# Mit Python von Caesar zur Public-Key Kryptographie

Thomas Grischott
KSS

30. Juni 2008

## 1 Die Caesarverschiebung

Caesar hat Nachrichten an seine Feldherren verschlüsselt, indem er jeden Buchstaben um drei Positionen im Alphabet verschoben hat (nach z kommt dabei wieder a).

Genau das tut unser erstes Skript:

```
# caesar1.py

print "Verschluesselung nach Caesar"
print "----------------------------"

alphabet = "abcdefghijklmnopqrstuvwxyz"

klartext = raw_input("Klartext eingeben: ")
klartext = klartext.lower()
schluessel = 3

geheimtext = ""
for buchstabe in klartext:
    if buchstabe == " ":
        geheimtext += " "
    else:
        nummer = alphabet.index(buchstabe)
        nummer += schluessel
        if nummer > 25:
            nummer -= 26
        geheimtext += alphabet[nummer]

print "Geheimtext:", geheimtext.upper()
```

Mit Zahlen geht alles leichter! Wir schreiben also

$$\begin{array}{rcl} 0 & \text{für} & \text{a}, \\ 1 & \text{für} & \text{b}, \\ 2 & \text{für} & \text{c}, \\ \vdots & & \vdots \\ \text{und } 25 & \text{für} & \text{z}. \end{array}$$

Dann wird beim Verschlüsseln

$$\begin{array}{rcl} 0 & \text{zu} & 3, \\ 1 & \text{zu} & 4, \\ 2 & \text{zu} & 5, \\ \vdots & & \vdots \\ 22 & \text{zu} & 25, \\ 23 & \text{zu} & 0 \text{ (aha!)}, \\ \vdots & & \vdots \\ \text{und } 25 & \text{zu} & 2. \end{array}$$

Allgemein wird aus dem Klartextbuchstaben $x$ der Geheimtextbuchstabe $Y$, wobei

$$\begin{aligned} Y &= \text{„26er-Rest“ von } x+3 \\ &= (x+3) \bmod 26 \end{aligned}$$

Die Schreibweise $(x+3) \bmod 26$ (lies: „$x$ plus 3 modulo 26“) bedeutet: Nimm den Rest beim Teilen von $x+3$ durch 26. In Python würde man das mit dem `%`-Operator so schreiben:

```
Y = (x+3)%26
```

Zeit für eine verbesserte Version unseres Caesarskriptleins:

```
# caesar2.py

print "Verschluesselung nach Caesar"
print "----------------------------"

alphabet = "abcdefghijklmnopqrstuvwxyz"

klartext = raw_input("Klartext eingeben: ")
klartext = klartext.lower()
schluessel = 3

geheimtext = ""
for buchstabe in klartext:
```

```
    if buchstabe == " ":
        geheimtext += " "
    else:
        nummer = alphabet.index(buchstabe)
        nummer += schluessel
        nummer %= 26
        geheimtext += alphabet[nummer]

print "Geheimtext:", geheimtext.upper()
```

Wollen wir um eine beliebige Anzahl Positionen verschieben können, verwenden wir die folgende dritte Version.

```
# caesar3.py

print "Verschluesselung nach Caesar"
print "----------------------------"

alphabet = "abcdefghijklmnopqrstuvwxyz"

klartext = raw_input("Klartext eingeben: ")
klartext = klartext.lower()
schluessel = input("Schluessel eingeben: ")

geheimtext = ""
for buchstabe in klartext:
    if buchstabe == " ":
        geheimtext += " "
    else:
        nummer = alphabet.index(buchstabe)
        nummer += schluessel
        nummer %= 26
        geheimtext += alphabet[nummer]

print "Geheimtext:", geheimtext.upper()
```

Die Anzahl Positionen, um die verschoben wird, heisst *Schlüssel*. Benutzt man dafür den Buchstaben $e$, dann lautet unsere Verschlüsselungsvorschrift jetzt

$$\boxed{Y = (x + e) \bmod 26}$$

Wie funktioniert nun aber das Entschlüsseln? Klar, man nehme den gleichen Schlüssel (wie fürs *Ver*schlüssleln) und subtrahiere ihn vom Geheimtext (modulo 26):

$$x = (Y - e) \bmod 26$$

Nun folgt aber eine Idee, die fundamental ist für die ganze moderne Verschlüsselungswissenschaft: Statt fürs *Ent*schlüsseln den *gleichen Schlüssel* wie fürs *Ver*schlüsseln zu verwenden und dafür das *Verfahren* abzu*ändern* (aus $(x+e)$ mod 26 wurde ja $(Y-e)$ mod 26), verwenden wir fürs Ver- und fürs Entschlüsseln *verschiedene Schlüssel* und benutzen dafür zweimal *dasselbe Verfahren*!

