ENSE 496AE
Group D
Algo_Crypto_image-in-audio

•••

Jeremy Cross 200319513
Bipin Suram 200439429
Taylen Jones 200354271

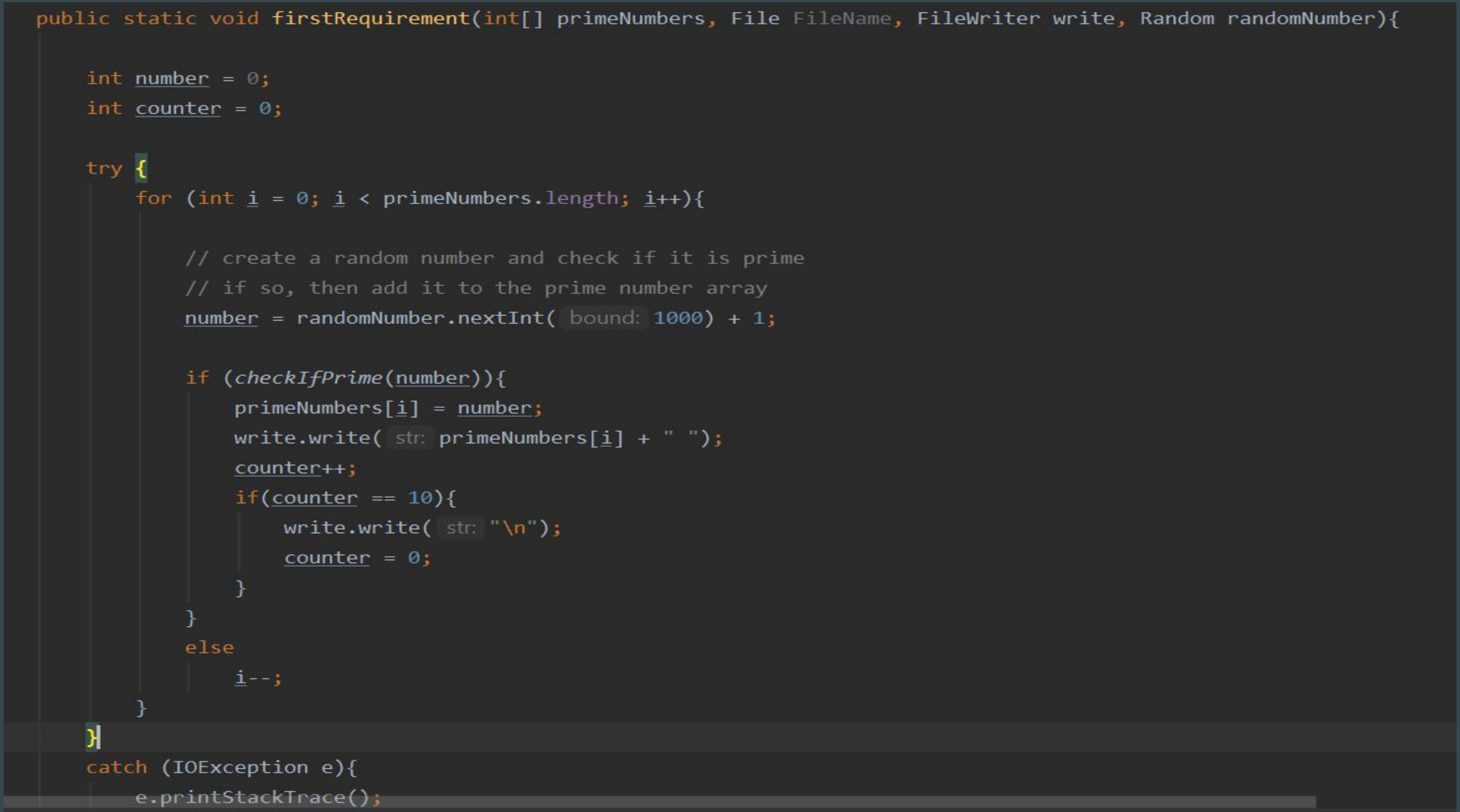
Requirements we covered for the assignment

1. Generate an array of 200 random integers
2. Generate a large list of prime numbers
3. Create a Diffie-Hellman Algorithm
4. Use key from DHA as seed value to generate a list of random integers
5. Create shuffle/unshuffle algorithm and apply it to an audio file

First Requirement

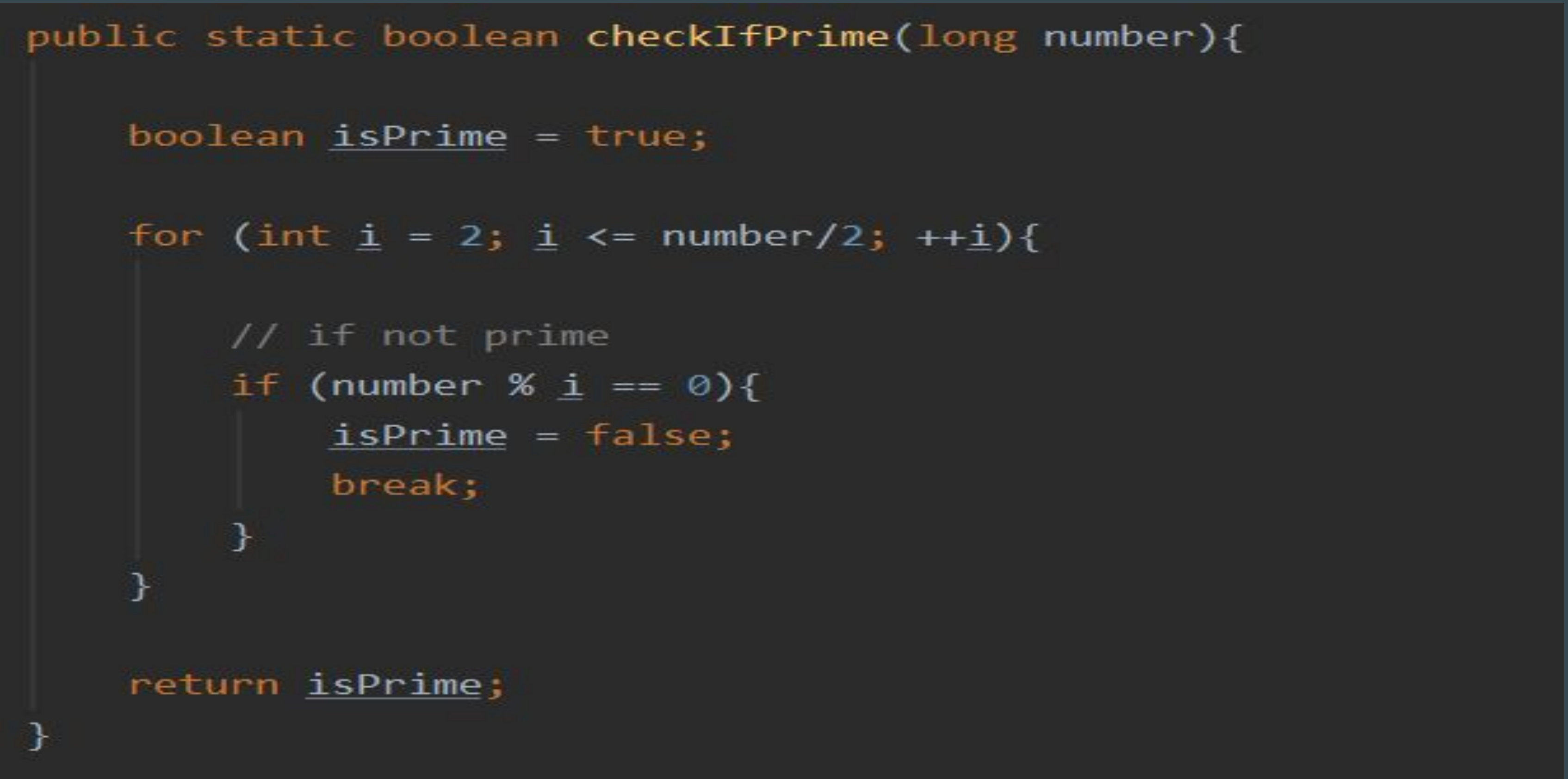
- The seed value used is 2,147,483,647 which is a prime and is the biggest number our program supports
- We followed the trial division approach to check the primality.
- The random numbers generated are prime numbers.
- We have generated 200 prime numbers below 1000.

