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)}
```

2 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)).

3 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
```

3.1 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
```

```
    y = _y

return (x + module) % module
```

---

4 Zu Aufgabe 4

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

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

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

```
for i in range(len(t)):
    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.

```
e = encode(t)
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.

---

5 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.

```
a = a % module
```

Danach wird ein entsprechendes `key_table` erzeugt.

```
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) ])
```

---

6 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
```

---

7 Zu Aufgabe 7

Datei wurde entsprechend benannt.

---

8 Zu Aufgabe 8

Das `affinecipher.py` Skript besteht aus zwei Teilen, der `crypt(path: str, key: str, fun)` Funktion sowie einem `__main__` Part, der nur ausgeführt wird, wenn man die Datei auch wirklich als Skript und nicht etwa als Modul nutzt.

8.1 crypt()

Die `crpyt()` Funktion hat drei Parameter, namentlich path, key und fun. Ihr wird der Pfad der zu ver- bzw. entschlüsselnden Datei als String, ein Schlüsselpaar ebenfalls als String, sowie entweder die Ver- oder Entschlüsselungsfunktion übergeben. Danach wird das Schlüsselpaar, mithilfe von `decode()`, in seine numerische Repräsentation umgewandelt. Schlussendlich wird die übergebene Funktion mit dem Schlüsselpaar sowie dem Inhalt der Datei als Argumente aufgerufen. Und der Rückgabewert auf der Kommandozeile ausgegeben.

```
1 def crypt(path: str, key: str, fun) -> None:
2     k1, k2 = decode(key)
3 
4     with open(path, "r") as f:
5         print(fun(k1, k2, f.read()))
```

Dadurch, dass die `acEncrypt()` und `acDecrypt()` Funktionen dieselben Schnittstellen besitzen, kann hier auf eine Funktion zurückgegriffen werden, der die jeweils passende Ver- oder Entschlüsselungsfunktion übergeben wird. Dies spart etwas Code und vermeidet Redundanzen.

8.2 ___main___

Hier steht der Großteil des Codes.

8.2.1 Kommand Line Arguments

Um an die jeweiligen Kommandozeilenargumente zu kommen, haben wir das `argparse` Modul importiert.

```
1 import argparse
```

Mit dem folgenden Methodenaufruf kann dann ein `ArgumentParser` erzeugt werden, dem außerdem eine Beschreibung des Skripts als String übergeben werden kann.

```
1 parser = argparse.ArgumentParser(description="Encrypt or decrypt \\
2 a file using the affine cipher")
```

Danach können dem erzeugten Parser-Objekt zu erwartende Argumente hinzugefügt werden. Es ist dabei möglich, Neben einem aussagekräftigen Namen, auch erwartete Werte sowie ein Hilfetext als Argumente zu übergeben.

```
1 parser.add_argument("mode", choices=["e", "d"], help="[e]ncrypt or \\
2 [d]ecrpyt the file")
3 parser.add_argument("key", help="String with exactly two lower \\
4 case ascii letters")
5 parser.add_argument("path", help="File path")
```

Mit dem Methodenaufruf `parse_args()` kann dann ein Objekt erzeugt werden, mit dem auf die Kommandozeilenargumente, wie auf normal Instanz-Attribute, zugegriffen werden kann.

```
args = parser.parse_args()

mode = args.mode
key = args.key
path = args.path
```

Der Vorteil von `argparse` gegenüber dem direkten aufrufen von `sys.argv` ist, dass auf einfache Weise nutzerfreundliche Kommandozeilenschnittstellen geschaffen werden können. Übergibt der Nutzer z.B. falsche Argumente, so wird ihm direkt der passende Hilfstext mit Nutzungsinformationen angezeigt.

Mit den obigen Statements erzeugt uns `parseargs` beim aufrufen des Skriptes mit `$./affinecipher.py -h` folgenden Output:

```
usage: affinecipher.py [-h] {e,d} key path

Encrypt or decrypt a file using the affine cipher

positional arguments:
  {e,d}       [e]ncrypt or [d]ecrpyt the file
  key         String with exactly two lower case ascii letters
  path        File path

optional arguments:
  -h, --help  show this help message and exit
```

8.2.2 Aufruf von crypt()

Je nach Betriebsmodus wird nach dem parsen der Argumente `crypt()` entweder mit der `acEncrypt()` oder der `acDecrypt()` aufgerufen, die sich wie oben beschrieben verhält.

