Duale Hochschule Baden-Württemberg Mannheim

Studienarbeit

Arithmetik endlicher Körper (speziell in der Charakteristik 2)

Studiengang Informatik

Studienrichtung Angewandte Informatik

Verfasser(in):	Lars Krickl
Matrikelnummer:	2512317
Kurs:	TINF22 AI2
Studiengangsleiter:	Prof. Dr. Holger D. Hofmann
Wissenschaftliche(r) Betreuer(in):	Prof. Dr. Reinhold Hübl
Bearbeitungszeitraum:	15.10.2024 - 15.04.2025

Inhaltsverzeichnis

1 Theorie **1**

2 Implementierung **2**

2.1 Polynomdarstellung 2

2.2 Hilfsfunktionen 3

2.2.1 Wegschneiden von Nullen 4

2.2.2 Erste Einser-Stelle finden 4

2.2.3 Grad eines Polynoms bestimmen 5

2.2.4 Polynom auf Nullstellen prüfen 6

2.2.5 Tupel nach Stellen aufteilen 7

2.2.6 Exponent zu einer Stelle finden 8

2.2.7 Ableitung bestimmen 8

2.3 Arithmetische Operationen 9

2.3.1 Addition von Polynomen 10

2.3.2 Multiplikation von Polynomen 11

2.3.3 Reduktion eines Produkts 12

2.3.4 Alternativer Algorithmus für die Multiplikation 14

2.3.5 Polynomdivision 18

2.4 Polynom-Algorithmen 20

2.4.1 Euklidischer Algorithmus 20

2.4.2 Minimalpolynome 22

2.4.3 Cantor-Zassenhaus-Algorithmus 26

2.5 Performance-Vergleiche 26

2.5.1 Vergleich der beiden Multiplikations-Algorithmen 26

2.5.2 Vergleich Minimalpolynom- mit Cantor-Zassenhaus-Algorithmus 26

3 Beispielaufgaben **27**

1 Theorie

x

2 Implementierung

In diesem Kapitel wird die Implementierung der Arithmetik endlicher Körper der Charakteristik zwei in Python erläutert. Dazu zählt die Polynomdarstellung, die Implementierung der verschiedenen Rechenoperationen, sowie weitere Algorithmen für den Umgang mit Polynomen.

2.1 Polynomdarstellung

Um in der Programmiersprache Python Polynome darzustellen, wird die Datenstruktur Tupel verwendet. In einem Tupel können mehrere Inhalte unveränderlich gespeichert werden. Da in der Charakteristik zwei lediglich Zahlen aus dem Binärsystem Vorfaktoren der verschiedenen Monome sein können, werden die Tupel nur mit Nullen und Einsen befüllt. Dadurch wird bestimmt, ob ein bestimmtes Monom in dem Polynom vorkommt (in diesem Fall mit einer 1 an der entsprechenden Stelle dargestellt) oder ob das Monom nicht in dem Polynom vertreten ist (durch eine 0 dargestellt).

In diesen Tupeln steht das Monom mit dem höchsten Exponenten an der ersten Stelle (entspricht der Stelle ganz links) und die Exponenten nehmen dann schrittweise nach rechts ab, sodass die letzte Stelle (ganz rechts) dem Monom mit dem Exponenten 0 entspricht. Diese Darstellung durch Tupel vereinfacht den Umgang mit Polynomen, da die ausführliche mathematische Schreibweise zusätzliche, unnötige Informationen wie die Variable X oder das $+$-Zeichen enthält. Für den Umgang mit Polynomen ist jedoch nur wichtig, welcher Vorfaktor an welcher Stelle des Polynoms steht, und diese beiden Informationen sind in der Schreibweise als binäre Tupel enthalten, wodurch sie sich anbietet für das Arbeiten mit Polynomen.

```python
def tuple_to_polynom(F):

    if all(elem == 0 for elem in F):
        return 0

    cases = {0: "1", 1: "X"}

    L = [
        cases.get(i, f"X^{i}")
        for i, coeff in enumerate(reversed(F))
        if coeff == 1
    ]

    return "(" + " + ".join(reversed(L)) + ")"
```

Um solch ein binäres Tupel in Python wieder in die mathematische Schreibweise eines Polynoms mit Vorfaktor, Variable X und Exponent für jedes Monom darzustellen, wurde die Funktion *"tuple_to_polynom"* erstellt. Wie der Name schon sagt, bekommt die Funktion ein Tupel als Funktionsparameter und gibt einen String zurück, der das vollständige Polynom umfasst.

Sollte das Tupel lediglich aus Nullen bestehen, gibt die Funktion 0 zurück, da das leere Tupel der Null in der mathematischen Schreibweise entspricht.

Ansonsten iteriert die Funktion über die einzelnen Elemente des Tupels und bestimmt daraus einen Teilstring. Bei der Erstellung dieser Teilstrings gibt es zwei verschiedene Fälle. Bei dem letzten Monom eines Tupels, also dem Monom ganz rechts, wird der Teilstring lediglich eine 1, da das letzte Monom den Exponenten null hat, und alles hoch null gleich eins ergibt. Für alle anderen Fälle, bei denen der Exponent größer als null ist, wird der Teilstring zu X^n, wobei n dem Exponenten des Monoms entspricht. Im Zuge dieser Funktion wird das Tupel vor der Erstellung des Lösungsstrings einmal umgekehrt, sodass nun das letzte Monom an erster Stelle steht, das vorletzte an zweiter Stelle steht, und so weiter. Dadurch gewinnt man den Vorteil, dass jetzt die Position des Monoms in dem Tupel gleich dem Exponenten des Monoms entspricht. Da bei Tupeln, ähnlich wie bei Arrays, das Zählen bei null anfängt, steht nach der Umkehrung das erste Monom (was ursprünglich das Letzte war) an der Position null in dem Tupel, was genau dem Exponenten dieses Monoms entspricht. All diese einzelnen Teilstrings werden dann aneinandergereiht, wobei zwischendrin immer ein $"+"$ beigefügt wird. So erhält man schließlich einen String, der das vollständige Polynom, wie man es auch in der mathematischen Schreibweise aufschreiben würde.

Anschließend werden um den Ergebnisstring noch Klammern gesetzt. Dies dient der Lesbarkeit der Tupel.

Diese Funktion, die aus einem Tupel wieder ein Polynom macht, erfüllt keinen direkten Zweck, der für weitere Operationen wichtig ist, sondern diese Funktion dient lediglich dazu, aus einem Tupel eine für den Anwender lesbare Darstellung zu bieten, da die Tupel, vor allem bei steigender Länge, unübersichtlich werden können.

2.2 Hilfsfunktionen

Während der Umsetzung werden für einige der implementierten Operationen Hilfsfunktionen benötigt. Diese sind in der Datei *"AuxiliaryFunctions.ipynb"* gesammelt,

da sie später von verschiedenen Stellen aus benötigt werden. Im Folgenden werden diese Funktionen genauer erläutert.

2.2.1 Wegschneiden von Nullen

Um führende Nullen von Polynomen, bzw. von binären Tupeln wegzuschneiden, wurde die Funktion *"cut_zeros_left"* implementiert, die dafür sorgt, dass das gegebene Tupel mit der ersten Eins, die in dem Tupel vorhanden ist, beginnt und alle vorherigen Stellen abgeschnitten werden.

```
1  def cut_zeros_left(F):
2
3      while F and F[0] == 0:
4          F = F[1:]
5
6      if F == ():
7          return (0,)
8
9      return F
```

Um diese führenden Nullen abzuschneiden, wird über das binäre Tupel iteriert und überprüft, ob die erste Stelle eine Null ist. Sollte diese Bedingung wahr sein, wird die erste Stelle abgeschnitten. Dieser Vorgang wird so lange wiederholt, bis die erste Stelle eine Eins ist.