Code to generate random integers



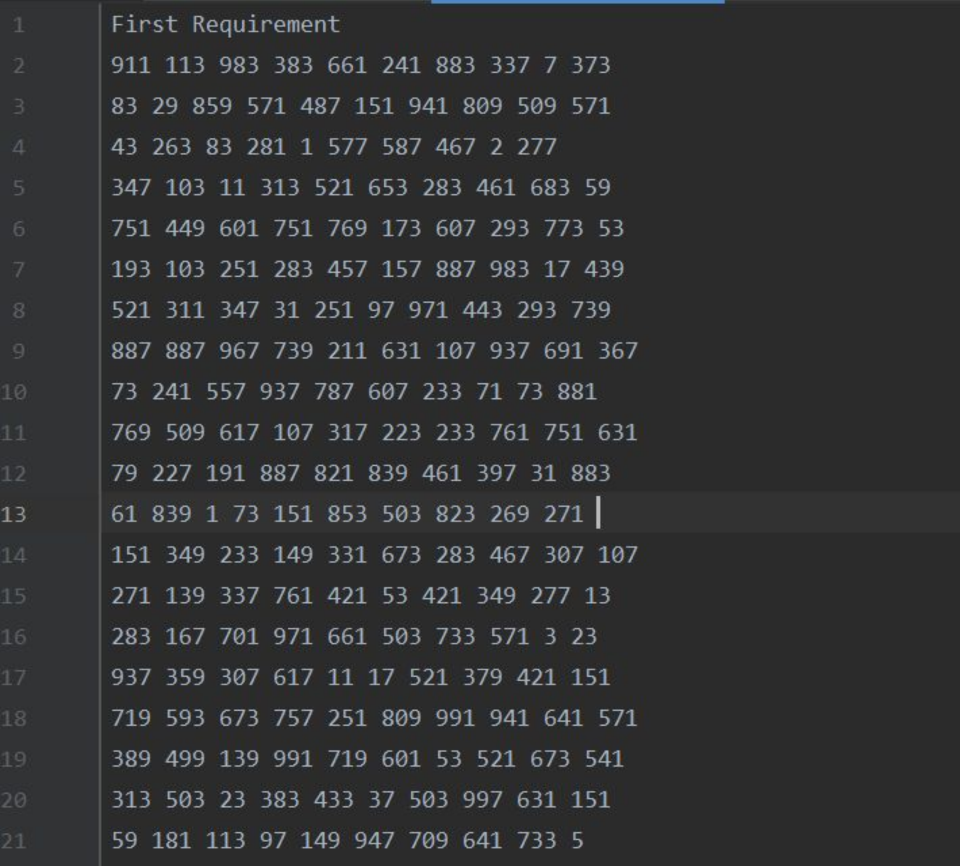
```java
public static void firstRequirement(int[] primeNumbers, File FileName, FileWriter write, Random randomNumber){

    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();
```

Check primality

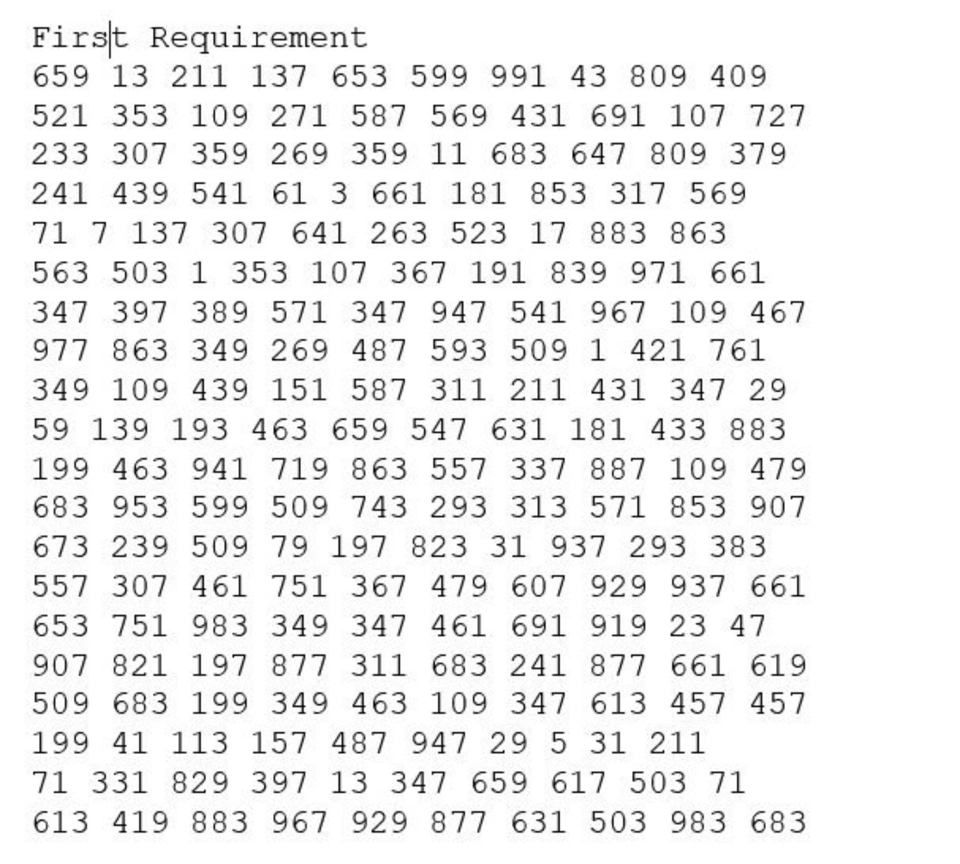


```
public static boolean checkIfPrime(long number){

    boolean isPrime = true;

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

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

    return isPrime;
}
```

Output in text and docx files(Windows OS)

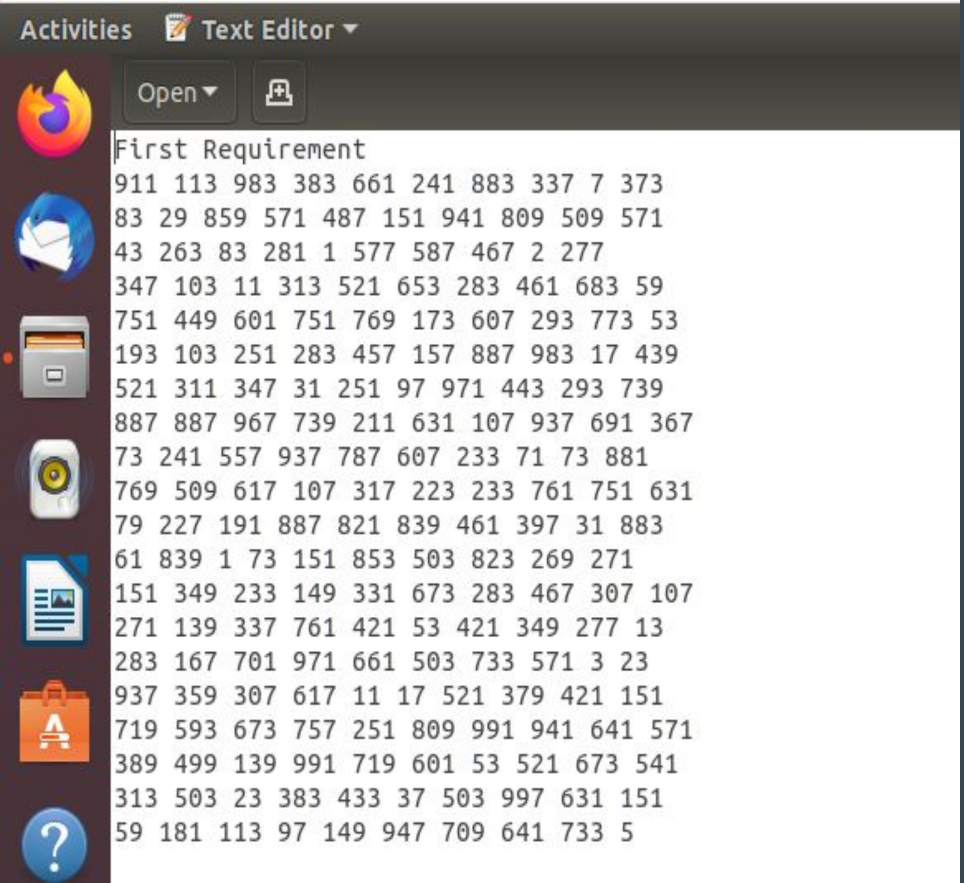


```
First Requirement
911 113 983 383 661 241 883 337 7 373
83 29 859 571 487 151 941 809 509 571
43 263 83 281 1 577 587 467 2 277
347 103 11 313 521 653 283 461 683 59
751 449 601 751 769 173 607 293 773 53
193 103 251 283 457 157 887 983 17 439
521 311 347 31 251 97 971 443 293 739
887 887 967 739 211 631 107 937 691 367
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
```

