Oppgave 1

1a)

$$z_{i+4} = z_i + z_{i+1} + z_{i+2} + z_{i+3} \pmod 2$$

Denne følga er definert ved at hvert tall etter det fjerde er summen modulo 2 av de fire foregående.

1a.1)

$$K = 1000$$

Følgen blir:

$$\textcolor{red}{10001}100011000$$

Altså en periode på 5.

1a.2)

$$K = 0011$$

Følgen blir:

$$\textcolor{red}{00110}0011000$$

Fortsatt en periode på 5.

1a.3)

$$K = 1111$$

Følgen blir:

$$\textcolor{red}{11110}11110$$

Atter en gang er perioden 5.

Konklusjon: Denne følgen gir en periode på 5, som er helt absurd dårlig – bare én høyere enn lengden på nøkkelen.

1b)

$$z_{i+4} = z_i + z_{i+3} \pmod 2$$

1b.1)

$$K = 1000$$

Følgen blir:

$$\textcolor{red}{100011110101100}100011110101100$$

Altså en periode på 15.

1b.2)

$$K = 0011$$

Følgen blir:

$$\textcolor{red}{001111010110010}001111\ldots$$

Fortsatt en periode på 15.

1a.3)

$$K = 1111$$

Følgen blir:

$$\textcolor{red}{111101011001000}11110101\ldots$$

Atter en gang er perioden 15.

Konklusjon: Denne følgen gir en periode på $2^n - 1 = 15$, som er den *beste* perioden en LFSR med $n = 4$ kan ha.

Oppgave 2

$$\mathcal{P} = \mathcal{C} = \mathbb{Z}_{29}$$

Her er K første del av nøkkelstrømmen ($z_1 = K$), og videre er strømmen gitt som $z_{i+1} = x_i$.

NB: Når vi dekrypterer, er det den *dekrypterte* teksten som brukes som input etter første runde.

Kryptering og dekryptering er som i Vigènere gitt ved

$$e_z(x) = (x + z) \pmod{29}$$

og

$$d_z(y) = (y - z) \pmod{29}$$

2a)

Skrev et par funksjoner (mye likt Vigènere-implementasjonen) for å utføre dette…

Kunne så kjøre

```
encrypt 17 "GODDAG"
```

og fik `XURGDG` som kryptert tekst.

2b)

Skrev litt Haskell-kode – måtte gjøre dekrypteringen annerledes enn i Vigènere-implementasjonen, siden den skulle bruke forrige krypterte verdi om og om igjen.

En kjøring av

```
bl $ decryptNums 29 5 [23,8,23,12,21,2,4,3,17,13,19]
```

ga tilbake `STEINSPRANG`

Matematisk:

$$x = 23\ 08\ 23\ 12\ 21\ 02\ 04\ 03\ 17\ 13\ 19, \ \ k = 5$$

$$
\begin{aligned}
x_1 &= (y_1 - k) \pmod{29} \\
x_2 &= (y_2 - x_1) \pmod{29} \\
&\vdots \\
x_n &= (y_n - x_{n-1}) \pmod{29}
\end{aligned}
$$

(men å gjøre det for hånd er *altfor* omstendelig, y'know)

Oppgave 3 - HMAC

$$
\begin{aligned}
K &= 1001 \\
\text{ipad} &= 0011 \\
\text{opad} &= 0101
\end{aligned}
$$

Funksjonen h kan enkelt lages med litt bitshifting og maskering:

```
h = lambda x: ((x*x % (256)) >> 2) & 15
```

Selve HMAC krever *flere* bitshifts til venstre for å konkatenere bitmønstrene.

```
hmac = lambda x: h(
    ((K^opad)<<4) # 4 bits
    +h(((K^ipad)<<4)+x) # 8 bits
)
```

3a)

Koden jeg skrev gir `0100` som HMAC av `0110`.

3b)

Jeg får `hmac(0b0111)` til å bli `0100` – den *stemmer faktisk* med den oppgitte HMAC-en, så det *trenger ikke* være grunn til å tro at meldingen *ikke* er autentisk.

Jeg og avsender vet nøkkelen, ingen andre skal vite det → *ren flaks* kreves for å forfalske HMAC-en. Men trengs det mye flaks for å gjette en HMAC her? Siden mønstrene i oppgave 3a og 3b begge gir samme HMAC, aner jeg et problem.

