UE 8

2025-01-06

Aufgabe 1

$$y^2 = x^3 - x + 4$$

Aufgabe 1a

Stellen Sie die Menge der Punkte über $\mathbb{R}^2$ graphisch dar:

Wir könnten 2 Funktionen ($y = \pm\sqrt{x^3 - x + 4}$) zeichnen, eine Funktion $f(x, y) = y^2 - x^3 + x - 4$ null-setzen und die Höhenlinie für $f(x, y) = 0$ zeichnen, etc.

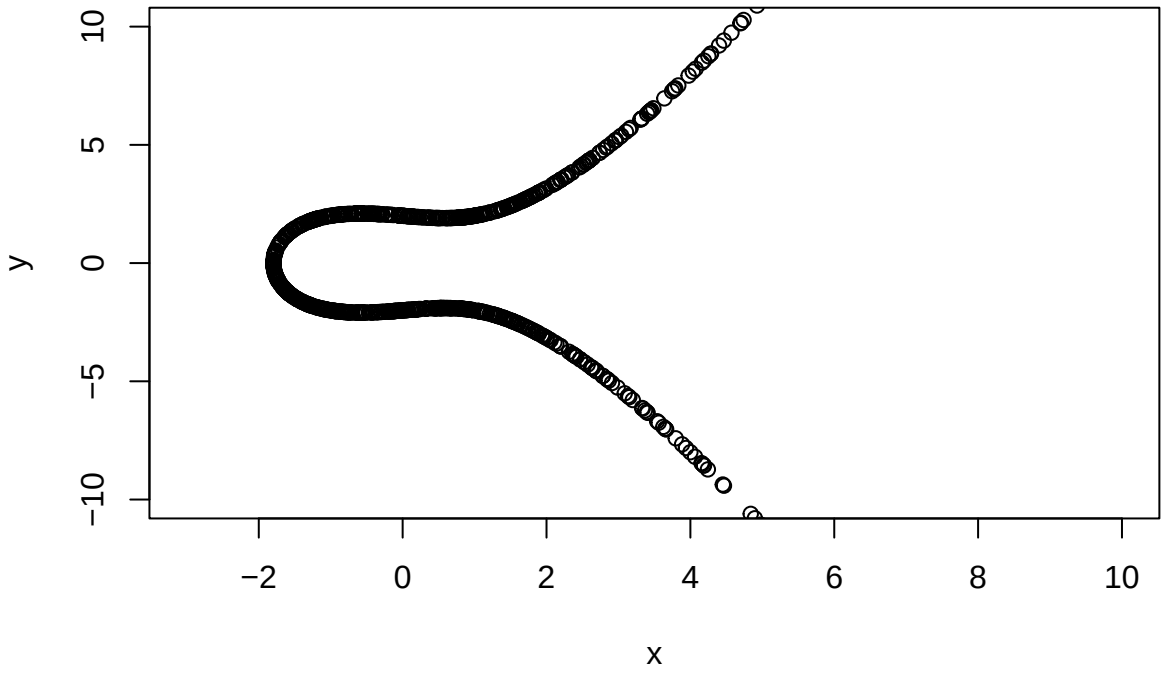
Wir entscheiden uns aber, eine Funktion zu definieren, die die Punktaddition implementiert und berechnen von 2 Startpunkten aus die ersten 1000 Punkte die daraus entstehen:

```
punktAddition <- function(p, q) {
  x1 <- p[1]
  y1 <- p[2]

  x2 <- q[1]
  y2 <- q[2]

  k <- (y2 - y1) / (x2 - x1)

  x3 <- k^2 - x1 - x2
  y3 <- k * (x1 - x3) - y1

  return(c(x3, y3))
}
```

