Schnelle Multiplikation

$O(n^2)$ ist für Anfänger

Johannes Hahn

06.06.25 – 09.06.25

Inhaltsverzeichnis

0 Crashkurs theoretische Informatik

0.1 Landau-Notation

0.1: Wir wollen uns damit auseinandersetzen, welche verschiedenen Algorithmen es gibt, um große Zahlen miteinander zu multiplizieren, und wie sie sich voneinander unterscheiden. Wir wollen insbesondere die Geschwindigkeit dieser Algorithmen untersuchen.

Das ist nur leider ein relativ schwieriges Problem. Offensichtlich ist die Laufzeit vom Input abhängig. Zehn-stellige Zahlen und 1.000.000-stellige Zahlen zu multiplizieren wird offensichtlich unterschiedlich lange dauern. Also wollen wir nicht zwei Zahlen miteinander vergleichen und herausfinden, welche die größere ist, sondern zwei Funktionen, die uns die Laufzeit eines Algorithmus in Abhängigkeit von der Größe der Inputs berechnet.

Es ist nicht von vornherein offensichtlich, wie das geschehen soll. Als Beispiel betrachte man

$$f(n) = 100n + 2 \quad \text{vs.} \quad g(n) = 3n^2$$

Welches ist die „bessere“ dieser Funktionen? Für $n \leq 33$ ist $g(n)$ die kleinere Zahl, für größere n hingegen $f(n)$. Es scheint als sei keiner der beiden hypothetischen Algorithmen, die hinter diesen beiden Funktionen stehen, optimal für alle Inputs. Stattdessen würde man viel mehr einen hybriden Algorithmus benutzen wollen, der das eine Verfahren für kleine und das andere für große Inputs implementiert. Aber natürlich weiß man das erst hinterher, wenn man die anderen Verfahren schon kennt...

0.2: Zusätzlich kommt hinzu, dass keine der Konstanten 100,2,3 in den Funktionen zuverlässig vorhergesagt werden kann. Wie viele Sekunden ein konkretes Programm für einen bestimmten Input tatsächlich benötigt, hängt von einer Vielzahl von Faktoren ab, die völlig unabhängig vom verwendeten Algorithmus sind:

a.) Auf welchem Rechner wird der Algorithmus ausgeführt? Hat er einen 2 GHz oder 4 GHz Prozessor? Wie groß sind die L1, L2, L3-Caches? Ist es ein Laptop? Ist er gerade im Batteriemodus? Scheint gerade die Sonne auf den Bildschirm?

b.) In welcher Programmiersprache ist das Programm geschrieben worden und wie wurde es kompiliert? C ? C++? Java? C++11 oder C++23 ? Welcher Compiler auf welcher Optimierungsstufe wurde verwendet? Java 17, 21 oder 24 ? Ist eine frische JVM oder eine aufgewärmte verwendet worden?

c.) Welche anderen Prozesse liefen gerade auf dem Rechner?

d.) ...

Und selbst wenn wir in der Lage wären, einige dieser Faktoren gut genug zu verstehen, um eine belastbare Vorhersage der Laufzeit zu machen, würde solch eine Vorhersage nicht lange halten. Im Herbst dieses Jahres[1] wird Java 25 veröffentlicht werden; möglicherweise

[1] Das wäre das Jahr 2025 für alle, die dies in der Zukunft lesen. Seid gegrüßt Zeitreisende!

ist unser Algorithmus auf der neuen JVM ein bisschen schneller. Unsere Analyse zum jetzigen Zeitpunkt wird jedenfalls nicht berücksichtigen können, was in den nächsten Monaten an Mikro-Optimierungen implementiert werden wird.

