ENSE 496AE Assignment Report

Group D-Algo_Crypto_image-in-audio

March 26 2020

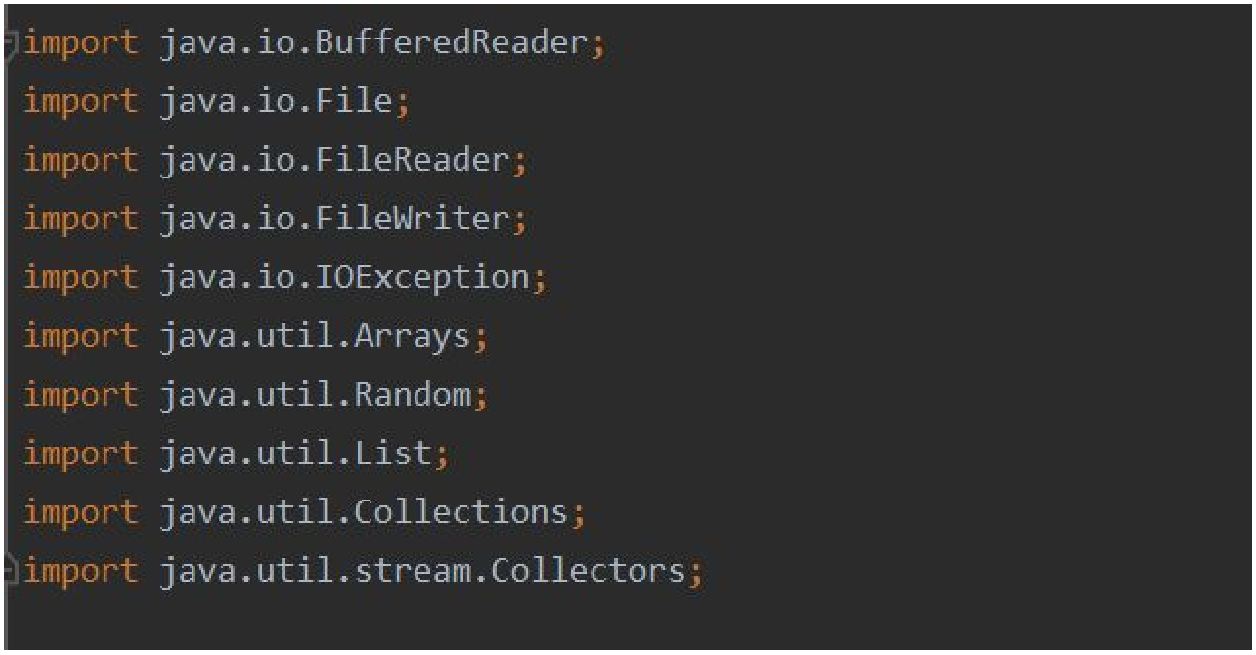
Jeremy Cross 200319513
Bipin Suram 200439429
Taylen Jones 200354271

Programming language used: JAVA

Github:
https://github.com/JeremyC238/ENSE-496AE-Presentations/tree/master/ENSE_496AE_Assignment_4

First Requirement: Generating an array of random integers

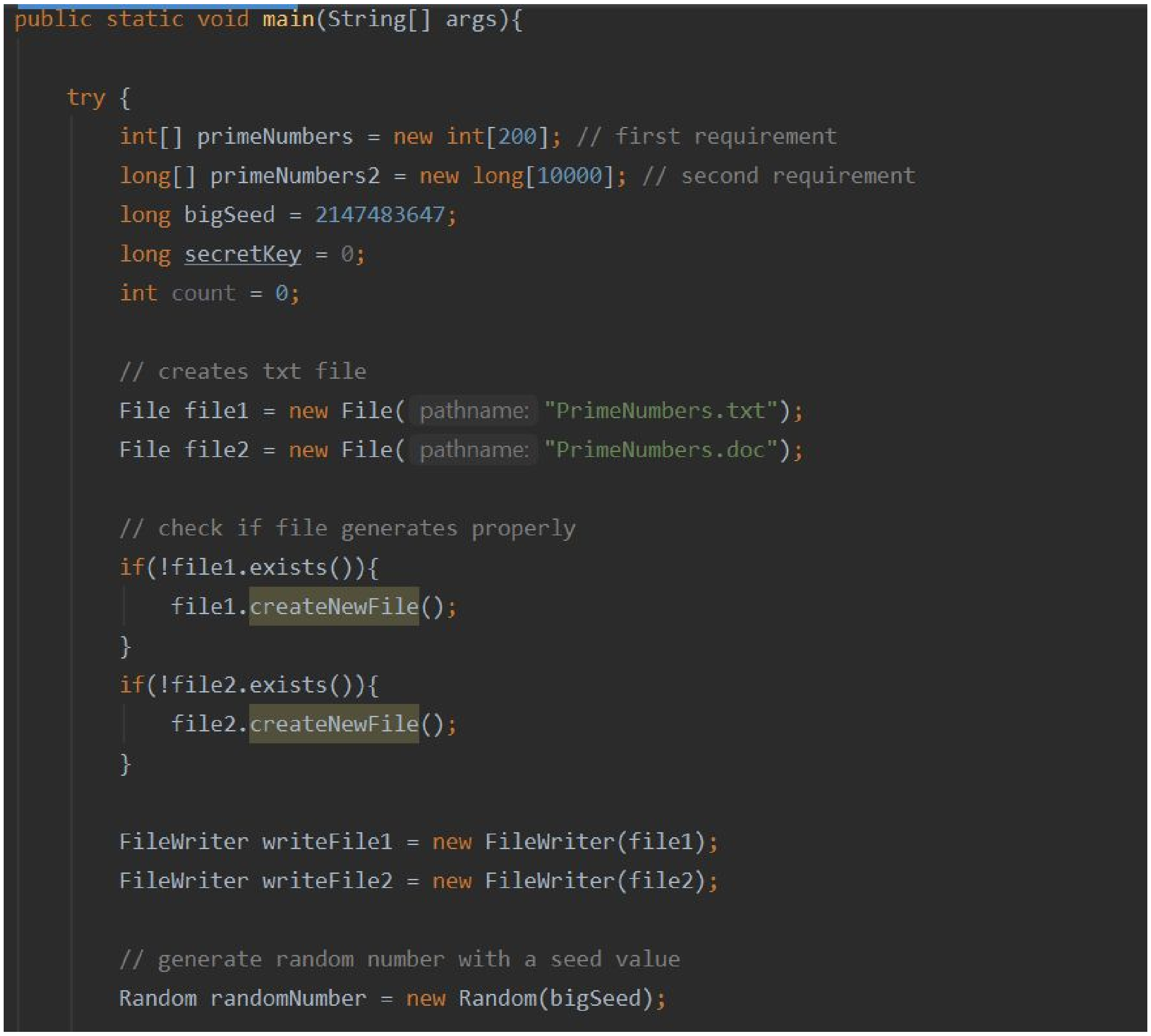
Below are the libraries that we have imported to execute all requirements.



```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Arrays;
import java.util.Random;
import java.util.List;
import java.util.Collections;
import java.util.stream.Collectors;
```

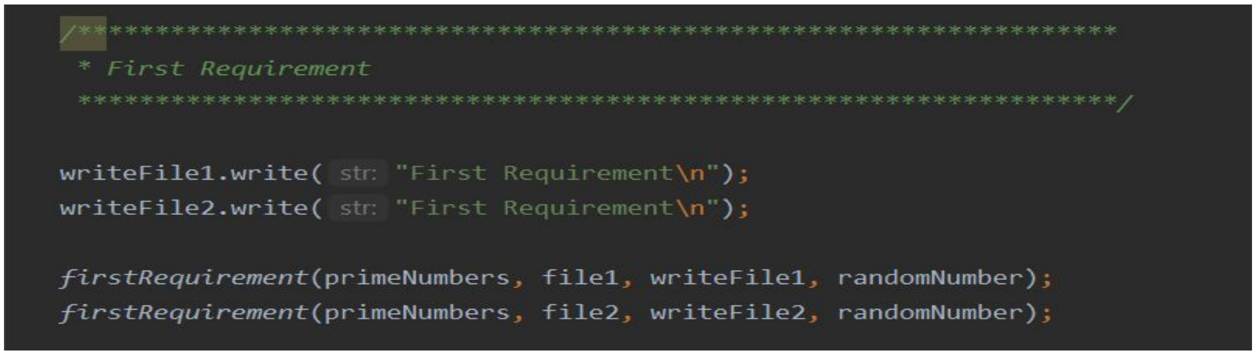
For the first requirement, we have generated 200 random integers and for the second requirement we have generated 10000 prime numbers.
Two files (one text and one docx.) are created to save the output into them.
The bigseed value taken is 2147483647 which is a prime number and is the largest prime that our program supports.

‘Random’ is the function that generates the random values with the seed value.



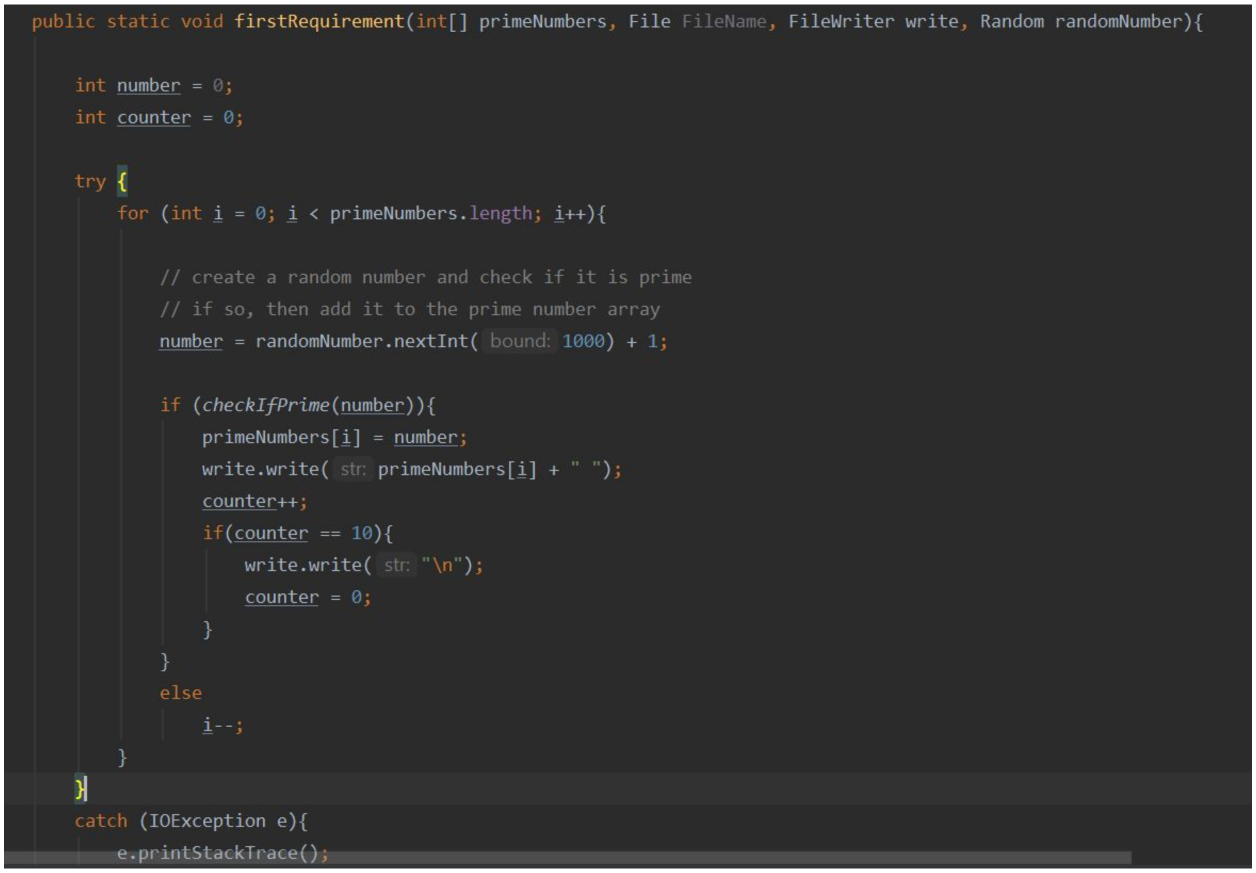
```
public static void main(String[] args){

    try {
        int[] primeNumbers = new int[200]; // first requirement
        long[] primeNumbers2 = new long[10000]; // second requirement
        long bigSeed = 2147483647;
        long secretKey = 0;
        int count = 0;

        // creates txt file
        File file1 = new File( pathname: "PrimeNumbers.txt");
        File file2 = new File( pathname: "PrimeNumbers.doc");

        // check if file generates properly
        if(!file1.exists()){
            file1.createNewFile();
        }
        if(!file2.exists()){
            file2.createNewFile();
        }

        FileWriter writeFile1 = new FileWriter(file1);
        FileWriter writeFile2 = new FileWriter(file2);

        // generate random number with a seed value
        Random randomNumber = new Random(bigSeed);
```

Below methods are called with the arguments mentioned.



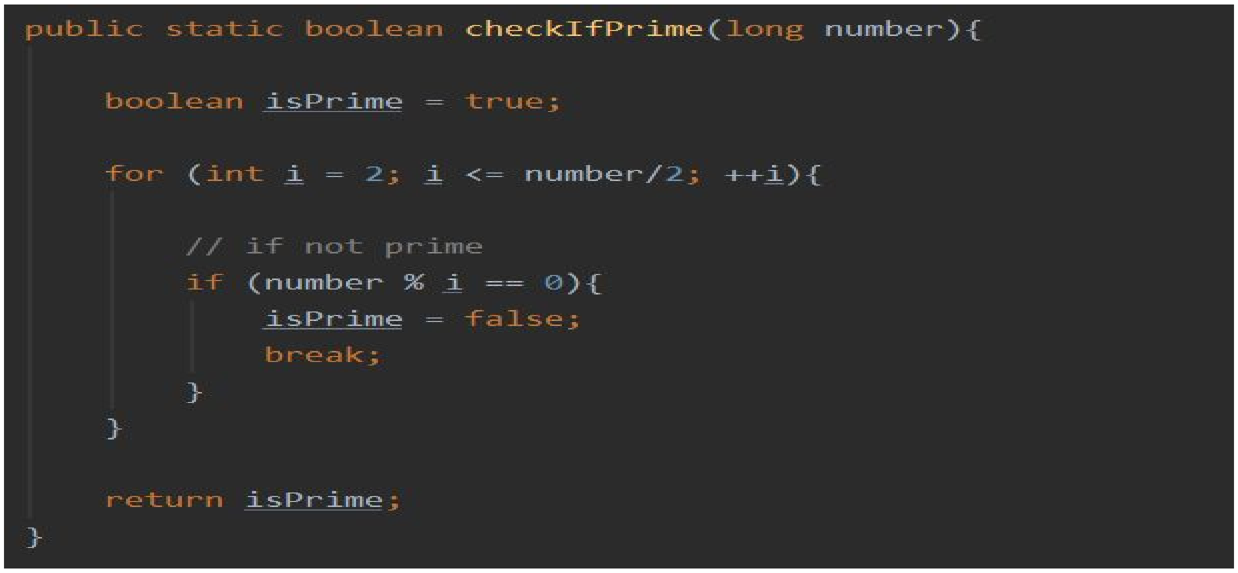
```
/*****************************************************************
 * First Requirement
 *****************************************************************/

writeFile1.write( str: "First Requirement\n");
writeFile2.write( str: "First Requirement\n");

firstRequirement(primeNumbers, file1, writeFile1, randomNumber);
firstRequirement(primeNumbers, file2, writeFile2, randomNumber);
```