Jetzt berechnen wir manuell 2 Punkte auf unserer Kurve (indem wir $x = 0, 1$ in die Gleichung einsetzen und ein beliebiges Vorzeichen für y wählen). Mittels der Punktaddition berechnen wir sukzessive weitere Punkte auf der Kurve

```
npts <- 1000
x1 <- data.frame(x=rep(0, npts), y=rep(0, npts))

p1 <- c(0, 2)
p2 <- c(1, 2)

for (i in 1:npts) {
  tmp <- punktAddition(p1, p2)
  x1$x[i] <- tmp[1]
  x1$y[i] <- tmp[2]
  p1 <- p2
  p2 <- tmp
}

x1 <- x1 %>% arrange(x, y)
```

```r
plot(x1, xlim=c(-3, 10), ylim=c(-10, 10))
```



Aufgabe 1b

Um alle Punkte in $\mathrm{GF}(43) = (\mathbb{Z}/43\mathbb{Z})^2$ zu finden, berechnen wir alle Punkte für $x = 0, \ldots, 42$.

Multiplikationstabelle Weil wir eine Wurzelfunktion brauchen, erstellen wir uns zuerst eine Multiplikations-Tabellen mit allen Elementen in GF(43). Beachte, in R beginnt die Indexierung bei 1.

```
multiplikationTable <- matrix(0, nrow = 42, ncol = 42)

for (i in 1:42) {
  for (j in 1:42) {
    multiplikationTable[i, j] = (i * j) %% 43
  }
}

# ersten 12 Zeilen und Spalten:
multiplikationTable[1:12, 1:12]
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
##  [1,]    1    2    3    4    5    6    7    8    9    10    11    12
##  [2,]    2    4    6    8   10   12   14   16   18    20    22    24
##  [3,]    3    6    9   12   15   18   21   24   27    30    33    36
##  [4,]    4    8   12   16   20   24   28   32   36    40     1     5
##  [5,]    5   10   15   20   25   30   35   40    2     7    12    17
##  [6,]    6   12   18   24   30   36   42    5   11    17    23    29
##  [7,]    7   14   21   28   35   42    6   13   20    27    34    41
##  [8,]    8   16   24   32   40    5   13   21   29    37     2    10
##  [9,]    9   18   27   36    2   11   20   29   38     4    13    22
## [10,]   10   20   30   40    7   17   27   37    4    14    24    34
## [11,]   11   22   33    1   12   23   34    2   13    24    35     3
## [12,]   12   24   36    5   17   29   41   10   22    34     3    15
```

Wurzeltabelle Jetzt suchen wir entlang der Diagonale alle Quadrate: der Zeilen/Spaltenindex ist die Wurzel vom Zelleninhalt.
(Bsp: Eintrag an Stelle $(7, 7)$ ist $6 \Rightarrow 7^2 = 6 \mod 43$)

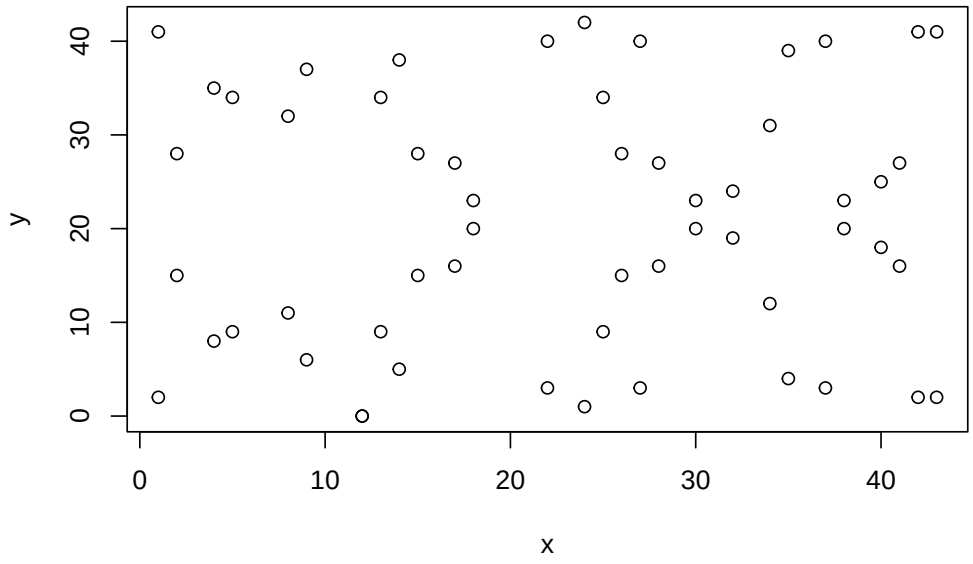
```
square43Table <- data.frame(n = 1:42, root1 = rep(NA, 42), root2 = rep(NA, 42))

# populate square table
for (i in 1:42) {
  # i*i = idx
  idx <- multiplikationTable[i, i] %% 43
  tableValue1 <- square43Table$root1[idx]
  tableValue2 <- square43Table$root2[idx]
  if (is.na(tableValue1)) {
    square43Table$root1[idx] <- i
  } else if (is.na(tableValue2)) {
    square43Table$root2[idx] <- i
  }
}

head(square43Table)
```

```
##   n root1 root2
## 1 1     1    42
## 2 2    NA    NA
```

```
## 3 3    NA    NA
## 4 4     2    41
## 5 5    NA    NA
## 6 6     7    36
```