0.3: Auf der anderen Seite ist die exakte Anzahl an Millisekunden vermutlich gar nicht so relevant und uns interessiert viel mehr, ob wir überhaupt warten müssen, ob es sich lohnt einen frischen Kaffee zu holen (oder eine Tafel Schokolade zu essen), ob wir das Ergebnis überhaupt in diesem Leben noch erfahren werden. Dafür ist der genaue Wert dieser Konstanten vermutlich nicht relevant.

Wir wollen also eine eher grobe Analyse durchführen, die von möglichst vielen hässlichen Faktoren der Realität unabhängig ist.
Wir wollen insbesondere keine exakten Zeiten messen. Typischerweise werden wir etwa die Laufzeit unserer Algorithmen nicht in Sekunden, sondern der Anzahl von „elementaren Operationen“ zählen. Was genau „elementar“ ist, hängt vom Kontext ab.

Klassisches Thema einer „Theoretische Informatik I“-Vorlesung wären z.B. die verschiedenen Sortieralgorithmen, für die man üblicherweise zählt, wie oft man zwei Elemente der zu sortierenden Liste miteinander vergleichen und ggf. ihre Positionen vertauschen muss, bis man die Liste sortiert hat.

Wir werden davon ausgehen, dass wir mit 1- und 2-stelligen Zahlen bereits addieren und multiplizieren können, mit 1.000.000-stelligen Zahlen aber nicht. Wir zählen also die Operationen mit kleinen Zahlen, die wir benötigen, um mit großen Zahlen zu rechnen.

Ein anderer Algorithmus, der ein anderes Ziel hat und sich nicht für die Details der Multiplikation interessiert, wird vielleicht die Multiplikation beliebiger Zahlen als eine Operation zählen.

0.4: Weil wir insbesondere am Multiplikationsproblem *sehr großer* Zahlen interessiert sind, werden wir außerdem in vielen Fällen vernachlässigen, wenn ein Algorithmus für kleine Inputs suboptimal ist. Wie bereits angemerkt, kann man das beheben, indem man verschiedene Algorithmen miteinander geschickt kombiniert. Für die theoretische Analyse soll uns aber vor allem das Verhalten für $n \to \infty$ interessieren.[2]

0.5 Definition (Landau-O-Notation)**:**
Es sei $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ eine Funktion. Die Menge $O(f)$ definieren wir als

$$O(f) := \{\, g : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid \exists c, n_0 > 0 \forall n \geq n_0 : f(n) \leq cg(n) \,\}$$

Wir schreiben der Kürze halber $O(f(n))$, wenn wir $O(f)$ meinen, also z.B. $O(n^2)$ anstatt $O(n \mapsto n^2)$.

[2] Man beachte, dass wir damit wirklich die Grenzen der Realität verlassen. Das beobachtbare Universum ist verdammt groß, aber trotzdem endlich. Es gibt eine Größe von Inputs, die einfach nicht ins Universum passt. Spätestens ab dieser Größe ist es für alle Zwecke der realen Welt sinnlos, sich noch über die Laufzeit von Algorithmen zu unterhalten. Wir werden es trotzdem tun, einfach weil's Spaß macht!

0.6: $g \in O(f)$ bedeutet anschaulich, dass $g(n)$ höchstens so schnell wächst wie $f(n)$ für sehr große Inputs.
$f(n) = n^2$ hat z.B. die Eigenschaft, dass sich $f(n)$ vervierfacht, wenn n sich verdoppelt. $g \in O(n^2)$ bedeutet somit, dass $g(2n)$ nicht viel mehr als viermal größer als $g(n)$ ist, zumindest wenn n hinreichend groß ist.

0.7: In dem meisten Fällen kann man die Definition auch umformulieren in

$$g \in O(f) \iff \limsup_{n\to\infty} \frac{g(n)}{f(n)} < \infty$$

da man meistens nicht $f(n) = 0$ für unendlich viele n hat. Man betrachtet i.d.R. monoton wachsende Funktionen f mit $\lim_{n\to\infty} f(n) = \infty$ für die Analyse.

