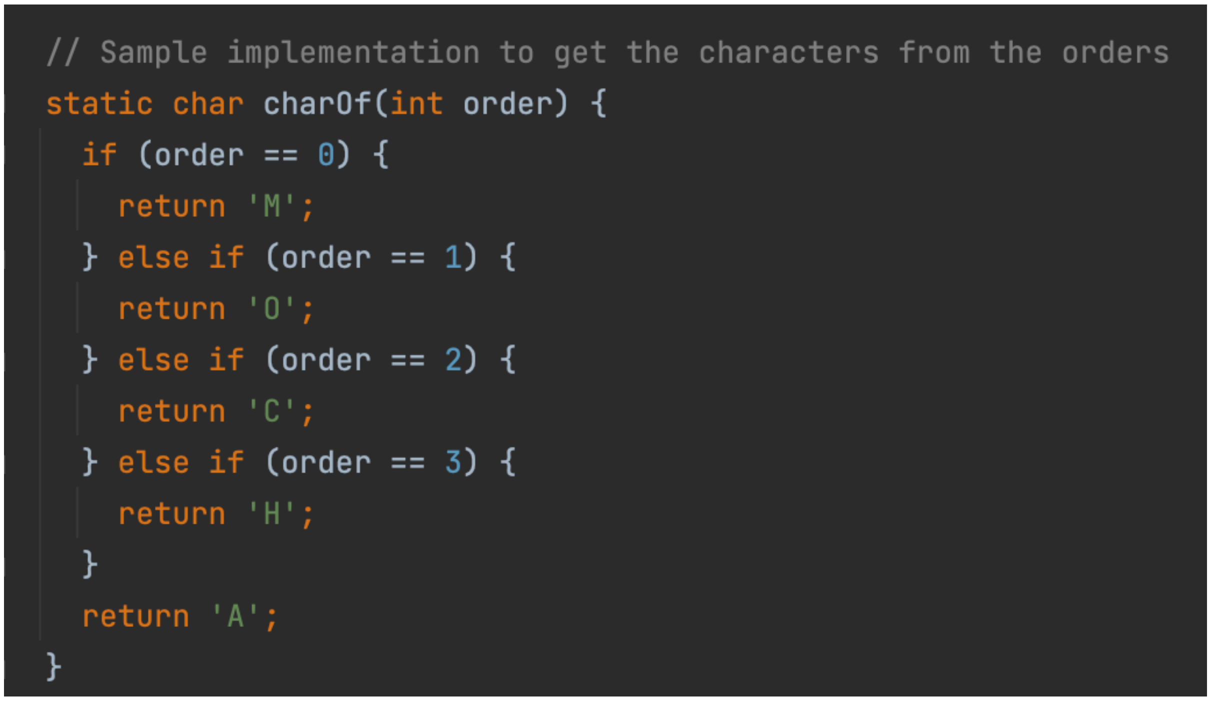
The ‘SecretKeyGuesser’ class provides 6 methods:

- Order(char c): this function returns the index of the input character of the string.



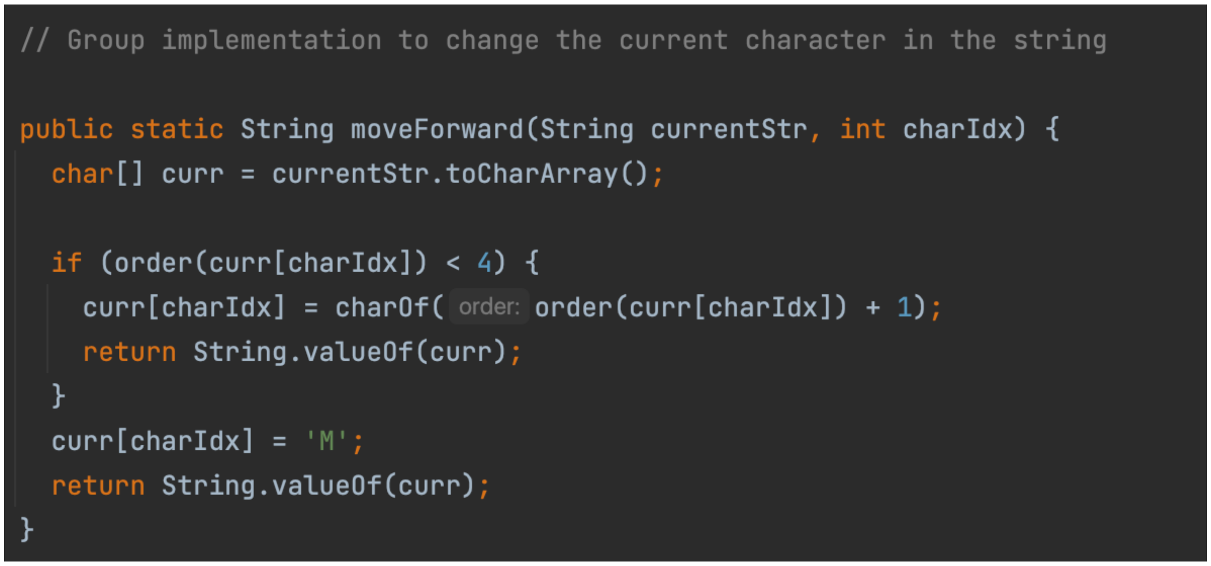
```
static int order(char c) {
  if (c == 'M') {
    return 0;
  } else if (c == 'O') {
    return 1;
  } else if (c == 'C') {
    return 2;
  } else if (c == 'H') {
    return 3;
  }
  return 4;
}
```

- charOf(int order): this function returns the characters from the orders.



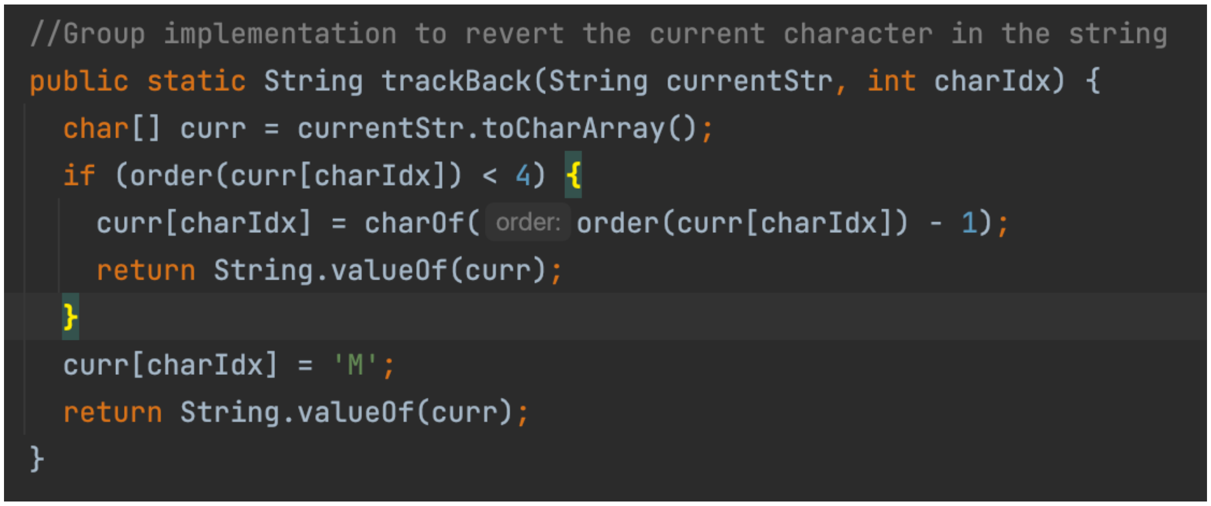
```java
// Sample implementation to get the characters from the orders
static char charOf(int order) {
  if (order == 0) {
    return 'M';
  } else if (order == 1) {
    return 'O';
  } else if (order == 2) {
    return 'C';
  } else if (order == 3) {
    return 'H';
  }
  return 'A';
}
```

- moveForward (String currentStr, int charIdx): this function increases the order of the current character and changes that character in the string.