Entschlüsselt wird also so:

$$\boxed{x = (Y + d) \bmod 26}$$

$e$ ist ab jetzt der Schlüssel fürs Verschlüsseln („*e*ncrypt“) , und $d$ ist der Schlüssel fürs Entschlüsseln („*d*ecrypt“).

Wie bestimmt man $d$ (aus $e$)? Ganz einfach, $d = 26 - e$! Wir können aber auch folgende hochkomplizierte Überlegung anstellen: Wird zuerst verschlüsselt und dann gleich wieder entschlüsselt, dann muss wieder der Klartext entstehen:

$$\begin{aligned} ((x + e) + d) \bmod 26 &= x \\ (x + (e + d)) \bmod 26 &= x \end{aligned}$$

Das heisst also, dass die Summe $e + d$ beider Schlüssel beim Teilen durch 26 den Rest 0 haben muss. Wir berechnen eine kleine Tabelle, aus der man ablesen kann, welches $d$ zu einem bestimmten $e$ addiert den 26er-Rest 0 liefert:

```
# additionstafel1.py

print "+ |",
for i in range(26):
    print i,
print
print 71*'-'
for i in range(26):
    print str(i)+" |",
    for j in range(26):
        print (i+j)%26,
    print
```

```
# additionstafel2.py

n = input("n = ")
print "%4s" % "+ |",
for i in range(n):
    print "%2i" % i,
print
print "%4s" % "--  ",
for i in range(n):
    print "%2s" % "--",
print
for i in range(n):
    print "%2s" % str(i)+" |",
    for j in range(n):
        print "%2i" % ((i+j)%n),
    print
```

(Die zweite Version berechnet die Tabelle in beliebiger Grösse $n$ statt nur für Zahlen von 0 bis 25. Ausserdem gibt sie die Tabelle etwas schöner formatiert aus.)

Man nennt $d$ auch das *additive Inverse* zu $e$ (modulo 26), weil $e$ und $d$ zusammen in der Welt der 26er-Reste ja 0 geben, sich in einer Additionsrechnung also gegenseitig aufheben.

Beim Caesarverfahren ist es offensichtlich kein Problem, $d$ aus $e$ zu bestimmen. In den nächsten Abschnitten machen wir uns auf die Suche nach Verschlüsselungsmethoden, bei denen das nicht so einfach geht. Wenn es nämlich unmöglich oder mindestens sehr schwierig ist, $d$ aus $e$ zu bestimmen, dann kann man $e$ veröffentlichen („in alle Welt hinausposaunen“). Wer immer mir eine geheime Nachricht senden möchte, kann diese dann mit meinem *öffentlichen Schlüssel* $e$ verschlüsseln. Weil niemand meinen *privaten Schlüssel* $d$ kennt (und ihn auch nicht aus dem bekannten öffentlichen Schlüssel $e$ berechnen kann), bleibe ich der Einzige, der die Nachricht entschlüsseln kann.

Bevor wir in diese spannende Suche einsteigen, schreiben wir aber noch schnell ein Skript, mit dem man – auf recht primitive Weise – viele caesarverschlüsselte Botschaften knacken kann. Die Idee dahinter ist: Der häufigste Buchstabe im Geheimtext entspricht wahrscheinlich dem im Deutschen häufigsten Klartextbuchstaben „e“.

```
# caesarknacken.py

print "Caesarverschluesselung knacken"
print "------------------------------"

alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

geheimtext = raw_input("Geheimtext eingeben: ")
geheimtext = geheimtext.upper()

haeufigkeiten = [geheimtext.count(buchstabe) for buchstabe in alphabet]
max_haeufigkeit = max(haeufigkeiten)
schluessel = alphabet.index('E')-haeufigkeiten.index(max_haeufigkeit)

klartext = ""
for buchstabe in geheimtext:
    if buchstabe == " ":
        klartext += " "
    else:
        nummer = alphabet.index(buchstabe)
        nummer += schluessel
        nummer %= 26
        klartext += alphabet[nummer]

print "Klartext:", klartext.lower()
```

Was heisst wohl der Geheimtext „HUGEHHUH“?