Für den Sonderfall, dass ein leeres Tupel in die Funktion gegeben wird, muss eine zusätzliche Bedingung eingeführt werden, da in einem leeren Tupel keine Eins gefunden werden kann, und somit eine Endlosschleife entstehen würde. Dieser Fall kann einfach abgefangen werden, indem in der Schleifenbedingung geprüft wird, ob das Tupel leer ist, und die Schleife nur durchlaufen wird, sollte das Tupel nicht leer sein.

2.2.2 Erste Einser-Stelle finden

Die Funktion *"find_pos_of_first_one"* dient dazu, die erste Stelle eines Polynoms, von links aus, zu finden, die mit einer Eins befüllt ist. Diese Funktion wird im späteren Verlauf benötigt, um den Grad einer Funktion ermitteln zu können.

```python
def find_pos_of_first_one(F):

    for i in range(len(F)):
        if F[i] == 1:
            return i

    return None
```

Um diese Stelle bestimmen zu können, wird über die Stellen des gegebenen Tupels iteriert. Sobald eine Eins gefunden wurde, wird der Index zurückgegeben, bei dem sich die Iteration zu dem Zeitpunkt befindet. Dieser Index stellt dann die Position der ersten Eins des Polynoms dar.

Sollte keine Eins gefunden werden, wird von der Funktion 0 zurückgegeben. Demnach ist der Grad des Polynoms gleich Null, da es lediglich aus Nullen besteht.

2.2.3 Grad eines Polynoms bestimmen

```python
def find_degree(F):

    FirstOne = find_pos_of_first_one(F)

    return (len(F)-1) - FirstOne if FirstOne != None else 0
```

Der Grad eines Polynoms wird bestimmt durch den höchsten Exponenten, der in dem jeweiligen Polynom zu finden ist. Um nun, mittels der Funktion ”find_degree”, den Grad eines Polynoms bestimmen zu können, muss zunächst die erste Stelle des Polynoms ermittelt werden, die eine Eins enthält. Dies ist durch die eben aufgeführte Hilfsfunktion *”find_pos_of_first_one”* möglich.

Anhand dieser Stelle, welche das Monom mit dem höchsten Exponent darstellt, kann nun der Grad des Polynoms ermittelt werden. Um den Grad ermitteln zu können, muss von der Gesamtlänge des Polynoms die Zahl, der Index der ersten Eins, abgezogen werden. Dadurch erhält man die Länge des Polynoms, wenn man von der ersten Eins aus anfängt zu zählen. Von dieser Länge muss nun noch eins abgezogen werden, da das letzte Monom den Exponenten null hat.

Als Ergebnis erhält man den höchsten Exponenten, der in dem Polynom zu finden ist, und dies entspricht dem Grad des Polynoms. Sollte das Polynom jedoch nur aus Nullen bestehen, so beträgt der Grad null.

2.2.4 Polynom auf Nullstellen prüfen

Mittel der Funktion *"has_root"* ist es möglich, ein Polynom darauf zu prüfen, ob es eine Nullstelle im Grundkörper $\mathbb{F}_2$ hat. Ein Polynom hat genau dann eine Nullstelle, wenn man in dem Polynom für die Variable X ein Element aus dem Grundkörper $\mathbb{F}_2$ einsetzen kann, wodurch das Polynom zu Null wird. In der Charakteristik zwei, das heißt in Körpern, die auf dem Grundkörper $\mathbb{F}_2$ basieren, gibt es hierbei lediglich die beiden Elemente Null und Eins, die in das Polynom eingesetzt werden können.

Die Null stellt genau dann eine Nullstelle dar, wenn das kleinste Monom den Vorfaktor Null hat. Denn in allen anderen Monomen des Polynoms wird eine Null eingesetzt, wodurch sich all diese zu Null addieren. Dadurch muss nur das niedrigste Monom betrachtet werden, da dies durch den Exponenten von Null nicht von der für die Variable eingesetzten Zahl abhängt. Demnach gilt: Wenn die letzte Stelle des Tupels eine Eins ist, stellt die Null keine Nullstelle dar, und wenn die letzte Stelle des Tupels eine Null ist, ist die Null eine Nullstelle des Polynoms.

Für die Eins hingegen müssen andere Bedingungen überprüft werden. Hierfür ist insbesondere die folgende Rechenregel wichtig, die aufgrund des Binärsystems gilt: $1 + 1 = 0$. Daraus ergibt sich, dass, wenn die Summe der Einser gerade ist, das Ergebnis Null wird, da sich immer zwei Einser gegenseitig auslöschen. Dadurch gilt, dass die Eins eine Nullstelle ist, sollten die Anzahl der Einser im binären Tupel gerade sein. Somit ist bei ungerader Anzahl der Einser im Tupel die Eins keine Nullstelle.

```python
def has_root(F):

    if F[-1] == 0:
        return True

    counter = F.count(1)

    return (counter % 2) == 0
```

In der Implementierung wurden die eben aufgeführten Bedingungen für Nullstellen, bezüglich der Elemente des Grundkörpers $\mathbb{F}_2$, folgendermaßen umgesetzt. Da für die Null lediglich die letzte Stelle entscheidend ist, ob sie eine Nullstelle für das Polynom ist, wird überprüft, ob die letzte Stelle eine Null ist. Sollte diese Bedingung stimmen, gibt die Funktion *'True'* zurück, da in diesem Fall eine Nullstelle vorliegt.

Um die Eins zu überprüfen, wird die Anzahl der vorkommenden Einsen gezählt. Soll-

te diese Anzahl gerade sein, gibt es eine Nullstelle und somit gibt die Funktion *‘True‘* zurück. Sollten diese beiden aufgeführten Bedingungen nicht eingetreten sein, hat das Polynom keine Nullstelle und die Funktion gibt *‘False‘* zurück.

2.2.5 Tupel nach Stellen aufteilen

Die Hilfsfunktion *”create_splitted_tuples”* bewirkt, dass ein Polynom in verschiedene Polynome aufgeteilt werden kann, sodass die Polynome nach der Aufspaltung lediglich eine Null besitzen. D.h. das ursprüngliche Tupel wird in so viele Tupel aufgespalten, wie die Anzahl Einsen des Tupels beträgt. Dabei behalten die aufgespaltenen Tupel die Information, wo die Einsen des ursprünglichen Tupels standen.

Beispielsweise wird das Tupel $(1, 0, 1, 1)$ in drei verschiedene Tupel aufgeteilt, da in dem ursprünglichen Tupel drei Einsen vorhanden sind. Sodass nun die Information über die Stellen, an denen sich die Einsen befinden, nicht verloren geht, werden die Einsen in dem neuen Tupel genau die gleiche Stelle haben, wie in dem ursprünglichen Tupel. Nun werden aus dem ursprünglichen Tupel $(1, 0, 1, 1)$ drei Tupel, mit jeweils einer Eins, an ihrer entsprechenden Stelle, erstellt. Diese sehen dann folgendermaßen aus: $(1, 0, 0, 0)$, $(0, 0, 1, 0)$ und $(0, 0, 0, 1)$. Dabei ist wichtig, dass alle Tupel die gleiche Länge wie das ursprüngliche Tupel haben.

```
1 def create_splitted_tuples(indizes, length):
2
3     return [tuple(1 if i == index else 0
4         for i in range(length))
5         for index in indizes]
```

Im Zuge der Implementierung werden dieser Funktion zwei Parameter mitgegeben. Zum einen eine Liste *’indizes’*, die bestimmt, an welchen Stellen später die Einsen stehen müssen, und zum anderen die Länge *’length’*, die bestimmt, wie viele Stellen die einzelnen Tupel haben müssen.