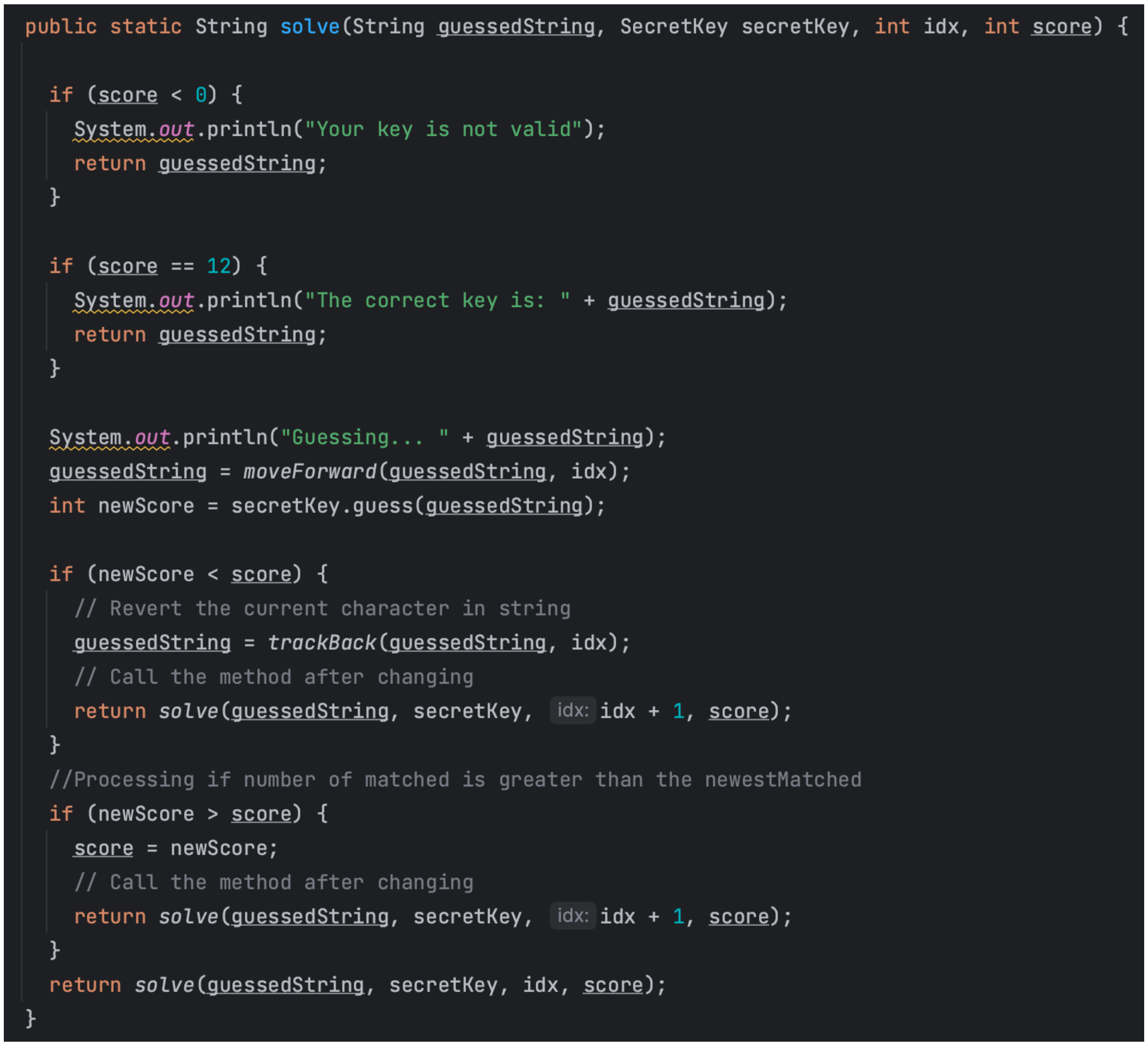
```java
// Group implementation to change the current character in the string

public static String moveForward(String currentStr, int charIdx) {
  char[] curr = currentStr.toCharArray();

  if (order(curr[charIdx]) < 4) {
    curr[charIdx] = charOf( order: order(curr[charIdx]) + 1);
    return String.valueOf(curr);
  }
  curr[charIdx] = 'M';
  return String.valueOf(curr);
}
```

- trackBack(String currentStr, int charIdx): this function decreases the order of the current character and changes that character in the string.



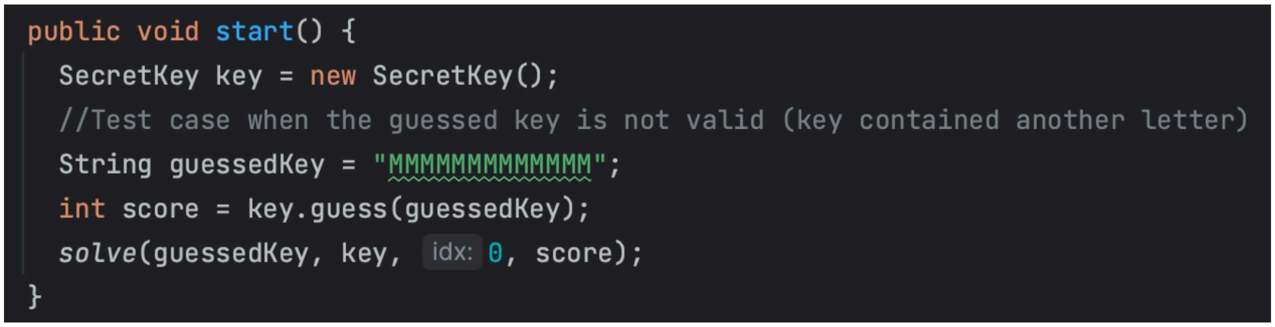
```java
//Group implementation to revert the current character in the string
public static String trackBack(String currentStr, int charIdx) {
  char[] curr = currentStr.toCharArray();
  if (order(curr[charIdx]) < 4) {
    curr[charIdx] = charOf( order: order(curr[charIdx]) - 1);
    return String.valueOf(curr);
  }
  curr[charIdx] = 'M';
  return String.valueOf(curr);
}
```

- solve (String guessedString, SecretKey secretKey, int idx, int score): This function changes one character in the data array to ‘M’, ‘O’, ‘C’, ‘H’, ‘A’ by calling functions moveForward() and trackBack(), then it calls the guess method to get its score. If the score is 12, so the correct key is found and the recursion will stop, or it will continue the process until the correct key is found. When the correct key is found, the function will display the correct key along with the number of guesses.