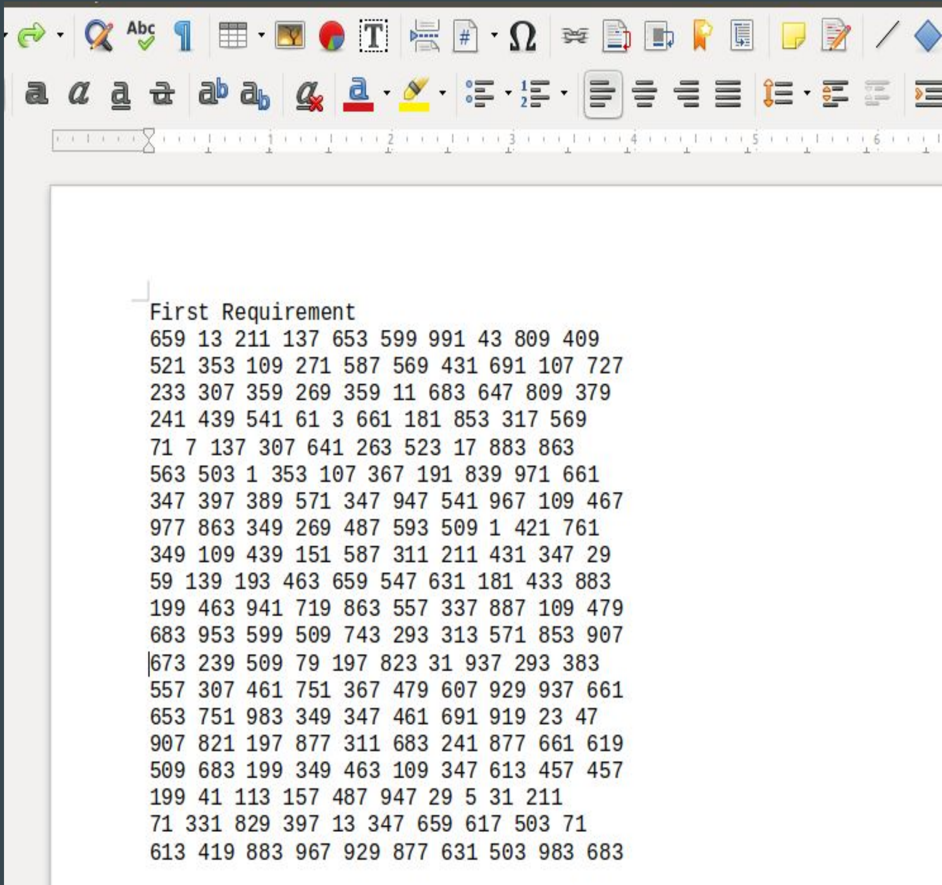


```
First Requirement
659 13 211 137 653 599 991 43 809 409
521 353 109 271 587 569 431 691 107 727
233 307 359 269 359 11 683 647 809 379
241 439 541 61 3 661 181 853 317 569
71 7 137 307 641 263 523 17 883 863
563 503 1 353 107 367 191 839 971 661
347 397 389 571 347 947 541 967 109 467
977 863 349 269 487 593 509 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
```

Output in text and docx files(Linux OS)



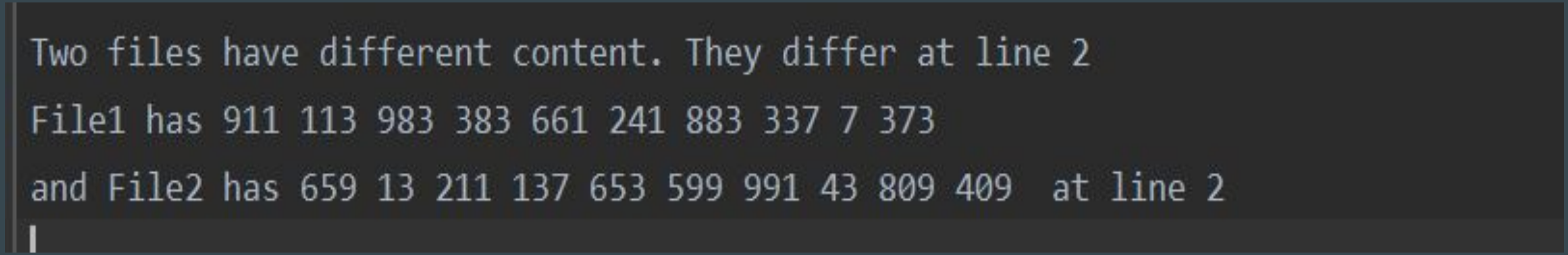
```
First Requirement
911 113 983 383 661 241 883 337 7 373
83 29 859 571 487 151 941 809 509 571
43 263 83 281 1 577 587 467 2 277
347 103 11 313 521 653 283 461 683 59
751 449 601 751 769 173 607 293 773 53
193 103 251 283 457 157 887 983 17 439
521 311 347 31 251 97 971 443 293 739
887 887 967 739 211 631 107 937 691 367
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
```