In der Funktion wird über den Funktionsparameter *’indizes’* iteriert und für jeden Index ein Tupel erstellt, das nur an der Stelle des Index eine Eins hat, und der Rest wird mit Nullen gefüllt. Diese Tupel haben die Länge des gegebenen Funktionsparameters *’length’*. Die von der Funktion zurückgegebene Tupel-Liste enthält nun die aufgespaltenen Tupel, der Länge *’length’*, die jeweils nur eine Eins an der zugewiesenen Stelle haben.

Diese Hilfsfunktion wird für die Implementierung eines Algorithmus für die Multipli-

kation von Polynomen benötigt. Dieser Algorithmus basiert darauf, dass eines der Polynome in mehrere Polynome aufgespalten wird, sodass das Distributivgesetz der Multiplikation einfacher umgesetzt werden kann.

2.2.6 Exponent zu einer Stelle finden

Die Hilfsfunktion *"find_exponent"* dient dazu, in einem Tupel zu einer angegebenen Stelle den entsprechenden Exponenten zu finden.

```
1 def find_exponent(F, pos):
2
3     return len(F) - pos - 1
```

Das heißt, die Funktion bekommt ein Tupel sowie eine Zahl als Parameter. Die Zahl bestimmt dabei, zu welcher Stelle im Tupel der entsprechende Exponent gefunden werden soll.

Der gesuchte Exponent wird bestimmt, indem von der Länge des Tupels die Stelle des gesuchten Exponenten innerhalb des Tupels abgezogen wird. Davon muss zusätzlich noch eins abgezogen werden, da die Zählweise der Stellen null-basiert ist. Würde man beim Zählen der Stellen bei eins anfangen, müsste man das Abziehen der zusätzlichen Eins weglassen.

2.2.7 Ableitung bestimmen

Mithilfe der Funktion *"find_derivative"* kann die Ableitung eines Polynoms bestimmt werden. Die Ableitung wird im späteren Verlauf für den Algorithmus von Cantor-Zassenhaus benötigt.

```
1 def find_derivative(F):
2
3     L = [find_exponent(F, i) % 2 if F[i] == 1 else 0
4                                  for i in range(len(F))]
5
6     return cut_zeros_left(tuple(L[:-1]))
```

Zu Beginn wird über alle Stellen des Polynoms *"F"*, welches abgeleitet werden soll, iteriert und für jede Stelle einzeln die zugehörige Ableitung bestimmt. Diese Stellen werden dann jeweils in die Liste *"L"* geschrieben, wobei diese Liste am Ende die Ableitung darstellt.

Eine Stelle kann in der Ableitung nur dann eins sein, wenn der Vorfaktor im ursprünglichen Polynom eine Eins ist. Der Vorfaktor muss eins sein, da, wenn er null wäre, die Ableitung eines einzelnen Monoms folgendermaßen aussehen würde: $r \cdot 0 \cdot X^{r-1}$ (wenn das ursprüngliche Monom $0 \cdot X^r$ ist). Und aufgrund der Multiplikation mit Null im Vorfaktor fällt das Monom somit in der Ableitung weg, und an die Lösungsliste *"L"* wird eine 0 für diese Stelle angefügt.

Wenn der Vorfaktor eins ist, muss zudem noch der Exponent ungerade sein, damit das Monom in der Ableitung ungleich Null ist. Sollte der Exponent nämlich gerade sein, sieht die Ableitung eines Monoms der Form X^r (wobei r gerade ist) so aus: $r \cdot X^{r-1}$. Da jedoch r gerade ist, ist es in der Charakteristik zwei gleich Null, da jede gerade Zahl modulo zwei gleich Null ist, wodurch erneut mit Null multipliziert wird. Dadurch wird ein Monom mit geradem Exponenten ebenfalls zu 0 in der Ableitung.

Demnach wird eine Stelle in der Ableitung nur dann eins, wenn die folgenden zwei Bedingungen gelten. Zum einen muss der Vorfaktor des Monoms eins sein, und zum anderen muss der Exponent ungerade sein, sodass dieser Exponent modulo zwei gleich eins ergibt.

Nachdem nun einzeln über die Stellen des Ausgangspolynoms *"F"* iteriert wurde, und für jedes Monom die Ableitung gebildet wurde, muss nun die letzte Stelle der Lösungsliste abgeschnitten werden. Das liegt daran, dass bei der Ableitung die Exponenten immer um je eins geringer werden. Durch das Entfernen der letzten Stelle wird die Länge der Liste um eins kleiner, und somit verringern sich alle Exponenten um eins. Zudem fallen Konstanten, und somit die letzte Stelle, bei der Ableitung weg, sodass bei diesem Abschneiden der letzten Stelle keine wichtige Information verloren geht.

Schließlich wird die Liste umgewandelt, als Tupel von der Funktion zurückgegeben, wobei noch alle führenden Nullen von links, mittels der Hilfsfunktion *"cut_zeros_left"*, abgeschnitten werden.

2.3 Arithmetische Operationen

Da die Charakteristik Zwei auf dem Binärsystem basiert, verhalten sich die arithmetischen Operationen innerhalb von Körpern der Charakteristik Zwei analog zu den Berechnungen im Binärsystem. Im Folgenden wird die Implementierung der arithmetischen Rechenoperationen für binäre Tupel, welche Polynome der Charakteristik Zwei darstellen, genauer erläutert.

2.3.1 Addition von Polynomen

Bei der Addition von Polynomen in Körpern der Charakteristik zwei gelten die Rechenregeln wie bei der Addition im Binärsystem. Diese lauten folgendermaßen. $0+0=0; 0+1=1; 1+0=1; 1+1=0$ (mit Übertrag 1).

```
1 def add_polynoms(F, G):
2
3     max_len = max(len(F), len(G))
4
5     F = (0,) * (max_len - len(F)) + F
6     G = (0,) * (max_len - len(G)) + G
7
8     return tuple([(F[i] + G[i]) % 2
9                   for i in range(max_len)])
```

Um nun diese Addition von Polynomen in Python zu implementieren, werden der Funktion *"add_polynoms"* zunächst zwei Polynome übergeben, die in Form von binären Tupeln dargestellt werden. Um diese beiden binären Tupel zu addieren, wird über die Stellen der Tupel iteriert, und jede Stelle einzeln, nach den oben gegebenen Rechenregeln, binär addiert. Die Ergebnisse werden dann schrittweise dem Lösungspolynom hinzugefügt.

Bevor die iterative Addition der Stellen der Polynome durchgeführt werden kann, muss jedoch sichergestellt werden, dass auch jeweils die passenden Stellen miteinander addiert werden. Als Voraussetzung dafür, dass immer die passenden Stellen miteinander addiert werden, gilt, dass die beiden Polynome, beziehungsweise die beiden Tupel, gleich lang sind. Um sicherzustellen, dass die beiden Tupel gleich viele Stellen haben, wird, vor der iterativen Addition, bestimmt, welches Tupel das längere der beiden ist, und diese Länge wird zwischengespeichert. Im Anschluss werden beide Tupel mit führenden Nullen von links aufgefüllt, bis beide die gespeicherte maximale Länge haben. Diese Nullen dienen lediglich zur richtigen Zuordnung der Stellen bei der Addition. Zudem ändern sie nichts an den Rechnungen, da die Zahl Null das neutrale Element in der Addition darstellt. Demnach wird durch diese Auffüllung mit Nullen das Ergebnis der Addition nicht beeinflusst.

Schließlich wird am Ende der Funktion das Lösungspolynom in Form eines binären Tupels zurückgegeben.

2.3.2 Multiplikation von Polynomen

Das Multiplizieren von zwei Polynomen funktioniert so, dass man die beiden Polynome mithilfe des Distributivgesetzes ausmultipliziert. Das bedeutet, dass jedes Monom des ersten Polynoms mit jedem Monom des zweiten Polynoms multipliziert wird.