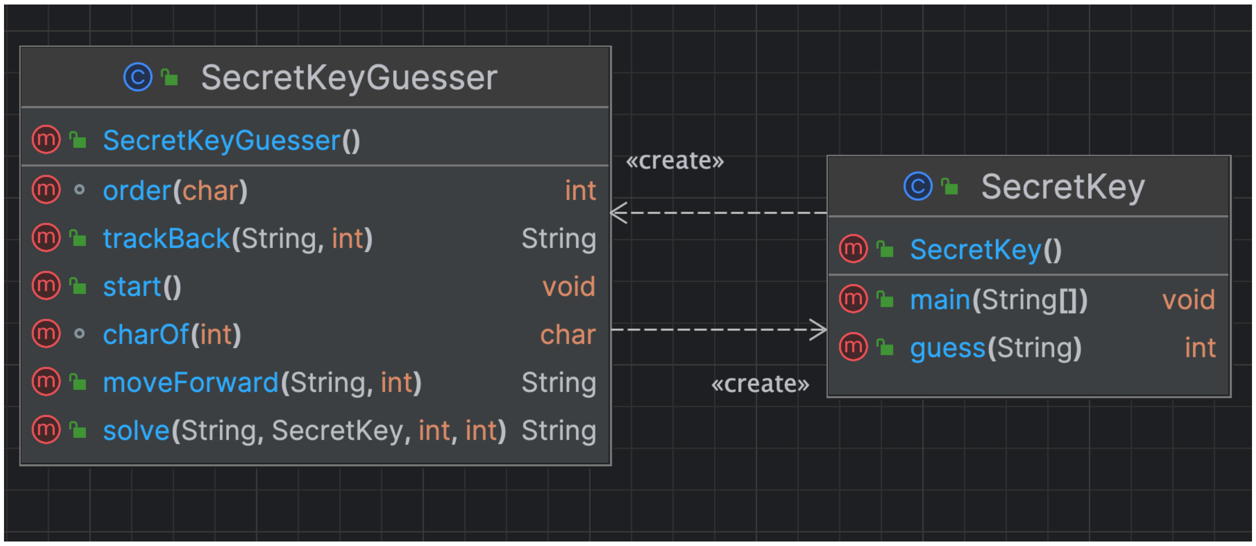


```java
public static String solve(String guessedString, SecretKey secretKey, int idx, int score) {

    if (score < 0) {
        System.out.println("Your key is not valid");
        return guessedString;
    }

    if (score == 12) {
        System.out.println("The correct key is: " + guessedString);
        return guessedString;
    }

    System.out.println("Guessing... " + guessedString);
    guessedString = moveForward(guessedString, idx);
    int newScore = secretKey.guess(guessedString);

    if (newScore < score) {
        // Revert the current character in string
        guessedString = trackBack(guessedString, idx);
        // Call the method after changing
        return solve(guessedString, secretKey, idx: idx + 1, score);
    }
    //Processing if number of matched is greater than the newestMatched
    if (newScore > score) {
        score = newScore;
        // Call the method after changing
        return solve(guessedString, secretKey, idx: idx + 1, score);
    }
    return solve(guessedString, secretKey, idx, score);
}
```

- start (): this function creates a new instance of ‘SecretKey’ class. It initializes the base guessed string with data “MMMMMMMMMMMMMMMM” and calls the ‘guess’ method to see what the score of this base case. Then, it passes all the required parameters to the ‘solve()’ method.



```
public void start() {
    SecretKey key = new SecretKey();
    //Test case when the guessed key is not valid (key contained another letter)
    String guessedKey = "MMMMMMMMMMMMM";
    int score = key.guess(guessedKey);
    solve(guessedKey, key, idx: 0, score);
}
```



II. Data structures and Algorithms:

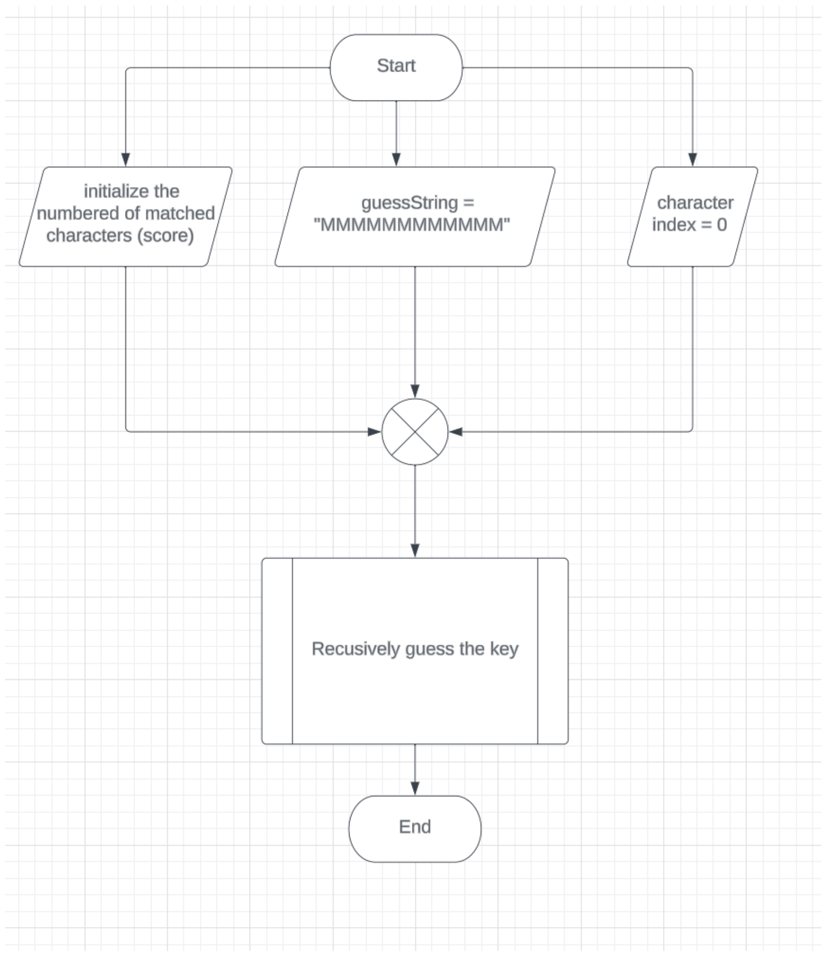
1. Data structures:

The data structures used in the algorithm are mainly Java built-in String and Characters. Where String is used to store the value of the secret key and guess key, the characters store value of ‘M’, ‘O’, ‘C’, ‘H’ or ‘A’ which are the possible characters in the secret key.

The reasons for using string and characters lie in the easy and convenient implementations. String is literally an array of characters. In this case, both secret key and guess key are string of 12 characters. The usage of string allows us to iterate and modify any character at specific index. So that, we can generate any possible permutations of guess key using the hint characters which supports a lot in guessing process.

