RMIT UNIVERSITY

Assignment 2:

Group Project

COSC2658 - Algorithm & Analysis

By Group 4
Lecturer: Tri Dang

Tran Quoc Trung	s3891724
Huynh Ky Thanh	s3884734
Ho Tran Minh Khoi	s3877653
Tran Minh Khoi	s3916827

Overview

The system is designed in SercretKeyGuesser class with an assessment-required method that is void start(). Then we have three other methods:

- int order(char c): This method will convert from characters ‘R’, ‘M’, ‘I’ and ‘T’ to numbers 0, 1, 2 and 3 respectively in order to easily change character value by adding 1 to the number.
- char charOf(int order): This method will convert from numbers 0, 1, 2 and 3 to characters ‘R’, ‘M’, ‘I’ and ‘T’ respectively.
- String next(String str, int index): this method will take two parameters which are a string standing for the guessing key and an int standing for the index of character of the guessing key. This method will change the character at the index of the guessing key in the following order: ‘R’ to ‘M’, ‘M’ to ‘I’ and ‘I’ to ‘T by using the order method, then add 1 to the returned number and use the charOf method to convert back to character.

Data Structures and Algorithm

We applied a Decrease and Conquer approach to this problem. The algorithm will solve the key from left to right. Therefore, it is a decrease-by-a-constant algorithm when the unsolved key will be decreased by one after each iteration through the key’s letters.

Our algorithm pseudo-code:

```
key = new SecretKey();
guessingKey = "RRRRRRRRRRRRRRRR";
n = guessingKey.length()
m = numberOfValidCharacter;

correctLetters = key.guess(guessingKey)

for i from 0 to n-1
      if correctLetters == n then
            return guessingKey
      for j from 0 to m-2
            newStr = guessingKey with character at index i convert to next
character in order of R to M, M to I and I to T
            newCorrectLetters = key.guess(newStr)

            if newCorrectLetters < correctLetters then
                  break
            else if newCorrectLetters > correctLetters then
                  correctLetters = newCorrectLetters
                  guessingKey = newStr
            else
                  guessingKey = newStr
```

Our algorithm will first use the provided guess method in SecretKey class to determine the number of correct letters in the guessing key (the 16-letter key provided in the SecretKeyGuesser class, this key will be originally 16 ‘R’ letters). Then the algorithm will loop through each letter in the guessing key:

At the beginning of each iteration, it will check if the number of correct letters matches the secret key length (which means they are the same), and it will break out of the loop if they are equals.

For each letter, it will use another loop, which loops 3 times (number of valid characters - 1 time), next() method will be called to replace the current letter with the next letter, for example, if the current letter is ‘R’, it will be changed to ‘M’, and it if is ‘M’, it will change to ‘I’ and similarly ‘I’ to ‘T’. After each change, it saves the new guessing key as newStr and uses the guess(newStr) method to get the new number of correct letters (newCorrectLetters) and then compare it with the previous number of correct letters (CorrectLetters). There are 3 cases:

- the newCorrectLetters is smaller than the CorrectLetters: it can claim that the previous letter is correct since changing it makes the number of correct letters smaller, then we will keep the old guessing key and break out of the inner loop the move to the next letter.
- the newCorrectLetters is bigger than the CorrectLetters: the new letter is correct so the guessing key and the CorrectLetters will be updated with the newStr and the newCorrectLetters, and it will break out of the loop to move to the next letter.
- the newCorrectLetters matches the CorrectLetters: it means that the letter is still not correct after the change so we will change it to the next letter until it reaches the correct letter.

After we loop through all the letters or break out from the outer loop, it will print out the guessing key which is also the correct key.

