PRP Project Report

How to Run:

1. Run the file “prp_m12804663/src/ PseudorandomPermutation.py”
    a. This will run through the permFamily, prpGen, EncCBC, and DecCBC functions with default values and should output a combination of the pictures below
2. If you would like to pass inputs, there are four options:
    a. EncCBC
        i. PseudorandomPermutation.py EncCBC
        ii. PseudorandomPermutation.py EncCBC 100111000011 4 1100 ../data/pseudopermutations.txt ../data/ciphertext.txt
    b. DecCBC
        i. PseudorandomPermutation.py DecCBC
        ii. PseudorandomPermutation.py DecCBC 4 1100 ../data/pseudopermutations.txt ../data/ciphertext.txt
    c. prpGen
        i. PseudorandomPermutation.py prpGen
        ii. PseudorandomPermutation.py prpGem 4 4 ../data/pseudopermutations.txt
    d. permFamily
        i. PseudorandomPermutation.py permFamily
        ii. PseudorandomPermutation.py permFamily 4 ../data/permutations.txt

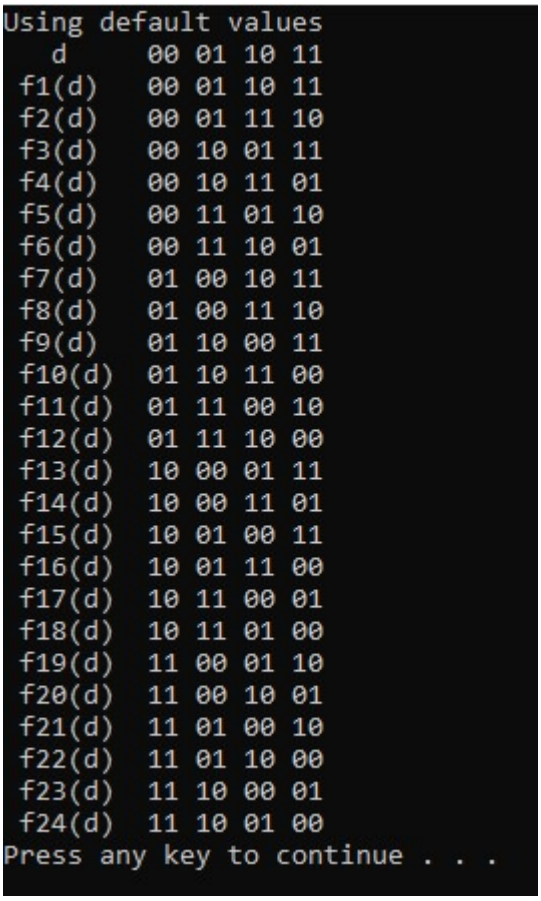
Description:

For this project, I decided to write the pseudo random permutation with CBC mode in the latest version of Python within Visual Studio 2019 and worked in Windows OS

PermFamily:

This function generates permutations for a permutation family with a given size n and writes it to “../data/permutations.txt”.

I programmed this function by first generating all the binary strings with the given length n and set each element in an array. Then I printed and wrote the d values (the column values) of the permutation table. Next, I generated the permutations of the array of binary strings using itertools permutations function and printed and wrote each of the rows of permutations.



PrpGen:

This function generates permutations for a pseudorandom permutation with each key having l bits and each d and r have n bits and writes it to the file "../data/pseudopermutations.txt".

I programmed this by separately generating all the binary strings of length n and length l and then generating a random IV to implement the probability factor. Next, I printed and wrote each d value from the generated strings. Then, I created all the key values from the generated strings of length l. Next I created the permutations using the set of generated strings using itertools permutations function. I then looped through the permutations and printed and wrote each out.

```
Using default values
Random IV: 1101
         d       0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111

k=0000, f0(d)    0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
k=0001, f1(d)    0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1111 1110
k=0010, f2(d)    0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1110 1101 1111
k=0011, f3(d)    0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1110 1111 1101
k=0100, f4(d)    0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1111 1101 1110
k=0101, f5(d)    0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1111 1110 1101
k=0110, f6(d)    0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1101 1100 1110 1111
k=0111, f7(d)    0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1101 1100 1111 1110
k=1000, f8(d)    0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1101 1110 1100 1111
k=1001, f9(d)    0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1101 1110 1111 1100
k=1010, f10(d)   0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1101 1111 1100 1110
k=1011, f11(d)   0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1101 1111 1110 1100
k=1100, f12(d)   0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1110 1100 1101 1111
k=1101, f13(d)   0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1110 1100 1111 1101
k=1110, f14(d)   0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1110 1101 1100 1111
k=1111, f15(d)   0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1110 1101 1111 1100
Press any key to continue . . .
```

EncCBC:

This function encrypts a message using CBC mode and writes it to "../data/ciphertext.txt".

I developed this function by first opening up the file, "../data/pseudopermutations.txt", which will return a pseudorandom permutation table generated from the previous function. Then I read each line and assigned them to variables. The first line of the file is the randomly generated IV. The second line includes the d values. And the rest of the lines are the k values and data points on the table. Then, I separated the input message into l lengths to make it easier to encrypt with the function table. Next, I looped through each element of the message list and checked if it was the first element. If it was, then the encryption would use the random IV. If not, it would use the previous message text. Then, it would generate an XOR of the input message chunk and the IV or the previous message text. After this, I check the table first by finding the correct key row associated with the given k input value. Then, for each XOR chunk generated, I find the correct d column and data point in the table. Then I combine each of these outputs into a single string, ciphertext, and output it to the console and file. Below are some examples of the same message being generated using different IV values. (Note, the IV is randomly generated by the prpGen function since decryption requires the IV to be known, although the project did not mention it, and this was the easiest way to do so)

IV = 1101

```
Using default values
Message list: ['1001', '1100', '0011']
XOR list: ['0100', '1000', '1011']
Ciphertext: 010010001011
Press any key to continue . . .
```

IV = 1010

```
Using default values
Message list: ['1001', '1100', '0011']
XOR list: ['0011', '1111', '1100']
Ciphertext: 001111111110
Press any key to continue . . .
```

IV = 1001

```
Using default values
Message list: ['1001', '1100', '0011']
XOR list: ['0000', '1100', '1111']
Ciphertext: 000011101111
Press any key to continue . . .
```

DecCBC:

This function decrypts a ciphertext using CBC mode and prints the message to the log

I created this function by first doing the same thing as encryption in retrieving each line of the permutation table and assigning each part to different variables. Then, I got the ciphertext from the file "../data/ciphertext.txt" and separated the ciphertext into l lengths to make it easier to use. Next, I found the correct key row with the given input k. For each ciphertext chunk, I found the associated function table d value using the correct k row and ciphertext chunk. Then I looped through each d value retrieved from the previous step and checked if it was the first one or not. If it was the first element, then I would use the IV value, otherwise I would use the previous ciphertext chunk value. Next, I generated an XOR of the d value and the IV or previous ciphertext value to create chunks of the final message. Then I combined the chunks and printed the decrypted message. Below is the output decrypting the first encryption image from earlier.

IV = 1101

```
Using default values
Cipher list: ['0100', '1000', '1011']
Function values: ['0100', '1000', '1011']
Message list: ['1001', '1100', '0011']
Final message output: 100111000011
Press any key to continue . . .
```