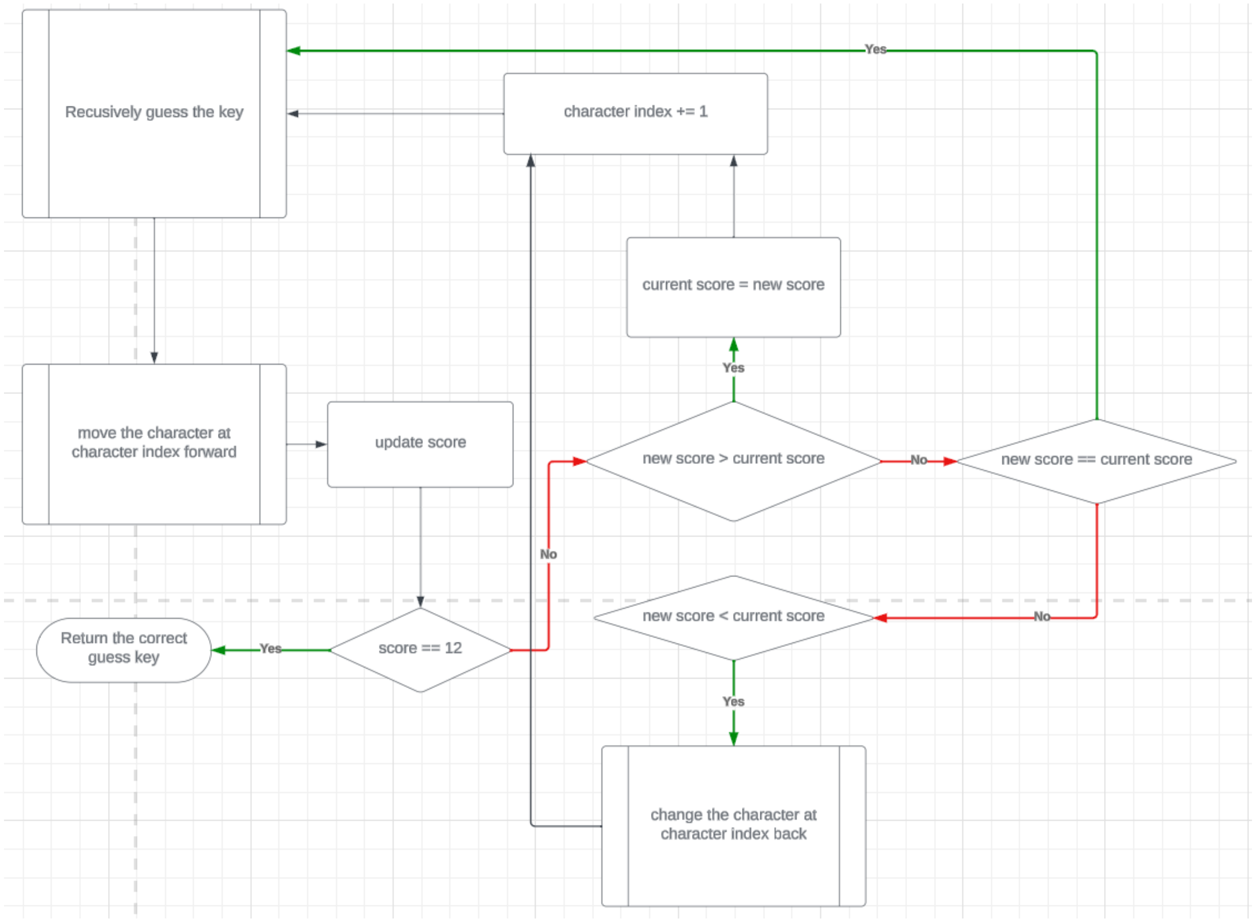
2. Algorithms:

The approach we use to solve the problem is that we recursively change and check the “score” - number of matched characters after each modification to the guess key. This approach sounds quite like brute force – a naïve algorithm that take approximate maximum 5^{12} guessing times to find the secret key. However, with some twist in the implementation, we manage to optimize the guessing process by relate on the “score”.


Start
initialize the numbered of matched characters (score)
guessString = "MMMMMMMMMMMM"
character index = 0
Recusively guess the key
End

(Figure 1)

For the initialization of the program, it creates the first guess string with 12 ‘M’ characters. Then it calls the “guess” function in class SecretKey to get the “score” – the number of matched characters between the secret key and the guess string. It also assigns 0 to the character index. (figure 1)



Recusively guess the key
character index += 1
Yes
current score = new score
Yes
move the character at character index forward
update score
new score > current score
No
new score == current score
No
new score < current score
No
Return the correct guess key
Yes
score == 12
Yes
change the character at character index back

(Figure 2)

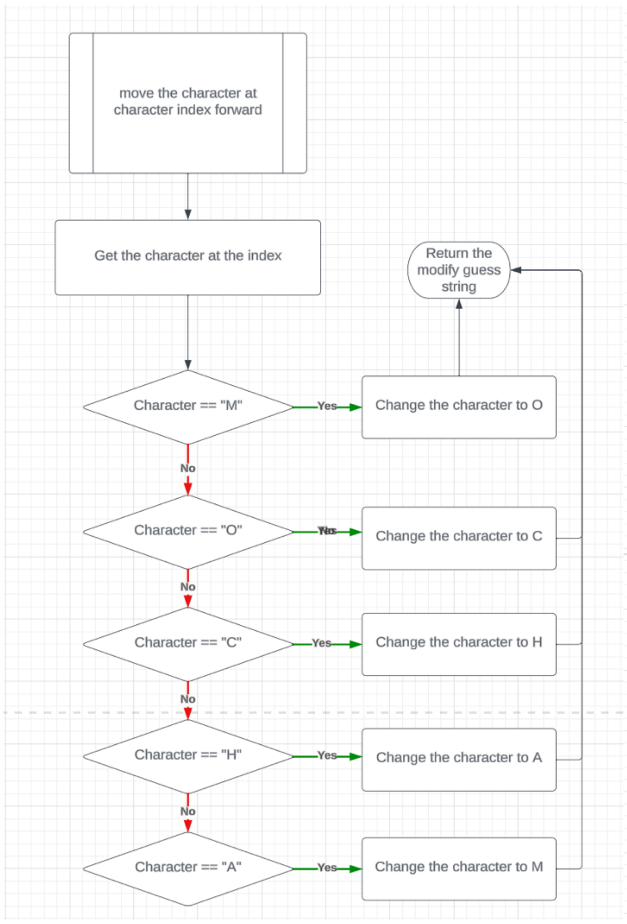
After setting up, the program begins the guessing process (figure 2). It calls the "solve" function recursively to update the character to the next possible character according to the character index (figure 1). After that it will compare the modified string with the secret key and update the new score. From now, the program compares the new score with the current score. This time, there are 3 possible probabilities:

- The new score > the current score: The programs then assign the current score by the new score and increment the character index by 1. The purpose behind this is to keep track of the highest score and move on the next character in the string for next guessing.
- The new score < the current score: The programs then reassign the current character with the previous character and increment the index by 1. The purpose for this is to get the back the correct character in the guess string (best score) and move on the next characters to continue.

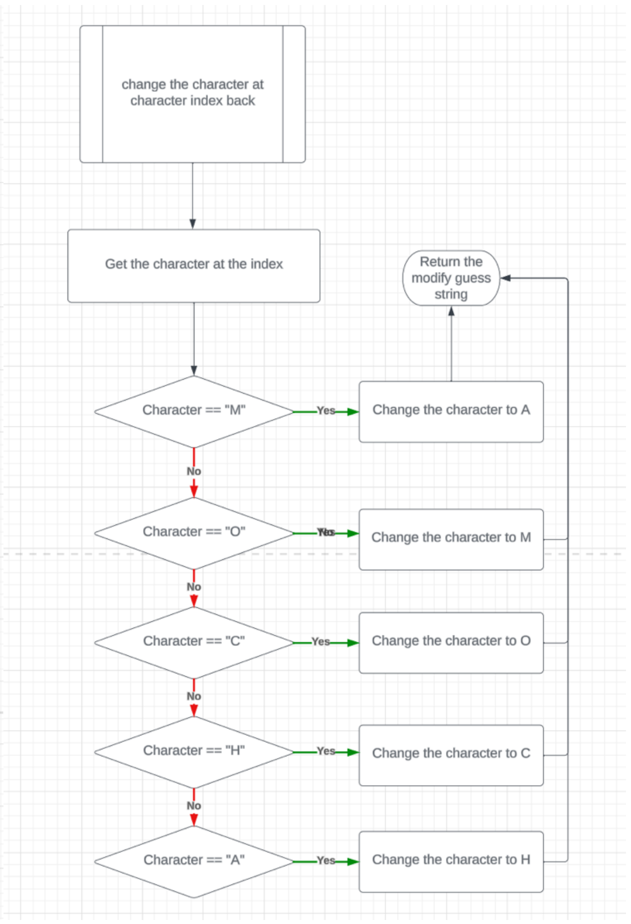
- The new score = the current score: The programs then continue changing the current character to the next possible characters (‘M’, ‘O’, ‘C’, ‘H’ or ‘A’). The reason for this is to continue updating the new score until it is greater or smaller to the current score.