0.8: Man beachte aber, dass dies keine Aussage über die exakten Werte von $g(n)$ macht. $0,0000001n^2$ ist genauso in $O(n^2)$ wie $1.000.000n^2 + 10^{10^{10^{42}}}$. Wenn ein Algorithmus einfach genug aufgebaut ist, kann man manchmal auch exakte Werte ermitteln, aber es ist typischerweise einfacher, nur auf die O-Komplexität zu schauen, weil man dann solche Unterschiede ignorieren kann.
Es kommt auch häufig vor, dass man den Term, der die Komplexität bestimmt, gut versteht, aber kleinere Terme ignorieren will. Dann würde man z.B. herausfinden, dass die betrachtete Funktion die Form $3n^2 + O(n)$ hat o.Ä.; der führende Term $3n^2$ bestimmt die Komplexität – es ist $O(n^2)$ –, aber die echten Werte sind nicht exakt $3n^2$, die Differenz liegt aber in der kleineren Klasse $O(n)$.

0.9: Man beachte auch, dass die O-Notation nur eine Abschätzung nach oben liefert. Wenn wir uns nicht clever anstellen, kann es also passieren, dass wir massiv überschätzen. Es ist z.B. völlig korrekt zu sagen, dass $f(n) = n^2 + 17$ in $O(n^{100})$ enthalten ist.

Es gibt ähnlich gelagerte Definitionen wie $o(f)$ und $\Theta(f)$, die für untere bzw. genaue Abschätzung gemacht sind.

Je komplexer ein Algorithmus ist, desto schwieriger ist es, genau zu erkennen, dass man die bestmögliche Abschätzung gefunden hat und in der Tat gibt es Algorithmen, für die die präzise Komplexität nicht bekannt ist.

Noch schlimmer wird es, wenn man nicht die Komplexität eines einzelnen Algorithmus betrachtet, sondern einfach nur ein bestimmtes zu lösendes Problem vorgibt und sich *alle* Algorithmen anschaut, die dieses Problem lösen könnten. Wenn man einen einzelnen Algorithmus findet, bekommt man eine Abschätzung nach oben, aber um die genaue Komplexität des Problems zu kennen, muss man *alle* Algorithmen betrachten und beweisen, dass keiner von ihnen in eine strikt bessere Laufzeitklasse fällt. Damit sind nicht alle *bekannten* Algorithmen gemeint, sondern alle nur denkbaren!

Selbst für viele elementare Probleme ist die optimale Laufzeit nicht bekannt! Für Ganzzahl-Multiplikation *vermutet* man, dass die Komplexität $\Theta(n \log(n))$ ist, aber wir können uns nicht sicher sein. Immerhin wissen wir sei 2019, dass es tatsächlich einen Algorithmus in $O(n \log(n))$ gibt, die obere Grenze also schon einmal stimmt.

Für Matrix-Multiplikation vermutet man $n^{2+o(1)}$, aber man kennt nicht einmal einen Algorithmus, der dem nahe käme. Der beste bekannte Algorithmus ist in $O(n^{2,371552})$, aber man weiß nicht, wie viel dichter an 2 man kommen kann.

0.10 Beispiel: a.) Es ist immer $f \in O(f)$.

b.) $O(1)$ ist die Menge aller beschränkten Funktionen, also der Funktionen f, für ein c existiert mit $f(n) \leq c$ für alle $n \in \mathbb{N}$.

c.) $O(1) \subsetneq O(n^2) \subsetneq O(n^3) \subsetneq O(n^4) \subsetneq \ldots$

d.) Für Polynome ist nur der Grad entscheidend: Ist $f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots$, dann $f \in O(n^k)$.

e.) Allgemeiner: O-Notation ignoriert „Terme kleinerer Ordnung“, d.h. wenn $g \in O(f)$ ist, dann ist $f + g$ immer noch in $O(f)$.