The program generates the list of random prime numbers below 1000 and writes the values to the file created.



```java
public static void firstRequirement(int[] primeNumbers, File FileName, FileWriter write, Random randomNumber){

    int number = 0;
    int counter = 0;

    try {
        for (int i = 0; i < primeNumbers.length; i++){

            // create a random number and check if it is prime
            // if so, then add it to the prime number array
            number = randomNumber.nextInt( bound: 1000) + 1;

            if (checkIfPrime(number)){
                primeNumbers[i] = number;
                write.write( str: primeNumbers[i] + " ");
                counter++;
                if(counter == 10){
                    write.write( str: "\n");
                    counter = 0;
                }
            }
            else
                i--;
        }
    }
    catch (IOException e){
        e.printStackTrace();
```

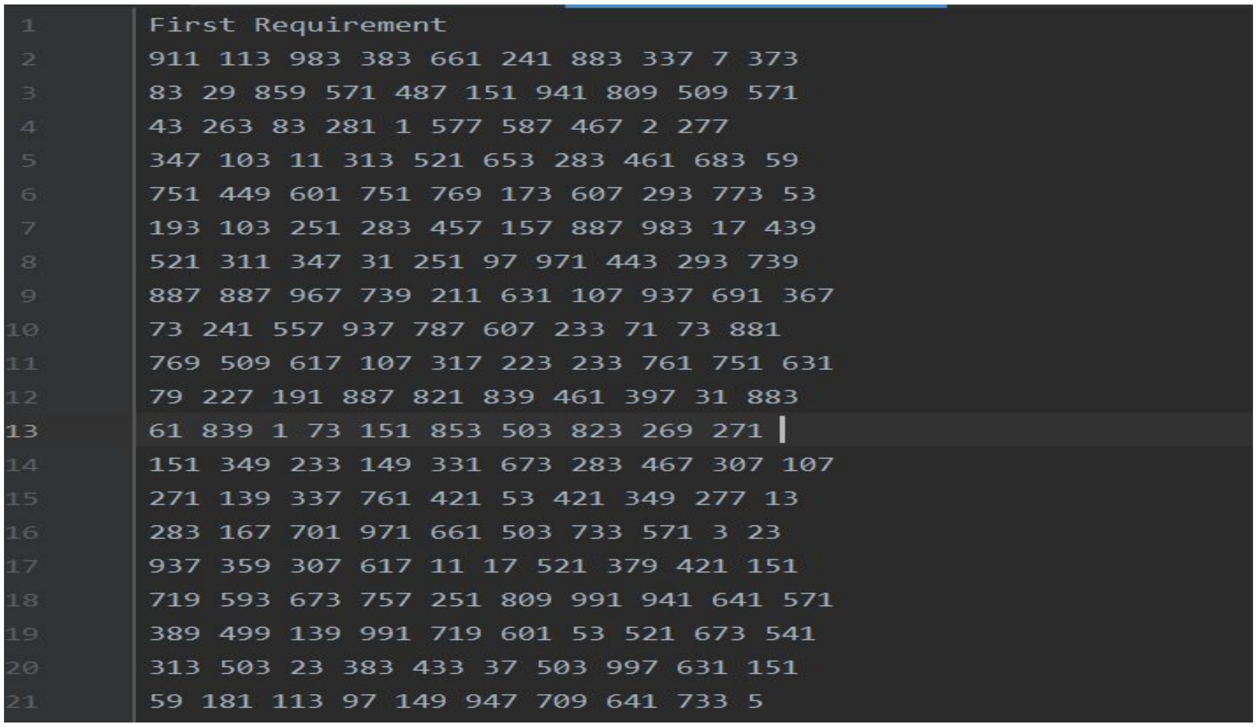
The code below checks if the ‘number’ is prime or not by the trial division approach. The program checks till the half of the current ‘number’ i.e., number/2 if it is divisible.



```java
public static boolean checkIfPrime(long number){

    boolean isPrime = true;

    for (int i = 2; i <= number/2; ++i){

        // if not prime
        if (number % i == 0){
            isPrime = false;
            break;
        }
    }

    return isPrime;
}
```

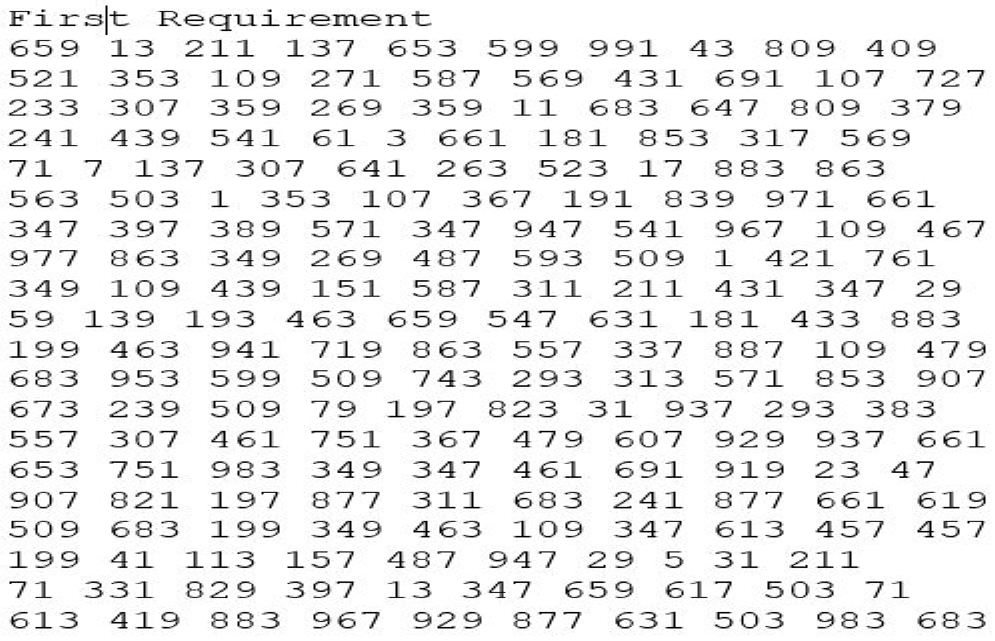
The output of the first requirement is as shown:

Output in the text document.



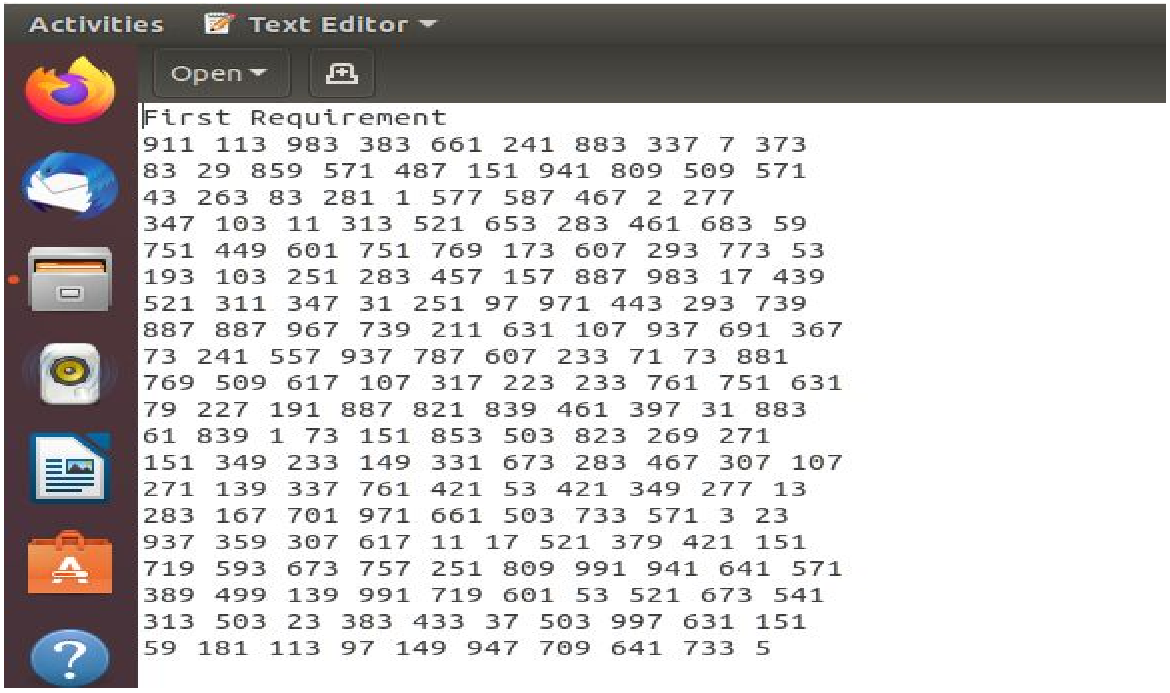
```
First Requirement
911 113 983 383 661 241 883 337 7 373
83 29 859 571 487 151 941 809 509 571
43 263 83 281 1 577 587 467 2 277
347 103 11 313 521 653 283 461 683 59
751 449 601 751 769 173 607 293 773 53
193 103 251 283 457 157 887 983 17 439
521 311 347 31 251 97 971 443 293 739
887 887 967 739 211 631 107 937 691 367
73 241 557 937 787 607 233 71 73 881
769 509 617 107 317 223 233 761 751 631
79 227 191 887 821 839 461 397 31 883
61 839 1 73 151 853 503 823 269 271
151 349 233 149 331 673 283 467 307 107
271 139 337 761 421 53 421 349 277 13
283 167 701 971 661 503 733 571 3 23
937 359 307 617 11 17 521 379 421 151
719 593 673 757 251 809 991 941 641 571
389 499 139 991 719 601 53 521 673 541
313 503 23 383 433 37 503 997 631 151
59 181 113 97 149 947 709 641 733 5
```

Interesting part is that the random function generates different set of random numbers in the 'docx' file as shown

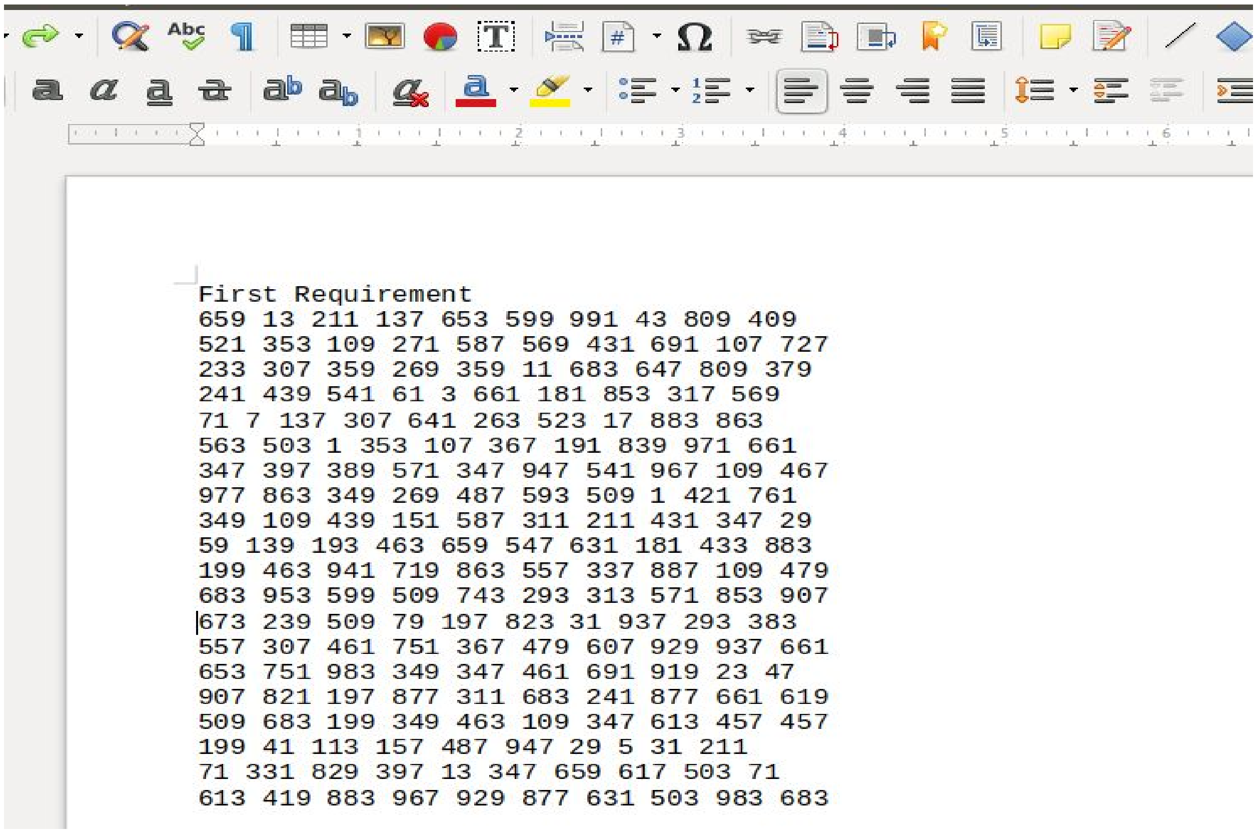


```
First Requirement
659 13 211 137 653 599 991 43 809 409
521 353 109 271 587 569 431 691 107 727
233 307 359 269 359 11 683 647 809 379
241 439 541 61 3 661 181 853 317 569
71 7 137 307 641 263 523 17 883 863
563 503 1 353 107 367 191 839 971 661
347 397 389 571 347 947 541 967 109 467
977 863 349 269 487 593 509 1 421 761
349 109 439 151 587 311 211 431 347 29
59 139 193 463 659 547 631 181 433 883
199 463 941 719 863 557 337 887 109 479
683 953 599 509 743 293 313 571 853 907
673 239 509 79 197 823 31 937 293 383
557 307 461 751 367 479 607 929 937 661
653 751 983 349 347 461 691 919 23 47
907 821 197 877 311 683 241 877 661 619
509 683 199 349 463 109 347 613 457 457
199 41 113 157 487 947 29 5 31 211
71 331 829 397 13 347 659 617 503 71
613 419 883 967 929 877 631 503 983 683
```

As asked, we have executed the same program on different OS(Linux) and the output remains same as long as the seed value doesn't change.