There is primary 2 types of changing the string character mentioned above. As we intendedly arrange the hint characters in an order from M to A (‘M’, ‘O’, ‘C’, ‘H’ or ‘A’):

- Update to the next possible characters (move the character forward): The program will get the character at the character index and check if it is in the following case. If the character satisfies one of the cases, change it to the next character in the array (figure 3). ‘M’ to ‘O’, ‘O’ to ‘C’, etc. (used when new score >= current score).
- Reassign with the previous character: The program will also check if the character matches any hint characters and change it back to the previous one (figure 4). ‘M’ to ‘A’, ‘O’ to ‘M’, etc. (used when new score < current score).


move the character at character index forward
Get the character at the index
Return the modify guess string
Character == "M"
Yes
Change the character to O
No
Character == "O"
Change the character to C
Character == "C"
Change the character to H
Character == "H"
Change the character to A
Character == "A"
Change the character to M

(Figure 3)


change the character at character index back
Get the character at the index
Return the modify guess string
Character == "M"
Yes
Change the character to A
No
Character == "O"
Change the character to M
Character == "C"
Change the character to O
Character == "H"
Change the character to C
Character == "A"
Change the character to H

(Figure 4)

With the mentioned flow, the program can efficiently guess the secret key without randomly assign the character like in brute force. It relates on the score to decide whether to continue updating the characters in the current index or move on to the next one. If the score increases, it must then be the correct character. Otherwise, it will continue updating the characters if the score remains unchanged and reassign to the previous character if the new score is fewer than the current score. Because of this, the program can guarantee making progress after each guess without wasting time and computer resources.

III. Complexity analysis

List of Acronyms

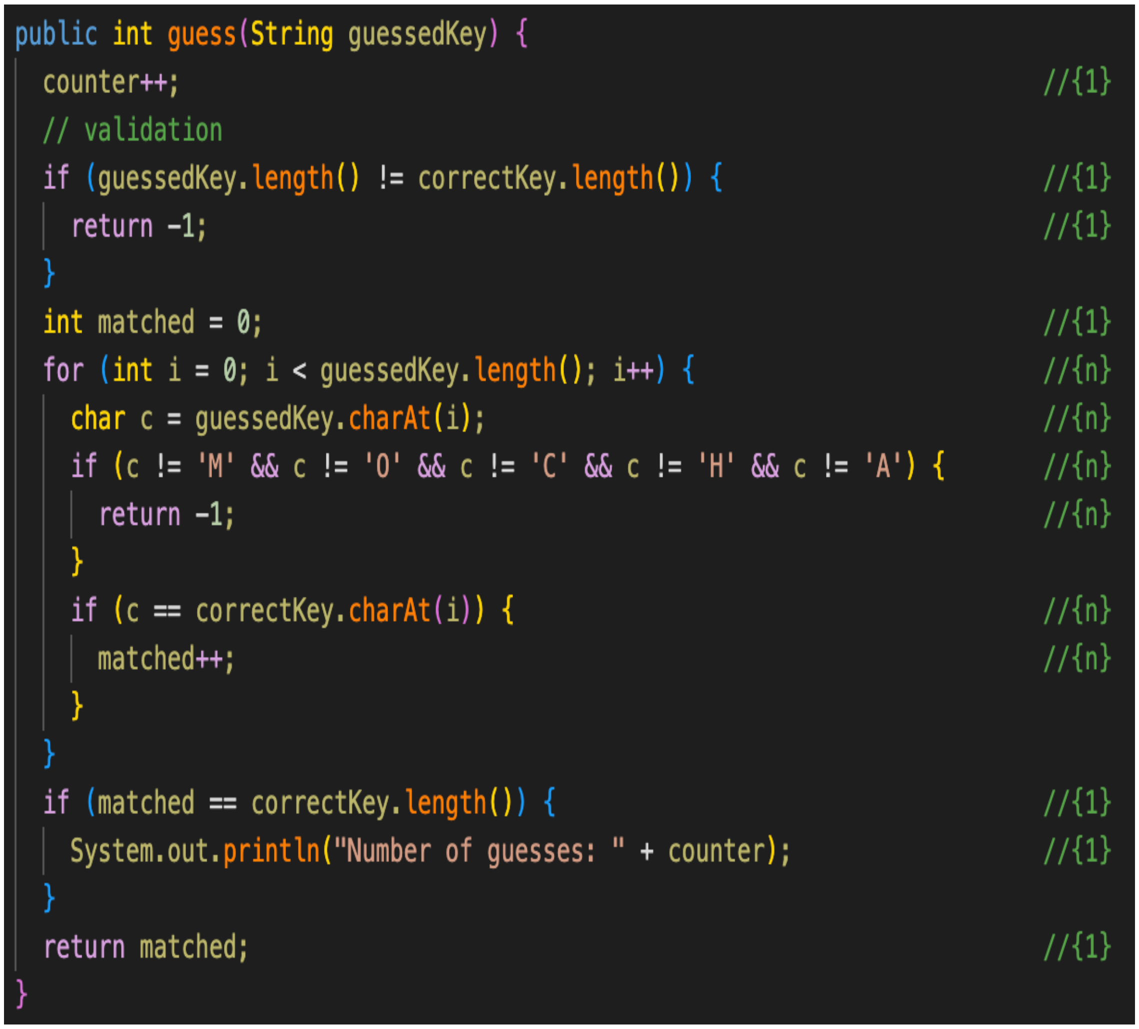
n: length of correctKey and guessedString, which is 12 in our case.

m: number of guessing characters, which is 5 in our case.

RMIT UNIVERSITY

1. SecretKey

a. Time complexity



```java
public int guess(String guessedKey) {
    counter++;                                                        //{1}
    // validation
    if (guessedKey.length() != correctKey.length()) {                 //{1}
        return -1;                                                    //{1}
    }
    int matched = 0;                                                  //{1}
    for (int i = 0; i < guessedKey.length(); i++) {                   //{n}
        char c = guessedKey.charAt(i);                                //{n}
        if (c != 'M' && c != 'O' && c != 'C' && c != 'H' && c != 'A') {   //{n}
            return -1;                                                //{n}
        }
        if (c == correctKey.charAt(i)) {                              //{n}
            matched++;                                                //{n}
        }
    }
    if (matched == correctKey.length()) {                             //{1}
        System.out.println("Number of guesses: " + counter);          //{1}
    }
    return matched;                                                   //{1}
}
```

The “guess” method’s only significant element is a loop to traverse through the “guessedKey” length, which gives it the time complexity of O(n).