0.11: Wir verwenden die O-Notation nicht nur für die Funktionen, die die Laufzeit von Algorithmen messen, sondern beispielsweise auch für Funktionen, die den Speicherbedarf messen.
Wir würden also z.B. sagen, dass ein Algorithmus bei einem Input der Größe n eine „Laufzeit von $O(n^6)$ und Speicherbedarf von $O(n^3 \log(n)^2)$ hat“. Damit meinen wir, dass die Funktion, die die Laufzeit in Abhängigkeit von n misst, in $O(n^6)$ enthalten ist, und die Funktion, die den Speicherbedarf in Abhängigkeit von n misst, in $O(n^3 \log(n)^2)$.

1 Level 0: $O(n^2)$ wie in der Grundschule

1.1 Das Problem

1.1: Konkret ist unser Problem folgendes:

Gegeben zwei natürliche Zahlen A, B, dargestellt durch ihre n Ziffern in Basis g

$$A = \sum_{i=0}^{n-1} a_i g^i, \quad B = \sum_{i=0}^{n-1} b_i g^i$$

mit $0 \leq a_i, b_i < g$, berechne die Ziffern $c_0, c_1, \ldots, c_{2n-1}$ des Produkts $C := A \cdot B$.

Wir messen die Laufzeit von Multiplikationsalgorithmen in Abhängigkeit von n. Wir nehmen an, dass Operationen auf einzelnen Ziffern eine konstante Laufzeit haben.

1.2: Die Annahme, dass Operationen mit einzelnen Ziffern eine konstante Laufzeit haben, ist in der Theorie dadurch gerechtfertigt, dass man eine Additions- und Multiplikationstabelle vorberechnen kann. Dann hat jede Addition oder Multiplikation von zwei Ziffern die Laufzeit von „Schau einen Eintrag in eine $n \times n$ Tabelle nach".
So funktionierte das letztendlich auch in der Grundschule. Das Einmaleins haben wir auswendig gelernt und schriftliches Addieren und Multiplizieren von größeren Zahlen nutzt den „Lookup" im Gedächtnis.

1.3: In Computerhardware sind i.A. keine Lookup-Tabellen implementiert. Die offensichtliche Wahl der Basis wäre $g = 2$. Die Addition und Multiplikation von einzelnen Bits sind einfach[3] direkt in Hardware zu implementieren, sodass kein Lookup nötig ist.

Etwas weniger offensichtlich ist es, auf „Wörtern" statt Bits zu arbeiten, d.h. $g = 2^8$, 2^{16} (richtig alte Hardware), 2^{32} (alte Hardware und einige moderne Mobilgeräte) oder 2^{64} (moderne Hardware, insb. PCs) zu wählen. Manche Hardware unterstützt sogar noch höhere Wortgrößen.

Während Speicher für eine 256×256 große Tabelle mit je 2 B großen Einträgen sicher noch verkraftbar wäre, ist eine $2^{32} \times 2^{32}$ Tabelle schon zu groß und eine $2^{64} \times 2^{64}$ Lookup-Tabelle völlig unrealistisch: 2^{64} viele 64 bit große Einträge sind 1.476×10^8 TB und 2^{128} viele 128 bit große Einträge sind $\approx 5.445 \times 10^{27}$ TB. Kein Datacenter der Welt hat diese Kapazität! Das ist ein Vielfaches der Gesamtgröße des Internets.

Also wird auch in diesen Fällen das Ergebnis einer einzelnen Multiplikation mit einer speziellen Instruktion direkt von der Hardware berechnet. Die Schaltkreise sind allerdings schon deutlich komplexer. Und in der Tat sind die Schaltkreise für größere Wörter i.d.R. zusammengesetzt aus Schaltkreisen, die Addition und Multiplikation kleinerer Wörter implementieren, d.h. es wird ein passender Multiplikationsalgorithmus in Hardware implementiert.