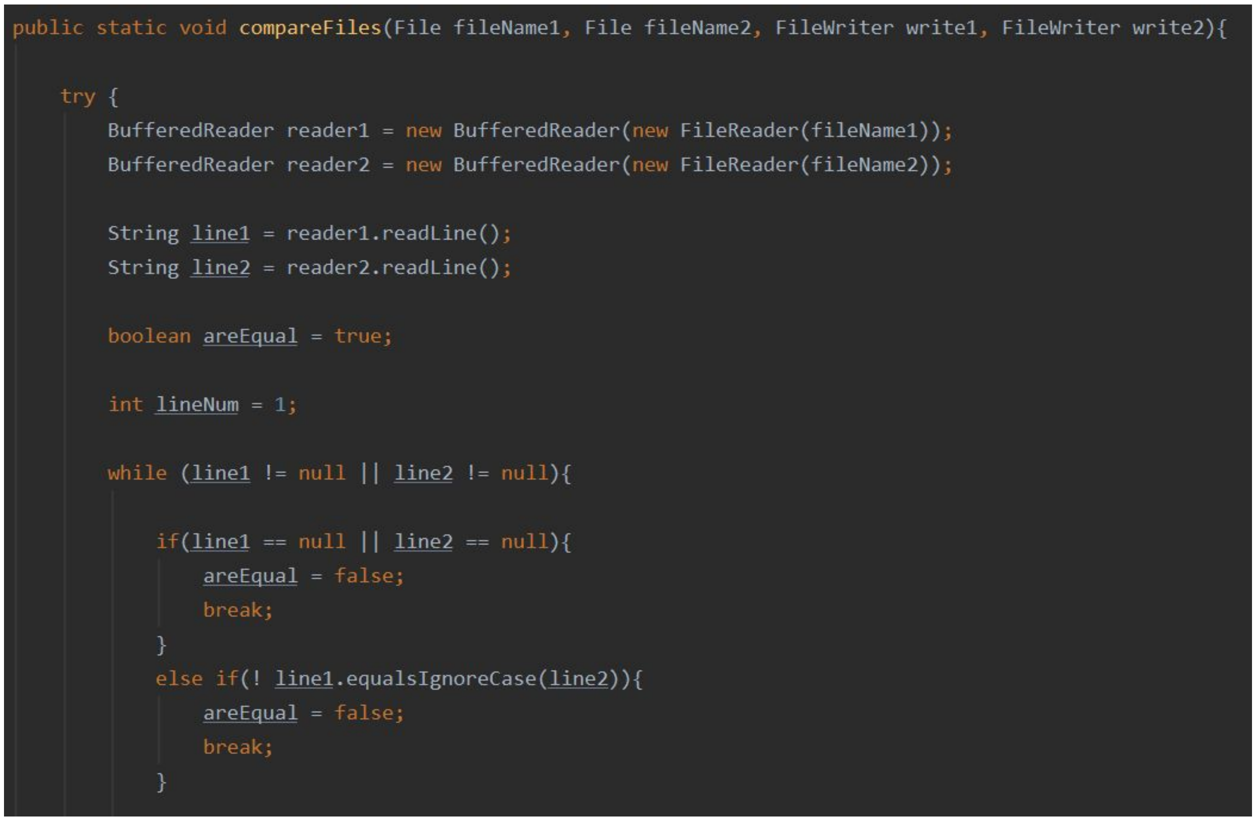


```
First Requirement
911 113 983 383 661 241 883 337 7 373
83 29 859 571 487 151 941 809 509 571
43 263 83 281 1 577 587 467 2 277
347 103 11 313 521 653 283 461 683 59
751 449 601 751 769 173 607 293 773 53
193 103 251 283 457 157 887 983 17 439
521 311 347 31 251 97 971 443 293 739
887 887 967 739 211 631 107 937 691 367
73 241 557 937 787 607 233 71 73 881
769 509 617 107 317 223 233 761 751 631
79 227 191 887 821 839 461 397 31 883
61 839 1 73 151 853 503 823 269 271
151 349 233 149 331 673 283 467 307 107
271 139 337 761 421 53 421 349 277 13
283 167 701 971 661 503 733 571 3 23
937 359 307 617 11 17 521 379 421 151
719 593 673 757 251 809 991 941 641 571
389 499 139 991 719 601 53 521 673 541
313 503 23 383 433 37 503 997 631 151
59 181 113 97 149 947 709 641 733 5
```

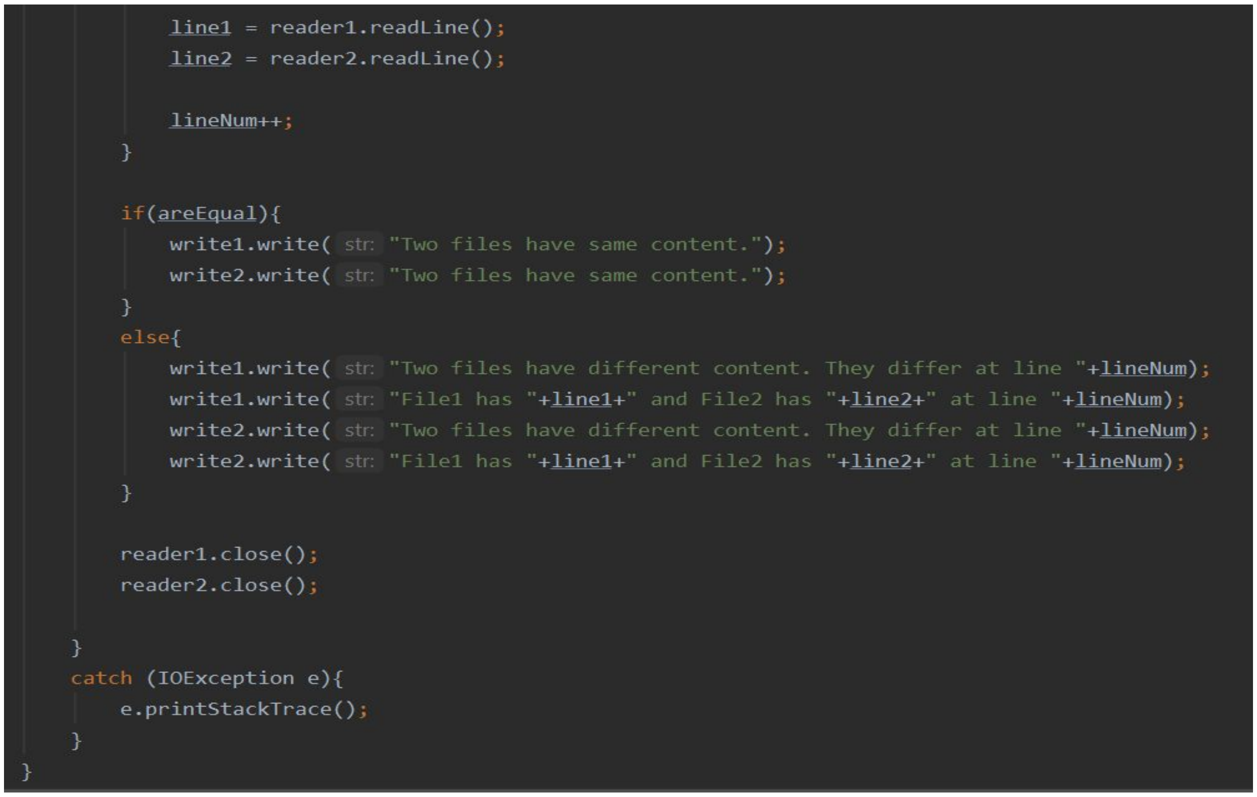


```
First Requirement
659 13 211 137 653 599 991 43 809 409
521 353 109 271 587 569 431 691 107 727
233 307 359 269 359 11 683 647 809 379
241 439 541 61 3 661 181 853 317 569
71 7 137 307 641 263 523 17 883 863
563 503 1 353 107 367 191 839 971 661
347 397 389 571 347 947 541 967 109 467
977 863 349 269 487 593 509 1 421 761
349 109 439 151 587 311 211 431 347 29
59 139 193 463 659 547 631 181 433 883
199 463 941 719 863 557 337 887 109 479
683 953 599 509 743 293 313 571 853 907
673 239 509 79 197 823 31 937 293 383
557 307 461 751 367 479 607 929 937 661
653 751 983 349 347 461 691 919 23 47
907 821 197 877 311 683 241 877 661 619
509 683 199 349 463 109 347 613 457 457
199 41 113 157 487 947 29 5 31 211
71 331 829 397 13 347 659 617 503 71
613 419 883 967 929 877 631 503 983 683
```

The screenshots below shows the code to compare any two given files.

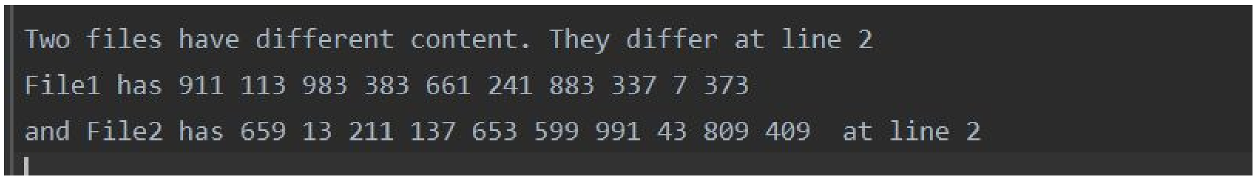


```java
public static void compareFiles(File fileName1, File fileName2, FileWriter write1, FileWriter write2){

    try {
        BufferedReader reader1 = new BufferedReader(new FileReader(fileName1));
        BufferedReader reader2 = new BufferedReader(new FileReader(fileName2));

        String line1 = reader1.readLine();
        String line2 = reader2.readLine();

        boolean areEqual = true;

        int lineNum = 1;

        while (line1 != null || line2 != null){

            if(line1 == null || line2 == null){
                areEqual = false;
                break;
            }
            else if(! line1.equalsIgnoreCase(line2)){
                areEqual = false;
                break;
            }
```



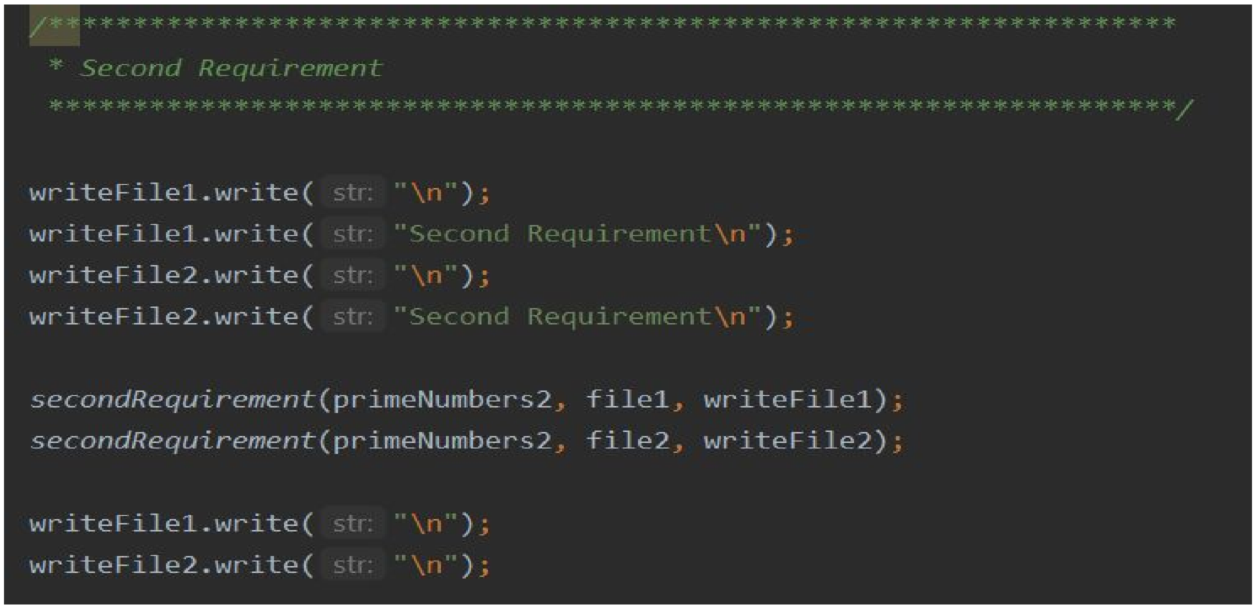
```java
            line1 = reader1.readLine();
            line2 = reader2.readLine();

            lineNum++;
        }

        if(areEqual){
            write1.write( str: "Two files have same content.");
            write2.write( str: "Two files have same content.");
        }
        else{
            write1.write( str: "Two files have different content. They differ at line "+lineNum);
            write1.write( str: "File1 has "+line1+" and File2 has "+line2+" at line "+lineNum);
            write2.write( str: "Two files have different content. They differ at line "+lineNum);
            write2.write( str: "File1 has "+line1+" and File2 has "+line2+" at line "+lineNum);
        }

        reader1.close();
        reader2.close();

    }
    catch (IOException e){
        e.printStackTrace();
    }
}
```

The above program compares the files byte by byte and throws the output as



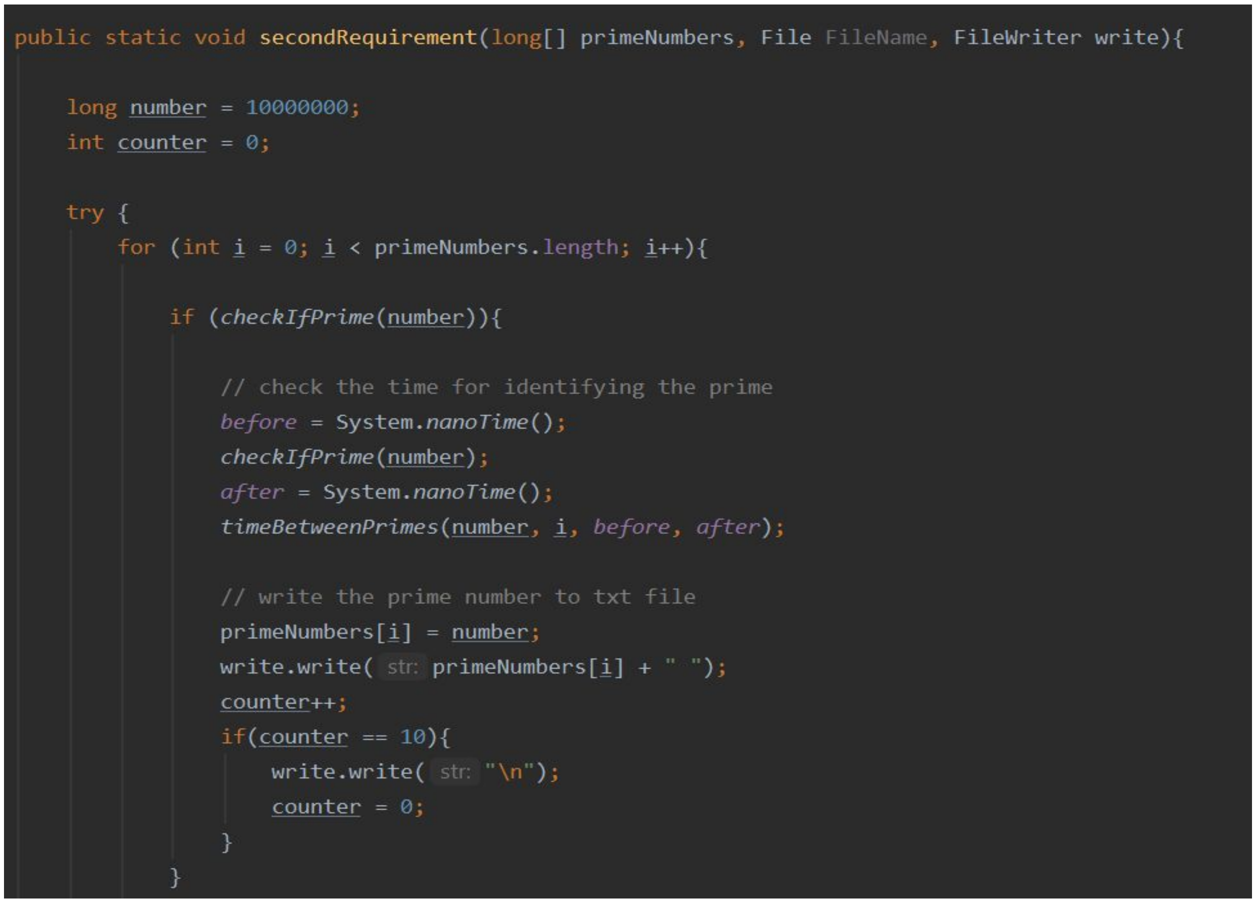
```
Two files have different content. They differ at line 2
File1 has 911 113 983 383 661 241 883 337 7 373
and File2 has 659 13 211 137 653 599 991 43 809 409  at line 2
```

Second Requirement: Generating prime bank along with the Discovery times and storing in a file.