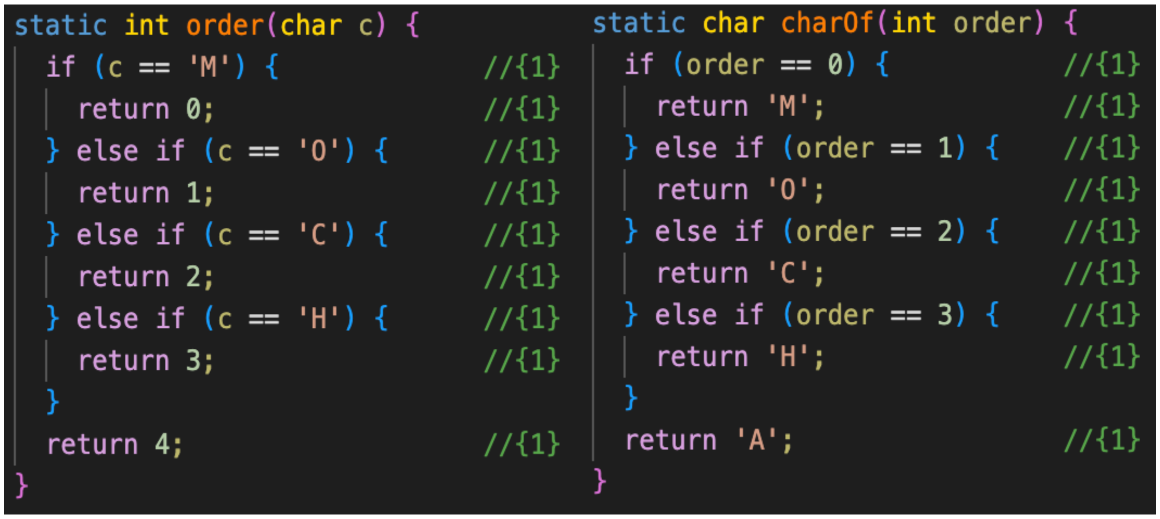
b. Space complexity

The SecretKey class has 2 attributes: a string variable “correctKey” and an integer variable “counter”, which gives it the space complexity of O(n+1), or O(n) after we eliminate the constant.

2. SecretKeyGuesser

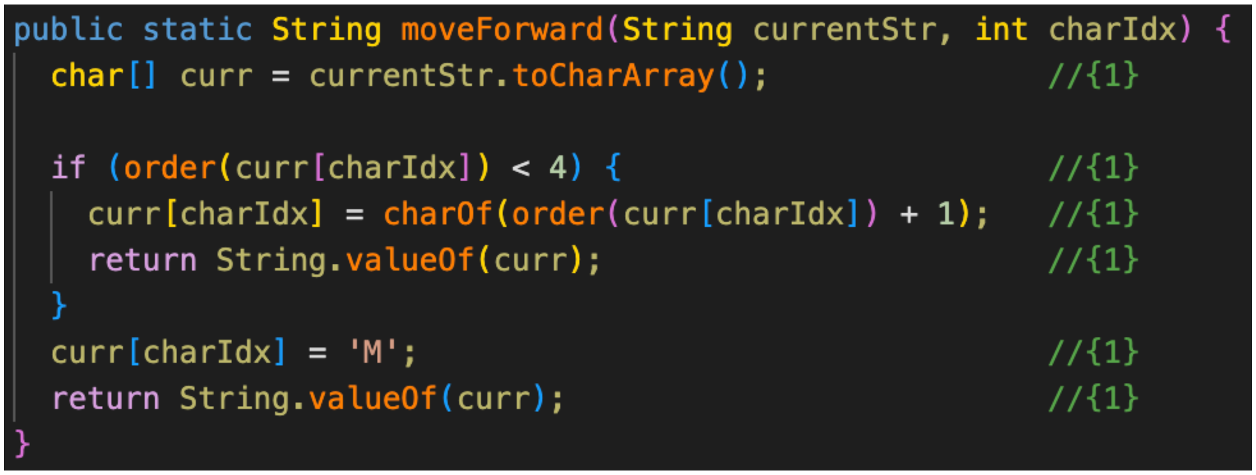
   a. Time complexity

After eliminating the insignificant elements and constants, the “order”, “charOf”, “moveForward” and “trackback” method all have constant time complexity of O (1).

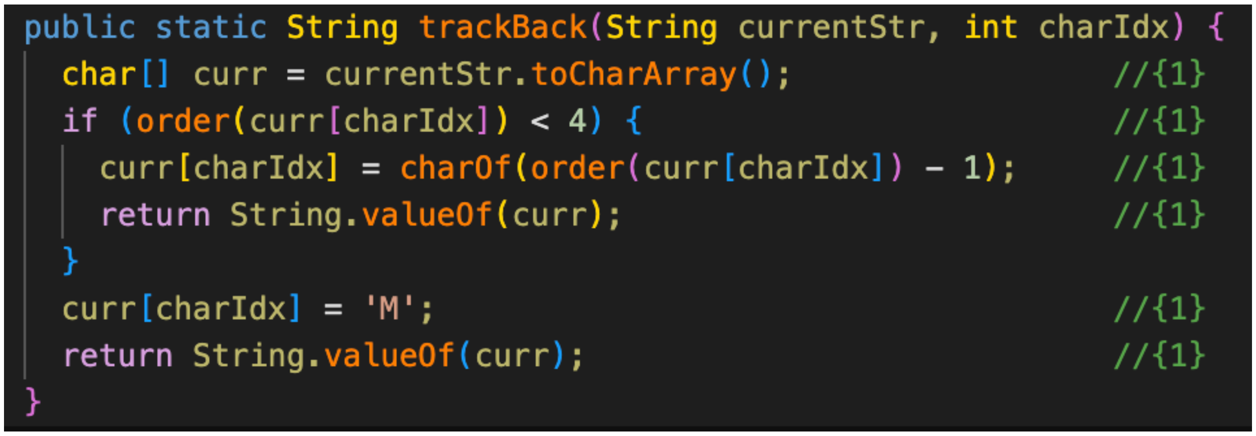


```
static int order(char c) {
    if (c == 'M') {                //{1}
        return 0;                  //{1}
    } else if (c == 'O') {         //{1}
        return 1;                  //{1}
    } else if (c == 'C') {         //{1}
        return 2;                  //{1}
    } else if (c == 'H') {         //{1}
        return 3;                  //{1}
    }
    return 4;                      //{1}
}
```

```
static char charOf(int order) {
    if (order == 0) {              //{1}
        return 'M';                //{1}
    } else if (order == 1) {       //{1}
        return 'O';                //{1}
    } else if (order == 2) {       //{1}
        return 'C';                //{1}
    } else if (order == 3) {       //{1}
        return 'H';                //{1}
    }
    return 'A';                    //{1}
}
```



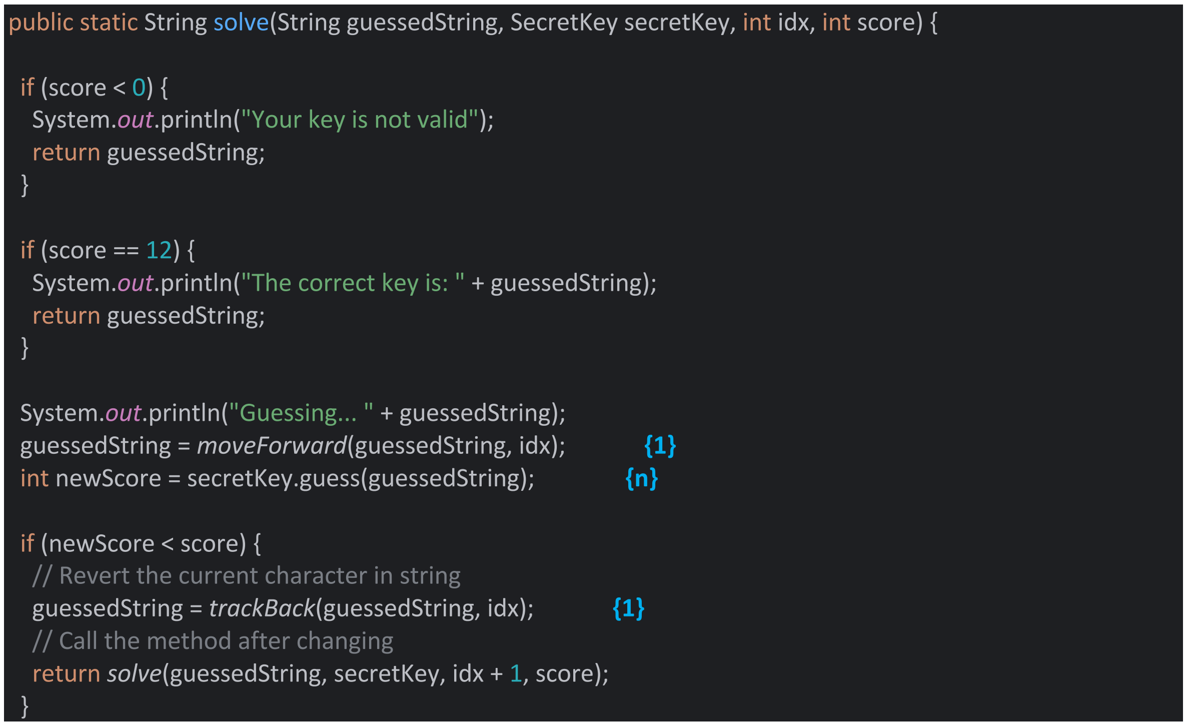
```
public static String moveForward(String currentStr, int charIdx) {
    char[] curr = currentStr.toCharArray();                //{1}

    if (order(curr[charIdx]) < 4) {                        //{1}
        curr[charIdx] = charOf(order(curr[charIdx]) + 1);  //{1}
        return String.valueOf(curr);                       //{1}
    }
    curr[charIdx] = 'M';                                   //{1}
    return String.valueOf(curr);                           //{1}
}
```



```
public static String trackBack(String currentStr, int charIdx) {
    char[] curr = currentStr.toCharArray();                        //{1}
    if (order(curr[charIdx]) < 4) {                                //{1}
        curr[charIdx] = charOf(order(curr[charIdx]) - 1);          //{1}
        return String.valueOf(curr);                               //{1}
    }
    curr[charIdx] = 'M';                                           //{1}
    return String.valueOf(curr);                                   //{1}
}
```

