Sichere Programmierung
Projekt 1

Julian Sobott
(76511)
David Sugar
(76050)

1 Zu Aufgabe 1

Aus der Aufgabenstellung war gegeben, dass die Funktion `decode(text)`, die Buchstaben des übergebenen Textes in entsprechende Zahlen aus $\mathbb{Z}_{26}$ umwandeln und diese dann in einer Liste zurückgeben soll. Daraus ergibt sich der Definitionsbereich $D = \{a, ..., z\}$ und Wertebereich $W = \{0, ..., 25\}$ mit $f : D \to W$ für die Symbole und $decode() : D^* \to W^*$ für Wörter beliebiger Länge.

$f : D \to W$ wird durch `alph_to_num` realisiert, einem Python `dict`, dass von ascii Kleinbuchstaben aufsteigend auf die Zahlen von Null bis 25 abbildet und wiederum innerhalb von `decode()` in einer Schleife verwendet wird um jeden einzelnen Buchstaben des übergebenen Textes umzuwandeln. Werte außerhalb des Definitionsbereiches werden vom gegebenen Algorithmus ignoriert.

```
1 alph_to_num = {k:v for v , k in enumerate(string.ascii_lowercase)}
```

1.1 Zu Aufgabe 2

Die Funktion `encode(text)` stellt die Umkehrfunktion von `decode()` dar, für alle $w \in \{a, ..z\}^*$. Sie nimmt als Eingabe eine Liste von Zahlen $a \in \mathbb{Z}_{26}$ und gibt eine entsprechende Zeichenkette (String) zurück.

Das Abbilden von Zahlen auf die entsprechenden Buchstaben wird durch $num\_to\_alph : \{0, .., 25\} \to \{a, .., z\}$ realisiert.

```
1 num_to_alph = {v:k for v , k in enumerate(string.ascii_lowercase)}
```

Um den String schlussendlich zu bauen benötigt es dann nur einen Einzeiler.

```
1 "".join([ num_to_alph[d] for d in int_list ])
```

Dadurch, dass `decode()` und `encode()` jeweils Funktion und Umkehrfunktion darstellen ergibt sich: w = encode(decode(w)).

1.2 Zu Aufgabe 3

Um das gewünschte Dictionary `key_table` zu erstellen, haben wir die gleichnamige Funktion `key_table(m: int)` implementiert. Diese ermöglicht es nicht nur `key_table` für $\mathbb{Z}_{26}$, sondern allgemein für $\mathbb{Z}_m, m \geq 2$ anzulegen. Dabei wird mithilfe einer For-Loop

über alle $a \in \{1, 2, .., m-1\}$ iteriert und deren multiplikativ Inverses, mithilfe von `mcrypt.mul_inverse(n, m)`, berechnet. Falls ein mult. Inverses existiert wird dieses dann nach dem Schema $a : a^{-1}$ dem Dictionary hinzugefügt. Sollte $m < 2$ oder ein invalider Datentyp übergeben werden, wird `None` zurückgegeben. Andernfalls gibt die Funktion das erstellte Dictionary zurück.

```
for i in range(m):
     i_neg = mul_inverse(i, m)  # Berechnung d. mult. Inv.

     if i_neg != None:
         d[i] = i_neg       # Es gibt ein mult. Inv. -> add to dict
```

1.2.1 Berechnung des multiplikativ Inversen

Zur Berechnung des multiplikativ Inversen wird die Funktion `mul_inverse(n: int, m: int)` genutzt, die auf dem erweiterten Euklid'schen Algorithmus beruht. Grundsätzlich gilt, wenn es zu $n \in Z_m$ eine Zahl $x \in \mathbb{Z}_m$ gibt mit $n * x = 1 (mod\, m)$, so wird x als multiplikativ Inverses zu n in m bezeichnet, schreibe n^{-1} oder $\frac{1}{n}$.

Die Definition des multiplikativ Inversen bedeutet jedoch, dass sich $n * x$ und 1 um ein Vielfaches von m unterscheiden, d.h. $n * x + m * y = 1$. Für Gleichungen dieser Art kann nun x und y mithilfe des erweiterten Euklid bestimmt werden.

