CT255 Assignment 1

Michael Mc Curtin ID: 21459584

Problem 1

> Study the code and summarize its functionality (bullet points will do), thereby referencing important lines of source code.

Main function:

Accepts an argument *arg*, 1 to 64 characters long (**L12**, **L14**). This argument is sent to the hash function **hashF1()** (**L13**).

The function prints the input and resulting hash (**L18**), then starts searching for hash collisions.

hashF1() function:

Takes a string input *s*.

Checks whether the string has the required length (**L36**).

The function then creates a 64 bit string out of the input, adding filler alphabetical characters if necessary. (**L40-41**).

Meanwhile, a 4-digit array *hashA* has been created. (**L30**).

The string is iterated through and *hashA* is populated by multiplying the current character's ASCII value (*byPos*) by different numbers:

```
                    hashA[0] += (byPos * 17); // Note: A += B means A = A + B
                    hashA[1] += (byPos * 31);
                    hashA[2] += (byPos * 101);
                    hashA[3] += (byPos * 79);
```

Next, each digit of *hashA* is replaced with itself modulo 255 (**L51-54**).

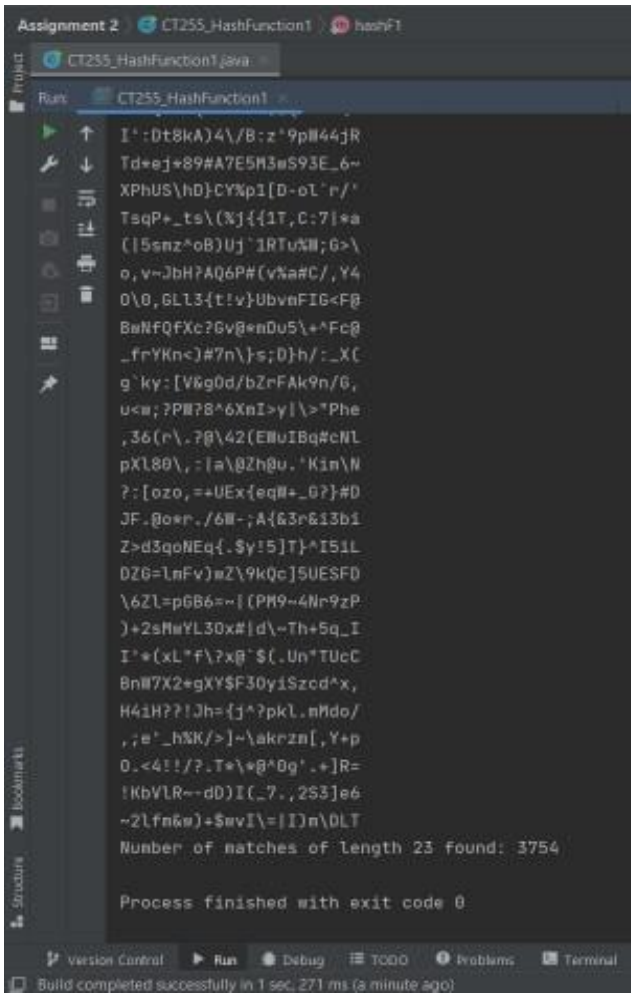
Finally, each digit of *hashA* is multiplied by (256 ^ digit position). These numbers are added together to form the hash. If the hash is negative, it is multiplied by -1. This value is then returned.

Problem 2

> Consider the input “Bamb0” (i.e. “Bamb” followed by a zero). The resulting hash value is 1079524045. Enhance the code to search for “Bamb0” hash collisions (i.e., different inputs that create the same hash value → weak collision resistance) via a brute-force search. What collision(s) can you find?