8.3 Inbetriebnahme

Um das Skript wie ein normal Programm ausführen zu können, muss dieses wissen welchen Interpreter es für die nachfolgenden Zeilen verwenden soll. Um dies festzulegen wird der sog. `shebang ( #! )` genutzt, gefolgt von dem absoluten Pfad des zu verwendenden Interpreters.

```
#! /usr/bin/python3
```

Diese Zeile sagt aus, dass die Nachfolgenden Zeilen mithilfe des `python3` Interpreters ausgeführt werden sollen.

Danach müssen dem Skript außerdem noch Ausführungsrechte (eng. execute permission) hinzugefügt werden. Dies geschieht über die Kommandozeile mit:

```
$ chmod +x affinecipher.py
```

Alternativ kann statt `+x` auch der entsprechende Oktalwert genutzt werden.

Mit `ls -l` lässt sich dann ggf. noch überprüfen ob der vorherige Befehl richtig ausgeführt wurde.

```
$ ls -l
...
-rwxr-xr-x 1 sugar sugar  864 Okt 23 15:38 affinecipher.py
...
```

Die Ziffern ganz links sagen uns, dass die Datei `affinecipher.py` user, group und other Ausführungsrechte (genau in dieser Reihenfolge beschrieben) hat.

Mit den wenigen hier beschriebenen Zeilen Code und unter Verwendung der Module `argparse` und `aclib` kann das in Aufgabe 8 geforderte Command Line Tool bereitgestellt werden, das Affine Chiffren nutzt um gewünschte Texte zu ver- oder entschlüsseln.

```python
#! /usr/bin/python3

import sys
import argparse

from aclib import decode, acDecrypt, acEncrypt


def crypt(path: str, key: str, fun) -> None:
    """
    Encodes/ Decodes the file pointed to by 'path'
    using the specified function 'fun'.
    """
    k1, k2 = decode(key)

    with open(path, "r") as f:
        print(fun(k1, k2, f.read()))


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Encrypt or...")
    parser.add_argument("mode", choices=["e", "d"], help="[e]...")
    parser.add_argument("key", help="String with exactly two...")
    parser.add_argument("path", help="File path")
    args = parser.parse_args()

    mode = args.mode
    key = args.key
    path = args.path

    if mode == "e":
        crypt(path, key, acEncrypt)
    elif mode == "d":
        crypt(path, key, acDecrypt)
```

9 Zu Aufgabe 9

Um zu testen ob das in Aufgabe 8 beschriebene Skript funktioniert wird die Datei `klartext.txt` verschlüsselt. Zu beachten ist, dass der Pfad zu der Datei entweder relativ zum Skript oder absolut sein muss. In unserem fall ist der Pfad also: `../../klartext.txt`. Um mit key `pn` zu verschlüsseln, sieht der Aufruf also wie folgt aus:

```
$ ./affinecipher.py e pn ../../klartext.txt
```

```
EJMOPADXMVDAVMPWWVEIPZINLLDVIXEINROV
```

---

10 Zu Aufgabe 10

Um die Datei `geheimtext.txt` zu entschlüsseln muss der Pfad entsprechend angepasst werden und der Modus in `d` geändert werden:

```
$ ./affinecipher.py d pn ../../geheimtext.txt
```

```
diesisteinstrenggeheimergeheimtextbittevertraulichbehandeln
```

---

11 Zu Aufgabe 11

Die Funktion `computeFrequencyTable` bekommt eine Liste übergeben und erstellt ein dictionary mit den Häufigkeiten jedes Wertes. Die Werte die in der Liste stehen können beliebig sein. Im Programm enthält die Liste jedoch nur Werte aus $\mathbb{Z}_{26}$. Das heißt das Dictionary stellt die Funktion $f : \mathbb{Z}_{26} \to \mathbb{N}$ dar. Dafür wird über jedes Element in der Liste iteriert. Wenn es schon im dictionary ist, wird der Zähler um 1 hoch gezählt, sonst wird es mit Wert 1 hinzugefügt.