Comparing files

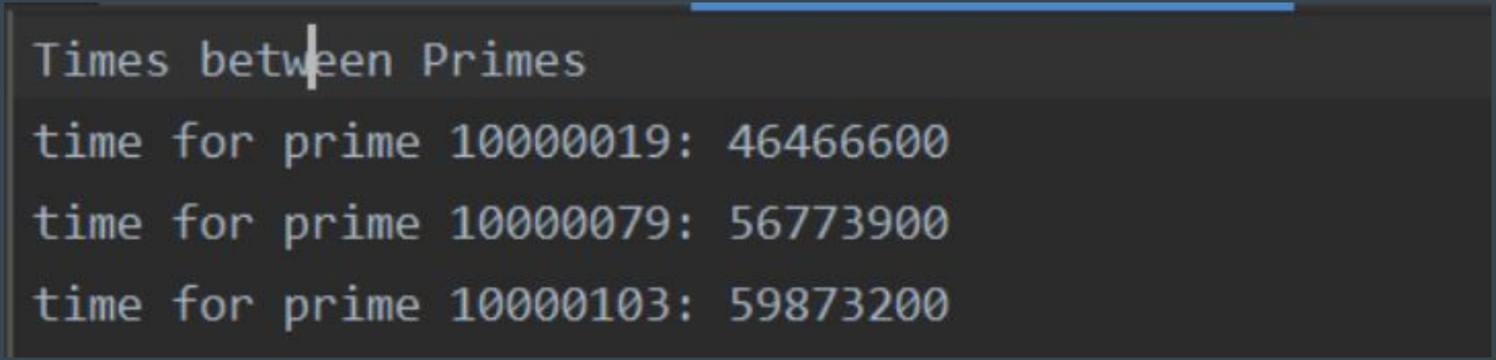
- The code checks byte by byte in two files and compares them.
- It provides the exact line where the two files differ.
- We have compared the text and docx files that were storing the values.
- The output of this code will look like



```
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

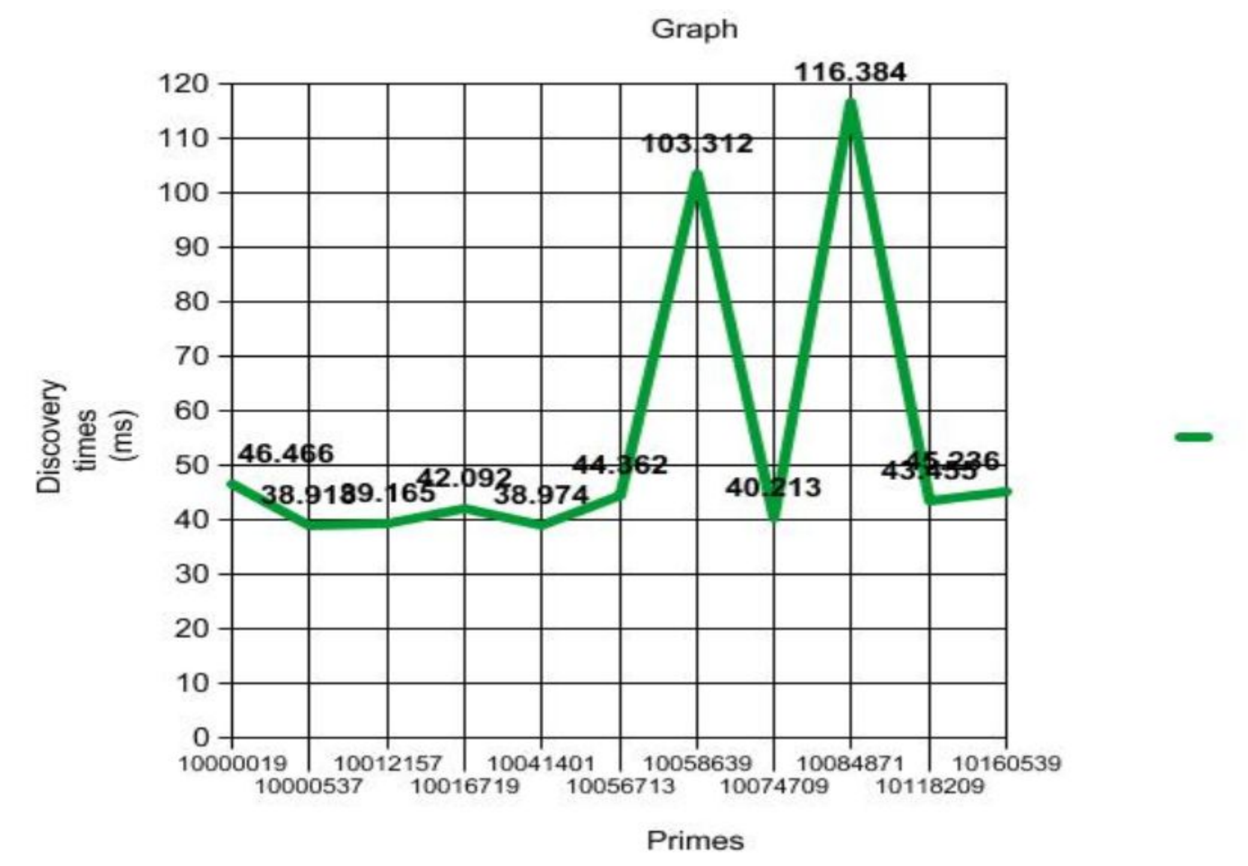
- Generated 10000 prime numbers starting from 10,000,000 and their corresponding times of each prime
- The machine took 24 minutes to complete the code and then it hang up.
- Output time is in nanoseconds.



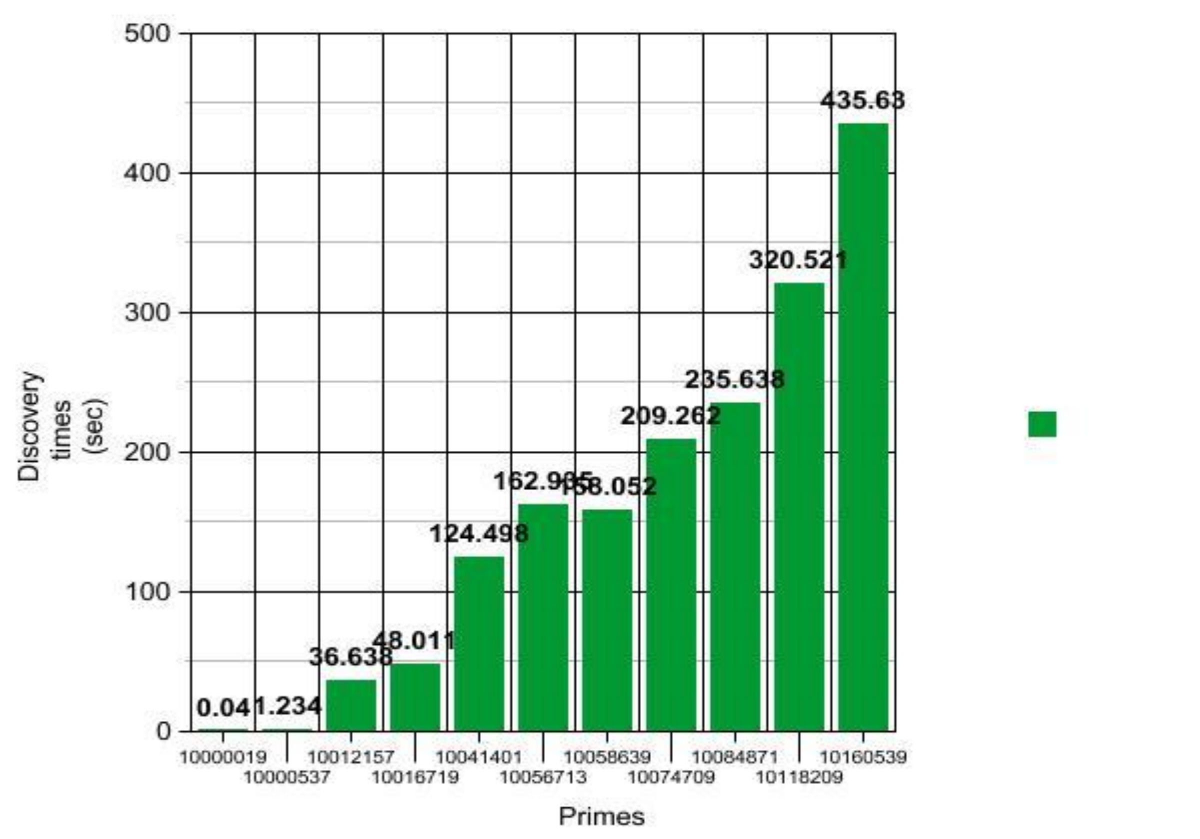
```
Times between Primes
time for prime 10000019: 46466600
time for prime 10000079: 56773900
time for prime 10000103: 59873200
```

