Week 2 – 30/09/2022

Exercício 1

```
import numpy as np

fst_prob = 0
snd_prob = 0

def solve(p, length):
    den = 2**length
    fst_index = (den%p)-1
    fst_prob = ((den//p)+1) / den
    snd_prob = (den//p) / den
    if (fst_index>0):
        print("Probabilidade de 0 a", fst_index, "=", N(fst_prob))
    print("Probabilidade de", fst_index+1,"a", p-1, "=", N(snd_prob))
    return N(fst_prob), N(snd_prob), fst_index
```

```
(p1,l1,f1) = solve(251,8)
```

length = 8 e p=251:

Probabilidade de 0 a 4 = 0.00781250000000000

Probabilidade de 5 a 250 = 0.00390625000000000

Esta probabilidade não é uniforme, mas podemos quantificar o quão distante esta distribuição está distante da uniforme através do cálculo da entropia.

Exercício 2

```
(p2,l2,f2) = solve(256,8)
```

length = 8 e p=256:

Probabilidade de 0 a 255 = 0.00390625000000000

Esta distribuição é uniforme, logo a entropia será o comprimento da bitstring, 8.

Exercício 3

Quando p não é um multiplo de 2^(length da bitstring), as distribuições não são completamente uniformes, pois existem sempre diferenças de probabilidades de um subconjunto de S para o outro.

Esta diferença é sempre 1/2^(length da bitstring).

Conforme aumentamos o tamanho da bit string, menores são estas diferenças, aproximando-se de uma distribuição uniforme.

Esta distância a uma distribuição uniforme é-nos dada pelo cálculo da entropia.

```
def entropy_general(l,k):
    den = 2**l
    fst_index = (den%k) - 1
    fst_prob = (den//k + 1) / den
    snd_prob = (den//k) / den

    expr1 = -fst_prob*math.log(fst_prob,2)*(fst_index+1)
    expr2=0
    if (snd_prob!=0):
        expr2 = -snd_prob*math.log(snd_prob,2)*(k-fst_index-1)
    return expr1+expr2
```

Entropia 1)

```
print(entropy_general(8,251))
```

7.9609375

A entropia deve ser entre 0 e o tamanho da bit string, ou seja, neste caso 8. A entropia é maximizada quando a distribuição é uniforme. Neste caso a distribuição é muito próxima da uniforme, logo a entropia é muito elevada.

Entropia 2)

```
print(entropy_general(8,256))
```

8.0

Como dito anteriormente, a distribuição é uniforme, logo a entropia é o tamanho da bit

string, ou seja, 8.

Exercício 4

Se k dividr o número de bitstrings possíveis (resto 0), as partições do conjunto C têm todas o mesmo tamanho, logo a probabilidade de cada partição é sempre a mesma, tendo assim uma distribuição uniforme.

Assim o tamanho do set (k) mais pequeno, para que a entropia seja o valor da entropia de uma distribuiçao uniforme é 2^(tamanho da bitstring). Ou seja, para um tamanho de bitstring=3, k deve ser 2^3.

Sempre que o tamanho do set (k) é um multiplo de 2^(tamanho da bitstring) a distribuição é uniforme, logo a entropia é o tamanho da bitstring.

Entropia

```
def entropy_general(l,k):
    den = 2**l
    fst_index = (den%k) - 1
    fst_prob = (den//k + 1) / den
    snd_prob = (den//k) / den

    expr1 = -fst_prob*math.log(fst_prob,2)*(fst_index+1)
    expr2=0
    if (snd_prob!=0):
        expr2 = -snd_prob*math.log(snd_prob,2)*(k-fst_index-1)
    return expr1+expr2
```

Exercício 5

Começando pela flag '-n', esta flag diz que apenas serão considerados 'length' bytes do input obtido, ou seja, no contexto dado, 32 bytes. De seguida, a opção '-e', serve para formatar o output de acordo com uma 'format string' passada. Neste caso, a dada no exemplo. E por fim passamos o diretório a partir de onde se extrai aleatoriedade, o '/dev/urandom'

a) $dd if=/dev/random bs=32 count=1

b) $openssl rand -out -hex 32

Exercício 6

Conforme o tamanho da chave aumenta, o tempo de geração cresce exponencialmente pois é computacionalmente mais exigente gerar primos com tamanhos cada vez maiores, o oposto acontece quando se diminui o tamanho da chave.

Exercício 7

Conforme no exercício anterior, o aumento do tamanho da chave leva a um aumento consideravelmente maior do tempo de geração dos parâmetros, este processo é bem maior que o de geração de chaves RSA.