Complexity analysis

```
key = new SecretKey();                                                  {1}
guessingKey = "RRRRRRRRRRRRRRRR";                                       {1}
n = guessingKey.length()                                                {1}
m = numberOfValidCharacter;                                             {1}

correctLetters = key.guess(guessingKey)                                 {1}

for i from 0 to n-1                                                     {n}
    if correctLetters == n then                                         {n}
        return guessingKey                                              {n}

    for j from 0 to m-2                                         {(m-1)*n}
        newStr = guessingKey with character at index i convert to next
character in order of R to M, M to I and I to T                 {(m-1)*n}
        newCorrectLetters = key.guess(newStr)                   {(m-1)*n}
        if newCorrectLetters < correctLetters then              {(m-1)*n}
            break                                               {(m-1)*n}
        else if newCorrectLetters > correctLetters then         {(m-1)*n}
```

```
        correctLetters = newCorrectLetters                    {(m-1)*n}
        guessingKey = newStr                                  {(m-1)*n}
    else                                                      {(m-1)*n}
        guessingKey = newStr                                  {(m-1)*n}
```

❖ For the calling of key.guess() in our algorithm, we consider the time complexity of this method to be constant instead of linear. Although the implementation of the SecretKey.guess() in the sample code is linear, we assumed that the actual implementation of the SecretKey.guess() method is black-box, which means that we are only aware of the input and output of the method without knowing the implementation in order to analyse the time complexity, and the actual implementation of SecretKey.guess() may also have an improvement in the algorithm compared to the sample one.

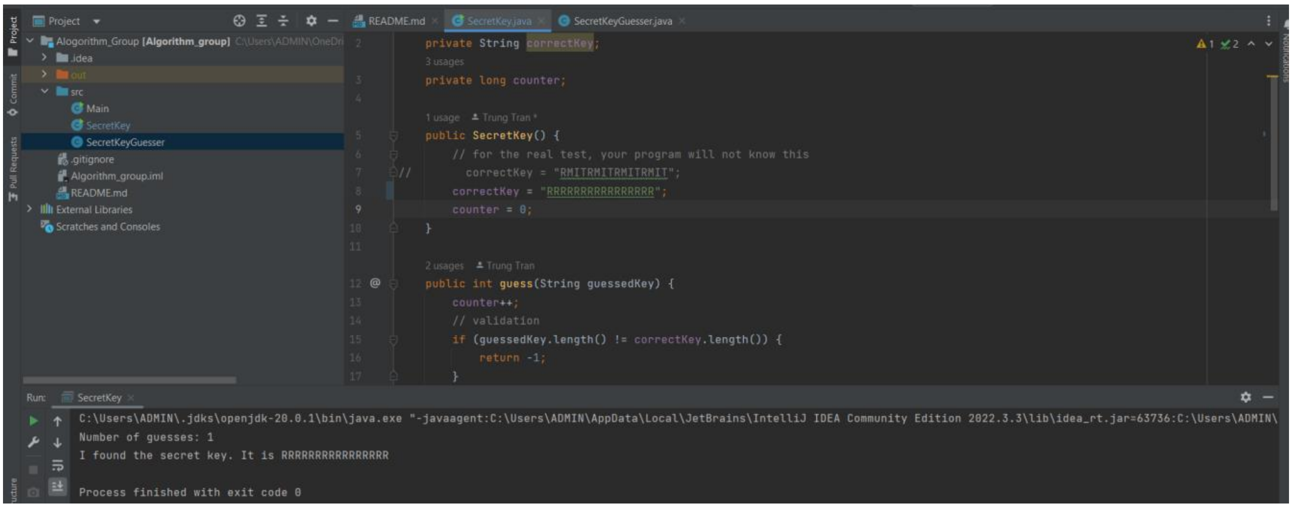
T(n, m) = 5*1 + 3*n + 10*(m-1)*n
= 5 + 3n + 10mn -10n
= 5 - 7n + 10mn

Keep the most significant element and remove the constants, it is n + mn
The big-O of the above algorithm is O(nm)