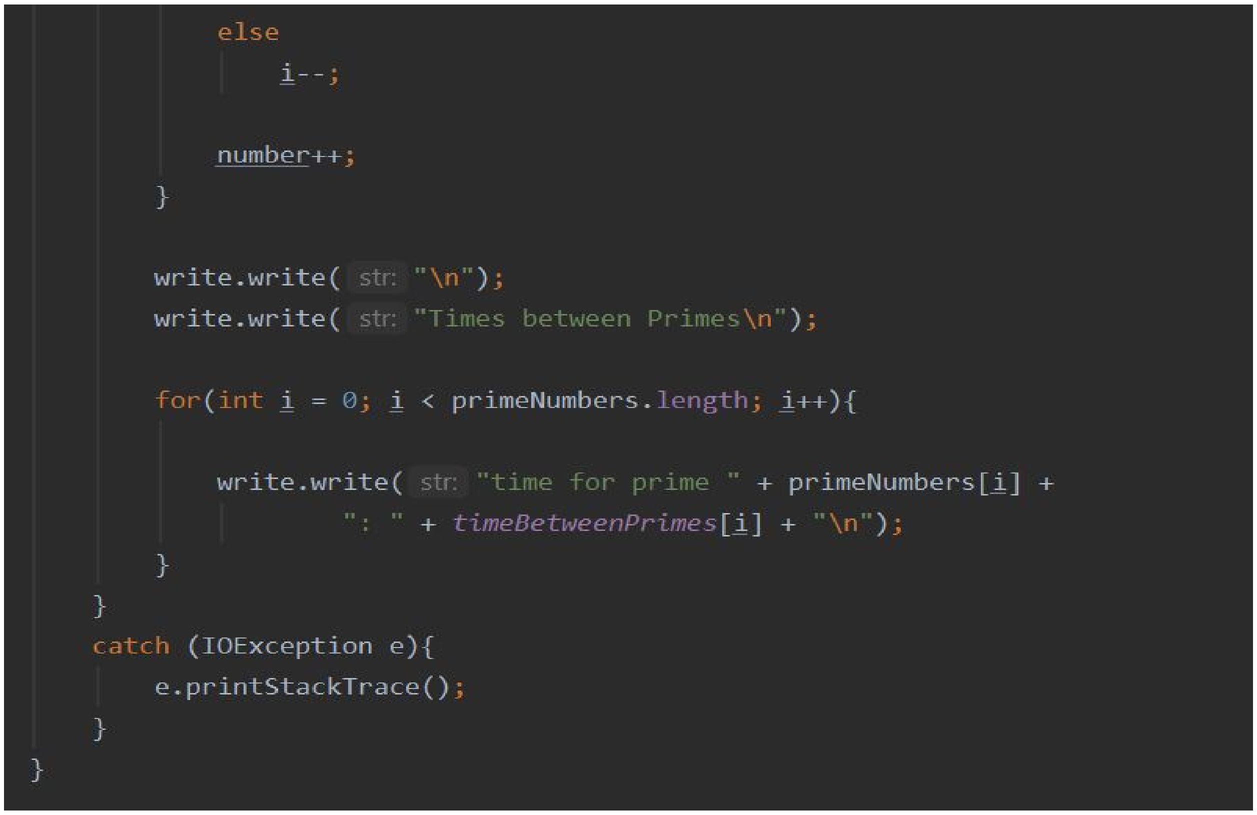


```
/*********************************************************************
 * Second Requirement
 *********************************************************************/

writeFile1.write( str: "\n");
writeFile1.write( str: "Second Requirement\n");
writeFile2.write( str: "\n");
writeFile2.write( str: "Second Requirement\n");

secondRequirement(primeNumbers2, file1, writeFile1);
secondRequirement(primeNumbers2, file2, writeFile2);

writeFile1.write( str: "\n");
writeFile2.write( str: "\n");
```

The method used for this requirement is

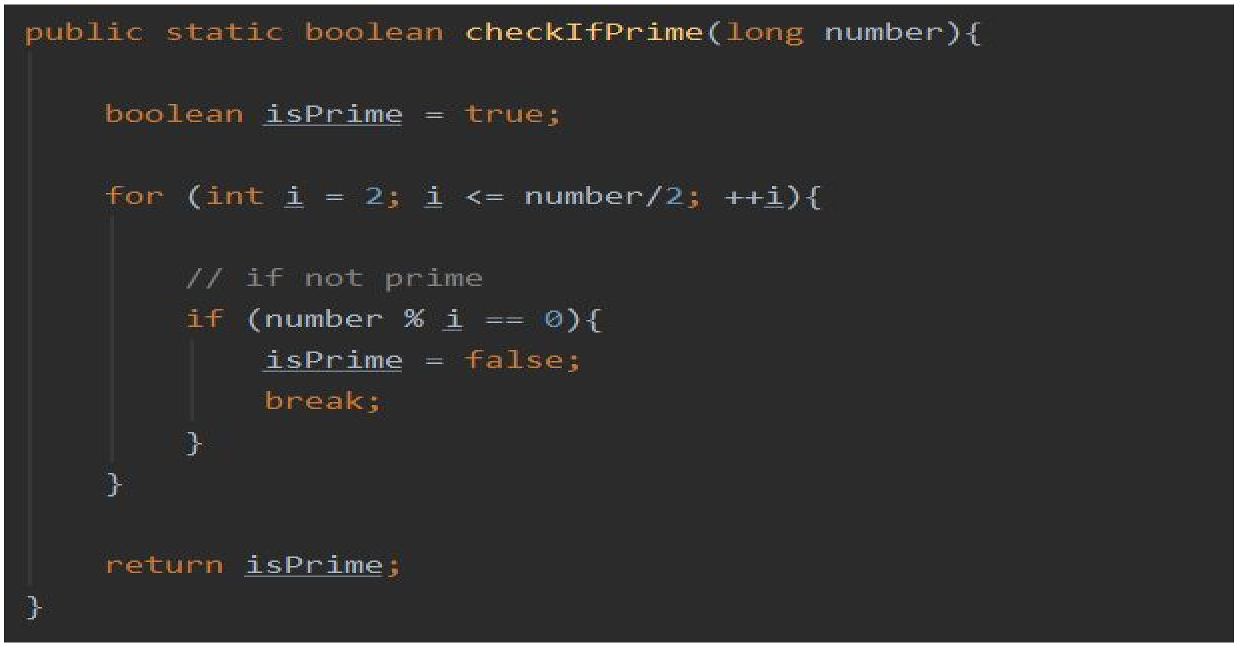


```
public static void secondRequirement(long[] primeNumbers, File FileName, FileWriter write){

    long number = 10000000;
    int counter = 0;

    try {
        for (int i = 0; i < primeNumbers.length; i++){

            if (checkIfPrime(number)){

                // check the time for identifying the prime
                before = System.nanoTime();
                checkIfPrime(number);
                after = System.nanoTime();
                timeBetweenPrimes(number, i, before, after);

                // write the prime number to txt file
                primeNumbers[i] = number;
                write.write( str: primeNumbers[i] + " ");
                counter++;
                if(counter == 10){
                    write.write( str: "\n");
                    counter = 0;
                }
            }
```



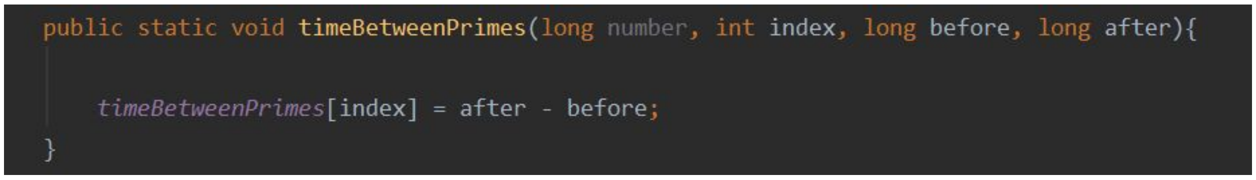
```
                else
                    i--;

                number++;
            }

            write.write( str: "\n");
            write.write( str: "Times between Primes\n");

            for(int i = 0; i < primeNumbers.length; i++){

                write.write( str: "time for prime " + primeNumbers[i] +
                        ": " + timeBetweenPrimes[i] + "\n");
            }
        }
        catch (IOException e){
            e.printStackTrace();
        }
}
```

The code to check the primality is



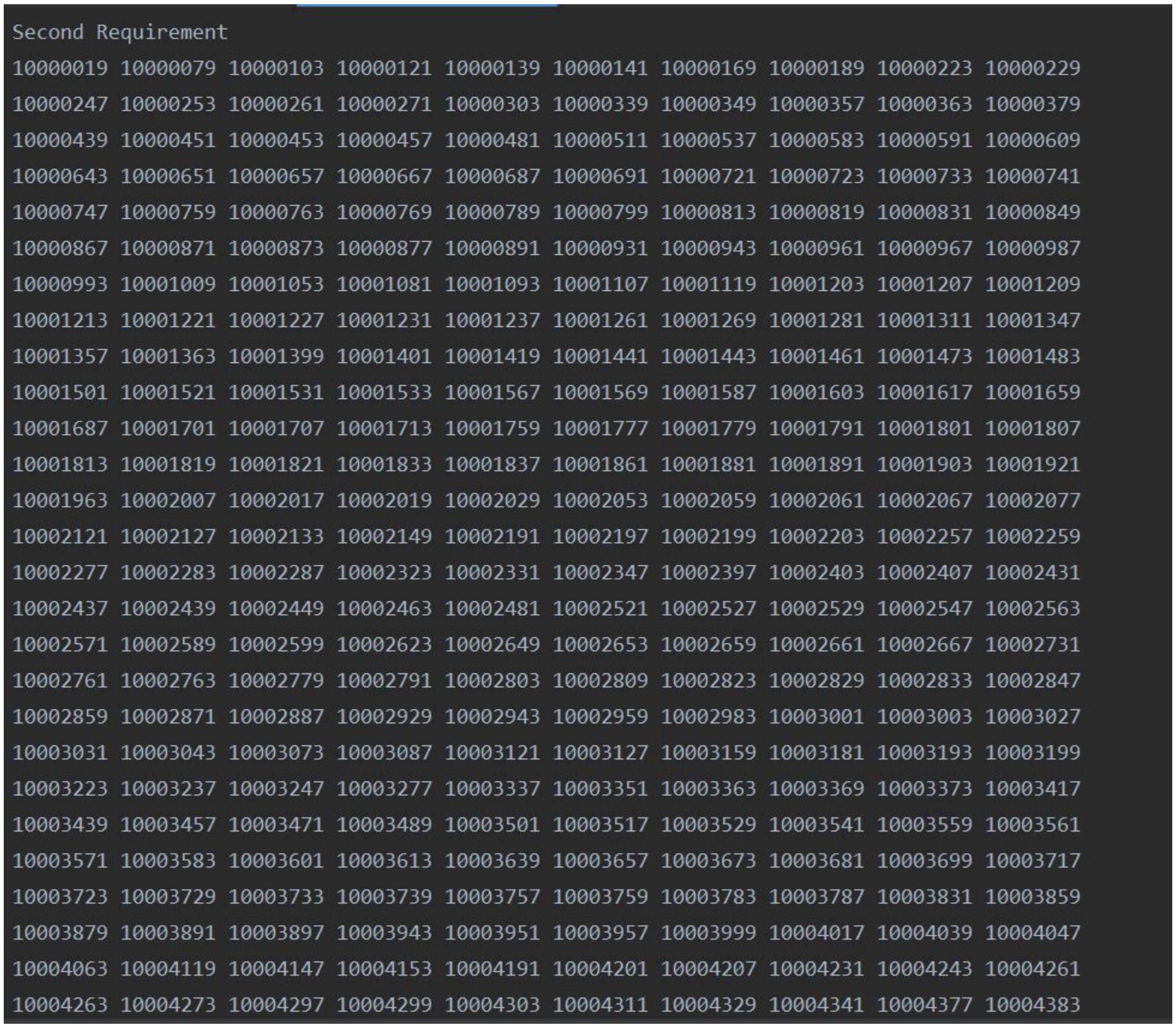
```
public static boolean checkIfPrime(long number){

    boolean isPrime = true;

    for (int i = 2; i <= number/2; ++i){

        // if not prime
        if (number % i == 0){
            isPrime = false;
            break;
        }
    }

    return isPrime;
}
```

Time taken to generate the prime numbers is generated by the ‘timeBetweenPrimes’ method which calculates the time in nanoseconds.



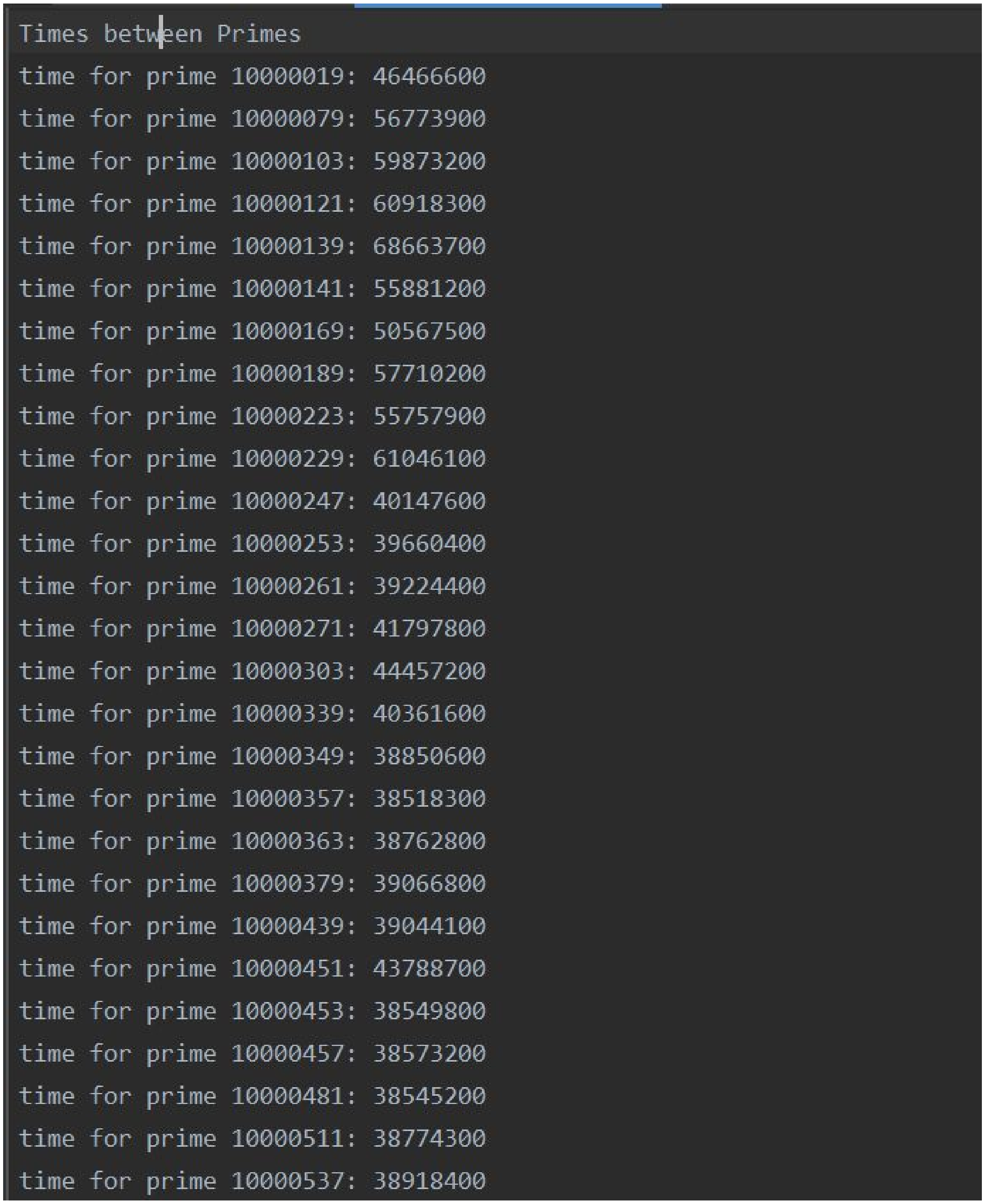
```
public static void timeBetweenPrimes(long number, int index, long before, long after){

    timeBetweenPrimes[index] = after - before;
}
```

The program generates an array of 10000 prime numbers in the sets of 10 for each row starting from 10000000.