## 2 Multiplikationschiffren

Da es bei der Caesarverschiebung zu einfach war, den privaten Schlüssel $d$ aus dem öffentlichen Schlüssel $e$ zu bestimmen, versuchen wir es als nächstes mit der Multiplikation anstelle der Addition:

$$Y = x \cdot e \bmod 26$$

Hier ist das zugehörige Pythonskript:

```
# multiplikationschiffre1.py

print "Verschluesselung durch Multiplikation"
print "-------------------------------------"

alphabet = "abcdefghijklmnopqrstuvwxyz"

klartext = raw_input("Klartext eingeben: ")
klartext = klartext.lower()
schluessel = input("Schluessel eingeben: ")

geheimtext = ""
for buchstabe in klartext:
    if buchstabe == " ":
        geheimtext += " "
    else:
        nummer = alphabet.index(buchstabe)
        nummer *= schluessel
        nummer %= 26
        geheimtext += alphabet[nummer]

print "Geheimtext:", geheimtext.upper()
```

Verschlüssle probehalber einmal das Wort „Abrakadabra“ (und wähle den Schlüssel 2). Was stellst du fest? Genau, „a“ bleibt „A“. Das ist einfach zu verstehen: „a“ wird ja durch die Zahl 0 dargestellt, und 0 bleibt beim Multiplizieren eben 0.

Über dieses Verhalten sind wir natürlich gar nicht erfreut, und wir ändern daher die Darstellung der Buchstaben durch Zahlen. Neu lassen wir die Zahl 0 für das Leerzeichen stehen, welches wir ja (vorerst) sowieso unverändert in den Geheimtext übernehmen. Die Zahl 1 steht neu für den Buchstaben a, 2 für b, 3 für c, ... und 26 für z.

Da das Alphabet inklusive Leerzeichen jetzt 27 Zeichen enthält, müssen wir nun 27er-Reste (statt wie bisher 26er-Reste) berechnen. Würden wir zum Alphabet noch die 10 Ziffern hinzunehmen, müssten wir mit 37er-Resten arbeiten. Allgemein lautet unsere Verschlüsselungsvorschrift also:

$$\boxed{Y = x \cdot e \bmod n}$$

Dabei bezeichnet der sogenannte Modul $n$ die Anzahl Zeichen im Alphabet.

```
# multiplikationschiffre2.py

print "Verschluesselung durch Multiplikation"
print "-------------------------------------"

alphabet = " abcdefghijklmnopqrstuvwxyz"

klartext = raw_input("Klartext eingeben: ")
klartext = klartext.lower()
schluessel = input("Schluessel eingeben: ")
modul = len(alphabet)

geheimtext = ""
for buchstabe in klartext:
    nummer = alphabet.index(buchstabe)
    nummer *= schluessel
    nummer %= modul
    geheimtext += alphabet[nummer]

print "Geheimtext:", geheimtext.upper()
```

Wie wird nun *ent*schlüsselt? Natürlich so:

$$\boxed{x = Y \cdot d \bmod n}$$

Aber was ist hier $d$?

Analog zu Abschnitt 1 muss gelten:

$$\begin{aligned} (x \cdot e) \cdot d \bmod n &= x \\ x \cdot (e \cdot d) \bmod n &= x \\ e \cdot d \bmod n &= 1 \end{aligned}$$

$d$ muss also zu $e$ hinzumultipliziert den $n$er-Rest 1 liefern. $d$ ist also das *multiplikative Inverse* zu $e$. Um zu einem $e$ das inverse $d$ zu bestimmen, benutzen wir wieder Tabellen:

```
# multiplikationstafel.py

n = input("Modul n: ")
print "%4s" % "* |",
for i in range(n):
    print "%2i" % i,
print
print "%4s" % "--  ",
for i in range(n):
    print "%2s" % "--",
```

```
print
for i in range(n):
    print "%2s" % str(i)+" |",
    for j in range(n):
        print "%2i" % ((i*j)%n),
    print
```

Hier folgen einige Tabellen, die mit dem obigen Skript berechnet wurden. Berechne selbst weitere!