The main logic of the solving algorithms lies in the "solve" functions where we use recursion to guess the key.

The "solve" method will traverse through "n" characters of "guessedString" string. In each character, the function will generate the maximum of "m" cases of guessing characters. It then uses the "guess" method (O(n)) of SecretKey class to check whether the number of correct positions increases in any case and proceed to move on to the next character if it does. Therefore, we can generally calculate the time complexity of "solve" to be O (n^2 * m).



```
public static String solve(String guessedString, SecretKey secretKey, int idx, int score) {

    if (score < 0) {
        System.out.println("Your key is not valid");
        return guessedString;
    }

    if (score == 12) {
        System.out.println("The correct key is: " + guessedString);
        return guessedString;
    }

    System.out.println("Guessing... " + guessedString);
    guessedString = moveForward(guessedString, idx);          {1}
    int newScore = secretKey.guess(guessedString);           {n}

    if (newScore < score) {
        // Revert the current character in string
        guessedString = trackBack(guessedString, idx);       {1}
        // Call the method after changing
        return solve(guessedString, secretKey, idx + 1, score);
    }
```

```
 //Processing if number of matched is greater than the newestMatched
 if (newScore > score) {
   score = newScore                              {1}
   // Call the method after changing
   return solve(guessedString, secretKey, idx + 1, score);
 }
 return solve(guessedString, secretKey, idx, score);
}
```

For specific calculations:

Consider each stack called of the recursion. According to the specification of the key, we have maximum 12 * 5 stacks (length of the key * number of possible characters). However, taking the generality into account, n * m stacks will be used in the analysis.

Stack 1: n + 1 (calling guess function take n times to complete + change the character take constant time)

Stack 2: n + 1 (calling guess function take n times to complete + change the character take constant time)

….

Stack n * m: n + 1 (calling guess function take n times to complete + change the character take constant time)

To sum up, the time complexity can possibly be the sum of complexity of all the stacks.

$T(n,m) = (n+1) * n * m = n^2 * m + n * m$

Omit the trivial part, we get the time complexity of $O(n^2 * m)$

The “start” method, which is the main part of our algorithm, roughly has the time complexity of $O(n^2 * m)$, like the “solve” method.

b. Space complexity

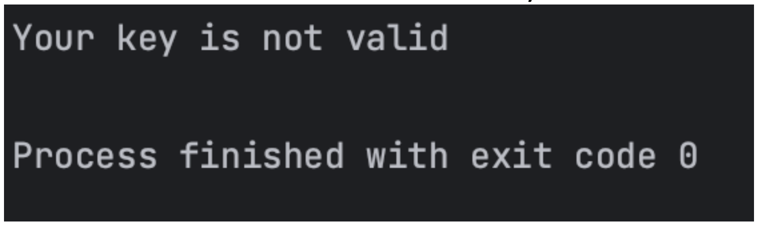
The algorithm in SecretKeyGuesser class will need to constantly store the “guessedKey” variable at any given time, which gives it the space complexity of O(n).

IV Evaluation

In order to meet the specification, our team created two classes: ***‘SecretKey’*** class and ***‘SecretKeyGuesser’*** class to process the logic. In the ***‘SecretKeyGuesser’*** class, following the instruction, we create a ***'SecretKey'*** instance and call the ***'guess()'*** method. On the other hand, in the ***'SecretKey'*** class,we include the initial correct key and the implementation of the ***'guess()'*** method. The code is readable and well-organized with the use of helper methods like: ***'moveForward'***, ***'trackBack'***, ***'solve'***.

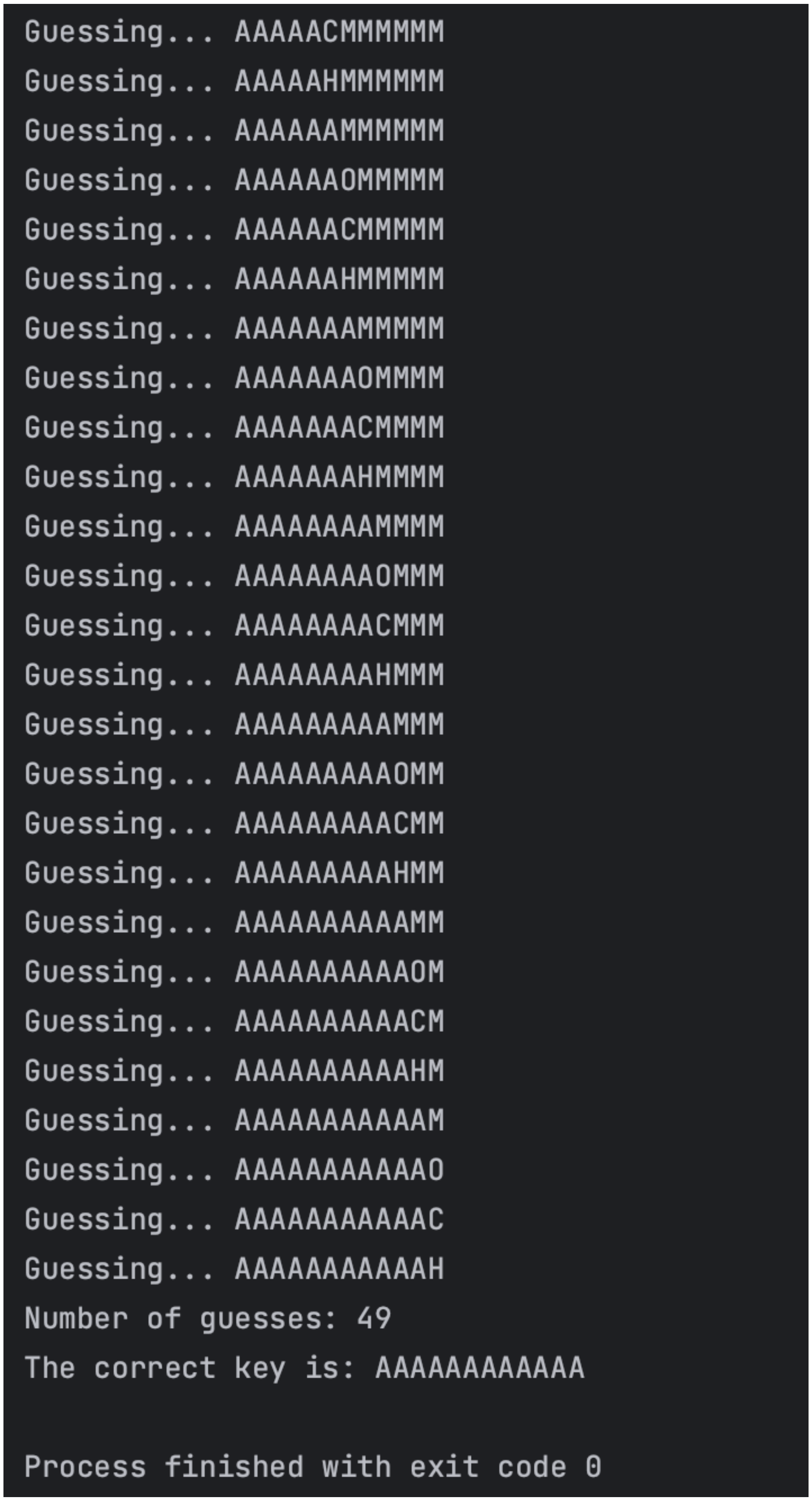
By increasing and decreasing character in response to comparison outcomes, we are able to determine the right logic for the algorithm. We try to minimize the number of guesses by adjusting only the necessary characters with the use of ***'start'*** method and ***'solve'*** method and a test case. The ***'guess'*** method is implemented to check the guessed key to find the right one and return the number of matched positions.