Graph Jetzt definieren wir die Funktion, die die Punkte auf der Kurve berechnet und plotten diese in Anschluss:

```r
# funktion, die zu einem x wert die 2 y werte berechnet für die 2 Punkte am Graphen
ec <- function(x, a = -1, b = 4) {
  rhs = (x^3 + a*x + b) %% 43
  if (rhs == 0) {
    return(c(0, 0))
  }

  val1 <- square43Table$root1[rhs]
  val2 <- square43Table$root2[rhs]
  return(c(val1, val2))
}

# data frame, wo die Punkte eingetragen werden
finiteEC <- data.frame(x = rep(0, 2*43), y = rep(0, 2*43))
for (i in 0:43) {
  vals <- ec(i)

  finiteEC$x[i] <- i
  finiteEC$y[i] <- vals[1]

  finiteEC$x[i+43] <- i
  finiteEC$y[i+43] <- vals[2]
}

plot(finiteEC)
```



Aufgabe 2

Markiere Punkte $P_1(5/9)$ und $P_2(2/15$ und deren Summe.

In der Punktaddition würden wir dividieren, daher brauchen wir eine Divisions-Tabelle:

Divisionstabelle Jetzt können wir daraus eine Divisionstabelle erstellen:

In jeder Zeile suchen wir den ersten Index, wo 1 in der Zelle steht. Dieser Index ist das Inverse zum Zeilenindex. (Bsp: in der 4. Zeile steht in der 11. Spalte $1 \Rightarrow 4 \cdot 11 = 1 \mod 43$).

```
division43Table <- data.frame(n = 1:42, inverse = rep(0, 42))

# populate division table
for (i in 1:42) {
  row <- multiplikationTable[i,] %% 43
  for (j in 1:42) {
    if (row[j] == 1) {
      division43Table$inverse[i] = j
      break
    }
  }
}

head(division43Table)
```

```
##   n inverse
## 1 1       1
## 2 2      22
## 3 3      29
## 4 4      11
## 5 5      26
## 6 6      36
```