Modul n: 5

| * | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 2 | 3 | 4 |
| 2 | 0 | 2 | 4 | 1 | 3 |
| 3 | 0 | 3 | 1 | 4 | 2 |
| 4 | 0 | 4 | 3 | 2 | 1 |

Modul n: 6

| * | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 2 | 3 | 4 | 5 |
| 2 | 0 | 2 | 4 | 0 | 2 | 4 |
| 3 | 0 | 3 | 0 | 3 | 0 | 3 |
| 4 | 0 | 4 | 2 | 0 | 4 | 2 |
| 5 | 0 | 5 | 4 | 3 | 2 | 1 |

Modul n: 15

| * | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 2 | 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 1 | 3 | 5 | 7 | 9 | 11 | 13 |
| 3 | 0 | 3 | 6 | 9 | 12 | 0 | 3 | 6 | 9 | 12 | 0 | 3 | 6 | 9 | 12 |
| 4 | 0 | 4 | 8 | 12 | 1 | 5 | 9 | 13 | 2 | 6 | 10 | 14 | 3 | 7 | 11 |
| 5 | 0 | 5 | 10 | 0 | 5 | 10 | 0 | 5 | 10 | 0 | 5 | 10 | 0 | 5 | 10 |
| 6 | 0 | 6 | 12 | 3 | 9 | 0 | 6 | 12 | 3 | 9 | 0 | 6 | 12 | 3 | 9 |
| 7 | 0 | 7 | 14 | 6 | 13 | 5 | 12 | 4 | 11 | 3 | 10 | 2 | 9 | 1 | 8 |
| 8 | 0 | 8 | 1 | 9 | 2 | 10 | 3 | 11 | 4 | 12 | 5 | 13 | 6 | 14 | 7 |
| 9 | 0 | 9 | 3 | 12 | 6 | 0 | 9 | 3 | 12 | 6 | 0 | 9 | 3 | 12 | 6 |
| 10 | 0 | 10 | 5 | 0 | 10 | 5 | 0 | 10 | 5 | 0 | 10 | 5 | 0 | 10 | 5 |
| 11 | 0 | 11 | 7 | 3 | 14 | 10 | 6 | 2 | 13 | 9 | 5 | 1 | 12 | 8 | 4 |
| 12 | 0 | 12 | 9 | 6 | 3 | 0 | 12 | 9 | 6 | 3 | 0 | 12 | 9 | 6 | 3 |
| 13 | 0 | 13 | 11 | 9 | 7 | 5 | 3 | 1 | 14 | 12 | 10 | 8 | 6 | 4 | 2 |
| 14 | 0 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |

Wir stellen – vielleicht mit einiger Verblüffung – fest: Im Fall $n = 6$ gibt es zu $e = 2$, 3 und 4 kein passendes $d$, und ebenso ist es im Fall $n = 15$ für $e = 3$, 5, 6, 9, 10 und 12!! In diesen Fällen ist ein Geheimtext nicht entschlüsselbar!

Überlege dreierlei:

- Wie zeigt sich dieses Problem beim konkreten Ver- und Entschlüsseln? Finde dazu zuerst jene Schlüssel $e$, welche bei $n = 27$ kein zugehöriges $d$ besitzen. Verschlüssle dann mit dem Skript `multiplikationschiffre2.py` das Wort „Ananas“ mit einem derartigen $e$. Was stellst du fest? Findest du andere, besonders problematische Wörter?
- Stelle eine Vermutung auf: Welche besondere Eigenschaft muss der Schlüssel $e$ bei gegebenem Modul $n$ besitzen, damit ein $d$ zum Entschlüsseln existiert? Studiere so viele Multiplikationstabellen, bis du eine Idee hast, und überprüfe diese Idee an weiteren Tabellen.
- Beim welchen besonderen Moduln $n$ gibt es immer, d. h. zu jedem $e$, ein $d$?

Ob du alle drei Fragen 100%-ig beantworten konntest, ist gar nicht so wichtig. Wichtig ist aber folgende grundsätzliche Erkenntnis: Man kann nicht einfach ein beliebiges Paar $(n, e)$ verwenden! *$n$ und $e$ müssen bestimmte Anforderungen erfüllen bzw. nach gewissen Kriterien ausgewählt werden!*

An dieser Stelle empfehle ich eine ausgiebige Spielrunde:

- Wähle „funktionierende“ $n$, $e$ und $d$.
- Gib $(n, e)$ als öffentlichen Schlüssel bekannt (aber behalte $d$ für dich).
- Sende deinem Nachbarn eine mit seinem öffentlichen Schlüssel verschlüsselte Nachricht.
- Entschlüssle eine an dich gerichtete Nachricht.
- Knacke eine nicht für dich bestimmte (evtl. vom Lehrer vorgegebene) längere Nachricht.