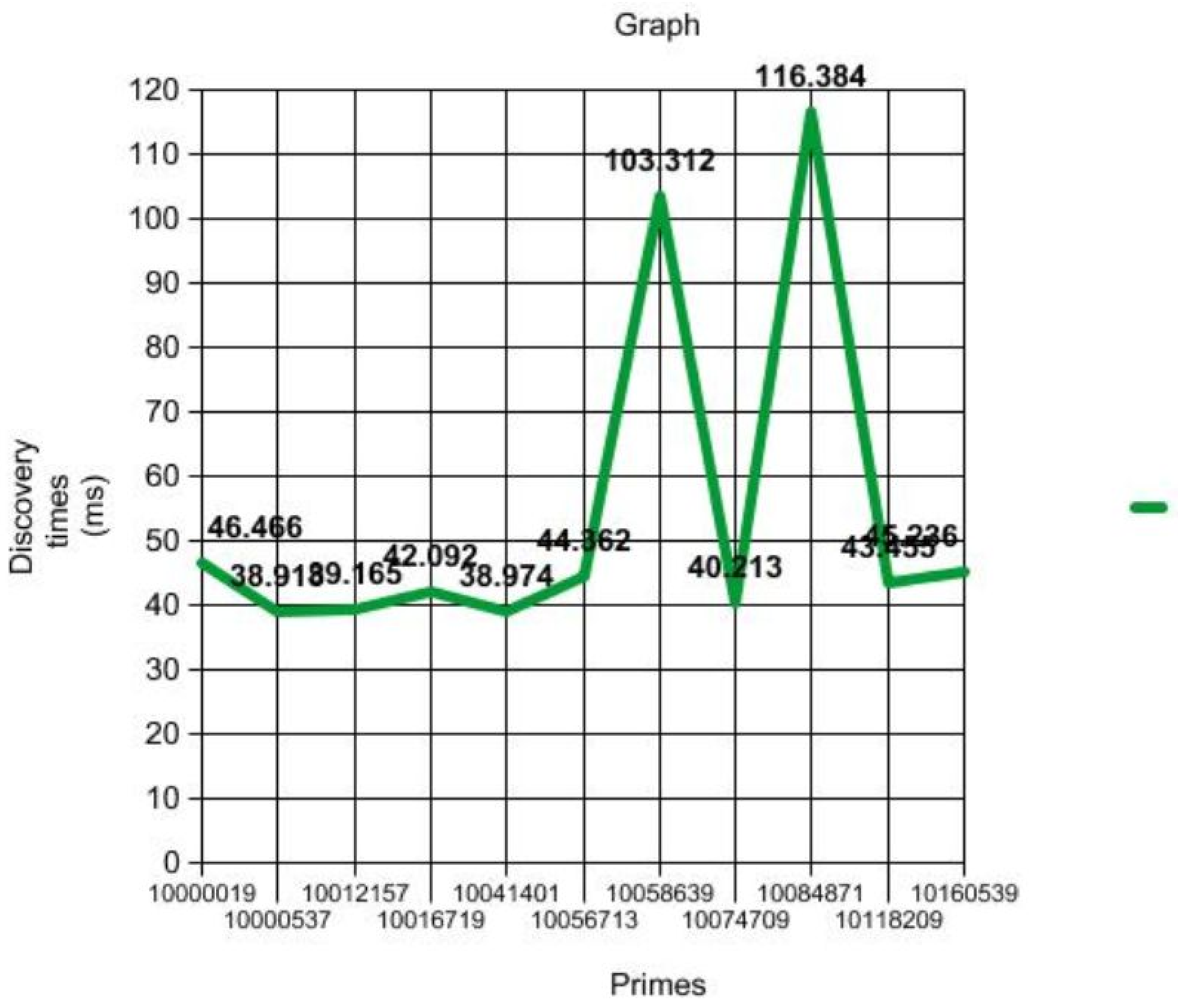
```
Second Requirement
10000019 10000079 10000103 10000121 10000139 10000141 10000169 10000189 10000223 10000229
10000247 10000253 10000261 10000271 10000303 10000339 10000349 10000357 10000363 10000379
10000439 10000451 10000453 10000457 10000481 10000511 10000537 10000583 10000591 10000609
10000643 10000651 10000657 10000667 10000687 10000691 10000721 10000723 10000733 10000741
10000747 10000759 10000763 10000769 10000789 10000799 10000813 10000819 10000831 10000849
10000867 10000871 10000873 10000877 10000891 10000931 10000943 10000961 10000967 10000987
10000993 10001009 10001053 10001081 10001093 10001107 10001119 10001203 10001207 10001209
10001213 10001221 10001227 10001231 10001237 10001261 10001269 10001281 10001311 10001347
10001357 10001363 10001399 10001401 10001419 10001441 10001443 10001461 10001473 10001483
10001501 10001521 10001531 10001533 10001567 10001569 10001587 10001603 10001617 10001659
10001687 10001701 10001707 10001713 10001759 10001777 10001779 10001791 10001801 10001807
10001813 10001819 10001821 10001833 10001837 10001861 10001881 10001891 10001903 10001921
10001963 10002007 10002017 10002019 10002029 10002053 10002059 10002061 10002067 10002077
10002121 10002127 10002133 10002149 10002191 10002197 10002199 10002203 10002257 10002259
10002277 10002283 10002287 10002323 10002331 10002347 10002397 10002403 10002407 10002431
10002437 10002439 10002449 10002463 10002481 10002521 10002527 10002529 10002547 10002563
10002571 10002589 10002599 10002623 10002649 10002653 10002659 10002661 10002667 10002731
10002761 10002763 10002779 10002791 10002803 10002809 10002823 10002829 10002833 10002847
10002859 10002871 10002887 10002929 10002943 10002959 10002983 10003001 10003003 10003027
10003031 10003043 10003073 10003087 10003121 10003127 10003159 10003181 10003193 10003199
10003223 10003237 10003247 10003277 10003337 10003351 10003363 10003369 10003373 10003417
10003439 10003457 10003471 10003489 10003501 10003517 10003529 10003541 10003559 10003561
10003571 10003583 10003601 10003613 10003639 10003657 10003673 10003681 10003699 10003717
10003723 10003729 10003733 10003739 10003757 10003759 10003783 10003787 10003831 10003859
10003879 10003891 10003897 10003943 10003951 10003957 10003999 10004017 10004039 10004047
10004063 10004119 10004147 10004153 10004191 10004201 10004207 10004231 10004243 10004261
10004263 10004273 10004297 10004299 10004303 10004311 10004329 10004341 10004377 10004383
```

The discovery times are also determined for each prime in nanoseconds



```
Times between Primes
time for prime 10000019: 46466600
time for prime 10000079: 56773900
time for prime 10000103: 59873200
time for prime 10000121: 60918300
time for prime 10000139: 68663700
time for prime 10000141: 55881200
time for prime 10000169: 50567500
time for prime 10000189: 57710200
time for prime 10000223: 55757900
time for prime 10000229: 61046100
time for prime 10000247: 40147600
time for prime 10000253: 39660400
time for prime 10000261: 39224400
time for prime 10000271: 41797800
time for prime 10000303: 44457200
time for prime 10000339: 40361600
time for prime 10000349: 38850600
time for prime 10000357: 38518300
time for prime 10000363: 38762800
time for prime 10000379: 39066800
time for prime 10000439: 39044100
time for prime 10000451: 43788700
time for prime 10000453: 38549800
time for prime 10000457: 38573200
time for prime 10000481: 38545200
time for prime 10000511: 38774300
time for prime 10000537: 38918400
```

Graph showing the Discovery times on vertical axis and the Primes on the horizontal axis

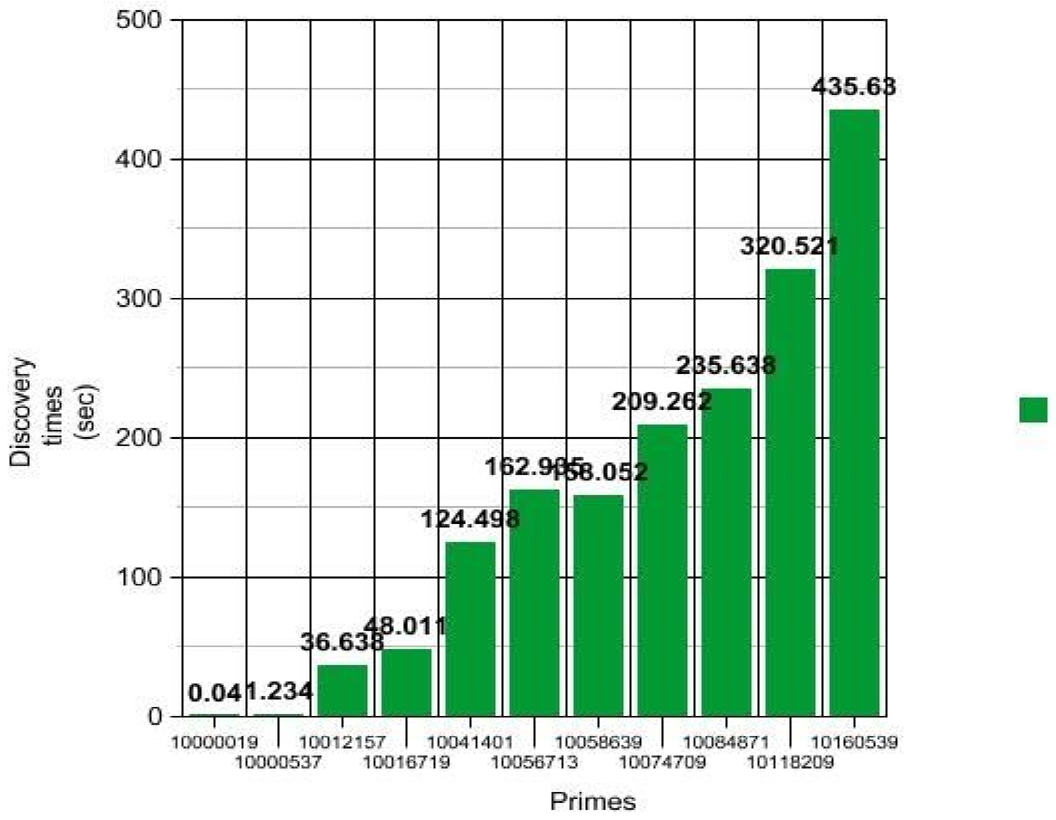


We have plotted the graph by taking 11 random primes from our prime bank and their respective times.

Almost all the primes took between 36ms - 48ms. But few primes took more than double the average time to generate.

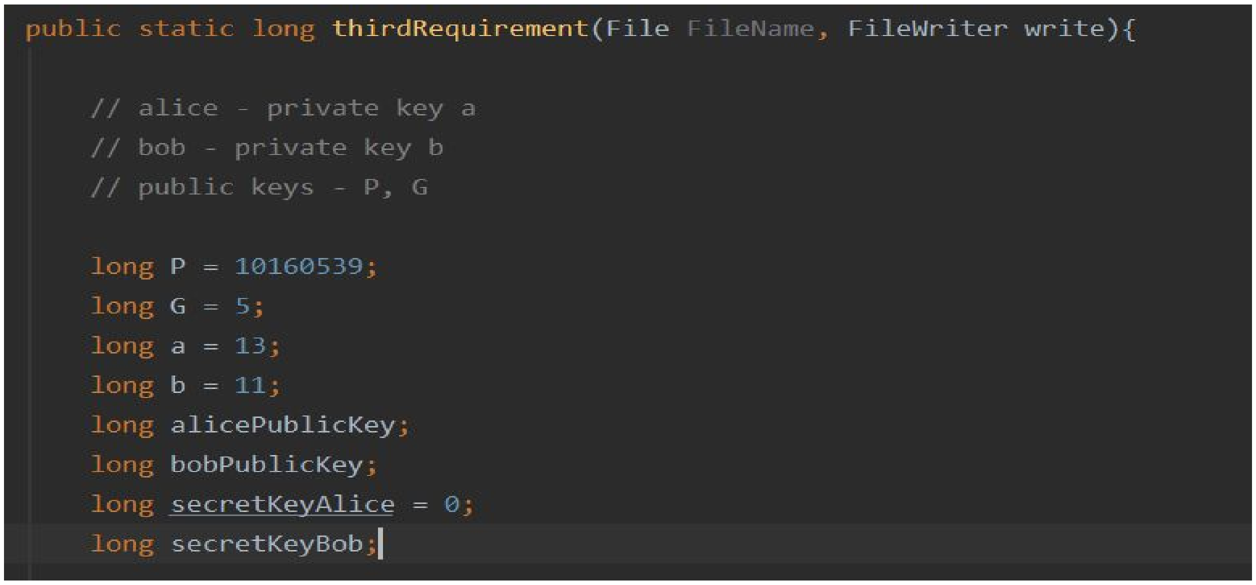
In the above graph, we can see the drastic elevation at the primes 10058639 and 10084871 which took 103.31ms and 116.38ms respectively.

Below is the graph where the times are taken in the incremental manner.