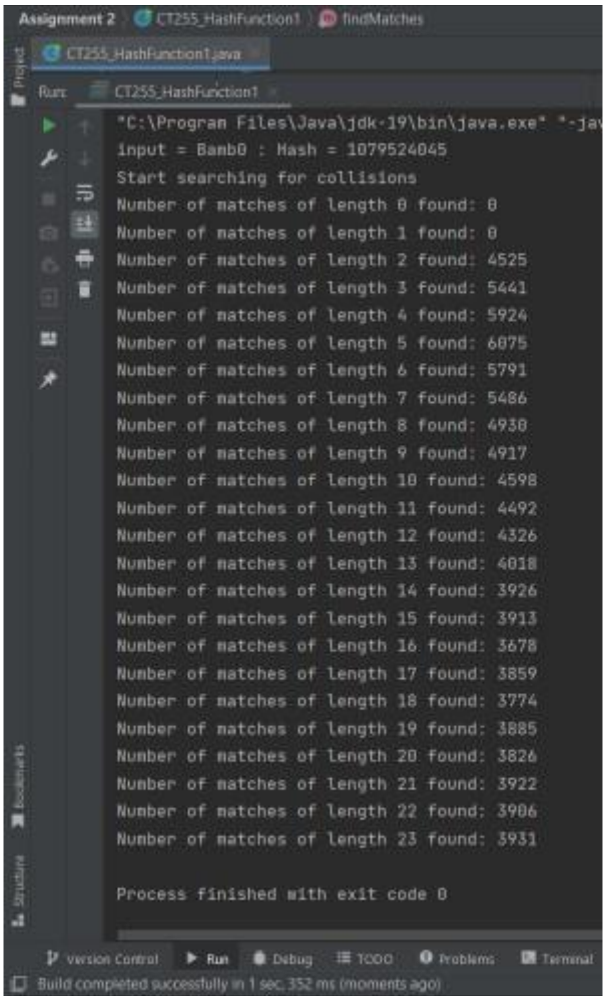
I wrote the following **findMatches()** function to perform this task:

```
    private static void findMatches(int stringLength){

        //  initialise variables

        char[] randomString =new char[stringLength];
        int matchCounter =0;


        // make 999999 attempts at finding a match

        for (int i=0; i < 999999; i++) {

            // populate the random string

            for (int j = 0; j < stringLength; j++) {
                 Random rnd =new Random();

                 String alphabet ="!\"#$%&\\'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\\\]^_`abcdefghijklmnopqrstuvwxyz{|}~?;/.,"

                char c = alphabet.charAt(rnd.nextInt(alphabet.length()));// pick
a random character from the alphabet
                 randomString[j] = c;

             }


            int result = hashF1(String.valueOf(randomString));

            // check if the generated hash matches our desired hash

            if (result ==1079524045) {
                /* System.out.println(String.format("%s",
String.valueOf(randomString))); // print out string (optional)*/
                 matchCounter ++;
             }


        }

        System.out.println(String.format("Number of matches of length %d found:
%d", stringLength, matchCounter));

    }
```



```
I':Dt8kA)4\/B:z'9p#44jR
Td*ej+89#A7E5M3mS93E_6~
XPhUS\hD}CY%p1[D-oL`r/'
TsqP+_ts\(%j{{1T,C:7|*a
(|5smz^oB)Uj`1RTu%#;G>\
o,v~JbH?AQ6P#(v%a#C/,Y4
O\0,GLl3{t!v}UbvmFIG<F@
BmNfQfXc?Gv@*mDu5\+^Fc@
_frYKn<)#7n\}s;D}h/:_X(
g`ky:[V&gOd/bZrFAk9n/G,
u<m;?P#?8^6XmI>y|\>"Phe
,36(r\.?@\42(E#uIBq#cNL
pXl80\,:|a\@Zh@u.'Kim\N
?:[ozo,=+UEx{eq#+_G?}#D
JF.@o*r./6#-;A{&3r&i3bi
Z>d3qoNEq{.$y!5]T}^I5iL
DZG=lmFv)mZ\9kQc]5UESFD
\6Zl=pGB6=~|(PM9~4Nr9zP
)+2sMmYL3Ox#|d\~Th+5q_I
I'*(xL"f\?x@`$(.Un"TUcC
Bn#7X2+gXY$F3OyiSzod^x,
H4iH??!Jh={j^?pkl.mMdo/
,;e'_h%K/>]~\akrzm[,Y+p
O.<4!!/?.T*\*@^Og'.+]R=
!KbVlR~-dD)I(_7.,2S3]e6
~2lfm&m)+$mvI\=|I)m\DLT
Number of matches of length 23 found: 3754

Process finished with exit code 0
```

Build completed successfully in 1 sec, 271 ms (a minute ago)



```
"C:\Program Files\Java\jdk-19\bin\java.exe" "-jav
input = Bamb0 : Hash = 1079524045
Start searching for collisions
Number of matches of length 0 found: 0
Number of matches of length 1 found: 0
Number of matches of length 2 found: 4525
Number of matches of length 3 found: 5441
Number of matches of length 4 found: 5924
Number of matches of length 5 found: 6075
Number of matches of length 6 found: 5791
Number of matches of length 7 found: 5486
Number of matches of length 8 found: 4930
Number of matches of length 9 found: 4917
Number of matches of length 10 found: 4598
Number of matches of length 11 found: 4492
Number of matches of length 12 found: 4326
Number of matches of length 13 found: 4018
Number of matches of length 14 found: 3926
Number of matches of length 15 found: 3913
Number of matches of length 16 found: 3678
Number of matches of length 17 found: 3859
Number of matches of length 18 found: 3774
Number of matches of length 19 found: 3885
Number of matches of length 20 found: 3826
Number of matches of length 21 found: 3922
Number of matches of length 22 found: 3906
Number of matches of length 23 found: 3931

Process finished with exit code 0
```

Build completed successfully in 1 sec, 352 ms (moments ago)

Problem 3

> Enhance the code in hashF1() to make it more robust, i.e., to reduce the risk of hash collisions. Explain your answer via comments in your code.