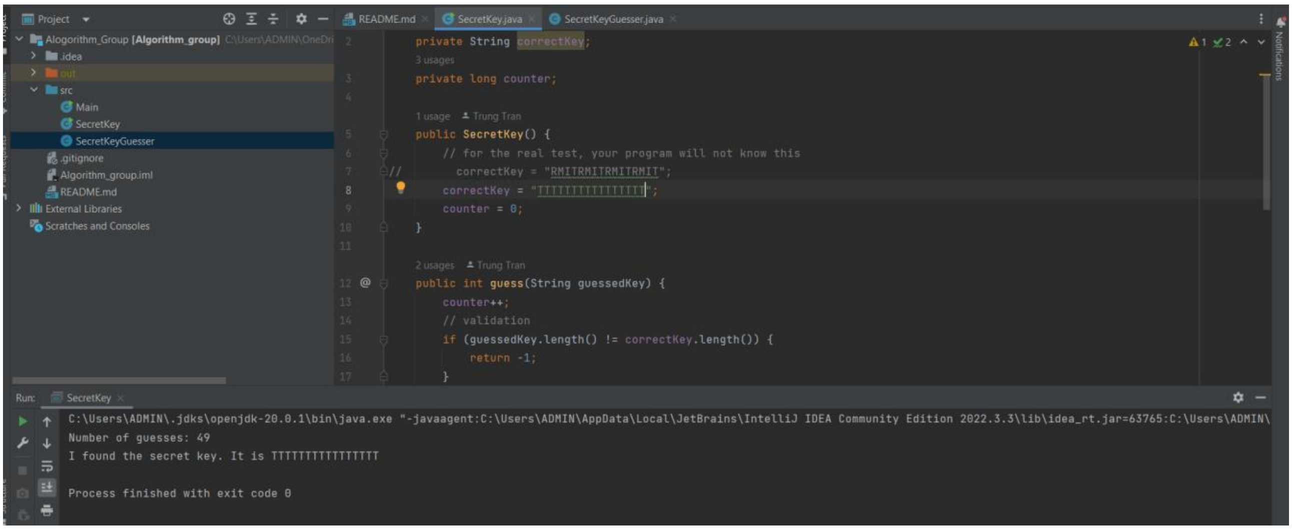
Evaluation

❖ Formula to calculate the number of calling SecretKey.guess method in our algorithm:
(m-1)n + 1 = mn - n + 1 (with m is the number of valid character, n is the key length)

Best case:
When the secret key is exactly the same as the default guessing key with any value of n and m (value of key length and number of valid characters). For example: “RRRRRRRRRRRRRRRR” (16 ‘R’ letters) for the base problem. The code will only call SecretKey.guess once, the outlet loop will break in the first iteration and print out the secret key

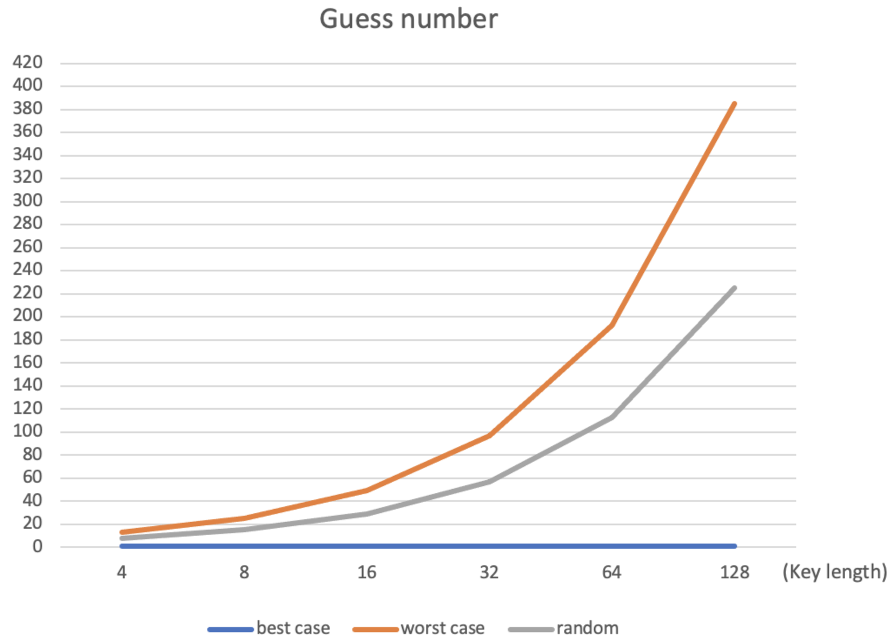


Worst case: worst case happens when all letters from the secret key are the final character in the valid characters list. For example: “TTTTTTTTTTTTTTTT” for the base problem since each letter will have the maximum number of changes. As a result, SecretKey.guess will be called 49 times (1 first time + 3 times for each letter when converse)