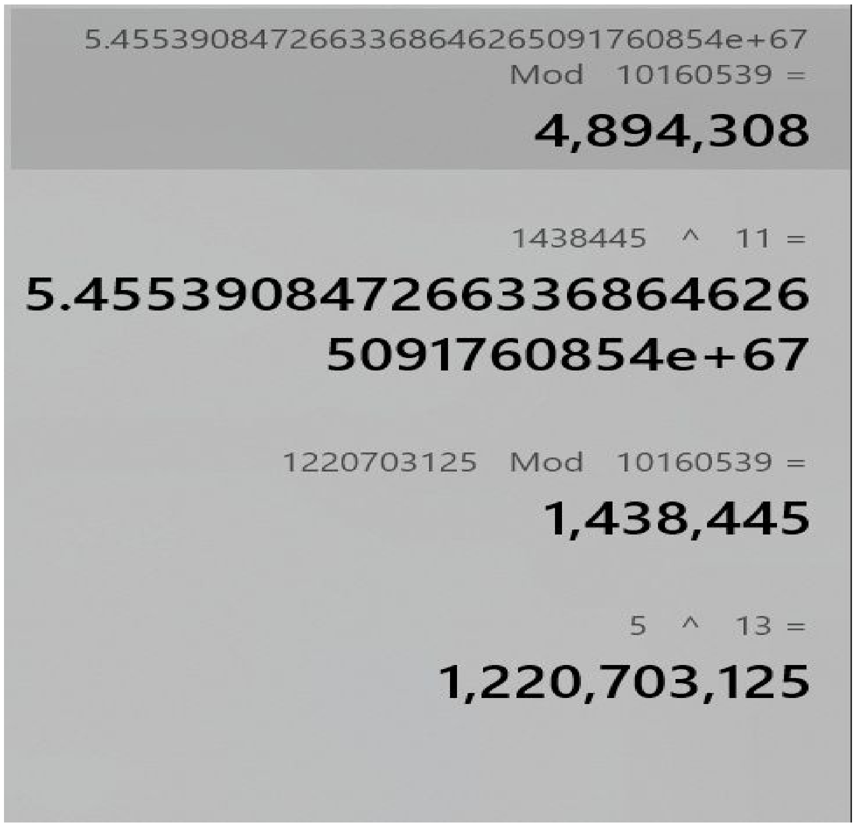
Third requirement: Bob and Alice application

Initially we have considered these values with P being the largest from our prime bank and the desired values for ‘a’ and ‘b’ but our program is unable to handle such huge calculations while calculating the secret keys for both Alice and Bob.



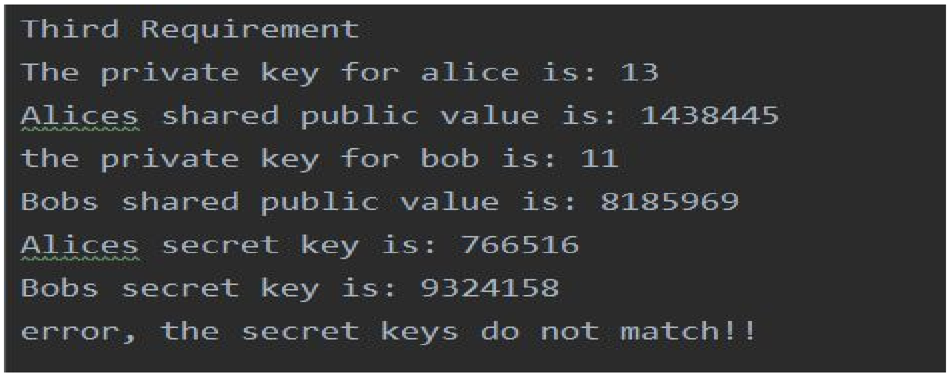
```
public static long thirdRequirement(File FileName, FileWriter write){

    // alice - private key a
    // bob - private key b
    // public keys - P, G

    long P = 10160539;
    long G = 5;
    long a = 13;
    long b = 11;
    long alicePublicKey;
    long bobPublicKey;
    long secretKeyAlice = 0;
    long secretKeyBob;
```

The output should be



```
5.4553908472663368646265091760854e+67
                          Mod  10160539 =
                                4,894,308

                         1438445  ^  11 =
5.4553908472663368646265091760854e+67

            1220703125  Mod  10160539 =
                                1,438,445

                               5  ^  13 =
                            1,220,703,125
```

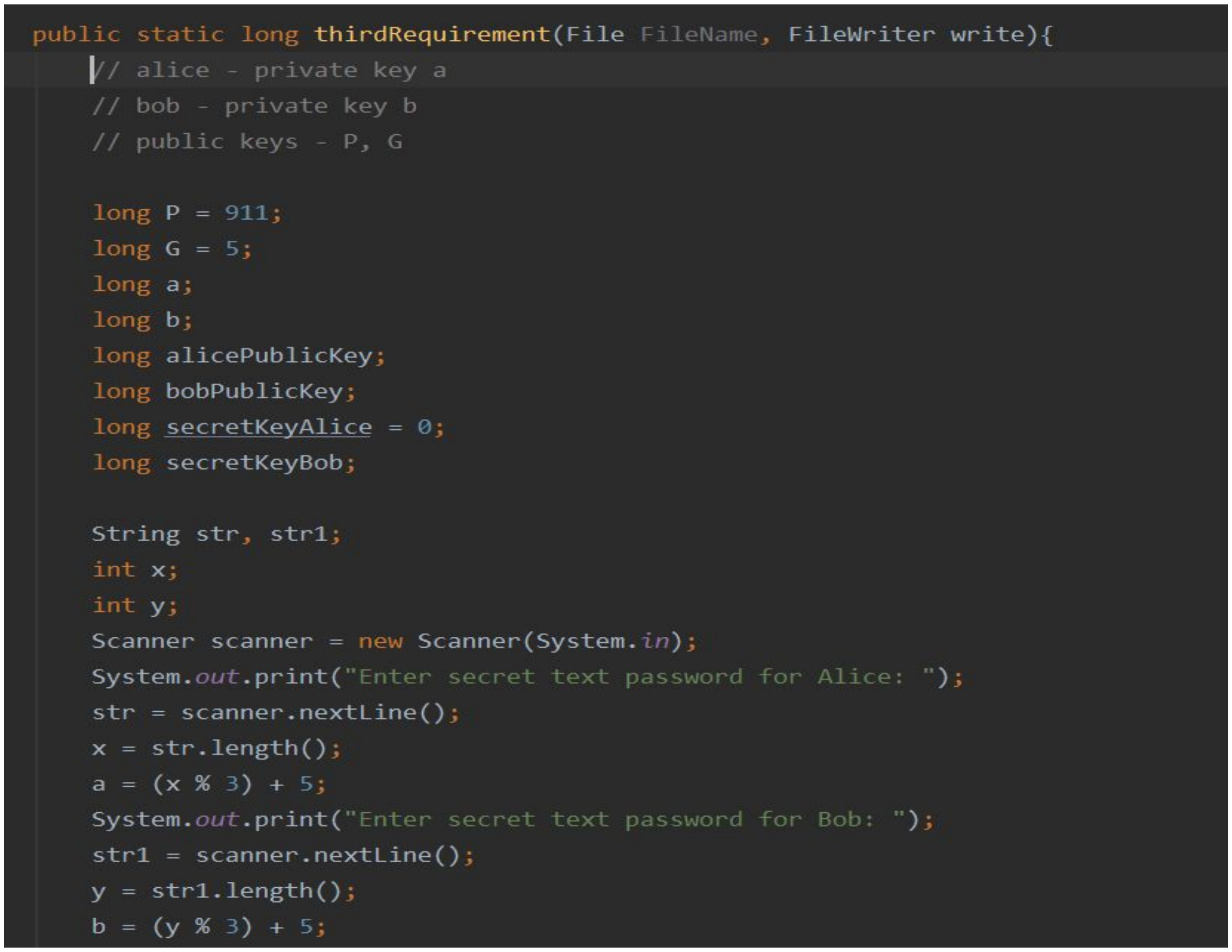
Due to the limitations in JAVA as it doesn't support such large numbers, the output we are getting is a garbage value



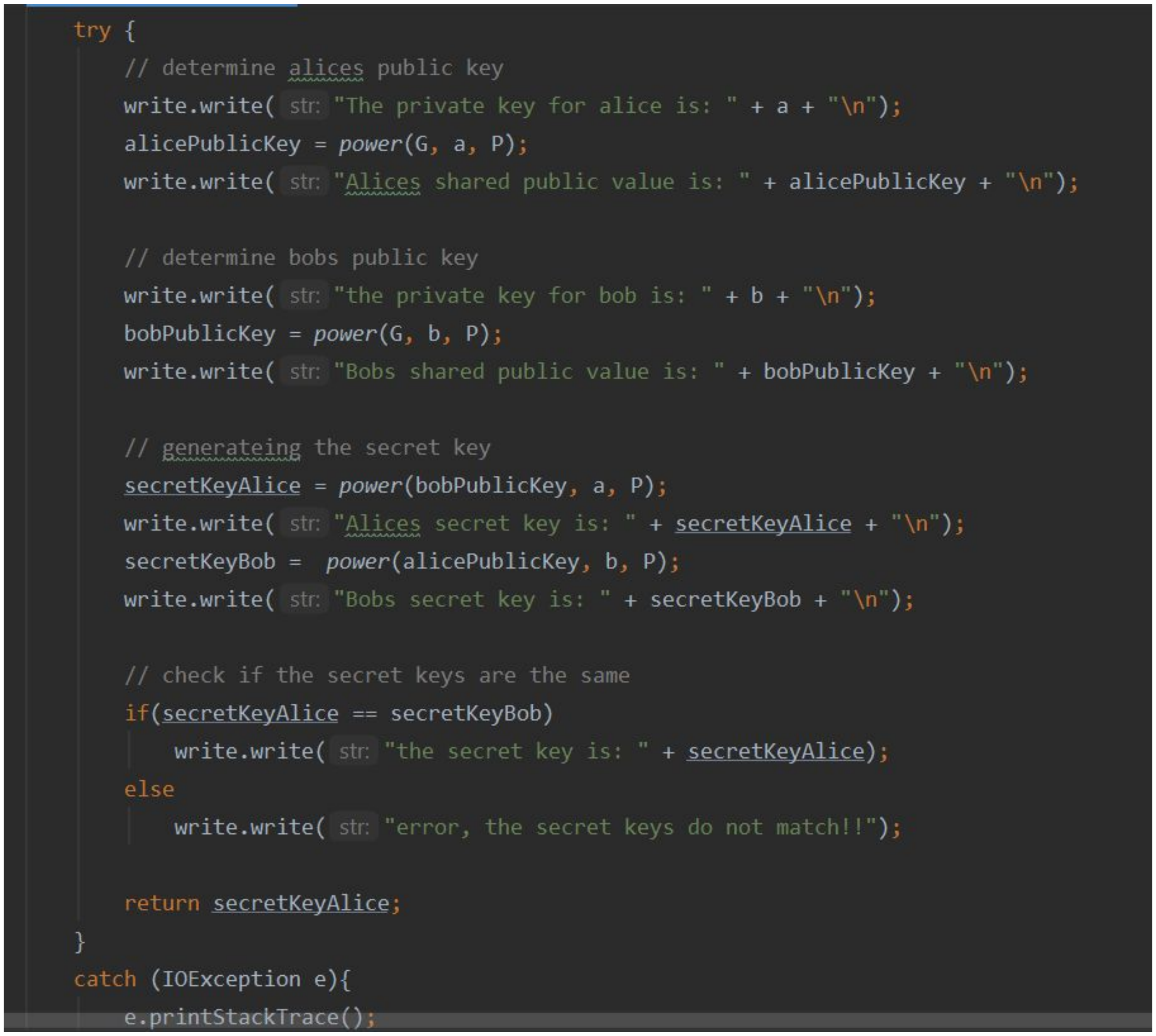
```
Third Requirement
The private key for alice is: 13
Alices shared public value is: 1438445
the private key for bob is: 11
Bobs shared public value is: 8185969
Alices secret key is: 766516
Bobs secret key is: 9324158
error, the secret keys do not match!!
```

We tried to increase the 'P' value but as the (g pow a) value will be low compared to P, the entire function of (g pow a) % P becomes the same as (g pow a) having no effect of P.

So we are confined to use smaller values to make our algorithm work.
The code takes the secret text passwords from Alice and Bob to generate the 'a' and 'b' values and then generates the shared keys.

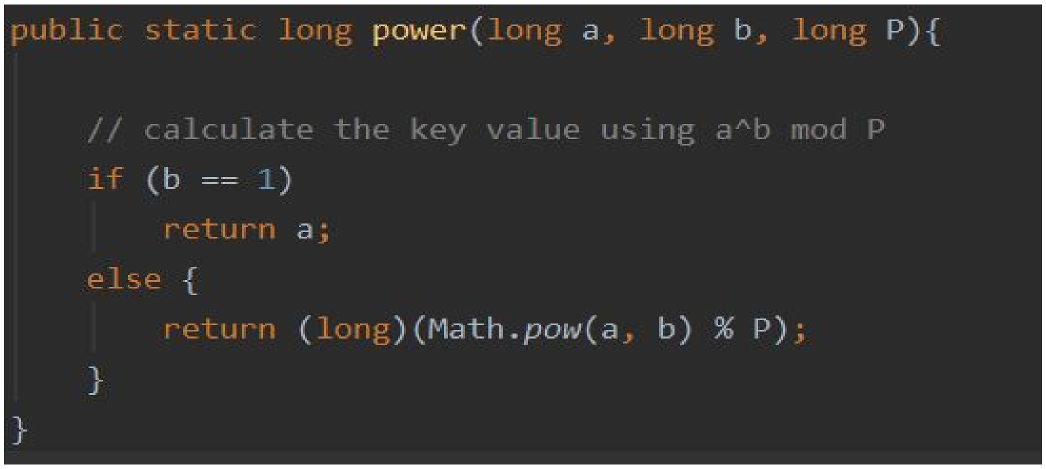


```java
public static long thirdRequirement(File FileName, FileWriter write){
    // alice - private key a
    // bob - private key b
    // public keys - P, G

    long P = 911;
    long G = 5;
    long a;
    long b;
    long alicePublicKey;
    long bobPublicKey;
    long secretKeyAlice = 0;
    long secretKeyBob;

    String str, str1;
    int x;
    int y;
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter secret text password for Alice: ");
    str = scanner.nextLine();
    x = str.length();
    a = (x % 3) + 5;
    System.out.print("Enter secret text password for Bob: ");
    str1 = scanner.nextLine();
    y = str1.length();
    b = (y % 3) + 5;
```



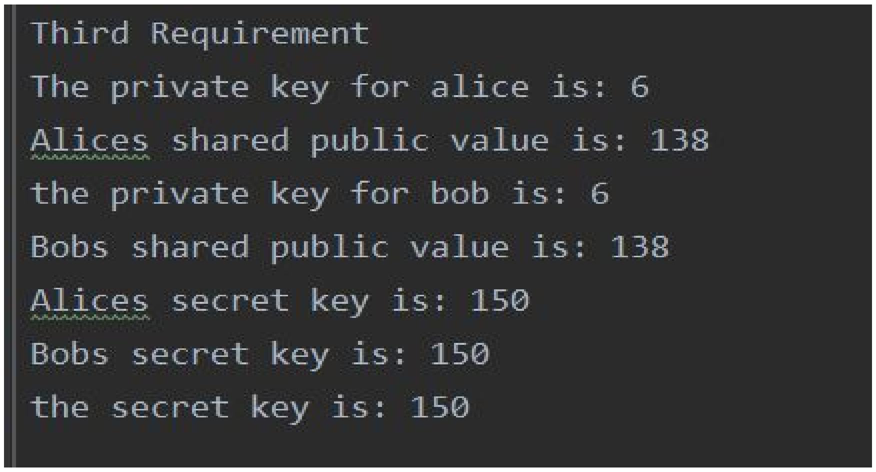
```java
    try {
        // determine alices public key
        write.write( str: "The private key for alice is: " + a + "\n");
        alicePublicKey = power(G, a, P);
        write.write( str: "Alices shared public value is: " + alicePublicKey + "\n");

        // determine bobs public key
        write.write( str: "the private key for bob is: " + b + "\n");
        bobPublicKey = power(G, b, P);
        write.write( str: "Bobs shared public value is: " + bobPublicKey + "\n");

        // generateing the secret key
        secretKeyAlice = power(bobPublicKey, a, P);
        write.write( str: "Alices secret key is: " + secretKeyAlice + "\n");
        secretKeyBob =  power(alicePublicKey, b, P);
        write.write( str: "Bobs secret key is: " + secretKeyBob + "\n");

        // check if the secret keys are the same
        if(secretKeyAlice == secretKeyBob)
            write.write( str: "the secret key is: " + secretKeyAlice);
        else
            write.write( str: "error, the secret keys do not match!!");

        return secretKeyAlice;
    }
    catch (IOException e){
        e.printStackTrace();
```