Bei diesem Ausmultiplizieren der einzelnen Monome kann es dazu kommen, dass mehrere Monome die gleiche Potenz haben. Diese müssen dann noch miteinander addiert werden. Da diese Polynome in Körpern der Charakteristik zwei sind, muss hierbei lediglich geprüft werden, ob die Anzahl Monome mit der gleichen Potenz gerade oder ungerade ist.

```
1  def multiply_polynoms(F, G):
2
3      S = [0] * (len(F) + len(G) - 1)
4
5      for i, coef1 in enumerate(F):
6          for j, coef2 in enumerate(G):
7
8                  S[i + j] ^= coef1 * coef2
9
10     return cut_zeros_left(tuple(S))
```

Die Multiplikation von zwei Polynomen wird durch die Funktion *"MultiplyPolynoms"* ermöglicht. Hierbei wird zu Beginn ein Lösungsarray definiert und alle Stellen mit Null initialisiert. Die Länge dieses Lösungsarrays wird von den Längen der beiden Faktoren bestimmt. Und zwar bekommt das Lösungsarray die Länge der Summe der beiden Faktoren. Davon muss jedoch noch eins abgezogen werden, da die Multiplikation mit dem letzten Monom keine zusätzliche Stelle liefert.

Wie bereits erwähnt, muss jedes Monom des einen Polynoms mit jedem Monom des anderen Polynoms multipliziert werden. Daraus folgt, dass für die Implementierung der Multiplikation von Polynomen zwei ineinander geschachtelte Schleifen benötigt werden, die jeweils über die Länge der beiden Polynome iterieren.

Innerhalb dieser beiden Schleifen wird dann die Multiplikation der einzelnen Monome, sowie die Addition von Monomen gleicher Potenzen durchgeführt. Dabei wird anhand der Schleifenindizes die aktuelle Stelle des Lösungspolynoms ermittelt, und der Inhalt dieser aktuellen Stelle wird mit dem Produkt der Multiplikation der einzelnen Monome mittels einer *'XOR'*-Anweisung verknüpft. Die *'XOR'*-Verknüpfung

entspricht in der Charakteristik zwei der Addition, wodurch die Multiplikation und die Addition zusammen in einer Anweisung durchgeführt werden können.

Im Anschluss wird das Lösungsarray noch zu einem Tupel umgewandelt. Für die Zwischenspeicherung der Lösung wurde ein Array benutzt, da Tupel in Python ”immutable” sind, das heißt, man kann den Inhalt nicht mehr ändern. Von diesem Tupel werden anschließend noch, mittels der Hilfsfunktion *”cut_zeros_left”*, die führenden Nullen von links abgeschnitten, da dies überflüssige Stellen sind, und dieses Tupel wird dann von der Funktion zurückgegeben.

2.3.3 Reduktion eines Produkts

Wenn man sich in einem Körper, hier der Charakteristik zwei, befindet, gibt es eine definierende Relation, die diesen Körper definiert. Dafür gilt, dass alle Elemente, die in diesem Körper sind, einen geringeren Grad als das Minimalpolynom, bzw. als die definierende Relation, haben müssen.

Da es bei dem eben aufgeführten Verfahren der Multiplikation von Polynomen dazu kommen kann, dass die Stellenanzahl, und damit der Grad des Lösungspolynoms nach der Multiplikation größer wird als der Grad des Minimalpolynoms zu der zugehörigen definierenden Relation, muss nach der Berechnung der Multiplikation noch eine Reduktion des Produkts mithilfe der definierenden Relation des Körpers durchgeführt werden.

```
1  def reduce_product(F, M):
2
3      Gf, Gm = find_degree(F), find_degree(M)
4
5      if Gf < Gm:
6          return (0,) * (Gm - len(F)) + F
7
8      F, M = cut_zeros_left(F)[::-1], cut_zeros_left(M)[::-1]
9
10     while len(F) >= len(M):
11
12         remaining_exponent = len(F) - 1 - Gm
13
14         temp_M = (0,) * remaining_exponent + M[:-1]
15
16         F = cut_zeros_left(add_polynoms(F[:-1], temp_M)[::-1])[::-1]
17
18     return (F + (0,) * (Gm - len(F)))[::-1]
```

Diese Reduktion funktioniert folgendermaßen. Zunächst werden der Funktion *”reduce_product”* zwei Polynome übergeben. Zum einen das Polynom *”F”*, das reduziert

werden soll, und zum anderen das Minimalpolynom *"M"*, das für die definierende Relation steht.

Als erstes werden die Grade beider Polynome mittels der Hilfsfunktion *"find_degree"* ermittelt. Sollte der Grad des Polynoms *"F"* kleiner als der Grad des Polynoms *"M"* sein, bedeutet das, dass das Polynom *"F"* nicht reduziert werden muss. Demnach kann dieses Polynom unverändert zurückgegeben werden, wobei es noch um führende Nullen ergänzt wird, damit es so viele Stellen hat, wie in dem Körper, der durch die Relation definiert wird, zugelassen sind.

Sollte der Grad des Polynoms *"F"* jedoch gleich oder höher als der Grad des Polynoms *"M"* sein, bedeutet das für das Polynom *"F"*, dass es reduziert werden muss. Dazu werden zuerst alle führenden Nullen von beiden Polynomen weggeschnitten und im Anschluss werden die beiden Polynome umgedreht. Durch diese Umkehrung der Polynome entsprechen nun die Stellen der Polynome den Exponenten der jeweiligen Monome.

Nun wird das Polynom *"F"* stellenweise mit der Relation reduziert, wobei pro Stelle, die weggeschnitten wird, der Grad des Polynoms um eins verringert wird. Dieser Vorgang wird so lange wiederholt, bis der Grad von *"F"* kleiner als der Grad des Minimalpolynoms *"M"* ist. Im Zuge dieser stellenweisen Reduktion wird zuerst der höchste Exponent des Polynoms ermittelt, indem von der Länge des Polynoms eins abgezogen wird. Von diesem höchsten Exponenten des Polynoms wird nun der Grad des Minimalpolynoms *"M"* abgezogen. Dieser Wert beschreibt nun, welcher Faktor nach dem Schritt der Vereinfachung übrig bleibt. Beispielsweise ist der Grad der Relation acht und der höchste Exponent des Polynoms *"F"* ist 13. Dann wird dieser Exponent 13 mittels der Potenzregel für gleiche Basen aufgeteilt in $X^8 * X^5$. Hierbei kann nun das Monom mit Exponent acht durch die Relation ersetzt werden.

Im weiteren Verlauf der Funktion wird nun die letzte Stelle, entsprechend dem Monom mit dem höchsten Exponenten, des Polynoms *"F"* abgeschnitten, da diese Stelle in dem aktuellen Schritt der Vereinfachung wegfällt. Anschließend wird ein temporäres Polynom erstellt, welches beschreibt, was zu *"F"* addiert werden muss. Dieses temporäre Polynom stellt den aktuellen Schritt der Vereinfachung dar, der eben beispielsweise anhand des Exponenten 13 erklärt wurde. Dazu werden dem temporären Polynom von links so viele Nullen angefügt, wie hoch der Faktor nach dem Schritt der Vereinfachung übrig bleibt. Anschließend wird das Minimalpolynom rechts an dem temporären Polynom angefügt, wobei die letzte Stelle weggelassen wird. Um zurück zu dem eben aufgeführten Beispiel zurückzukommen, entspre-

chen die Nullen dem Monom mit Exponent fünf, und das Minimalpolynom ohne den höchsten Exponenten stellt die Relation dar.

Anschließend wird das Polynom *"F"* mit dem eben aufgeführten temporären Polynom addiert, und somit wird ein Schritt der Vereinfachung durchgeführt. Dann werden noch alle führenden Nullen von rechts weggeschnitten. Dazu wird das Tupel umgedreht, alle führenden Nullen von links weggeschnitten, und anschließend wird das Tupel erneut umgedreht. Diese Schritte der Vereinfachung werden so lange wiederholt, bis der Grad von *"F"* kleiner als der Grad des Minimalpolynoms *"M"* ist.