Anfangs wird jedoch erst einmal geprüft, ob n teilerfremd zu m ist und parallel die einzelnen Teiler jeder Division für später in einer Liste gespeichert.

```
q = []                  # Liste von Teilern

while m != 0:
    q += [n // m]       # Teiler hinzufügen
    (n, m) = (m, n % m)

if n != 1:
    return None         # n und m Teilerfremd ?
```

Dies ist zwingend notwendig, denn sollte gelten, dass n und m einen gemeinsamen Teiler $t > 1 \in \mathbb{Z}$ besitzen, dann gilt: $n = t\hat{n}$ und $m = t\hat{m}$, d.h. $nx = 1 (mod\, m) = t\hat{n}x - t\hat{m}y = 1 = t(\hat{n}x - \hat{m}y) = 1$. Es gibt jedoch kein $t > 1$, das diese Gleichung erfüllt, demnach müssen n und m teilerfremd sein.

Danach werden mittels der Teiler aus q, x und y berechnet, wobei nur x von Interesse ist und als Rückgabewert dient. Das letzte Statement dient dazu, dass x in $\mathbb{Z}_m$ liegt.

```
q.reverse()
x = 1
y = 0

for t in q:
    _x = y
    _y = x - (_x * t)
    x = _x
```

```
9       y = _y
10
11  return (x + module) % module
```

1.3 Zu Aufgabe 4

`acEncrypt(a, b, text)` nutzt zuerst `decode()` um mithilfe des gegebenen Textes eine entsprechende Liste von Ganzzahlen zu erzeugen.

```
1  t = decode(plain_text)
```

Danach wird über jedes Element der Liste iteriert und dieses mithilfe des Schlüssels `(a, b)` verschlüsselt.

```
1  for i in range(len(t)):
2      t[i] = (a * t[i] + b) % module
```

Schlussendlich wird die verschlüsselte Liste an `encode()` übergeben, die alle Zahlen wieder in einen String umwandelt, der danach zurückgegeben wird.

```
1  e = encode(t)
2  return e.upper()
```

Anfangs prüft die Funktion, ob überhaupt die richtigen Datentypen übergeben wurden. Sollte dies nicht der Fall sein, wird vom `logger` eine entsprechende Nachricht mit dem Level `Warning` ausgegeben und ein leerer String zurückgegeben. Sollte a nicht teilerfremd zu 26 sein, wird ebenfalls eine Nachricht geloggt, dieses mal mit dem Level `Info` und ein leerer String zurückgegeben.

Damit Einträge des Levels Info angezeigt werden, muss das jeweilige Script mit der Option `'-v' (verbose)` ausgeführt werden.

1.4 Zu Aufgabe 5

Die Fehlerprüfung von `acDecrypt()` ist identisch zu der von `acEncrypt`.

Nachdem auf Fehler geprüft wurde, wird als erstes dafür gesorgt, dass der Teilschlüssel a in $\mathbb{Z}_{26}$ liegt. Dadurch wird sichergestellt, das bei einem späteren Hash-Table lookup auch an der richtigen Stelle 'gesucht' wird.

```
1  a = a % module
```

Danach wird ein entsprechendes `key_table` erzeugt.

```
1  table = key_table(module)
```

Schlussendlich wird mit $(y-b)*a^{-1} = x$ jede Ziffer des übergebenen Cipher-Textes wieder entschlüsselt und zu einem String zusammengefügt, der als Rückgabewert dient. a^{-1} wird dabei in `table` mithilfe von `a` nachgeschlagen.

```
return encode([ ((y - b) * table[a]) % modulo for y in decode(cipher_text) ])
```

1.5 Zu Aufgabe 6

```
if __name__ == '__main__':
    # Aufgabe 6
    pt = "strenggeheim"
    k1 = "db"
    ct = "IFFYVQMJYFFDQ"
    k2 = "pi"

    k1_1, k1_2 = decode(k1)
    k2_1, k2_2 = decode(k2)

    ptoc = acEncrypt(k1_1, k1_2, pt)
    ctop = acDecrypt(k2_1, k2_2, ct)

    print("Aufgabe 6:")
    print(ptoc)
    print(ctop)
```

```
Aufgabe 6:
DGANOTTNWNZL
affinechiffre
```

1.6 Zu Aufgabe 7

Datei wurde entsprechend benannt.

1.7 Zu Aufgabe 8