Solange alle Multiplikation von zwei Wörtern stets durch den gleichen Schaltkreis ermittelt wird, kann aber ebenfalls davon ausgegangen werden, dass die Laufzeitkosten konstant sind.

[3] Wie „einfach" es auch sein mag, wenige hundert Atome große Schaltkreise zu bauen...

1.4: Gewisse Operationen zählen wir nicht zur Laufzeit des Algorithmus:

Wir werden keine Additionen zählen, die Schleifenvariablen inkrementieren. Die sind Teil des Konstrukts „Schleife“, nicht unseres spezifischen Algorithmus. Und wie die Schleifen tatsächlich ausgeführt werden, ist außerdem zu schwer vorhersagbar, um konkrete Anzahlen an Additionen vorhersagen zu können (Stichwort „loop-tiling“, „loop-unrolling“, „auto-vectorization“, „SIMD“, ...)

Wir werden uns außerdem keine Gedanken um Speicherzugriffe machen. Speicher ist in unserer Analyse kostenlos.
Auch hier ist die Realität wieder sehr anders; die echte Laufzeit echten Codes auf echter Hardware ist sehr wohl davon abhängig, wie häufig im Hauptspeicher gelesen oder geschrieben wird, ob Caches effizient genutzt werden können etc. Es gibt auch Theorie für diese Aspekte (Stichwort „cache-friendliness“, „cache-oblivious algorithms“ etc.), aber wir werden das in diesem Kurs nicht untersuchen.

1.2 Der erste Algorithmus

1.5: Der offensichtliche Algorithmus für Multiplikation beliebiger natürlicher Zahlen ergibt sich direkt durch Anwendung des Distributivgesetzes:

$$\left(\sum_{i=0}^{n-1} a_i g^i\right) \cdot \left(\sum_{j=0}^{n-1} b_j g^j\right) = \sum_{i,j} (a_i b_j) g^{i+j} = \sum_{k=0}^{2n-2} \left(\sum_{i+j=k} a_i b_j\right) g^k$$

Indem wir alle n^2 Produkte $a_i b_j$ berechnen und geeignet aufsummieren, erhalten wir das Produkt $A \cdot B$.

In der Grundschule lernt man meist ein tabellarisches Verfahren, das schematisch in etwa so abläuft, dass in jeder Zeile das Produkt einer einzelnen Ziffer a_i von A mit B ausgerechnet wird, um i viele Dezimalstellen verschoben wird (was die Multiplikation mit 10^i darstellt), sowie am Ende alle Zeilen aufsummiert werden.

In jeder Zeile werden dafür die Produkte $a_i \cdot b_j$ ausgerechnet. Grob vereinfacht würde das also so aussehen:

a_2	a_1	a_0	$\cdot$	b_3	b_2	b_1	b_0
				a_0b_3	a_0b_2	a_0b_1	a_0b_0
			a_1b_3	a_1b_2	a_1b_1	a_1b_0	0
		a_2b_3	a_2b_2	a_2b_1	a_2b_0	0	0
	c_6	c_5	c_4	c_3	c_2	c_1	c_0

Natürlich sind die Produkte $a_i \cdot b_j$ der einzelnen Ziffern i.A. größer als eine Ziffer, sodass man in jeder Zeile und von einer Zeile zur nächsten jeweils die Überträge beachten muss.

A1 Algorithmus (Schriftliches Multiplizieren)**:**
Wir benutzen die Schulmethode direkt. Einziger Unterschied ist, dass wir ein laufendes Zwischenergebnis pflegen und dies kontinuierlich updaten, anstatt n viele Zwischenergebnisse zu speichern und erst am Ende alle aufzuaddieren.

```
List<Digit> multiply(List<Digit> a, List<Digit> b){
  // initialize with a.length + b.length zeros
  List<Digit> c := new List<Digit>(a.length+b.length);

  for(i := 0; i < a.length; i++){
    Digit z := 0;

    for(j := 0; j < b.length; j++){
      // p := c[i+j] + a_i * b_j + z is at most a 2-digit number !!
      Digits {p_1,p_0} := c[i+j] + a[i] * b[j] + z;

      c[i+j] := p_0;
      z      := p_1;
    }
    c[i + b.length] := z;
  }

  return c;
}
```

1.6 Proposition (Korrektheit)**:**
Der Algorithmus A1 ist korrekt. Er benötigt n^2 Multiplikation, $O(n^2)$ Additionen und benötigt $2n + O(1)$ Speicher.