De mulige bitmønstrene er $\mathcal{P} = 0, 1^4$. Skrev et par linjer for å teste fordelingen av mulige HMAC-outputs:

```
hmacs = Counter(hmac(i) for i in range(16))
print('\n'.join(f'{k:04b}: {v}' for k, v in hmacs.items()))
```

Får da at fordelingen av bitmønstre som denne HMAC mapper til, er:

```
0000: 6
0001: 2
0100: 6
1001: 2
```

Det er altså *lurt* å velge `0100` som en forfalsket HMAC – sannsynligheten er $\frac{6}{2+2+6+6} = \frac{6}{16} = \frac{3}{8} = 37,5\%$ for at dette er den rette! (nå skal det sies at angriper ikke vet nøkkelen, så angriper vet ikke *nødendigvis* at dette er en lur HMAC-verdi)

Tilsvarende går det an å si at det er $\frac{1}{6} \approx 16,7\%$ sannsynlighet for at denne meldingen er den som ga HMAC `0100`. Det er *ikke* en spesielt høy sannsynlighet, så jeg vil si at vi *ikke med sikkerhet* (eller, med 16.7% sikkerhet) kan si at `0111` er en autentisk melding.

Oppgave 4 - CBC-MAC

Ut fra definisjonen ser jeg at $\vec{y}$ kan skrives slik:

$$\vec{y} = [IV,\ e_K(IV \oplus x_1),\ e_K(e_K(IV \oplus x_1) \oplus x_2),\ e_K(e_K(e_K(IV \oplus x_1) \oplus x_2), \oplus x_3)]$$

- og vi er *bare* ute etter y_1 tom. y_n

I Haskell var det så enkelt som dette, siden dette er en left fold:

```
cbcmac k = drop 1 . reverse . foldl (\ys x -> ((x `xor` head ys) + k) `
    mod` 16 : ys) [0]
```

Eller litt ryddigere:

```
cbcmac k = drop 1 . reverse . foldl (\ys x -> cipher (head ys) x : ys)
    [0]
  where cipher y x = ((x `xor` y) + k) `mod` 16
```

Eller *enda enklere*, siden en `foldl` der vi vil spare på hvert resultat er en typisk operasjon:

```
cbcmac k = drop 1 . scanl cipher 0
  where cipher y x = ((x `xor` y) + k) `mod` 16
```

(der xor-ingen av input til chifferet er lagt inn i chifferfunksjonen for å gjøre første linje kortere, det er egentlig litt misvisende)

Resultatet:

`CBC-MAC` av x → `0000 0010 1011 1101`

`CBC-MAC` av x' → `0101 1100 0000 0010`

Oppgave 5 - AES-kryptering

Med rundenøkkelen `67 71 35 c4 ff da e5 ff 1c 54 e1 fd 7f 2e 88 b7` skulle vi utføre én runde av en forenklet versjon av AES, uten `mixColumns`-steget.

Dette inkluderte ikke engang `addRoundKey` på slutten av første runde, siden vi bare fikk oppgitt én rundenøkkel.

5a)

Krypteringen besto av

1. En første `addRoundKey`
2. `subBytes`, der hver byte ble erstattet med et oppslag i S-boksen
    - (bare implementert som en hardkodet tabell)
3. `shiftRows`, som simpelthen bruker `numpy.roll` for det den er verdt.

Krypterte `24 59 66 0c 99 da 9b 00 d6 55 fd 20 e9 ff 46 95` og fikk `1A 33 74 90 63 7C 3E 34 9C 8B ED F3 93 E8 16 C1`

Dekryptering ga samme svar tilbake.

5b)

Dekrypteringen besto av

1. Invers av `shiftRows` → motsatt retning på rotasjonene
2. Invers av `subBytes` → oppslag i invers S-boks
3. Invers av `addRoundKey` → akkurat det samme, siden $\oplus k$ er sin egen invers

Dekrypterte `26 FA 83 E7 2D CD 5D B8 C4 DC EB 12 70 CF D6 1E` og fikk `44 05 94 af 2e ce d4 bd 09 af a0 05 5e c6 14 07`

Oppgave 6 - AES-nøkkelstrøm

(kildekoden til denne oppgaven ligger også i `aes.py`)

Skrev altfor spesifikk kode for akkurat dette tilfellet (vi skulle bare finne 6 av ordene, og de 4 første er gitt…)

Oppsummert i de to mest relevante linjene (der `RCON` er en ferdiglagd tabell):

```
words[4] = words[0] ^ np.roll(words[3], -1) ^ [RCON[1],0,0,0]
words[5] = words[1] ^ words[4]
```

Gikk ut fra wikipedia-artikkelen om dette, som ga en ganske fin beskrivelse av metoden.

Linjene ovenfor tilsvarer

$$W_{i-N} \oplus \text{RotWord}(W_{i-1}) \oplus rcon_{i/N}$$

og

$$W_{i-N} \oplus W_{i-1}$$

De seks første ordene i denne forenklede nøkkelstrømmen:

```
2B 7E 15 16
28 AE D2 A6
AB F7 15 88
09 CF 4F 3C
E5 31 29 1F
CD 9F FB B9
```