Graph (Discovery time Vs Primes)

- Time taken for each prime.
- The average time taken to generate each prime is in between 36ms - 48ms.
- But few primes took double the time to get generated.



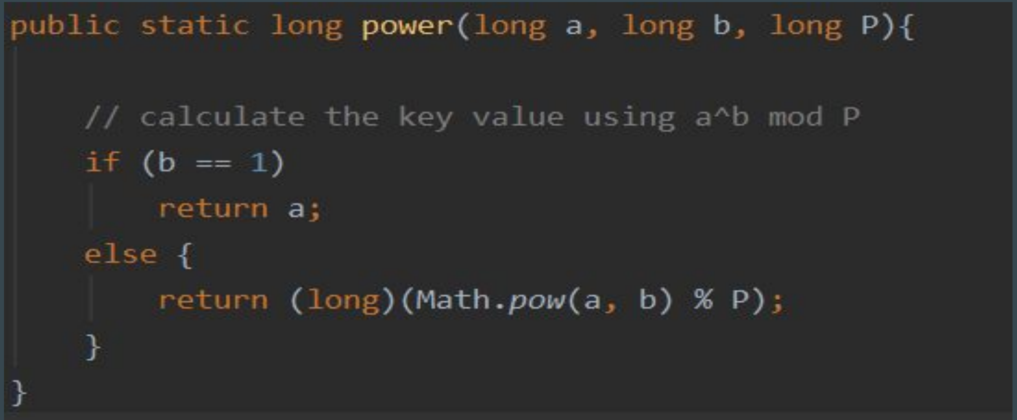
Graph(times in incremental way)



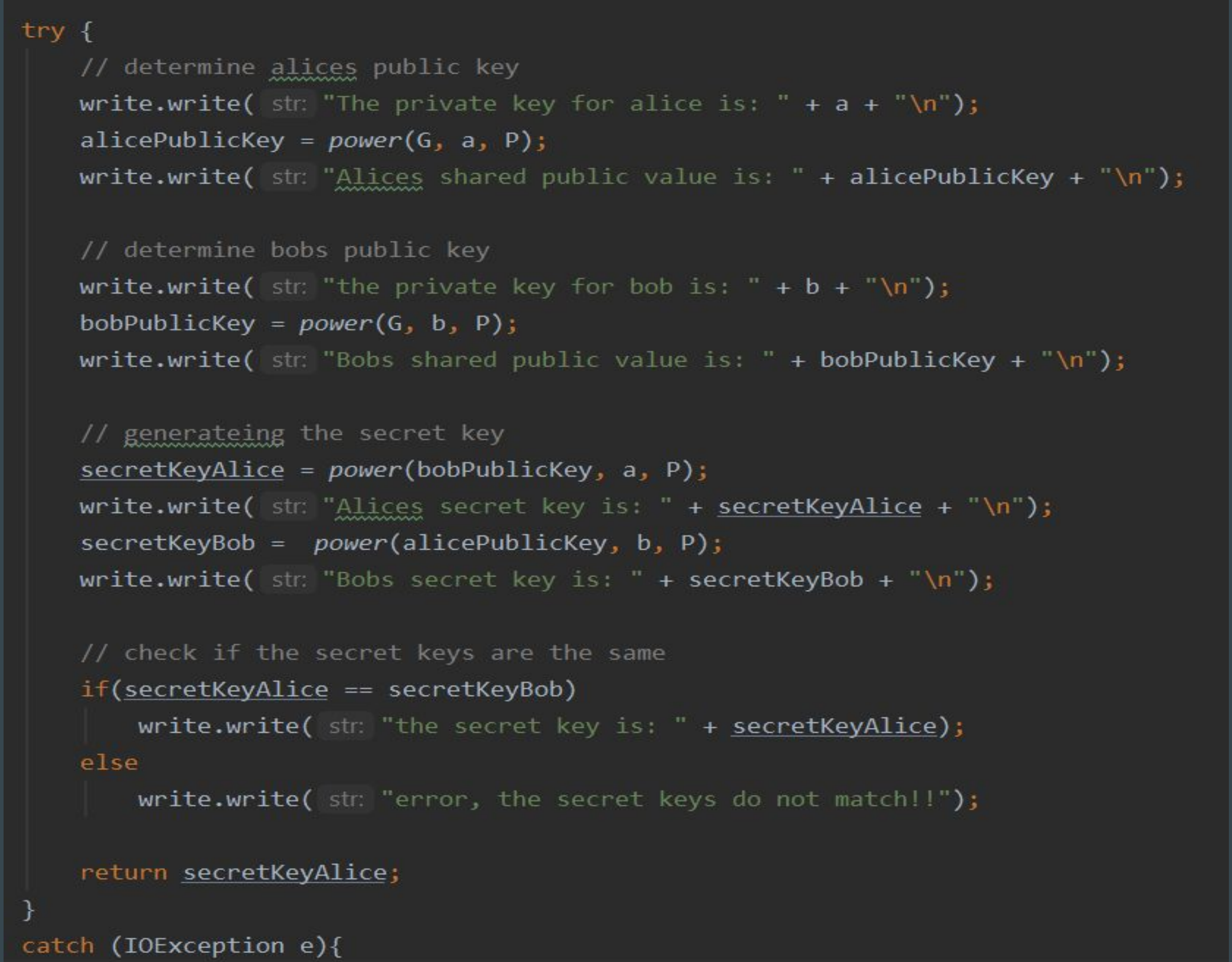
Third Requirement

Diffie-Hellman Algorithm

- We were forced to choose lower values for ‘a’, ‘b’ and ‘p’ to implement this requirement as per the limitations of the programming language.
- As the P value increases, the value of function (g pow a) will be lower than P and the public key (g pow a mod P) becomes same as (g pow a).
- The values ‘a’ and ‘b’ are generated from the secret text passwords separately for Alice and Bob.

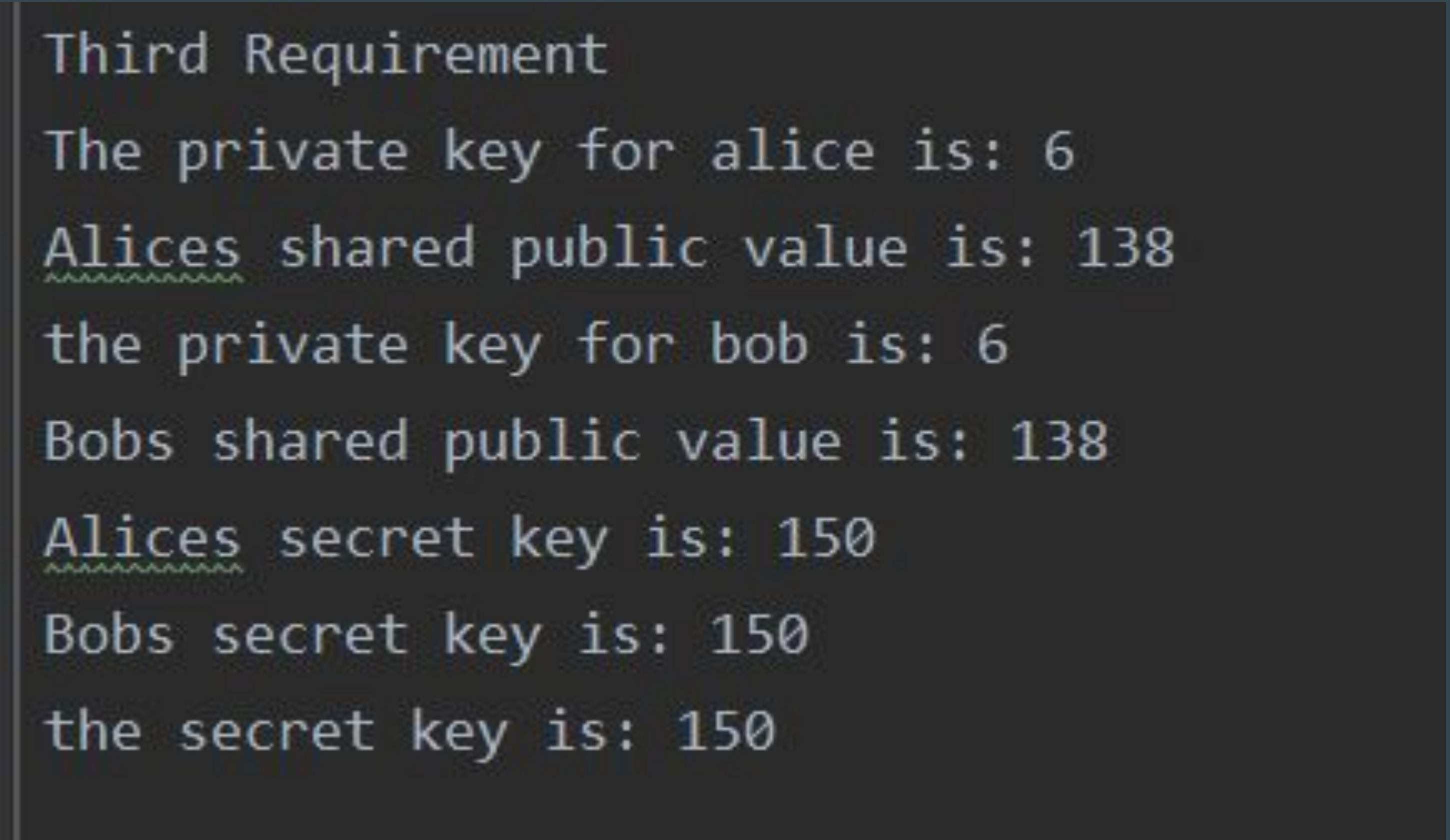