Beweis. Das meiste ist relativ klar. Wir beginnen, indem wir C mit den Ziffern $000\ldots000$ initialisieren. Dann addieren wir für jedes i die Zahl $a_i g^i \cdot B$ zu C, indem wir $a_i \cdot B$ berechnen und auf die Ziffern $\ldots c_{i+2}c_{i+1}c_i$ addieren, und speichern das Ergebnis wieder in C.

Wir addieren in jeder Iteration der inneren Schleife die 1er-Ziffer des Produkts $p := a_i \cdot b_j$ an die korrekte Stelle c_{i+j}. Wir addieren außerdem den Übertrag der letzten Iteration an diese Stelle und merken uns den Übertrag, den wir dabei erhalten, für die nächste Iteration.

Damit das funktioniert, müssen wir die mit `!!` markierte Behauptung beweisen:
Das Produkt zweier Ziffern $a_i, b_j \leq g-1$ ist höchstens $(g-1)^2 = g^2 - 2g + 1$ und die Addition von zwei weiteren Ziffern $c, z \leq g-1$ führt zu einem Maximum von $g^2 - 1$, was echt kleiner g^2 ist, also wirklich maximal zwei Ziffern hat. Der Übertrag z ist also auch in der nächsten Iteration maximal eine Ziffer groß. □

1.7: Wie wir eben gezeigt haben, ist es möglich, einen Schaltkreis zu bauen, der vier Wörter a, b, c, z als Input nimmt, und in einer einzigen Operation das 2-Wort breite Ergebnis $ab + c + z$ berechnet.

Es ist von der konkreten Prozessorarchitektur abhängig, ob es tatsächlich eine solche Instruktion gibt. Wenn es sie gibt, sind überhaupt keine Additionsinstruktionen zusätzlich zu den Multiplikationen für unseren Algorithmus nötig.

Wenn es diesen Schaltkreis nicht gibt, dann gibt es oft trotzdem $(a, b, c) \mapsto ab + c$, womit wir nur noch eine einzige Addition zusätzlich zur Multiplikation benötigen.

1.8: Der Algorithmus ist offenbar nicht davon abhängig, dass A und B die gleiche Anzahl an Ziffern haben. Etwas präziser könnten wir die Laufzeiten also als $O(nm)$ ausdrücken, wobei n und m die Längen der beiden Zahlen sind.

Darin enthalten ist die Teilaussage, dass die Multiplikation mit einer kleinen Konstanten in $O(n)$ realisierbar ist. Das merken wir uns für später, weil wir es verwenden werden.

1.9: Man beachte auch, dass die asymptotische Laufzeit nicht von der Wahl von g abhängig ist. Wenn wir eine andere Basis g' wählen, dann sind A und B Zahlen mit $\approx \frac{\log(g)}{\log(g')} n$ Ziffern. Die Basis zu ändern, ändert also die Laufzeit nur um einen konstanten Faktor.