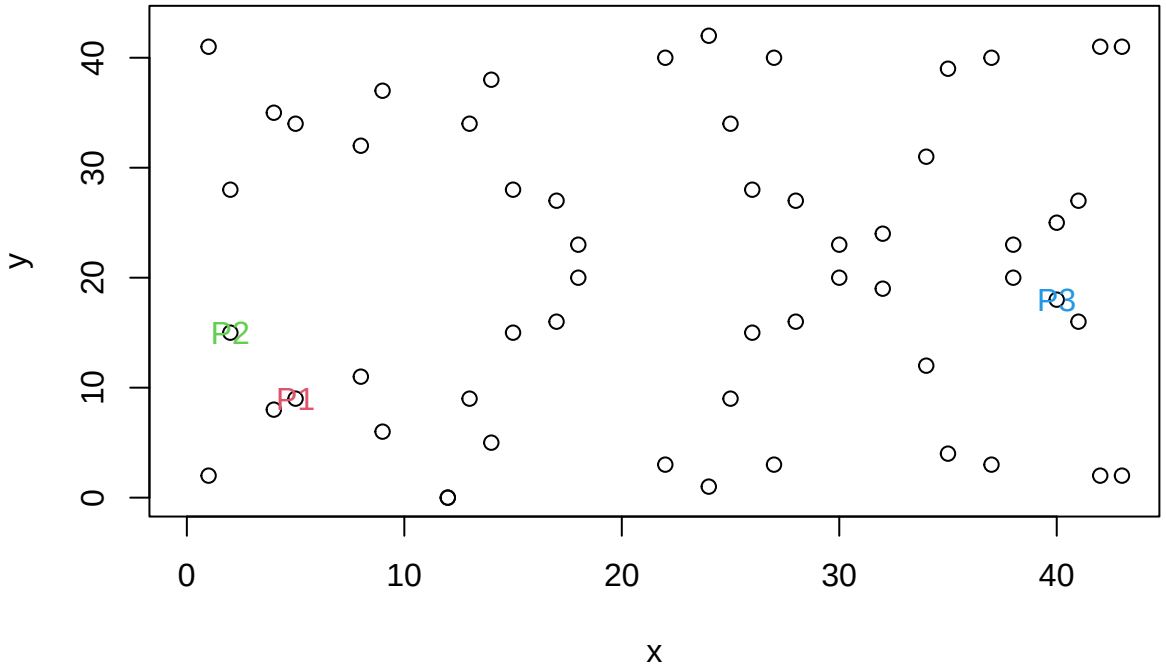
Punktaddition Jetzt definieren wir die Punktaddition auf dem Körper GF(43):

```
punktAddition43 <- function(p, q) {
  x1 <- p[1]
  y1 <- p[2]

  x2 <- q[1]
  y2 <- q[2]

  k <- (y2 - y1) * division43Table$inverse[(x2 - x1) %% 43]

  x3 <- k^2 - x1 - x2
  y3 <- k * (x1 - x3) - y1

  return(c(x3 %% 43, y3 %% 43))
}
```

So können wir alle Punkte plotten und zusätzlich die Punkte P_1, P_2 und $P_3 := P_1 + P_2$ markieren:

```
plot(finiteEC, xlim=c(0, 43), ylim=c(0, 43))

# P1: roter text
text(5,  9, "P1", col=2)
```

```
# P2: grüner text
text(2, 15, "P2", col=3)

# P3 = P1 + P2
v <- punktAddition43(c(5, 9), c(2, 15))
# P3: blauer text
text(v[1], v[2], "P3", col=4)
```



Aufgabe 3

Unser Graph hat so viele Punkte:

```
finiteEC %>% filter(!is.na(y)) %>% nrow()
```

```
## [1] 56
```

Obwohl wir $86 = 2 \cdot 43$ Punkte erwarten könnten, gibt es für manche x Werte keine Punkte auf der Kurve. Das passier, wenn in GF(43) keine Wurzeln von $x^3 - 1 + 4$ gezogen werden können.

Aufgabe 4

Wie viele grundlegenden Körperoperationen (Multiplikation/Addition/Invertierung) benötigen Sie für die Berechnung von

a) $x = a^{675} \mod 1021$

Mittels Square-and-Multiply brauchen wir 13 Operationen.

Es gilt $675 = 2^9 + 2^7 + 2^5 + 2^1 + 2^0$. Wir berechnen

Rechnung	Anzahl Operationen
a^{2^0}	0
a^{2^1}	1
a^{2^2}	1
$\dots$	$\dots$
a^{2^9}	1
$a^{2^9+\dots+2^0} = a^{2^9} \cdot \dots \cdot a^{2^0}$	4

Für die Quadrierungen brauchen wir 9 Multiplikationen, für das Multiply dann noch 4 weitere Multiplikationen $\Rightarrow$ 13 Körperoperationen.