```
public static long power(long a, long b, long P){

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



```
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){
```

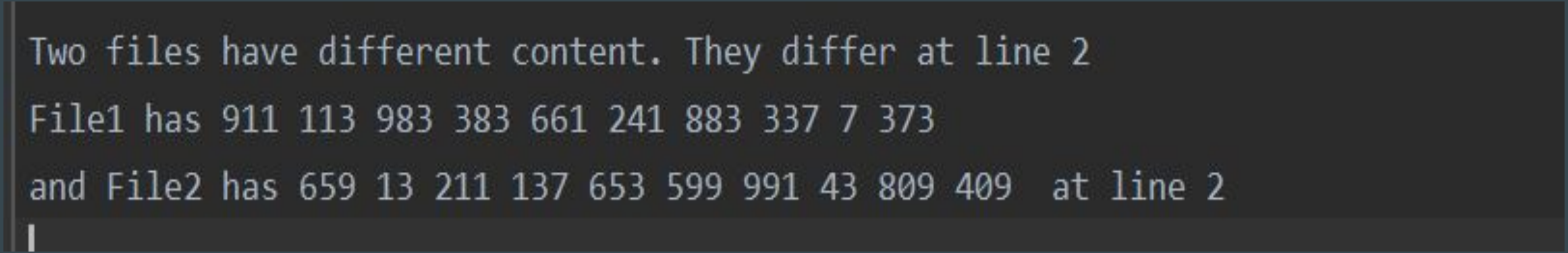
Output



```
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
```

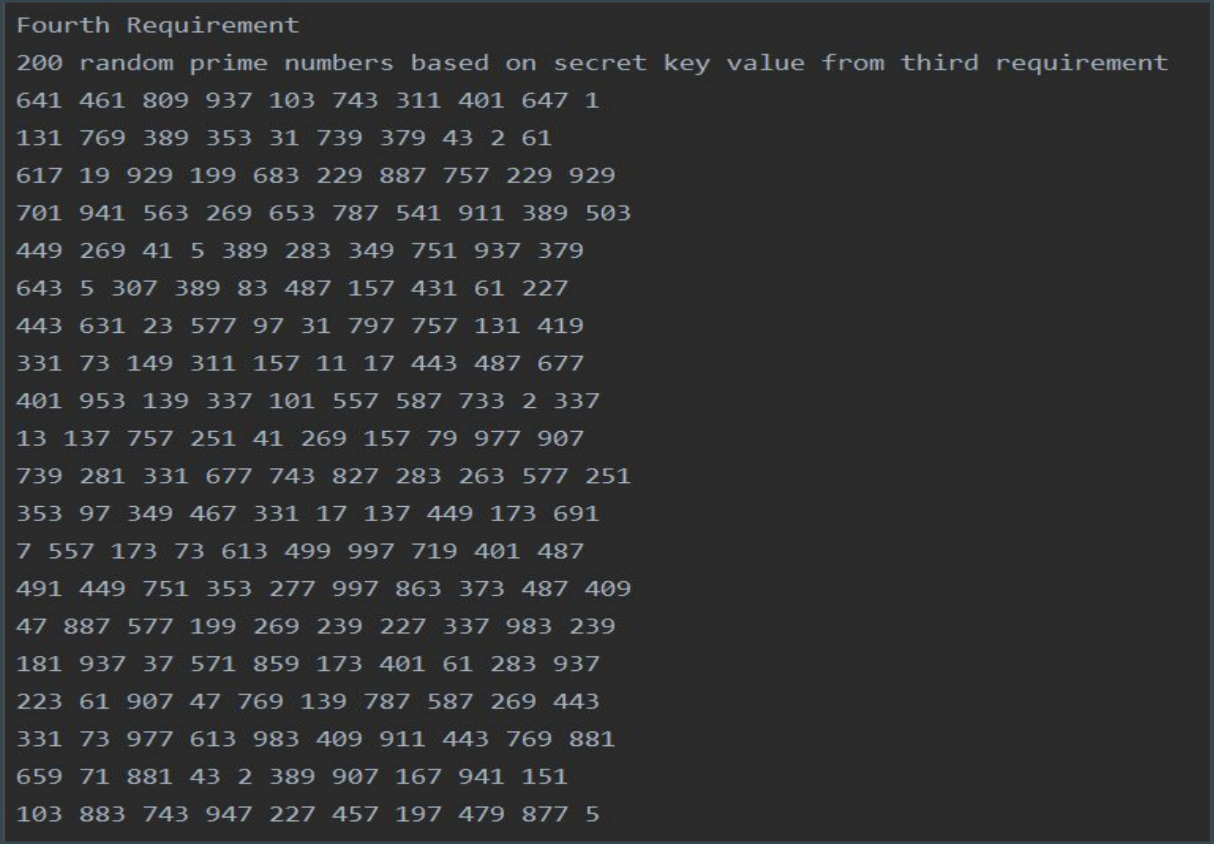
Fourth Requirement

- Generated random integers based on the seed value or shared key from the third requirement.
- We have used the same code of the first requirement.
- But the random numbers generated were different as the seed value has changed.
- We noticed that as long as the seed value is same, we can get the same set of random integers. If the seed value changes, the random set also changes.
- We compared both the files



```
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
```

Output



```
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
```

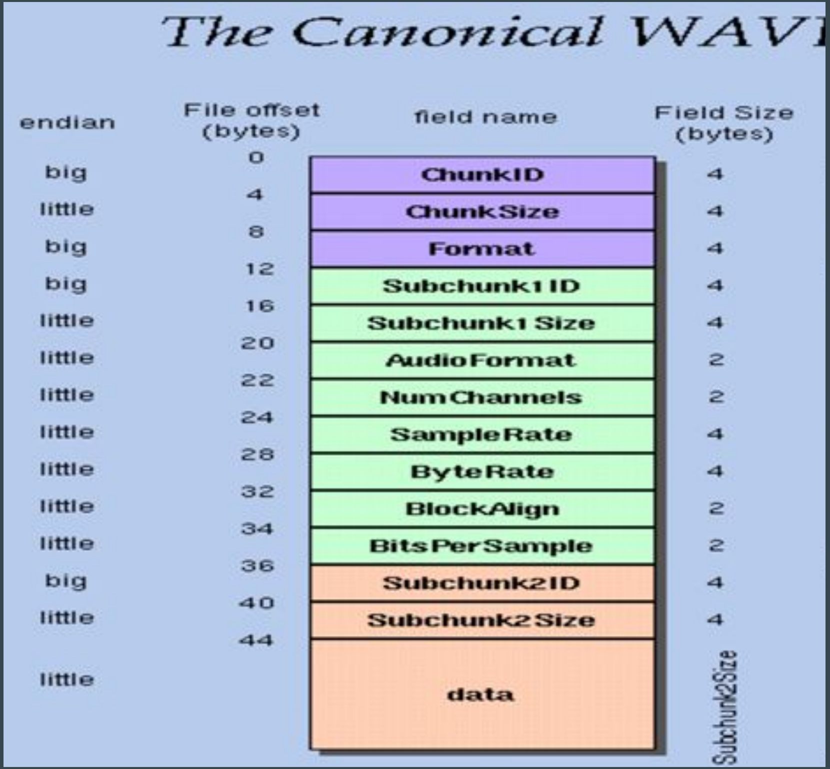
Fifth Requirement

- Fisher - Yates shuffling algorithm
- Shuffle
  - We Store our bytes for our audio file in a list of strings (every byte is stored as a string)
  - Going backwards over the size of our array containing the bytes, and we mark the index positions that each of one of those bytes is going to swap with, we create an integer array for these index positions
  - Those index positions are found by using our random number generator (RNG) with seed value to generate a random number between 0 and that specific index position
  - We continue going backwards through the size of the array until every byte has an index position to swap with
  - Then based on the index information we go backwards through the array containing the bytes and swap the bytes based on those index positions
  - We now have our shuffled array of bytes

- Fisher - Yates shuffling algorithm
- Unshuffle
  - The same logic applies here, but in reverse
  - Going forward through the array of bytes, we swap all the bytes in the array with the index positions until the end of the array is reached
  - At this point all bytes from the array are swapped back to their original index positions

How did we shuffle/unshuffle the audio?