Eine weitere Erkenntnis aus dieser Spielrunde sollte sein: Die Multiplikationschiffre ist fast ebenso einfach zu knacken wie das Caesarverfahren. Auch hier kann man nach dem häufigsten Buchstaben im Geheimtext suchen. Das Verfahrern ist also nicht grundsätzlich sicherer. Hingegen ist es bereits etwas schwieriger geworden, den privaten Schlüssel $d$ aus dem öffentlichen Schlüssel $e$ zu bestimmen.

Zum Schluss dieses Abschnitts halten wir noch fest: Was passiert, wenn man einen Klartext zuerst mit $d$ multipliziert und danach noch einmal mit $e$? Da die Multiplikation kommutativ ist (d. h. „auf die Reihenfolge kommt's nicht an“), erhält man dasselbe, wie wenn man zuerst mit $e$ verschlüsselt und dann mit $d$ entschlüsselt, nämlich wieder den ursprünglichen Klartext! In andern Worten: Was zuerst mit dem eigenen privaten Schlüssel verschlüsselt wurde, kann mit dem zugehörigen öffentlichen Schlüssel entschlüsselt werden. Wenn ich also eine verschlüsselte Nachricht mit dem öffentlichen Schlüssel von Mr X entschlüsseln kann *und sicher bin, dass dieser öffentliche Schlüssel wirklich Mr X gehört*, dann weiss ich, dass Mr X die Nachricht verfasst und mit seinem privaten Schlüssel verschlüsselt hat. Eine Verschlüsselung mit dem eigenen privaten Schlüssel wirkt also wie eine *Unterschrift*.

# 3 Blockchiffren

Bisher haben wir jeweils einzelne Buchstaben verschlüsselt. Das hat den Vorteil, dass es einfach geht, aber den Nachteil, dass die einzelnen Buchstaben auch im Geheimtext mit charakteristischen Häufigkeiten vorkommen. Wir konnten deshalb die Geheimtexte recht leicht knacken.

In diesem Abschnitt verschlüsseln wir darum Blöcke von mehreren Buchstaben. Zwar kommen kurze Blöcke (insb. die Buchstabenpaare) in vernünftigen Texten noch mit deutlich unterschiedlichen Häufigkeiten vor, bei längeren Blöcken sind die Häufigkeitsunterschiede aber so winzig klein, dass man sie nicht mehr ausnutzen kann.

Wir betrachten nur Blöcke der Länge 3. Der Block ␣␣␣ (drei Leerzeichen) erhält die Nummer 0, ␣␣a die Nummer 1, ␣␣b die Nummer 2, ..., ␣␣z die Nummer 26, ␣a␣ die Nummer 27 usw. bis zum Block zzz mit der Nummer 19682. Das „Blockalphabet“ umfasst insgesamt $n = 27^3 = 19683$ „Zeichen“.

(In der Praxis wählt man noch längere Blöcke. Viele gebräuchliche Verfahren arbeiten mit Bitmustern der Länge 64, also mit Sequenzen, die aus 64 Nullen oder Einsen bestehen. Das entspricht einer Blocklänge von 8 Zeichen des ASCII-Alphabets.)

Blockchiffren kann man unabhängig davon einsetzen, ob man mit dem Caesarverfahren oder mit einer Multiplikationschiffre oder noch einer anderen Methode verschlüsselt. Das folgende Skript verwendet die Caesarverschiebung:

```
# blockcaesar.py

print "Caesarverschluesselung auf Bloecken der Laenge 3"
print "------------------------------------------------"

alphabet = " abcdefghijklmnopqrstuvwxyz"
blockalphabet = [i+j+k for i in alphabet for j in alphabet for k in alphabet]
modul = len(blockalphabet)

klartext = raw_input("Klartext eingeben: ")
klartext = klartext.lower()
while len(klartext)%3:
    klartext += " "
schluessel = input("Schluessel eingeben: ")

geheimtext = ""
for b in range(len(klartext)/3):
    block = klartext[3*b:3*b+3]
    nummer = blockalphabet.index(block)
    nummer += schluessel
    nummer %= modul
    geheimtext += blockalphabet[nummer]

print "Geheimtext:", geheimtext.upper()
```

Ändere das Skript, so dass es eine Multiplikationschiffre einsetzt, und experimentiere damit.