The power function looks like



```java
public static long power(long a, long b, long P){

    // calculate the key value using a^b mod P
    if (b == 1)
        return a;
    else {
        return (long)(Math.pow(a, b) % P);
    }
}
```

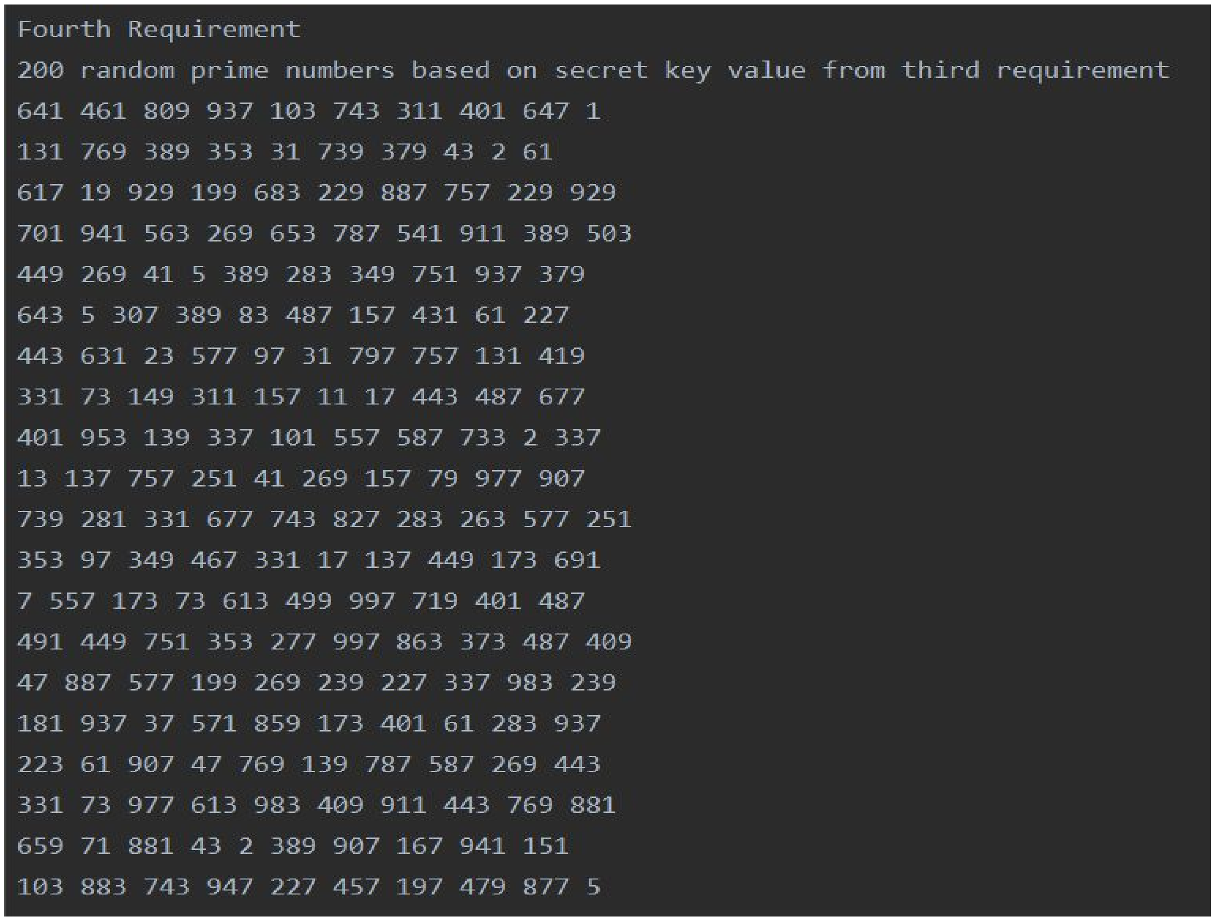
The output of this is



```
Third Requirement
The private key for alice is: 6
Alices shared public value is: 138
the private key for bob is: 6
Bobs shared public value is: 138
Alices secret key is: 150
Bobs secret key is: 150
the secret key is: 150
```

Fourth Requirement: Generating random integers based on the Bob and Alice Shared key

This requirement uses the same code as in the first requirement but with the seed value as generated in the third requirement.



```
Fourth Requirement
200 random prime numbers based on secret key value from third requirement
641 461 809 937 103 743 311 401 647 1
131 769 389 353 31 739 379 43 2 61
617 19 929 199 683 229 887 757 229 929
701 941 563 269 653 787 541 911 389 503
449 269 41 5 389 283 349 751 937 379
643 5 307 389 83 487 157 431 61 227
443 631 23 577 97 31 797 757 131 419
331 73 149 311 157 11 17 443 487 677
401 953 139 337 101 557 587 733 2 337
13 137 757 251 41 269 157 79 977 907
739 281 331 677 743 827 283 263 577 251
353 97 349 467 331 17 137 449 173 691
7 557 173 73 613 499 997 719 401 487
491 449 751 353 277 997 863 373 487 409
47 887 577 199 269 239 227 337 983 239
181 937 37 571 859 173 401 61 283 937
223 61 907 47 769 139 787 587 269 443
331 73 977 613 983 409 911 443 769 881
659 71 881 43 2 389 907 167 941 151
103 883 743 947 227 457 197 479 877 5
```

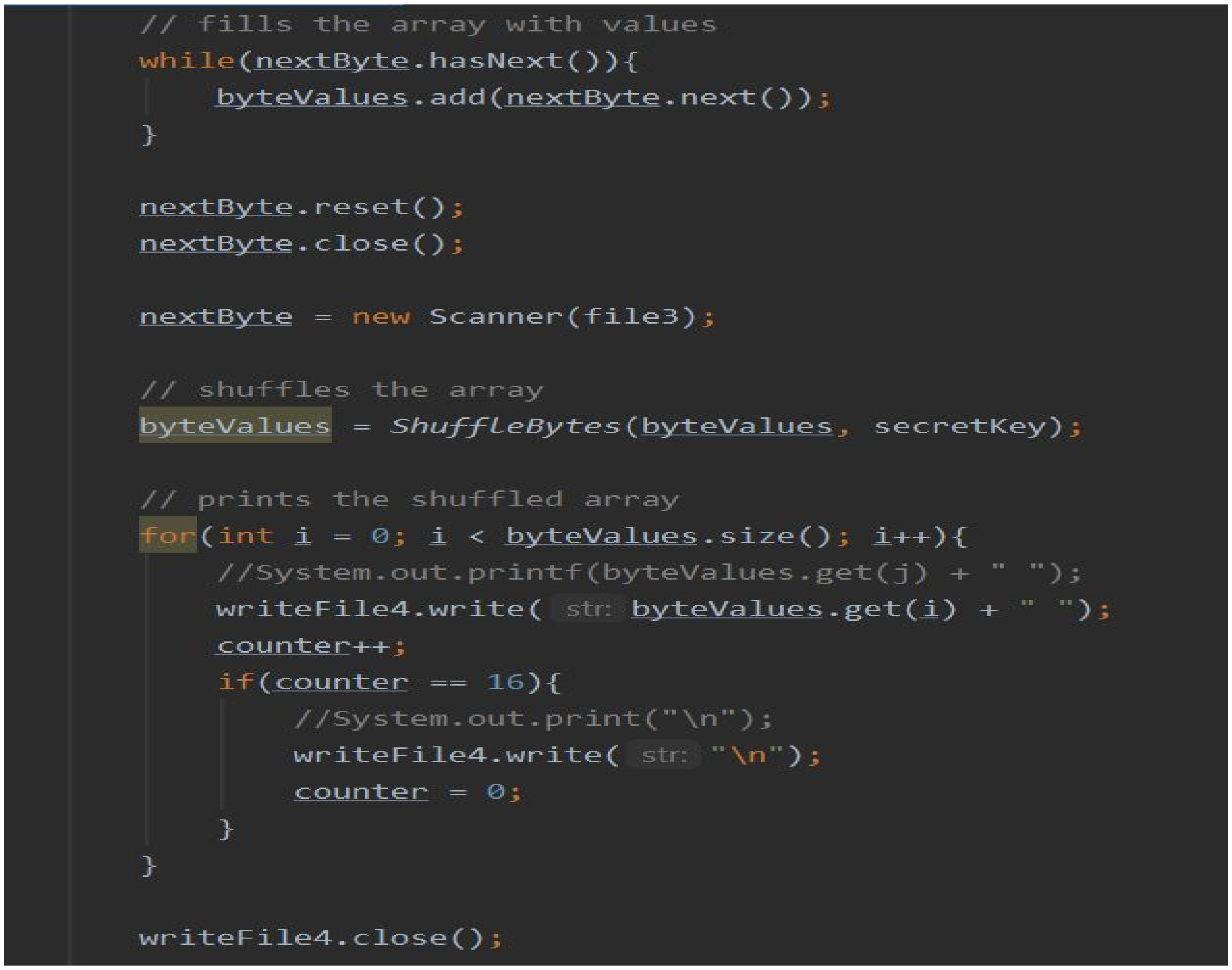
As the seed values in the first and the third requirement are different, the random numbers generated are different. We cannot get the same set of random integers unless the seed value is the same.

Fifth Requirement: Shuffling algorithm

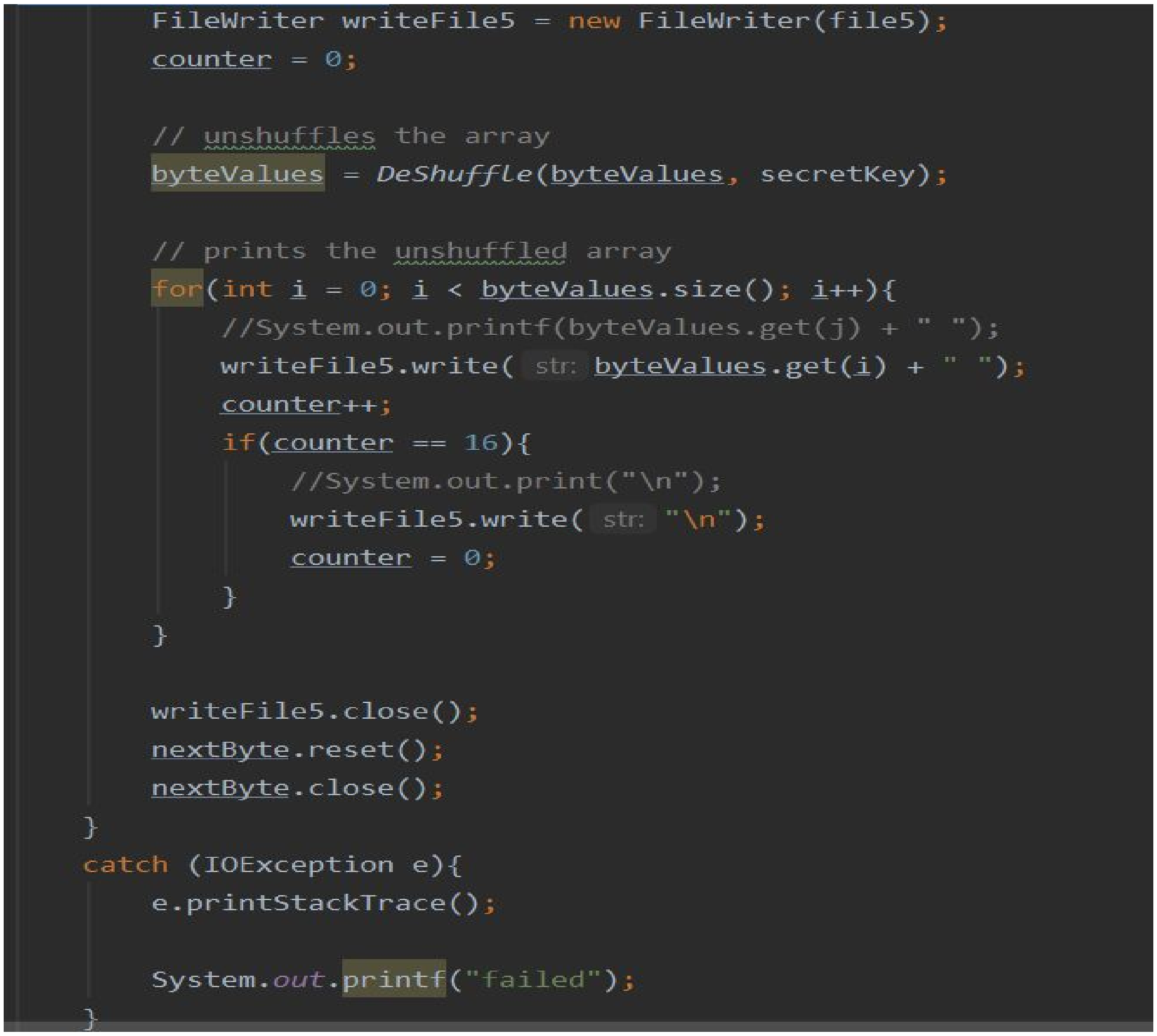
For this requirement we converted the audio file to hex using an online converter, storing every byte as a string in an array. The array is then parsed backwards, indexing the bytes and creating another integer array for these positions. Those index positions are found by using our random number generator with a seed value to generate a random number between 0 and that specific index, continuing until every byte has been swapped. This creates our shuffled array, which can then be converted back to a playable audio .mp4 file.

The unshuffling applies the same process as shuffling the array, except in reverse starting at the front of the array. All bytes are swapped back with the corresponding index position until the entire array has been sorted back to the original array.