Wenn die Modulo Operationen mitgezählt werden, haben wir $2 \cdot 9 + 4 + 1 = 23$ Operationen.

b) $Q = 675 * P$ über GF(1021)

Wir verwenden Punktadditionen und Punktverdoppelungen um Q zu berechnen:

$$675P = 2^9P + 2^7P + 2^5P + 2P + P.$$

Eine Punktaddition benötigt 1 Inversion, 2 Multiplikationen und 1 Quadrierung $\Rightarrow$ 4 Operationen.
Eine Punktverdoppelung benötigt 1 Inversion, 2 Multiplikationen und 2 Quadrierungen $\Rightarrow$ 5 Operationen.

Daher haben wir wegen 9 Punktverdoppelungen und 4 Punktadditionen insgesamt $9 \cdot 5 + 4 \cdot 4 = 61$ Operationen.

Aufgabe 5

Bestimmen Sie für die folgenden H_i, ob es sich jeweils um Hashfunktionen im Sinne der Definitionen der Vorlesung handelt. Geben Sie jeweils alle Eigenschaften an, die die gegebene Funktion besitzt, sowie, gegebenenfalls, die sie nicht besitzt, und begründen Sie Ihre Entscheidung auch kurz.

(Anmerkung: m bezeichnet im Folgenden eine beliebige binäre Datei der Länge n, die stets als Zahl $< 2n$ interpretiert, aber auch in kleinere Einheiten aufgebrochen werden kann)

Erinnerung: **Hashfunktion**

1. Kompression: bildet beliebig lange Texte auf n bits ab
2. Preimage-Resistenz: für $y = h(x)$ ist es praktisch unmöglich x zu finden
3. schwache Kollisionsresistenz: für alle x ist es praktisch unmöglich ein $x'! = x$ zu finden mit $h(x) = h(x')$
4. starke Kollisionsresistenz: es ist praktisch unmöglich ein Paar $x \neq x'$ zu finden mit $h(x) = h(x')$

a) $H_1(m) = m \mod 2^{20}$

1. Kompression: ja
2. Preimage-Resistance: nein, es kann zwar nicht eindeutig das Inverse gefunden werden, aber viele möglivhe Kandidaten ($x = k \cdot 2^{20} + H_1(m)$) ausprobiert.
3. Collision-Resistance: nein, es ist trivial ein x' bzw. ein Paar (x, x') zu finden.

b) $H_2(m) = \text{SHA-256}(m) \mod 10^{10} - 1$

SHA-256 ... *Secure Hash Funktion*: https://en.wikipedia.org/wiki/SHA-2

1. Kompression: ja
2. Preimage-Resistance: ja (aber es gibt Attacken)
3. Collision-Resistance: ja (aber es gibt Attacken)

H_2 erbt Hash-Eigenschaften von SHA-256.

c) $H_3(m) = \text{MD5(SHA-256(m))}$

Hier werden folgenden Funktionen verwendet:

- MD5: *Message-Digest Algorithm 5* (Hashfunktion)
- SHA-256: Hashfunktion, siehe b)

1. Kompression: Ja
2. Preimage Resistance: Ja (aber es gibt Attacken)
   - MD5: theoretisch, weil Komplexität 2^{123.4}
   - SHA-256: ja, siehe b)
3. Collision-Resistance: Ja (aber es gibt Attacken)
   - MD5: nein (https://en.wikipedia.org/wiki/MD5)
   - SHA-256: ja, siehe b)

H_3 erbt Hash-Eigenschaften von SHA-256.

d) $H_4(m) = \text{AES-128(m,k)}$

AES ist ein Block-Cipher, keine Hash-Funktion

1. Kompression: nein
2. Preimage-Resistance: ja, AES wird verwendet, um geheime Informationen zu verschlüsseln, ein Preimage zu finden ist sehr schwer
3. Collision-Resistance: unbekannt

H_4 ist keine Hash-Funktion

e) $H_5(m) = 2^m \bmod 2^{1.279} - 1$

H_5 kann eine Hash Funktion sein:

1. Kompression - ja, dank Modulo
2. Preimage-Resistance - ja, siehe diskreter Logarithmus
3. Kollisions-Resistenz - ja, siehe diskreter Logarithmus