---

12 Zu Aufgabe 12

Die `printFrequencyTable` ist eine Hilfsfunktion, die die Häufigeiten jedes Buchstabens, formatiert in der Konsole ausgibt. Als Übergabeparameter erhält die Funktion ein Dictionary, wie es in Aufgabe 11 erstellt wurde. Für die Ausgabe ist nur eine Zeile an Code notwendig. Es wird über das Dictionary iteriert und jedes Element aus dem Dictionary wird ein Elemenmt in einer Liste. Dieses Elememt ist ein formatierter String, in dem auch die `encode` Funktion aufgerufen wird. Alle Listenelemente werden anschließend zu einem String zusammengefügt und durch ein '\n' getrennt.

```
print("\n".join([f"{encode([char])}: {freq}" for char, freq in table.items()])
```

---

13 Zu Aufgabe 13

Die Funktion `computeMostFrequentChars` soll die häufigsten Buchstaben als Liste zurückgeben. Als Eingabe wird ein dictionary wie es in `computeFrequencyTable` erstellt wird, übergeben. Da ein dictionary nicht sortiert werden kann, muss dies zuerst in eine Liste umgewandelt werden. Die Elemente sind Tupel aus key, value. Wichtig hierbei ist, dass die frequency (value) an erster Stelle im Tupel steht, da sie beim sortieren die höhere Priorität hat. Wie hier in dem Beispiel zu sehen, wenn aufsteigend sortiert wird.

```
>>> sorted([(2, 1), (1, 3)])
[(1, 3), (2, 1)]
```

Anschließend muss die Liste in **reverse** sortiert werden, um die häufigsten Buchstaben am Anfang zu haben. Der zweite Paramter `n` gibt an wie viele Element die Liste haben soll. Hierfür mit hilfe von slicing die Liste auf die gewünschte Größe kopiert. Zum Schluss wird mit Hilfe von 'list comprehension' noch eine Liste erstellt, die nur noch dekodierten Buchstaben (keys im Dictionary) enthält.

Obwohl der Algorithmus in einer Zeile Code möglich wäre haben wir uns im finalen Code für die ausfühliche Version entschieden. Dadurch istder Code besser lesbar und kann leichter erweitert werden.

Einzeiler:

```
return [tup[1] for tup in sorted(
    [(freq, char) for char, freq in freq_table.items()],
     reverse=True)[0:n]]
```

Finaler Code:

```
l = [(freq, char) for char, freq in freq_table.items()]
l.sort(reverse=True)
l_cut = l[0:n]
final = [tup[1] for tup in l_cut]
return final
```

---

14 Zu Aufgabe 14

Die Funktion `computeKeyPairs` gibt eine Liste zurück die durch die Menge $L = \{(a, b) | a, b \in char\_list \wedge a \neq b\}$ beschrieben wird. Hierfür muss in einer geschachtelten Schleife jeweils über die übergebene Liste iteriert werden und wenn die Elemente unterschiedlich sind, der neuen Liste angehängt werden.

```
return [(c1, c2) for c1 in char_list for c2 in char_list if c1 != c2]
```

---

15 Zu Aufgabe 15

Der Funktion `analyzeCipherText` wird ein verschlüsselter Text und eine Liste von Buchstabenpaaren aus Aufgabe 14 übergeben. Das Ziel ist nun aus den Buchstabenpaaren, Schlüsselpaare zu berechnen und damit den Geheimtext entschlüsseln.

15.1 Berechnung des Schlüssels

Verschlüsseln eines Textes: $y \equiv ax + b \pmod{26}$

Wird nun eine Tabelle mit den häufigsten Buchstaben in deutschen Texten stellt man fest, dass **E** und **N** die häfigsten Buchstaben sind. setzt man nun ihre dekodierten Zahlenwerte in die Formel ein erhält man.

$$\begin{aligned} C_E &\equiv a \cdot 4 + b \pmod{26} \\ C_N &\equiv a \cdot 13 + b \pmod{26} \end{aligned}$$