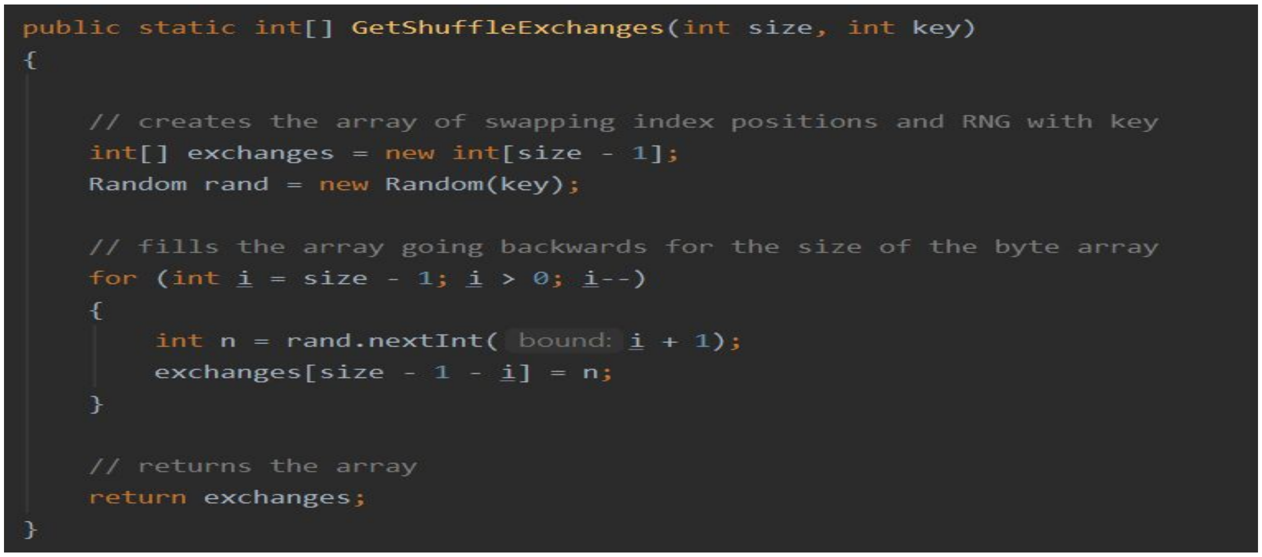
We have used Fisher-Yates modern algorithm for this requirement.
The below code will fill the array with the hex values of the audio and also the shuffling and deshuffling methods are called.



```java
// fills the array with values
while(nextByte.hasNext()){
    byteValues.add(nextByte.next());
}

nextByte.reset();
nextByte.close();

nextByte = new Scanner(file3);

// shuffles the array
byteValues = ShuffleBytes(byteValues, secretKey);

// prints the shuffled array
for(int i = 0; i < byteValues.size(); i++){
    //System.out.printf(byteValues.get(j) + " ");
    writeFile4.write( str: byteValues.get(i) + " ");
    counter++;
    if(counter == 16){
        //System.out.print("\n");
        writeFile4.write( str: "\n");
        counter = 0;
    }
}

writeFile4.close();
```

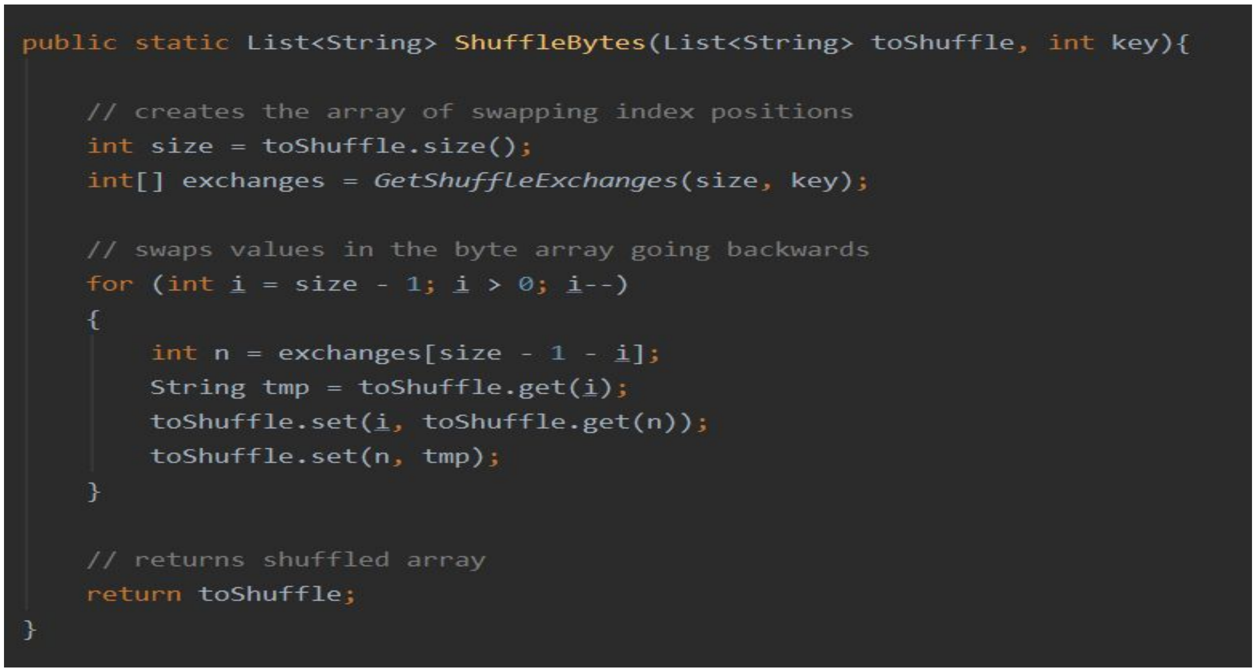


```
        FileWriter writeFile5 = new FileWriter(file5);
        counter = 0;

        // unshuffles the array
        byteValues = DeShuffle(byteValues, secretKey);

        // prints the unshuffled array
        for(int i = 0; i < byteValues.size(); i++){
            //System.out.printf(byteValues.get(j) + " ");
            writeFile5.write( str: byteValues.get(i) + " ");
            counter++;
            if(counter == 16){
                //System.out.print("\n");
                writeFile5.write( str: "\n");
                counter = 0;
            }
        }

        writeFile5.close();
        nextByte.reset();
        nextByte.close();
    }
    catch (IOException e){
        e.printStackTrace();

        System.out.printf("failed");
    }
```



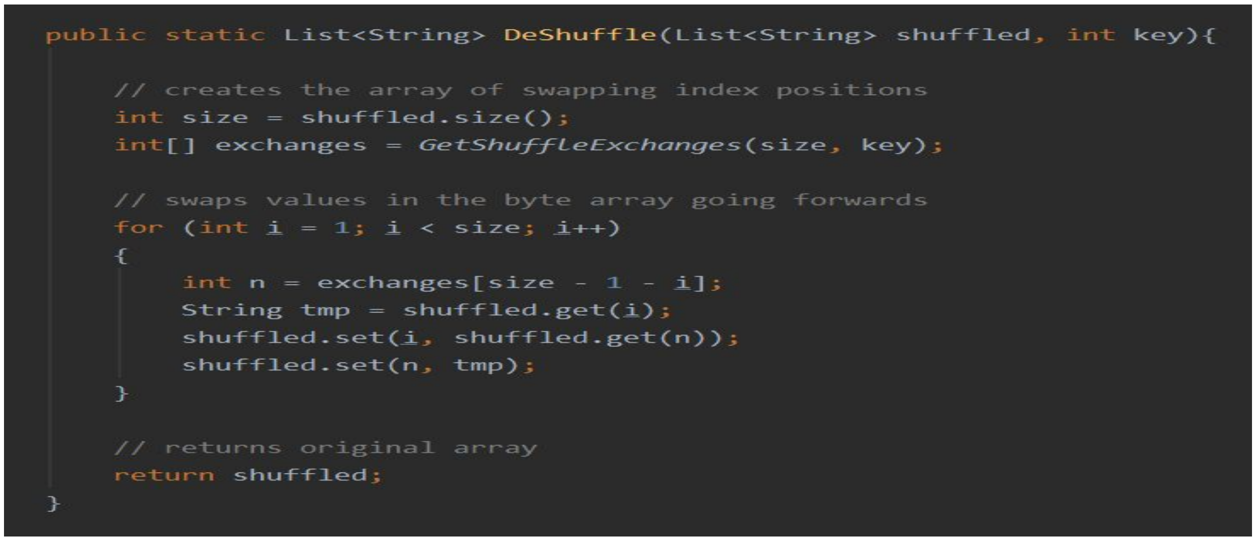
```
public static int[] GetShuffleExchanges(int size, int key)
{

    // creates the array of swapping index positions and RNG with key
    int[] exchanges = new int[size - 1];
    Random rand = new Random(key);

    // fills the array going backwards for the size of the byte array
    for (int i = size - 1; i > 0; i--)
    {
        int n = rand.nextInt( bound: i + 1);
        exchanges[size - 1 - i] = n;
    }

    // returns the array
    return exchanges;
}
```

This code will shuffle the byte values based on the indexes creates in the previous step.



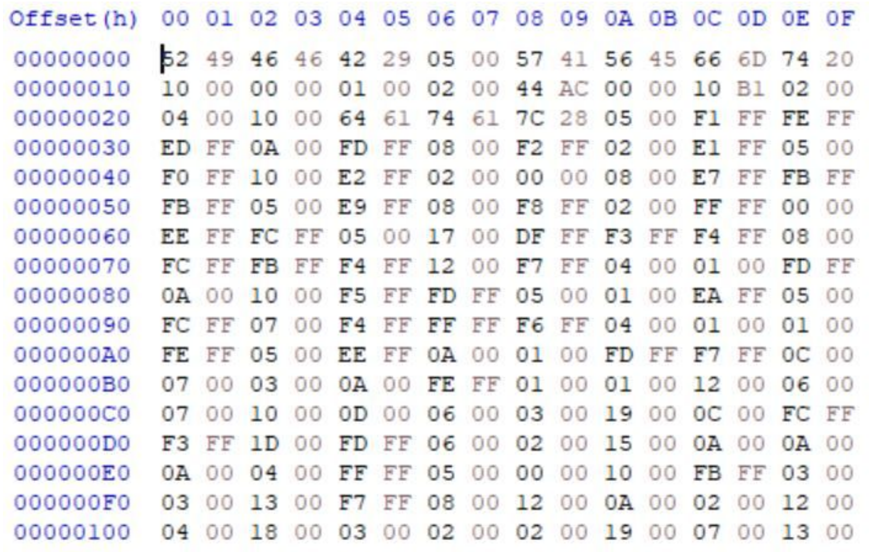
```
public static List<String> ShuffleBytes(List<String> toShuffle, int key){

    // creates the array of swapping index positions
    int size = toShuffle.size();
    int[] exchanges = GetShuffleExchanges(size, key);

    // swaps values in the byte array going backwards
    for (int i = size - 1; i > 0; i--)
    {
        int n = exchanges[size - 1 - i];
        String tmp = toShuffle.get(i);
        toShuffle.set(i, toShuffle.get(n));
        toShuffle.set(n, tmp);
    }

    // returns shuffled array
    return toShuffle;
}
```

The program below will deshuffle the array back to its original indexes.



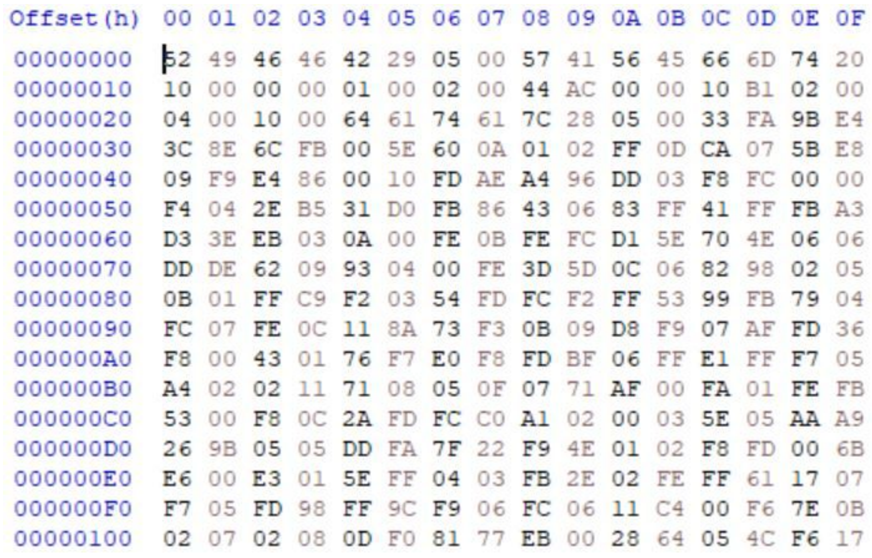
```
public static List<String> DeShuffle(List<String> shuffled, int key){

    // creates the array of swapping index positions
    int size = shuffled.size();
    int[] exchanges = GetShuffleExchanges(size, key);

    // swaps values in the byte array going forwards
    for (int i = 1; i < size; i++)
    {
        int n = exchanges[size - 1 - i];
        String tmp = shuffled.get(i);
        shuffled.set(i, shuffled.get(n));
        shuffled.set(n, tmp);
    }

    // returns original array
    return shuffled;
}
```

The original file in the hex form. The header information ends at 44byte. Audio data starts at F1 in line 3.



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  52 49 46 46 42 29 05 00 57 41 56 45 66 6D 74 20
00000010  10 00 00 00 01 00 02 00 44 AC 00 00 10 B1 02 00
00000020  04 00 10 00 64 61 74 61 7C 28 05 00 F1 FF FE FF
00000030  ED FF 0A 00 FD FF 08 00 F2 FF 02 00 E1 FF 05 00
00000040  F0 FF 10 00 E2 FF 02 00 00 00 08 00 E7 FF FB FF
00000050  FB FF 05 00 E9 FF 08 00 F8 FF 02 00 FF FF 00 00
00000060  EE FF FC FF 05 00 17 00 DF FF F3 FF F4 FF 08 00
00000070  FC FF FB FF F4 FF 12 00 F7 FF 04 00 01 00 FD FF
00000080  0A 00 10 00 F5 FF FD FF 05 00 01 00 EA FF 05 00
00000090  FC FF 07 00 F4 FF FF FF F6 FF 04 00 01 00 01 00
000000A0  FE FF 05 00 EE FF 0A 00 01 00 FD FF F7 FF 0C 00
000000B0  07 00 03 00 0A 00 FE FF 01 00 01 00 12 00 06 00
000000C0  07 00 10 00 0D 00 06 00 03 00 19 00 0C 00 FC FF
000000D0  F3 FF 1D 00 FD FF 06 00 02 00 15 00 0A 00 0A 00
000000E0  0A 00 04 00 FF FF 05 00 00 00 10 00 FB FF 03 00
000000F0  03 00 13 00 F7 FF 08 00 12 00 0A 00 02 00 12 00
00000100  04 00 18 00 03 00 02 00 02 00 19 00 07 00 13 00
```

The below is the shuffled array by using our algorithm. Here after the 44 bytes the data is shuffled and we see that the F1 in line 3 is changed to 33.



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  52 49 46 46 42 29 05 00 57 41 56 45 66 6D 74 20
00000010  10 00 00 00 01 00 02 00 44 AC 00 00 10 B1 02 00
00000020  04 00 10 00 64 61 74 61 7C 28 05 00 33 FA 9B E4
00000030  3C 8E 6C FB 00 5E 60 0A 01 02 FF 0D CA 07 5B E8
00000040  09 F9 E4 86 00 10 FD AE A4 96 DD 03 F8 FC 00 00
00000050  F4 04 2E B5 31 D0 FB 86 43 06 83 FF 41 FF FB A3
00000060  D3 3E EB 03 0A 00 FE 0B FE FC D1 5E 70 4E 06 06
00000070  DD DE 62 09 93 04 00 FE 3D 5D 0C 06 82 98 02 05
00000080  0B 01 FF C9 F2 03 54 FD FC F2 FF 53 99 FB 79 04
00000090  FC 07 FE 0C 11 8A 73 F3 0B 09 D8 F9 07 AF FD 36
000000A0  F8 00 43 01 76 F7 E0 F8 FD BF 06 FF E1 FF F7 05
000000B0  A4 02 02 11 71 08 05 0F 07 71 AF 00 FA 01 FE FB
000000C0  53 00 F8 0C 2A FD FC C0 A1 02 00 03 5E 05 AA A9
000000D0  26 9B 05 05 DD FA 7F 22 F9 4E 01 02 F8 FD 00 6B
000000E0  E6 00 E3 01 5E FF 04 03 FB 2E 02 FE FF 61 17 07
000000F0  F7 05 FD 98 FF 9C F9 06 FC 06 11 C4 00 F6 7E 0B
00000100  02 07 02 08 0D F0 81 77 EB 00 28 64 05 4C F6 17
```