The validation is easy to understand, the algorithm is terminated if an invalid key is given. The ***'guess'*** method also gives out the validation for the length of the guessed key and the character it contains. It will return -1 for invalid keys.



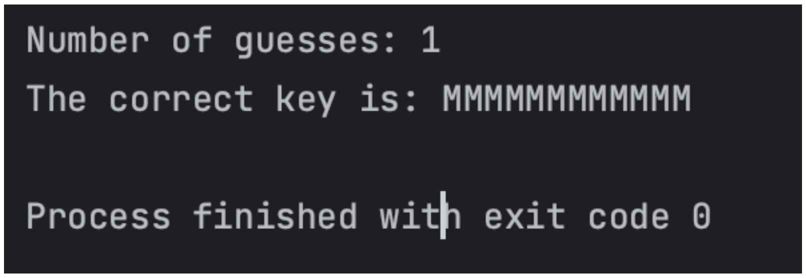
```
Your key is not valid

Process finished with exit code 0
```

The result of our algorithm compared to the Brute-Force one is a huge gap. With the worse case like twelve characters "A", our algorithm took 49 guesses to find out the correct key; This is because the algorithms need to go through all possible characters and change them (4 time each * 12 = 48) and Brute-Force implementation took more than five minutes and no sign of stopping to print out the result.



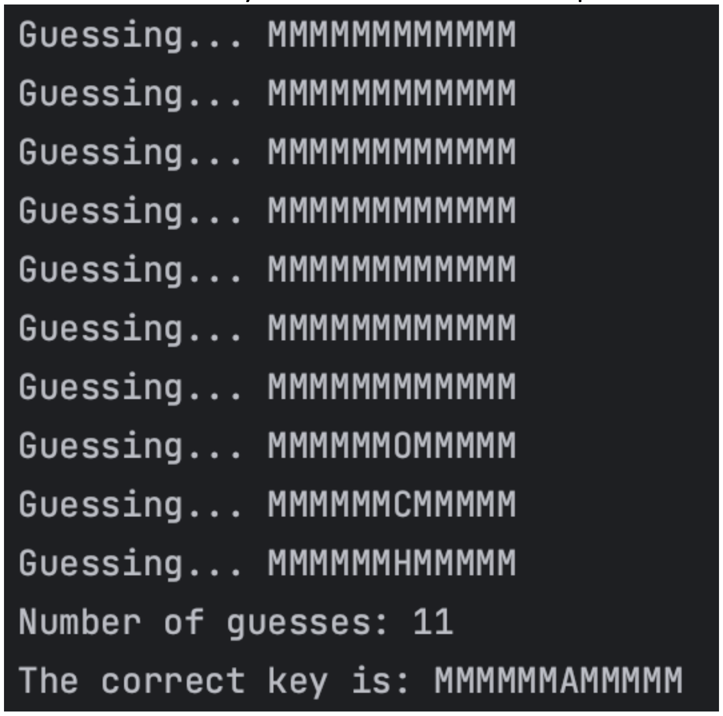
```
Guessing... AAAAACMMMMMM
Guessing... AAAAAHMMMMMM
Guessing... AAAAAAMMMMMM
Guessing... AAAAAAOMMMMM
Guessing... AAAAAACMMMMM
Guessing... AAAAAAHMMMMM
Guessing... AAAAAAAMMMMM
Guessing... AAAAAAAOMMMM
Guessing... AAAAAAACMMMM
Guessing... AAAAAAAHMMMM
Guessing... AAAAAAAAMMMM
Guessing... AAAAAAAAOMMM
Guessing... AAAAAAAACMMM
Guessing... AAAAAAAAHMMM
Guessing... AAAAAAAAAMMM
Guessing... AAAAAAAAAOMM
Guessing... AAAAAAAAACMM
Guessing... AAAAAAAAAHMM
Guessing... AAAAAAAAAAMM
Guessing... AAAAAAAAAAOM
Guessing... AAAAAAAAAACM
Guessing... AAAAAAAAAAHM
Guessing... AAAAAAAAAAAM
Guessing... AAAAAAAAAAAO
Guessing... AAAAAAAAAAAC
Guessing... AAAAAAAAAAAH
Number of guesses: 49
The correct key is: AAAAAAAAAAAA

Process finished with exit code 0
```

With the best case like twelve characters “M”, the two implementations seem to have the same result which is 1.



```
Number of guesses: 1
The correct key is: MMMMMMMMMMMM

Process finished with exit code 0
```

However, this is because the initialized guess string fortunately matches with the correct key. The result is not consistent. For example, we just need to change one of the characters in the secret key to another one like “A”. Brute force just takes forever to run while with our approach we can find the key in a small number of steps like 11.



```
Guessing... MMMMMMMMMMMM
Guessing... MMMMMMMMMMMM
Guessing... MMMMMMMMMMMM
Guessing... MMMMMMMMMMMM
Guessing... MMMMMMMMMMMM
Guessing... MMMMMMMMMMMM
Guessing... MMMMMMMMMMMM
Guessing... MMMMMMOMMMMM
Guessing... MMMMMMCMMMMM
Guessing... MMMMMMHMMMMM
Number of guesses: 11
The correct key is: MMMMMMAMMMMM
```

The requirement in both implementations is all the same (go through a combination of 12-character key and find out the correct one). But the difference between the two algorithms is our algorithm is adjusting the characters based on the feedback from the ***‘guess’*** method. On the other hand, the Brute-Force algorithm tests all possible combinations of the 12-character key, which would consumpt a lot of time if the string is getting longer and longer. Besides, both handle invalid inputs correctly, both use the ***‘SecretKey’*** class and ***‘SecretKeyGuesser’*** to ensure that. The logic is the same, is to compare the guesses. The output display is the same between both.

Overall, we seem to meet all the requirements. The code is readable and well-organized. We use comments to provide the purpose of the code we use. But in some scenarios, the code appears robust in handling.