- We found an audio clip in this case, clip of running water
- The audio was originally 11 seconds long
- We trimed it down to 1 second to reduce the number of bytes in the file
- We then used a hex editor to extract the data bytes of the file (everything starting after byte 44)
- Finally we used the Fisher Yates shuffling algorithm with a large key value from the third requirement to shuffle/unshuffle the data bytes

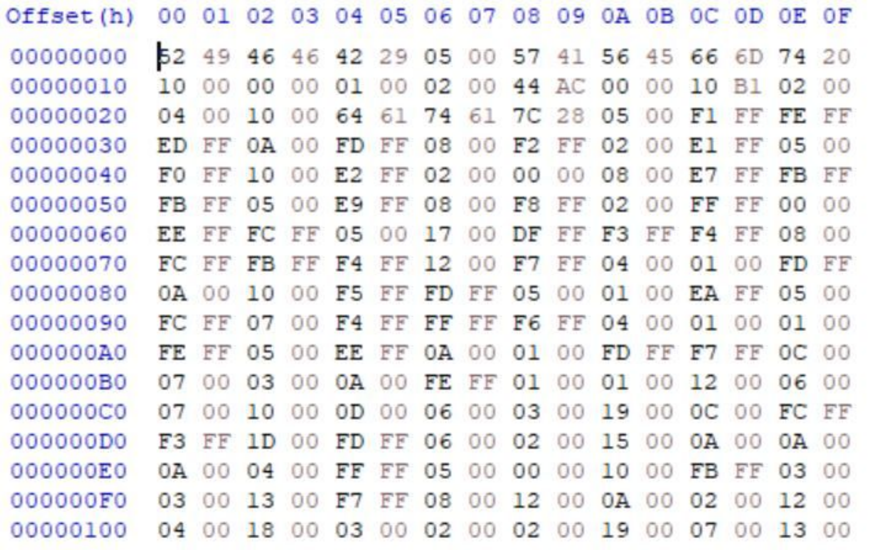


The Canonical WAV

endian	File offset (bytes)	field name	Field Size (bytes)
big	0	ChunkID	4
little	4	ChunkSize	4
big	8	Format	4
big	12	Subchunk1 ID	4
little	16	Subchunk1 Size	4
little	20	AudioFormat	2
little	22	NumChannels	2
little	24	SampleRate	4
little	28	ByteRate	4
little	32	BlockAlign	2
little	34	BitsPerSample	2
big	36	Subchunk2ID	4
little	40	Subchunk2Size	4
little	44	data	Subchunk2Size

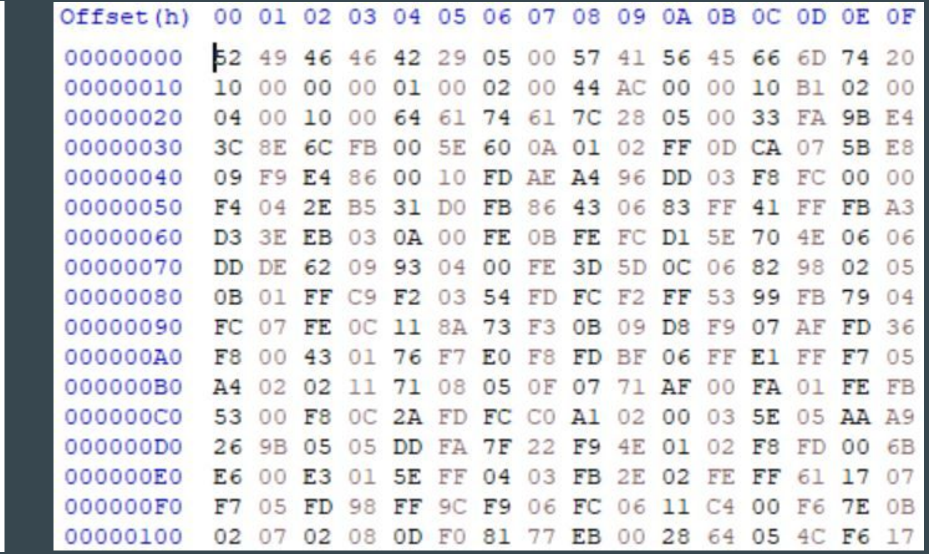
Bytes in the Hex File

original/unshuffed bytes
header info ends at byte 44
audio data start at byte F1



```
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
```