Falls du die zweite der drei Fragen aus dem letzten Abschnitt beantworten konntest, dann weisst du auch, welche Schlüssel $e$ dann nicht verwendet werden dürfen. Warum ist $e = 1000$ ein guter Schlüssel? Der inverse Schlüssel dazu ist $d = 12853$. Ihn herauszufinden, dürfte mit einer Multiplikationstabelle bereits mühsam sein. Aber dass $1000 \cdot 12853$ beim Teilen durch 19683 wirklich Rest 1 hat, lässt sich mit Python einfach überprüfen.

# 4 Das RSA-Verfahren

Nach dem Addieren und dem Multiplizieren wird jetzt potenziert! Wir verschlüsseln mit

$$\boxed{Y = x^e \bmod n}$$

und entschlüsseln mit

$$\boxed{x = Y^d \bmod n}$$

Weil $(x^e)^d = x^{e \cdot d}$ ist, könnte man vermuten, dass $d$ wie im Abschnitt 2 das multiplikative Inverse zu $e$ sein muss. Leider darf man aber im Exponenten nicht einfach modulo $n$ rechnen! Ein Beispiel zeigt's: $2 \cdot 3 = 6$, $2 \cdot 3 \bmod 5 = 1$, aber $2^6 \bmod 5 = 64 \bmod 5 = 4$ ist nicht das gleiche wie $2^1 \bmod 5 = 2$. Die Anforderungen an $n$ und $e$ sowie die Bestimmung von $d$ sind also etwas komplizierter als bei der Multiplikationschiffre.

Ohne weitere Begründung folgt jetzt das Rezept, wie man $n$, $e$ und das zugehörige $d$ wählen muss, damit sowohl das Verschlüsseln als auch das Entschlüsseln klappt:

1. Wähle zwei *Primzahlen* $p$ und $q$.
2. Berechne $n = p \cdot q$.
3. Wähle $e$ und $d$ so, dass $e \cdot d$ um 1 grösser ist als ein beliebiges Vielfaches von $(p-1) \cdot (q-1)$.

Ein erstes Beispiel: $p = 7$, $q = 13$. Dann ist $n = 7 \cdot 13 = 91$. Wir wählen $e = 5$ und $d = 29$. Dann ist $e \cdot d = 145$ um 1 grösser ist als das Doppelte von $(p-1) \cdot (q-1) = 72$.

Ein zweites Beispiel: $p = 101$, $q = 103$. Dann ist $n = 101 \cdot 103 = 10403$. Weiter wählen wir $e = 77$ und $d = 8213$. Verifiziere, dass $e \cdot d = 632401$ um 1 grösser ist als das 62-fache von $(p-1) \cdot (q-1) = 10200$.

Findest du selbst weitere Beispiele für $n$, $e$ und $d$? Falls du dabei grössere Zahlen in ihre Faktoren zerlegen musst, hilft dir möglicherweise folgende Funktion weiter:

```
# faktorisieren.py

def faktorisiere(x):
    faktoren = []
    faktor = 2
    while x%faktor:
        faktor += 1
    faktoren.append(faktor)
    if faktor < x:
        weiterefaktoren = faktorisiere(x/faktor)
        faktoren.extend(weiterefaktoren)
    return faktoren

zahl = input("Zu faktorisierende Zahl: ")
print "Faktoren:", faktorisiere(zahl)
```

Bevor wir endlich verschlüsseln, müssen wir noch ein kleines Problem klären. Bisher war $n$ ja gleich der Anzahl Zeichen oder Blöcke im Alphabet. Weil $n$ beim RSA-Verfahren aber das Produkt von $p$ und $q$ ist, welche wir selbst wählen, kann $n$ nicht mehr für alle Teilnehmer gleich sein. Wir wollen aber natürlich nicht für jeden Teilnehmer ein neues Alphabet mit der passenden Anzahl Zeichen erfinden.