Ist diese Bedingung nun erreicht, ist die Reduktion des Polynoms *"F"* mittels der definierenden Relation des Körpers, bzw. mittels des Minimalpolynoms, vollendet. Das Ergebnis wird nun noch um führende Nullen ergänzt, bis es die maximale Länge hat, die in dem jeweiligen Körper möglich ist. Außerdem wird das Ergebnis noch umgekehrt, sodass nun das Monom mit dem höchsten Exponenten wieder an erster Stelle steht.

2.3.4 Alternativer Algorithmus für die Multiplikation

Der bereits aufgeführte Algorithmus, um Polynome zu multiplizieren, basiert darauf, dass die beiden Polynome stellenweise miteinander ausmultipliziert werden. Und anschließend wird das Lösungspolynom in einem separaten Schritt mittels der definierenden Relation, bzw. dem zugehörigen Minimalpolynom reduziert, sodass das Lösungspolynom auch ein Element des zu betrachtenden Körpers (der Charakteristik zwei) ist.

Alternativ gibt es noch einen anderen Algorithmus, mit dem diese Multiplikation möglich ist. Bei diesem Algorithmus passieren die beiden Schritte des vorherigen Algorithmus, also zum einen die reine Multiplikation der beiden Polynome, und zum anderen die Reduktion mittels des Minimalpolynoms, zusammen in einem Schritt.

Bei diesem Algorithmus wird einer der Polynomfaktoren schrittweise aufgeteilt, sodass die beiden Polynome nicht miteinander nach dem Distributivgesetz ausmultipliziert werden, sondern dass das eine Polynom schrittweise mit den Potenzen des anderen Polynoms multipliziert. Sollte dabei der Grad des Polynoms so groß werden, wie der Grad der Relation, wird der höchste Exponent direkt mit der Relation ersetzt, also wird die Relation zum aktuellen Stand des Lösungspolynoms addiert. Dieser Vorgang wird so lange wiederholt, bis über alle Stellen des einen Polynoms iteriert worden ist.

```python
def alternative_multiplication(F, G, M):

    if find_degree(F) >= find_degree(M) or find_degree(G) >= find_degree(M):
        return False

    R = cut_zeros_left(M[1:])

    pos_of_ones = [index for index, value in enumerate(F) if value == 1]

    tuple_list = create_splitted_tuples(pos_of_ones, len(F))

    L = ()

    for i in range(len(tuple_list)):

        if tuple_list[i][-1] == 1:
            L = add_polynoms(L,G)
        else:
            distance = len(F) - 2 - tuple_list[i].index(1)

            iteration_count = distance + 1

            temp = G

            for j in range(iteration_count):

                temp = add_polynoms(temp[1:] + (0,), temp[0] * R)

            L = add_polynoms(L, temp)

    return L
```

In der Implementierung werden der Funktion *älternative_multiplication"* zunächst drei Funktionsparameter übergeben. Dazu zählen die beiden Polynomfaktoren *"F"* und *"G"* sowie das Minimalpolynom *"M"*, welches die definierende Relation des Körpers darstellt.

Im ersten Schritt überprüft die Funktion, ob die zwei Polynomfaktoren überhaupt zulässig sind. Dazu müssen die Grade der beiden Polynome kleiner als der Grad des Minimalpolynoms sein. Wenn der Grad von einem der beiden Polynome nicht kleiner als der Grad des Minimalpolynoms ist, bedeutet das, dass dieses Polynom dann kein Element des Körpers ist, welcher durch das Minimalpolynom, bzw. die darauf basierende definierende Relation, bestimmt wird.

Als nächstes wird ein neues Tupel *"R"* erstellt, welches die definierende Relation darstellt. Die definierende Relation kann aus dem Minimalpolynom *"M"* bestimmt werden, in dem lediglich die erste Stelle, also das Monom mit dem höchsten Exponenten, weggelassen wird. Beispielsweise kann das Minimalpolynom $M(X) = X^3 + X + 1$ (in Tupelschreibweise: *(1,0,1,1)*) als definierende Relation der Form $X^3 = X + 1$ (in Tupelschreibweise: *(0,1,1)*) geschrieben werden. Dabei geht zwar die Information verloren, welches die Stelle mit dem höchsten Exponenten ist, doch

diese Information kann weiterhin über den Grad des Minimalpolynoms ermittelt werden. Da in diesem Algorithmus mit der definierenden Relation gerechnet werden muss, ist es sinnvoll, diese in dem Tupel *"R"* zu speichern.

Anschließend werden alle Stellen des Tupels *"F"* ermittelt, an denen eine Eins steht. Dies wird mittels einer Schleife, die über die Stellen von *"F"* iteriert und in eine Liste schreibt, welche Stellen eine Eins beinhalten, ermöglicht.

Da wir nun in einer Liste genau die Stellen haben, an denen *"F"* eine Eins hat, können wir mittels der Hilfsfunktion *"create_splitted_tuples"* eine Liste an Tupeln erstellen, die jeweils genau eine Eins haben, wobei diese an genau den Stellen stehen, an denen sie auch in dem ursprünglichen Tupel *"F"* stehen. Diese Tupel haben alle jeweils die Länge des ursprünglichen Tupels, da diese Länge als zweiter Funktionsparameter, neben den Stellen, an denen sich in dem Ausgangstupel die Einsen befinden, mitgegeben wurde.

Mittels dieser Liste an Tupeln wird im weiteren Verlauf die Multiplikation funktionieren, indem der zweite Tupel-Faktor *s*chrittweise mit den einzelnen Tupeln der eben erstellten Tupel-Liste multipliziert wird. Sollte in dieser Berechnung der Grad des Produkt-Polynoms auf den Grad des Minimalpolynoms anwachsen, so wird die höchste Stelle durch die definierende Relation ersetzt, und diese wird dann zu dem Produkt-Polynom hinzuaddiert.

Nun wird das Lösungspolynom *"L"* definiert, in dem im weiteren Verlauf schrittweise die Lösung der Multiplikation eingefügt wird.

Anschließend wird in einer Schleife über die Anzahl der Elemente in der erstellten Tupel-Liste iteriert. Nun wird geprüft, ob im aktuellen Tupel der Iteration die letzte Stelle eine Eins ist. Da die letzte Stelle den Exponenten Null darstellt, bedeutet das, sollte die letzte Stelle eine Eins sein, dass das zweite Polynom *"G"* lediglich mit Eins multipliziert werden muss. Das kann über die Addition des Polynoms *"G"* zum Lösungspolynom *"L"* dargestellt werden.

Sollte die letzte Stelle des aktuellen Tupels keine Eins sein, muss ermittelt werden, an welcher Stelle die Eins in dem Polynom steht, um damit die richtige Potenz bestimmen zu können, die für den nächsten Schritt der Multiplikation benötigt wird. Im nächsten Schritt wird die Distanz von der Stelle, an der sich die Eins des Tupels befindet, zur vorletzten Stelle berechnet. Denn für den Algorithmus ist diese Distanz entscheidend, um zu wissen, wie oft die schrittweise Multiplikation des Po-

lynoms *"G"* mit dem Monom X durchgeführt werden muss. Beispielsweise bedeutet die Distanz 2, dass der Exponent zu der zugehörigen Einser-Stelle 3 beträgt. Dieses Monom wiederum kann dann zu dreimal dem Monom X aufgespalten werden (jeweils durch eine Multiplikation getrennt). Denn laut Potenzregel für die Multiplikation gilt: $X^3 = X * X * X$. Dann muss der Algorithmus dreimal das Monom X mit dem Polynom *"G"* multiplizieren.