shuffled bytes
header info ends at byte 44
audio data start at byte 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
```

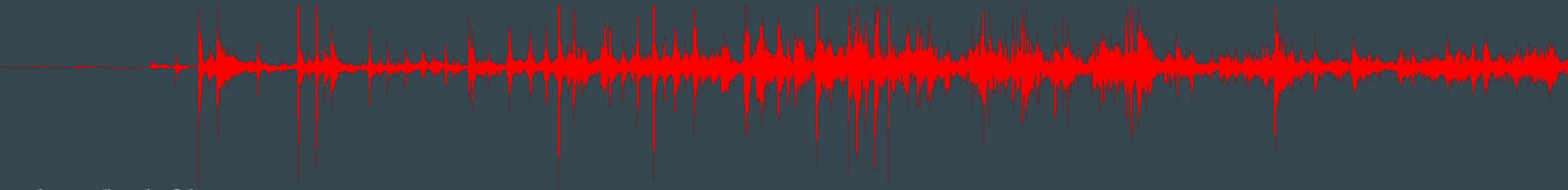
Audio files

🔊 pour_glass_water_shuffled.wav - original

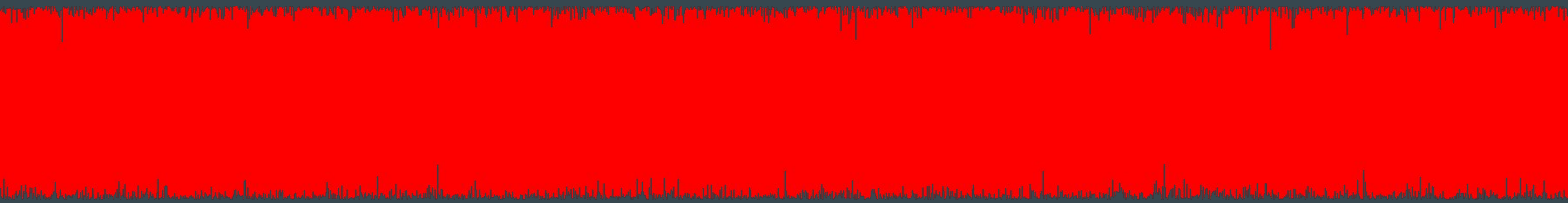
🔊 pour_glass_water_shuffled.wav

🔊 pour_glass_water_unshuffled.wav

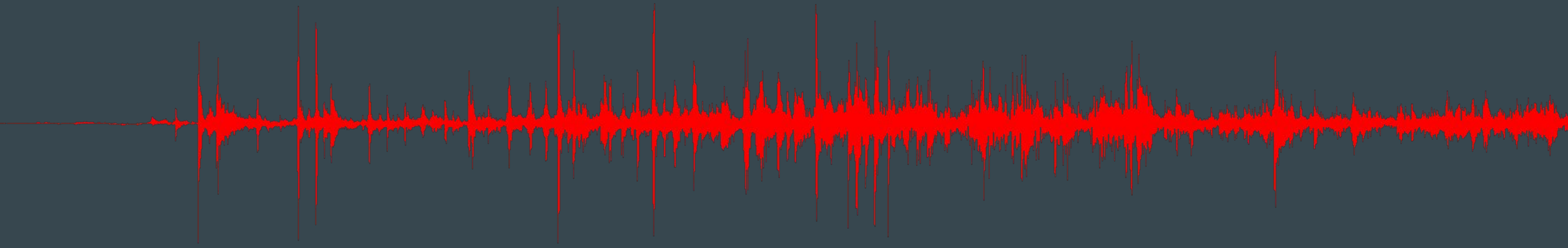
Original file



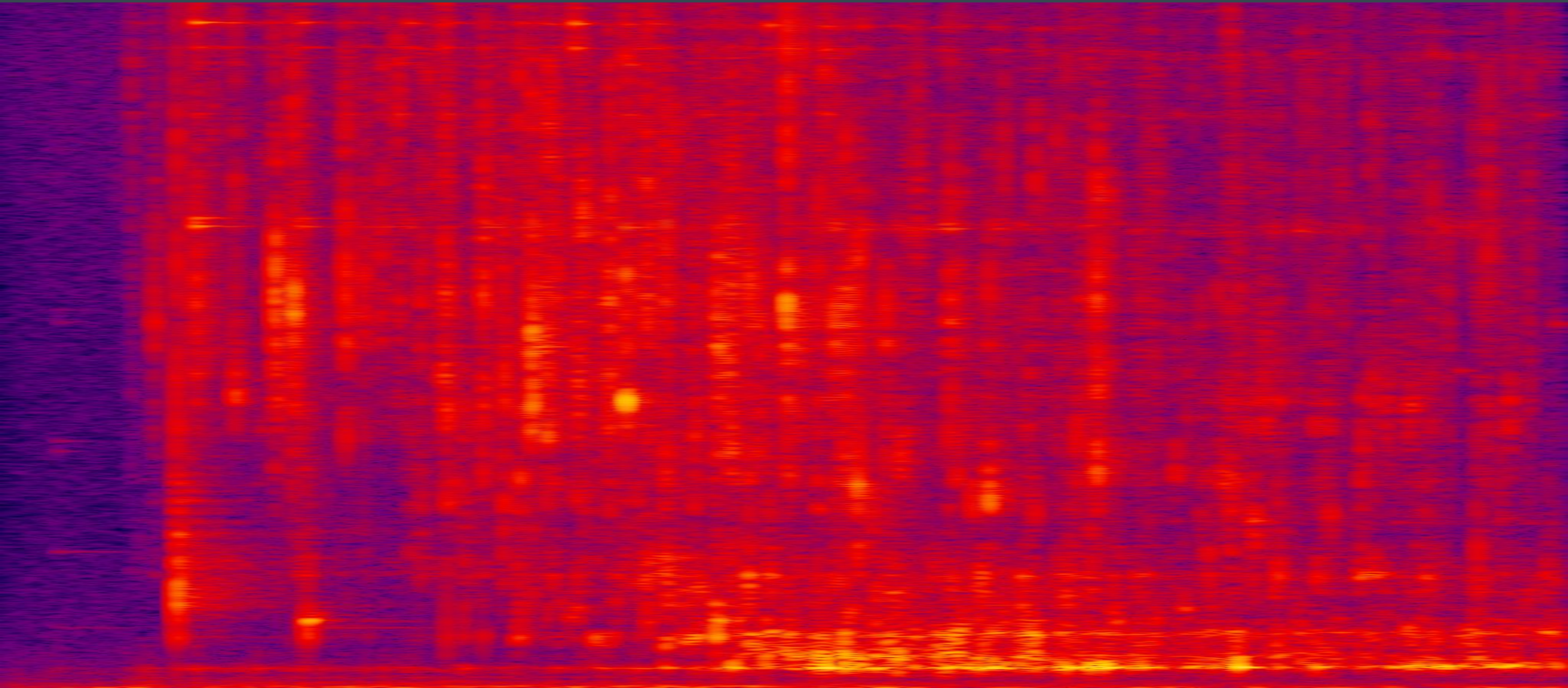
Shuffled file



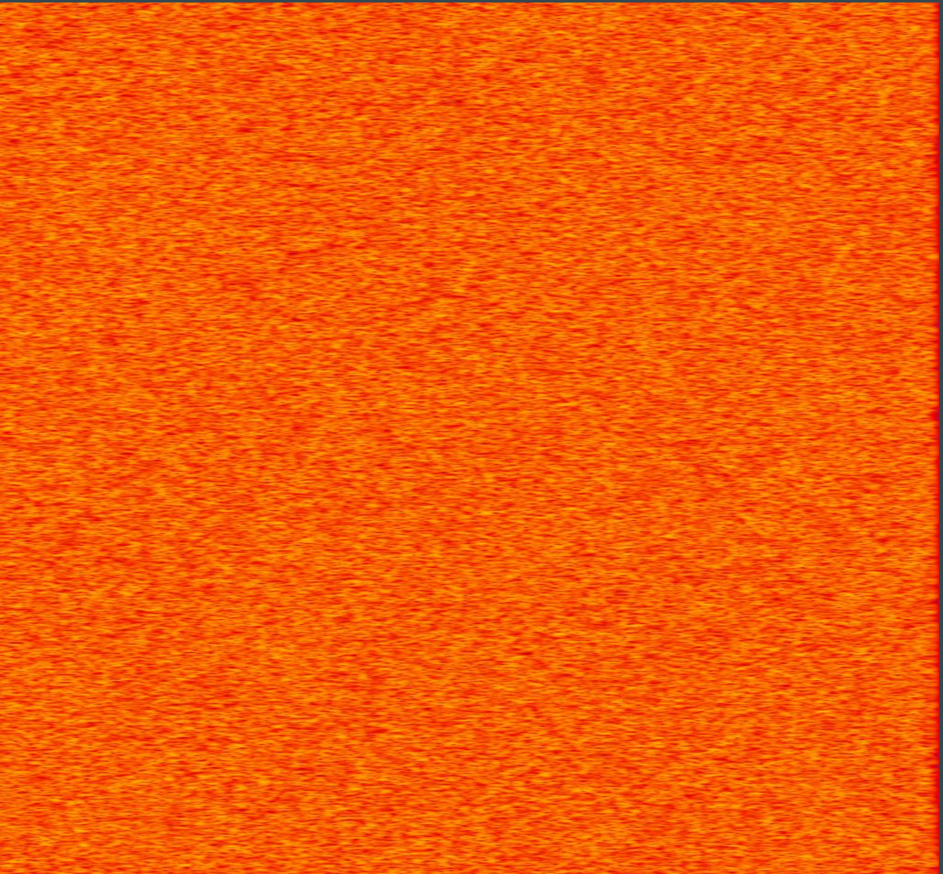
Unshuffled file



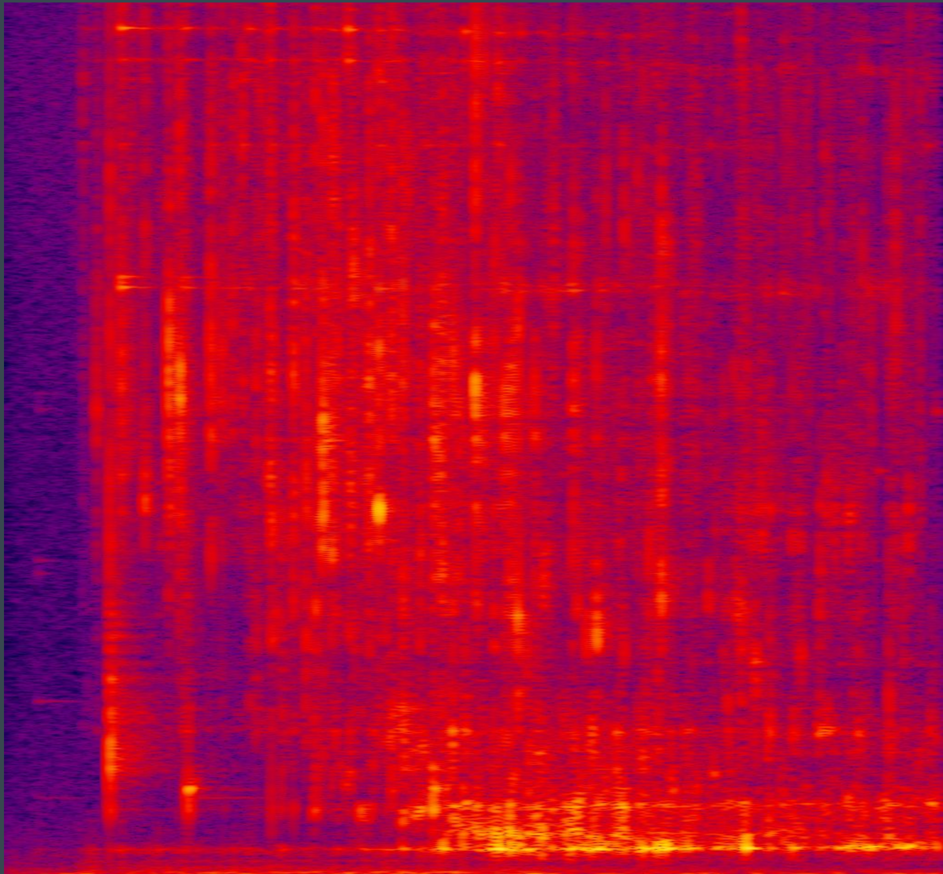
Spectrogram of original audio file



Shuffled



Unshuffled



Questions?

References

https://stackoverflow.com/questions/3541378/reversible-shuffle-algorithm-using-a-key

https://www.geeksforgeeks.org/shuffle-a-given-array-using-fisher-yates-shuffle-algorithm/

https://www.youtube.com/watch?v=tLxBwSL3IPQ&t=200s

https://nces.ed.gov/nceskids/createagraph/

https://convert.ing-now.com/mp3-audio-waveform-graphic-generator/download/spectrogram/8444003617f7769ce00d9c0cef3d972c/?v=1585196907204/