Wie lösen wir das Problem? Ganz einfach, wir geben einfach einen Bereich vor, in welchem alle $n$ liegen müssen. Wenn wir beispielsweise Blöcke der Länge 2 verschlüsseln wollen und wie in Abschnitt 3 zu den 26 Buchstaben noch das Leerzeichen hinzunehmen, dann gibt es $27^2 = 729$ verschiedene Blöcke. Wir dürfen daher kein $n$ wählen, das kleiner als 729 ist, denn sonst würden wir als Ergebnis der modulo-$n$-Rechnung nicht alle Blocknummern erhalten. Wenn wir andererseits als obere Grenze $27^3 = 19683$ wählen, dann sind wir sicher, das die Ergebnisse der modulo-$n$-Rechnungen nicht über 19682 liegen, und zu Zahlen in diesem Bereich gibt es immer einem Block der Länge 3, wenn wir die Codierung aus Abschnitt 3 verwenden. Wir müssen also in Kauf nehmen, dass aus einem Buchstabenpaar im Klartext ein Buchstabentripel im Geheimtext wird.

```
# rsaverschluesseln.py

print "RSA-Verschluesselung"
print "--------------------"

alphabet = " abcdefghijklmnopqrstuvwxyz"
klartextbloecke = [i+j for i in alphabet for j in alphabet]
geheimtextbloecke = [i+j+k for i in alphabet for j in alphabet\
                     for k in alphabet]

klartext = raw_input("Klartext eingeben: ")
klartext = klartext.lower()
while len(klartext)%2:
    klartext += " "
modul = input("Modul n eingeben (729 <= n <= 19683): ")
schluessel = input("Schluessel e eingeben: ")

geheimtext = ""
for b in range(len(klartext)/2):
    block = klartext[2*b:2*b+2]
    nummer = klartextbloecke.index(block)
    nummer **= schluessel
    nummer %= modul
    geheimtext += geheimtextbloecke[nummer]

print "Geheimtext:", geheimtext.upper()
```

Wie du siehst, müssen zum Verschlüsseln $n$ und $e$ bekannt sein. Beim RSA-Verfahren ist der öffentliche Schlüssel also das Paar $(n, e)$.

Zum Entschlüsseln braucht man dann den privaten Schlüssel ($n$, $d$):

```
# rsaentschluesseln.py

print "RSA-Entschluesselung"
print "--------------------"

alphabet = " ABCDEFGHIJKLMNOPQRSTUVWXYZ"
klartextbloecke = [i+j for i in alphabet for j in alphabet]
geheimtextbloecke = [i+j+k for i in alphabet for j in alphabet\
                     for k in alphabet]

geheimtext = raw_input("Geheimtext eingeben: ")
geheimtext = geheimtext.upper()
while len(geheimtext)%3:
    geheimtext += " "
modul = input("Modul n eingeben (729 <= n <= 19683): ")
schluessel = input("Schluessel d eingeben: ")

klartext = ""
for b in range(len(geheimtext)/3):
    block = geheimtext[3*b:3*b+3]
    nummer = geheimtextbloecke.index(block)
    nummer **= schluessel
    nummer %= modul
    klartext += klartextbloecke[nummer]

print "Klartext:", klartext.lower()
```

Nun sind wir fast am Ende der Geschichte angekommen. Ein Wort noch zur Sicherheit des RSA-Verfahrens: Die Verschlüsselung von „Zweierpäckchen“, wie sie die obigen Skripte durchführen, lässt sich noch gut durch eine Suche nach dem häufigsten „Dreierpäckchen“ im Geheimtext knacken. Verschlüsselt man dagegen längere Blöcke, ist dieser Weg nicht mehr gangbar. Die einzige Möglichkeit, die RSA-Verschlüsselung zu knacken, besteht dann darin, $n$ in seine Primfaktoren $p$ und $q$ zu zerlegen. Sind $p$, $q$ und $e$ erst bekannt, kann man $d$ leicht finden (z. B. durch geschicktes Pröbeln). Allerdings ist es den Mathematikern bisher nicht gelungen, eine genügend grosse Zahl $n$ innerhalb vernünftiger Zeit in ihre Primfaktoren zu zerlegen!

## Literatur


[1] Hermann Puhlmann. Kryptographie verstehen – Ein schülergerechter Zugang zum RSA-Verfahren. Fachbereich Mathematik der TU Darmstadt. August 1998.
`ftp://ftp.mathematik.tu-darmstadt.de/pub/department/preprints/2000.ps.gz`
`http://www.zum.de/Faecher/Materialien/rubin/Programme/kryptographie.ps`