Diese Anzahl, die bestimmt, wie oft das Monom X mit dem Polynom *"G"* multipliziert werden muss, ergibt sich aus der eben errechneten Distanz (von der Stelle der Eins zur vorletzten Stelle), indem sie um eins erhöht. Wie man am Beispiel gesehen hat, ergibt eine Distanz von 2, dass der Algorithmus dreimal durchlaufen muss. Diese Information wird als *ïteration_count"* gespeichert, da sie im weiteren Verlauf des Algorithmus benötigt wird.

Nun wird ein neues Tupel *temp* erstellt, und mit dem Inhalt von *"G"* befüllt. Dieses Tupel dient als Zwischenspeicher, der in den Iterationsstufen des Algorithmus dazu dient, zu speichern, was in jedem Schritt zum Lösungspolynom *"L"* hinzuaddiert werden muss. Ein neues Tupel zur Zwischenspeicherung muss erstellt werden, da der Inhalt des Tupels *"G"* unverändert erhalten bleiben muss.

Nun folgt eine Schleife, wobei die Durchlaufzahl von dem eben ermittelten *ïteration_count"* bestimmt wird. Innerhalb der Schleife wird der Inhalt des Tupels *"temp"* erneuert. Und zwar dadurch, dass das Tupel *"temp"* einmal nach links geshiftet wird. Dies wird umgesetzt, indem die erste Stelle abgeschnitten wird und am rechten Ende eine zusätzliche 0 hinzugefügt wird. Somit bleibt die Länge des Tupels erhalten. Zu diesem verschobenen Tupel wird nun die Relation (mittels des Tupels *"R"*) hinzuaddiert. Jedoch findet diese Addition nur dann statt, wenn die erste Stelle, die von dem Tupel *temp* beim Shiften abgeschnitten wurde, eine Eins war. Dies wird sichergestellt, indem das Tupel für die Relation *"R"* mit der weggeschnittenen Stelle von *"temp"* multipliziert wird.

Anschließend wird das Tupel für die Zwischenspeicherung *"temp"*, für jeden Schleifendurchlauf für die aufgespaltenen Tupel, zu dem Lösungspolynom *"L"* hinzuaddiert.

Am Ende der Funktion wird das Lösungstupel *"L"* zurückgegeben. Dieses Tupel enthält nun das Produkt der Multiplikation der beiden Ausgangstupel und ist bereits mittels des Minimalpolynoms bzw. mittels der definierenden Relation reduziert, sodass das Lösungstupel auch ein Element des Körpers ist, der durch die Relation

definiert wird.

2.3.5 Polynomdivision

Die Division von Polynomen funktioniert so, dass man die beiden Polynome mit Rest teilt. Dazu schaut man, wie oft der Nenner in den Zähler passt, und das, was dabei übrig bleibt, wird der Rest. Das Vorgehen dabei ist wie beim schriftlichen Dividieren von reellen Zahlen, nur dass hierbei mit Monomen, d.h. mit Potenzen zur Variable X gerechnet wird. Der Fakt, dass wir in der Charakteristik zwei sind, erleichtert diese Polynomdivision in gleich zwei verschiedenen Aspekten. Zum einen gibt es als Vorfaktor zu den jeweiligen Monomen lediglich die beiden Möglichkeiten null und eins, das heißt, der Vorfaktor bestimmt nur, ob das Monom vorhanden ist oder nicht. Zum anderen gelten die üblichen Rechenregeln der Charakteristik zwei, das bedeutet, dass es keine Subtraktion gibt, da diese im Binärsystem gleichbedeutend mit der Addition ist. Dazu vereinfacht die Rechenregel $1+1=0$ die Polynomdivision um einiges, da, wenn in einer Addition zwei Monome mit dem gleichen Exponenten addiert werden müssen, streichen sich die beiden Monome aufgrund dieser Rechenregel weg.

Der genaue Vorgang der Polynomdivision läuft folgendermaßen ab. Zu Beginn wird geguckt, wie oft der Nenner in den Zähler passt. Dies kann über deren Exponenten bestimmt werden, und zwar passt der Nenner so oft in den Zähler, wie die Differenz der beiden höchsten Exponenten von Zähler und Nenner ist. Dann wird zum Ergebnis ein Monom mit genau diesem Exponenten geschrieben. Als nächstes folgt die Rückrechnung, das heißt, das Monom, welches eben zum Ergebnis hinzugefügt wurde, wird mit dem Nenner multipliziert. Dieses Ergebnis wird anschließend mit dem ursprünglichen Zähler addiert. Die Summe aus dieser Rechnung stellt nun den neuen Zähler dar. Also wird der beschriebene Vorgang im nächsten Schritt mit diesem neuen Zähler und dem ursprünglichen Nenner durchgeführt. Dieser Prozess wird so lange wiederholt, bis beim Schritt der Addition die Summe entweder null oder ein Polynom mit kleinerem Grad als dem Grad des Nenners ergibt.

Sollte die Summe null ergeben, bedeutet das, dass die Polynomdivision ohne Rest aufgegangen ist. In diesem Fall ist somit der Nenner ein echter Teiler des Zählers, also kann man den Zähler in die zwei Faktoren zerlegen. Zum einen in den Nenner und zum anderen in das Polynom, welches als Ergebnis der Polynomdivision herauskommt. Sollte jedoch das Ergebnis der Summe ein Polynom ergeben, dessen Grad kleiner ist als der Grad des Nennerpolynoms, bedeutet das, dass dieses Polynom nicht mehr in den Nenner passt, und somit stellt dieses Polynom den Rest dar,

der bei der Polynomdivision übrig bleibt. In diesem Fall führt die durchgeführte Polynomdivision zu dem Ergebnis-Polynom, wobei der eben aufgeführte Rest zusätzlich übrig bleibt.

```
1  def polynom_division(F, G):
2
3      if G == (1,):
4          return F, (0,)
5
6      L = []
7
8      starting_len = len(F)
9
10     for counter in range(len(F)):
11
12         if len(F[counter:]) < len(G):
13             return tuple(L), tuple(F)
14
15         L.append(F[counter])
16
17         if F[counter] == 1:
18             F = add_polynoms(F[counter:], G + (0,) * (len(F[counter:]) - len(G)))
19
20         F = (0,) * (starting_len - len(F)) + F
```

In der Implementierung werden der Funktion *"polynom_division"* die beiden Polynome als Parameter gegeben. Die Funktion gibt zwei verschiedene Werte zurück, zum einen das Ergebnis der Polynomdivision und zum anderen den übrigbleibenden Rest. Als erstes wird geprüft, ob das Nenner-Polynom gleich 1 ist. Sollte dies der Fall sein, wird als Ergebnis das Zähler-Polynom und 0 als Rest zurückgegeben.

Sollte das Nenner-Polynom ein nicht-triviales Polynom sein, wird zunächst eine Liste *"L"* definiert, in die die Lösung geschrieben wird. Zudem wird die Ausgangslänge des Tupels *"F"* gespeichert, da diese später unverändert benötigt wird, jedoch ändert sich diese im weiteren Verlauf dynamisch.

Nun wird über die Länge des Zähler-Tupels *"F"* iteriert, und diese Schleife bricht ab, sobald dieses Tupel kürzer als das Nenner-Tupel *"G"* ist. Das ist gleichbedeutend damit, dass der Grad des Zählers kleiner als der Grad des Nenners ist. Die Länge von *"F"* wird dabei in jedem Iterationsschritt um eins verringert, da einzeln über dessen Stellen iteriert wird. Sollte diese Abbruchbedingung erreicht sein, wird die Lösungsliste *"L"*, in Form eines Tupels, als Ergebnis zurückgegeben. Zudem wird das Tupel *"F"* als Rest zurückgegeben, da das, was noch in diesem Tupel übrig bleibt, den Rest der Polynomdivision darstellt.