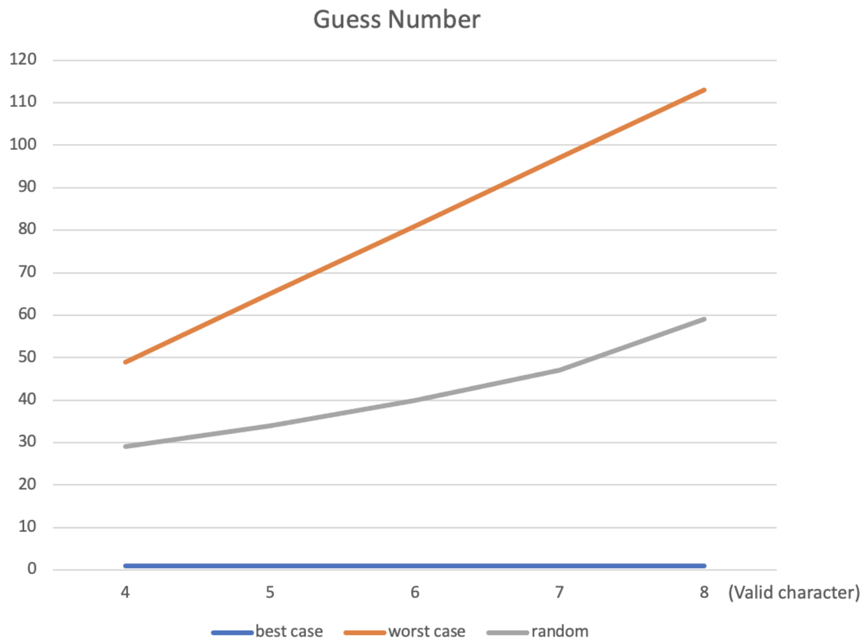
For different key lengths, the number of SecretKey.guess calls in worst cases can be calculated by the mn - n + 1 formula (defined above). The following table and graph illustrate the growth of the number of SecretKey.guess calling with different key lengths in the best case, worst case and random with 4 valid characters defined in the based problem (for random, keys are: “RMIT”, “RMITRMIT”, … depending on the key length)

key length	best case	worst case	random
4	1	13	8
8	1	25	15
16	1	49	29
32	1	97	57
64	1	193	113
128	1	385	225



For the different amount of valid character, the following table and graph illustrate the growth of the number of SecretKey.Guess calling with different number of valid characters in the best case, worst case and random with the key length of 16 in the based problem:

valid character	best case	worst case	random
4	1	49	29
5	1	65	34
6	1	81	40
7	1	97	47
8	1	113	59



Sercet keys that we used to analyse:

valid character	best case	worst case	random
4 (A to D)	AAAAAAAAAAAAAAAA	DDDDDDDDDDDDDDDD	ABCDABCDABCDABCD
5 (A to E)	AAAAAAAAAAAAAAAA	EEEEEEEEEEEEEEEE	ABCDEABCDEABCDEA
6 (A to F)	AAAAAAAAAAAAAAAA	FFFFFFFFFFFFFFFF	ABCDEFABCDEFABCD
7 (A to G)	AAAAAAAAAAAAAAAA	GGGGGGGGGGGGGGGG	ABCDEFGABCDEFGAB
8 (A to H)	AAAAAAAAAAAAAAAA	HHHHHHHHHHHHHHHH	ABCDEFGHABCDEFGH

For the worst-case scenarios with different key length, the number of SecretKey.guess calling (mn - n + 1) and time complexity is O(mn) is much more efficient than applying Brute Force Permutation to try every scenario of the key (the number of SecretKey.Guess calling will be m^n since each letter has m options), which was implemented in the sample code.

key length	Our algorithm	Brute Force Permutation
4	13	256
8	25	65536
16	49	4294967296
32	97	18446744073709552000
64	193	$3.402823669 \times 10^{38}$

And for the worst-case scenarios with different numbers of valid character, the number of SecretKey.guess calling (mn - n + 1) is much more efficient than applying Brute Force Permutation to try every scenario of the key (the number of SecretKey.guess calling will be m^n since each letter has m options), which was implemented in the sample code.

Number Valid character	Our algorithm	Brute Force Permutation
4	49	4294967296
5	25	152587890625
6	49	2821109907456
7	97	33232930569601
8	193	281474976710656