To further secure **hashF1()**, I added the following code before the modulo operations:

```
            hashA[0] = extraEncryption(hashA[0]);
            hashA[0] = extraEncryption(hashA[1]);
            hashA[0] = extraEncryption(hashA[2]);
            hashA[0] = extraEncryption(hashA[3]);
```

This calls my **extraEncryption()** function:

```
    private static int extraEncryption(int x) {
        // perform 5 bit left shift on the number then subtract the original to
get multiplication by 31
        // similar to Java's own String.hashCode() function
        return ( x << 5 - (x));
    }
```

which simply performs a 5-bit bitwise left shift on the operand then subtracts the original operand from the result.

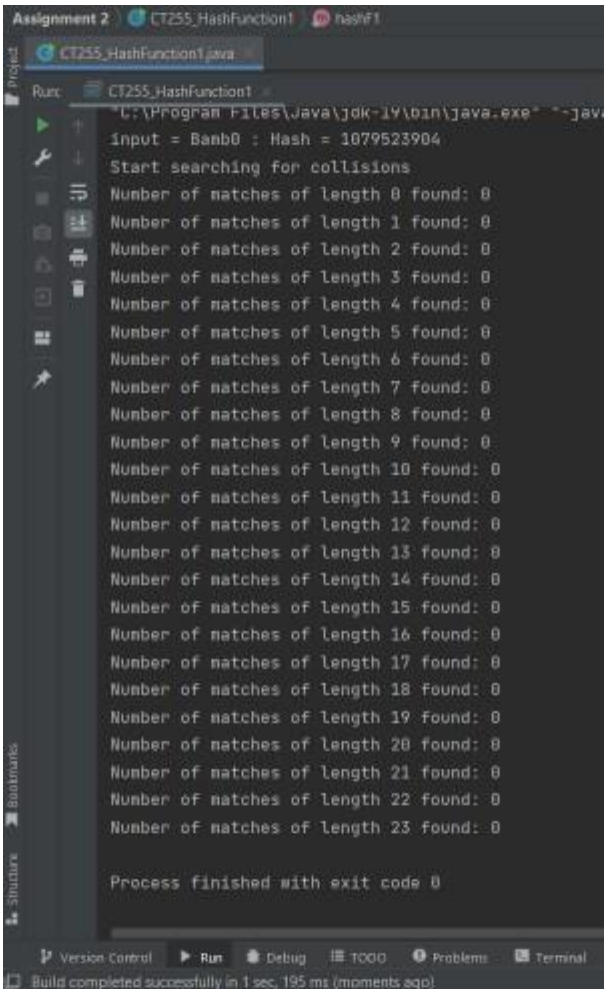
I chose this particular operation because it results in the multiplication of the operand by 31. Java's own **String.hashCode()** function

```
    s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
```

performs a similar multiplication.

Various reasons behind this multiplication by 31 have been proposed, some due to performance and some due to convenience.

However, in 1997, Goodrich and Tamassia computed that creating a hash code using a cyclic shift of 31, 33, 37, 39, or 41 bits produced fewer than 7 hash collisions when generating hashes from over 50,000 English words. [*Data Structures and Algorithms in Java, 9.2.3*]



```
"C:\Program Files\Java\jdk-19\bin\java.exe" "-jav
input = Bamb0 : Hash = 1079523904
Start searching for collisions
Number of matches of length 0 found: 0
Number of matches of length 1 found: 0
Number of matches of length 2 found: 0
Number of matches of length 3 found: 0
Number of matches of length 4 found: 0
Number of matches of length 5 found: 0
Number of matches of length 6 found: 0
Number of matches of length 7 found: 0
Number of matches of length 8 found: 0
Number of matches of length 9 found: 0
Number of matches of length 10 found: 0
Number of matches of length 11 found: 0
Number of matches of length 12 found: 0
Number of matches of length 13 found: 0
Number of matches of length 14 found: 0
Number of matches of length 15 found: 0
Number of matches of length 16 found: 0
Number of matches of length 17 found: 0
Number of matches of length 18 found: 0
Number of matches of length 19 found: 0
Number of matches of length 20 found: 0
Number of matches of length 21 found: 0
Number of matches of length 22 found: 0
Number of matches of length 23 found: 0

Process finished with exit code 0
```

Build completed successfully in 1 sec, 195 ms (moments ago)

My testing shows that applying the **extraEncryption()** function results in a significantly lower number of hash collisions, therefore I am satisfied that it achieves the desired result.