Sollte die Abbruchbedingung noch nicht erreicht sein, wird zum Lösungstupel *"L"* die aktuelle Stelle der Iteration von *"F"* hinzugefügt. Wenn diese Stelle nun eine Null war, bedeutet das, dass keine Rückrechnung erforderlich ist, und somit wurden lediglich die Potenzen im Lösungstupel um eins erhöht. Sollte diese Stelle jedoch eine Eins sein, muss eine Rückrechnung durchgeführt werden. Für diese Rückrechnung müssen zwei Polynome addiert werden. Zum einen das Zähler-Polynom *"F"* beginnend bei der aktuellen Iterationsstufe. Und zum anderen das Nenner-Polynom *"G"*, wobei bei diesem Tupel noch Nullen hinzugefügt werden müssen, damit die beiden Tupel für die Addition die gleiche Länge haben.

Am Ende jeder Iteration wird *"F"* von links mit Nullen gefüllt, bis es wieder die Länge des Ausgangstupels hat. Dadurch werden die Potenzen des Zähler-Tupels angepasst, damit diese für den weiteren Funktionsverlauf übereinstimmend sind.

2.4 Polynom-Algorithmen

In diesem Abschnitt werden weitere Algorithmen beschrieben, die für den Umgang mit Polynomen geeignet sind. Zum einen der euklidische Algorithmus zur Bestimmung des ggT's von zwei Polynomen. Und zum anderen zwei Algorithmen, die ein Polynom darauf prüfen, ob es ein Minimalpolynom ist, bzw. ob das Polynom irreduzibel ist.

2.4.1 Euklidischer Algorithmus

Der euklidische Algorithmus wird verwendet, um den größten gemeinsamen Teiler (kurz ggT) zu finden. Sollte dieser ggT gleich eins sein, bedeutet das, dass die beiden Zahlen teilerfremd sind, also sie haben keinen gemeinsamen, nicht-trivialen Teiler.

Der euklidische Algorithmus funktioniert so, dass man die beiden Zahlen, zu denen der ggT gefunden werden soll, so lange mit Rest durcheinander teilt, bis der Rest gleich 0 ergibt. Anschließend führt man eine Rückrechnung durch, sodass der ggT als Produkt von den beiden ursprünglichen Zahlen geschrieben werden kann. Die Vorfaktoren der beiden ursprünglichen Zahlen werden Bézout-Koeffizienten genannt.

Der euklidische Algorithmus für Polynome funktioniert analog zu den beschriebenen Verfahren, mit dem Unterschied, dass mittels Polynomdivision gerechnet werden muss, anstatt einfacher Division mit Rest für die natürlichen Zahlen.

```python
def euclid(m, n):

    r0, r1 = m, n

    a0 = b1 = (1,)
    a1 = b0 = (0,)

    while True:

        if tuple_to_polynom(r1) == 0:
            return r0, a0, b0

        r1 = cut_zeros_left(r1)

        q, r = polynom_division(r0, r1)

        if r == (0,):
            return r1, a1, b1

        a0_old, b0_old = a0, b0

        a0, b0 = a1, b1

        a1 = add_polynoms(a0_old, multiply_polynoms(q,a1))
        b1 = add_polynoms(b0_old, multiply_polynoms(q,b1))

        r0, r1 = r1, r
```

In der Implementierung muss der euklidische Algorithmus für Polynome etwas angepasst werden. Dabei wird die Rückwärtsrechnung durch Vorwärtsrechnen ersetzt, da man sich für die Rückwärtsrechnung alle Zwischenschritte merken muss, während man sich beim Vorwärtsrechnen lediglich den letzten Schritt merken muss. Daher ist dieses Verfahren für die Implementierung geeigneter.

Für den Algorithmus werden die folgenden Variablen benötigt, $a_0, a_1, b_0, b_1, r_0, r_1$. Dabei stehen *ä"* und *"b"* jeweils für die beiden Vorfaktoren der ursprünglichen Polynome und *"r"* für das Rest-Polynom. Die Indizes 0 und 1 stehen dabei für die alten, bzw. neuen Werte der jeweiligen Polynome.

Der Funktion werden die beiden Polynome *"m"* und *"n"* als Parameter übergeben. Zu Beginn werden die Variablen r_0 und r_1 initial auf die beiden Funktionsparameter gesetzt. Während der Implementierung müssen die beiden folgenden Gleichungen immer gelten:

$$r_0 = a_0 \cdot m + b_0 \cdot n$$

$$r_1 = a_1 \cdot m + b_1 \cdot n$$

Damit diese Gleichungen für die initiale Befüllung gelten, müssen die Variablen folgendermaßen befüllt werden. Für die erste Gleichung: $a_0 = 1$, $b_0 = 0$ und für die zweite Gleichung: $a_1 = 0$, $b_1 = 0$.

Anschließend folgt eine Schleife, die solange läuft, bis der ggT der beiden Zahlen gefunden wurde, wobei neben dem ggT auch die beiden Bézout-Koeffizienten zu den beiden Polynomen zurückgegeben werden.

Innerhalb dieser Schleife wird dann geprüft, ob das Tupel r_1 leer ist, also ob es lediglich aus Nullen besteht. Sollte dieses Tupel leer sein, so beschreibt r_0 den ggT von *"m"* und *"n"*, mit zugehörigen Bézout-Koeffizienten a_0 und b_0. Diese drei Werte werden somit von der Funktion zurückgegeben.

Ansonsten werden von r_1 alle führenden Nullen weggeschnitten, und dann wird auf die beiden Tupel r_0 und r_1 mithilfe der bereits definierten Funktion *"polynom_division"* eine Polynomdivision durchgeführt. Dabei wird das Ergebnis in die Variable q und der Rest in r gespeichert. Wenn dabei der Rest r null ist, so ist der ggT gefunden und steht in $r1$. Dieser ggT wird zusammen mit den beiden Bézout-Koeffizienten a_1 und b_1 von der Funktion zurückgegeben.

Sollte in dieser Schleifeniteration der ggT noch nicht gefunden worden sein, so werden alle Variablen für den nächsten Iterationsschritt angepasst. Dafür werden a_0 und b_0 zunächst mittels a_0old und b_0old zwischengespeichert und anschließend werden a_0 und b_0 mit den Werten aus a_1 und b_1 gefüllt. Die beiden Werte a_1 und b_1 werden wiederum durch die folgenden Gleichungen erneuert:

$$a_1 = a_0old + q \cdot a_1$$

$$b_1 = b_0old + q \cdot b_1$$

Dann wird noch das Tupel r_0 durch r_1 neu befüllt und r_1 bekommt den Inhalt von r.

Mit diesen neuen Variableninhalten folgt nun der nächste Iterationsschritt der Schleife. Diese Schritte werden so lange wiederholt, bis einer der beiden Fälle, die oben aufgeführt sind, eintritt, bei denen der ggT gefunden wird.

2.4.2 Minimalpolynome

Minimalpolynome sind Polynome, die keinen nicht-trivialen Teiler haben, das heißt, sie sind durch kein nicht-triviales Polynom ohne Rest teilbar. Minimalpolynome ha-

ben die Eigenschaft, dass aus ihnen eine Relation geschaffen werden kann, die Körper der Form $\mathbb{F}_q$ mit $q = p^l$ definieren.

Ist ein Polynom

$$F(X) = X^l + r_{l-1} \cdot X^{l-1} + ... + r_1 \cdot X + r_0$$

ein Minimalpolynom des Körpers $\mathbb{F}_q$. So gibt es ein $\alpha \, \epsilon \, \mathbb{F}_q$, sodass die Relation

$$\alpha^l = -r_{l-1} \cdot \alpha^{l-1} - ... - r_1 \cdot \alpha - r_0$$

den Körper $\mathbb{F}_q$ definiert.

Da in der Charakteristik die Subtraktion analog zur Addition ist, kann die definierende Relation in der Form

$$\alpha^l = r_{l-1} \cdot \alpha^{l-1} + ... + r_1 \cdot \alpha + r_0$$

geschrieben werden.

Damit ein Polynom ein Minimalpolynom sein kann, darf es zum einen keine Nullstelle über dem Grundkörper $\mathbb{F}_2$ haben. Und zum anderen darf es nicht in zwei nicht-triviale Faktoren aufgespalten werden können. Das bedeutet, es darf keinen nicht-trivialen Teiler haben, der das Polynom ohne Rest teilt.

Um bestimmen zu können, ob ein Polynom einen nicht-trivialen Teiler hat, benötigt man zunächst alle Minimalpolynome bis zum Grad $\lfloor \frac{d}{2} \rfloor$ (wenn d der Grad des Polynoms ist, das auf ein Minimalpolynom überprüft wird, ist). Das liegt daran, dass ein Polynom nur in mindestens zwei Faktoren zerfallen kann, und wenn es einen nicht-trivialen Teiler hat, dessen Grad größer als $\lfloor \frac{d}{2} \rfloor$ ist, so hat es auch einen nicht-trivialen Teiler, dessen Grad kleiner als $\lfloor \frac{d}{2} \rfloor$ ist. Deshalb reicht die Überprüfung auf nicht-triviale Teiler bis zum Grad $\lfloor \frac{d}{2} \rfloor$.

Beispielsweise muss ein Polynom des Grades 5 auf echte Teiler bis zum Grad $\lfloor \frac{5}{2} \rfloor = 2$ überprüft werden. Dafür muss das Polynom zuerst auf Nullstellen im Grundkörper $\mathbb{F}_2$ überprüft werden, wodurch auf nicht-triviale Teiler des Grades 1 überprüft wird. Zudem muss es noch auf nicht-triviale Teiler des Grades 2 überprüft werden. Da es für den Grad 2 lediglich das Minimalpolynom $X^2 + X + 1$ gibt, reicht es, das Polynom des Grades 5 durch dieses Minimalpolynom zu teilen. Sollte bei dieser Polynomdivision ein Rest übrig bleiben, so ist $X^2 + X + 1$ kein Teiler des Polynoms vom Grad 5. Somit hat das Grad keine echten Teiler vom Grad 1 oder 2. Daraus folgt,

dass es auch keine echten Teiler vom Grad 3 oder 4 hat, da, wenn es einen Teiler vom Grad 3 (bzw. vom Grad 4) hätte, wäre der andere Faktor, in den das Polynom zerfällt, vom Grad 2 (bzw. vom Grad 1). Jedoch wurden bereits echte Teiler vom Grad 1 und 2 ausgeschlossen, wodurch auch die Grade 3 und 4 ausgeschlossen werden können.

```python
def is_minpol(F):

    if has_root(F):
        return False

    max_div_degree = find_degree(F) // 2

    list_of_minpols = get_all_minpols(max_div_degree)

    for G in list_of_minpols:

        Solution, Remainder = polynom_division(F, G)

        if tuple_to_polynom(Remainder) == 0:
            return False

    return True
```

Die Funktion *"is_minpol"* bekommt ein Polynom, in Form eines binären Tupels, als Parameter und gibt zurück, ob das Polynom ein Minimalpolynom ist oder nicht. Dabei wird zunächst mittels der Hilfsfunktion *"has_root"* überprüft, ob das Polynom, bzw. das binäre Tupel eine Nullstelle im Grundkörper $\mathbb{F}_2$ hat. Wenn das Polynom eine Nullstelle hat, bedeutet das, dass das Polynom kein Minimalpolynom sein kann, weshalb die Funktion in diesem Fall *'False'* zurückgibt.

Als nächstes wird der maximale Teilergrad bestimmt, in den das Polynom, vom Grad d zerfallen kann. Dieser maximale Teilergrad beträgt $\lfloor \frac{d}{2} \rfloor$. Daraufhin wird eine Liste an Minimalpolynomen bestimmt, deren Grad von 2 bis $\lfloor \frac{d}{2} \rfloor$ reicht. Dazu wurde die Funktion *"get_all_minpols"* implementiert.

```python
def get_all_minpols(max_degree):

    return [polynom for degree in range(2, max_degree + 1)
                    for polynom in product([0,1], repeat = degree + 1)
                    if polynom[0] and is_minpol(polynom)]
```

Diese Funktion bekommt als Parameter den maximalen Teilergrad, bis zu dem die Minimalpolynome bestimmt werden sollen. Diese von der Funktion *"get_all_minpols"* generierten Polynome haben einen Grad von 2 bis zum gegebenen maximalen

Grad. Mittels der Funktion *"product"* von der Python-Bibliothek *"itertools"* werden alle möglichen Tupelkombinationen erstellt, deren Länge jeweils zu dem Grad des Tupels passt. Anschließend wird noch überprüft, dass die erste Stelle des Tupels eine 1 ist, da ansonsten der Grad nicht mehr zu dem Tupel passt, und es wird mittels der Funktion *"is_minpol"* nur die Tupel zurückgegeben, die auch ein Minimalpolynom sind.

Dabei fällt auf, dass die beiden Funktionen *"is_minpol"* und *"get_all_minpols"* sich gegenseitig aufrufen, das heißt, es werden schrittweise alle Minimalpolynome bis zu dem bestimmten Teilergrad generiert, und dabei muss in jedem Schritt überprüft werden, ob die möglichen Teilerpolynome auch selbst bereits Minimalpolynome sind.

Das bedeutet, dass diese Implementierung für höhere Grade der zu überprüfenden Polynome immer ineffizienter wird, da sich die beiden Funktionen immer häufiger gegenseitig aufrufen müssen. Für Grade bis ca. 10 funktioniert diese Implementierung noch, jedoch steigt die Ausführungszeit der Funktion exponentiell. Im späteren Verlauf wird der Algorithmus von Cantor-Zassenhaus implementiert, der Polynome auf Irreduzibilität, und damit auch auf Minimalpolynome, überprüft, der deutlich effizienter ist. Mit diesem Algorithmus können dann auch Polynome mit Graden größer als zehn auf Minimalpolynome überprüft werden.

Nachdem die Funktion *"is_minpol"* die Liste aller Minimalpolynome bis zum Grad $\lfloor \frac{d}{2} \rfloor$ bestimmt hat, wird über diese Liste iteriert, und für jedes Element dieser Liste wird eine Polynomdivision, mittels der Funktion *"polynom_division"* mit dem zu überprüfenden Polynom *"F"* durchgeführt. Sollte eine dieser Polynomdivisionen ohne Rest aufgehen, bedeutet das, dass das Polynom *"F"* einen nicht-trivialen Teiler hat, und demnach kein Minimalpolynom sein kann. In diesem Fall wird von der Funktion *'False'* zurückgegeben.

Sollte wiederum keine Polynomdivision, von *"F"* mit den Minimalpolynomen bis zum Grad $\lfloor \frac{d}{2} \rfloor$, ohne Rest aufgehen, so ist *"F"* ein Minimalpolynom, und somit gibt die Funktion *'True'* zurück.

2.4.3 Cantor-Zassenhaus-Algorithmus

```python
def cantor_zassenhaus(F):

    dF = find_derivative(F)

    if dF == (0,):
        return False

    ggT_dF, _, _ = euclid(F,dF)

    if find_degree(ggT_dF) >= 1:
        return False

    for l in range(1, find_degree(F) // 2):

        h = (1,) + (0,) * (2 ** l)

        h = add_polynoms(h, (1,0))

        ggT_h, _, _ = euclid(F, h)

        if find_degree(ggT_h) >= 1:
            return False

    return True
```

2.5 Performance-Vergleiche

2.5.1 Vergleich der beiden Multiplikations-Algorithmen

2.5.2 Vergleich Minimalpolynom- mit Cantor-Zassenhaus-Algorithmus

3 Beispielaufgaben

x

Literatur

[1] a. c, x. Zugriff am xx. xxxxxxx 2025.

[2] test. test. *test*, test(test):test, test.