1.10: Es ist nicht einmal notwendig, die beiden Inputs A und B bzgl. derselben Basis auszudrücken. In der Tat ist das genau das Prinzip hinter der „ägyptischen Multiplikation", auch „(russische) Bauernmultiplikation" genannt:
Drückt man eine der beiden Zahlen in Basis 2 aus, etwa A, so reduziert sich das Problem darauf, aus B die Zahlen $2B$, $4B$, $8B$, ... zu berechnen, da die Multiplikationen $a_i \cdot B$ entweder 0 oder B ist (denn a_i ist ja 0 oder 1 in Basis 2), also keinen zusätzlichen Rechenaufwand verursachen:

$$(a_{n-1}2^{n-1} + \ldots + a_2 2^2 + a_1 2 + a_0) \cdot B = \sum_{\substack{0 \le i \le n-1 \\ a_i = 1}} 2^i B$$

Man kann also mit B beginnen und die Zahl immer wieder zu sich selbst addieren, um $2B$, $4B$, ... zu erhalten. Und während dessen, jedes Mal, wenn das passende Bit in A gesetzt ist, eine weitere Addition auf das Zwischenergebnis durchführen.

A2 Algorithmus (Double-and-add)**:**
Input: A in Binärdarstellung $a_{n-1}2^{n-1} + \ldots a_2 2^2 + a_1 2 + a_0$, B in beliebiger Basis $b_{m-1}g^{m-1} + \ldots b_2 g^2 + b_1 g + b_0$.
Output: Das Produkt $C = A \cdot B$ in Basis g.
Pseudocode:

```
List<Digit> multiply(List<Bit> a, List<Digit> b){
  // calculate the number of base-g-digits in A
  n := floor( a.length * log(2)/log(g) ) + 1;
  c := new List<Digit>(n + b.length);

  // stores 2^i*b
  b2 := b;
  for(i := 0; i<a.length; i++){
    if(a[i] == 1){
      c := add(c, b2);
    }
```

```
    b2 := add(b2, b2);
  }

  return c;
}
```

1.11 Proposition (Korrektheit und Laufzeit)**:**
Der Algorithmus ist korrekt, hat Laufzeit $O(n^2)$ (alles Additionen) und Speicherbedarf $4n + O(1)$.

Beweis. Übung. □

1.12: Da man in der Praxis eben nicht mit beliebigen g, sondern mit Zweierpotenzen $g = 2^k$ arbeitet, ist für die Berechnung von $\frac{\log(2)}{\log(g)}$ keine Fließkomma-Arithmetik nötig, da sich alles zu $\frac{1}{k}$ kürzt. Die Länge von A in bits ist typischerweise klein genug, damit die Division `a.length / k` in einer einzigen Hardware-Instruktion berechnet werden kann.

Mehr noch: Wenn g selbst eine Zweierpotenz ist, dann besteht eigentlich kein Unterschied zwischen `List<Bit>` und `List<Digit>` und es ist überhaupt keine zusätzliche Aktion nötig.

Aufgaben

Aufgabe 1.1. – Parallelisierung
Beschleunige Algorithmus A1 durch Parallelisieren einiger Schritte um einen konstanten Faktor, sodass der Speicherverbrauch weiterhin $O(n)$ bleibt.

Aufgabe 1.2. – Square-and-multiply
Es sei ein Monoid $(M, \cdot)$ gegeben und eine Black-box Funktion, die die Multiplikation berechnet.

a.) Finde einen Algorithmus, der für gegebenes $x \in M$ und $N \in \mathbb{N}$ die Potenz x^N in $\leq 2\log_2(N)$ Multiplikationen berechnet.

b.) Was hat das mit dem Problem des Multiplizierens zu tun?

Aufgabe 1.3.
Beweise Proposition 1.11.

2 Level 1: $O(n^{1.585})$ Karatsuba

2.1: Über Jahrtausende ist keine Methode bekannt gewesen, wesentlich schneller als $O(n^2)$ zu multiplizieren. Erst 1960 wurde von Anatoly Karatsuba[4] eine schnellere Methode entdeckt.

Karatsubas Methode arbeitet rekursiv wie viele anderen, die wir uns ansehen werden: Anstatt in der eigentlich beabsichtigten kleinen Zahlenbasis g (z.B. $g = 2$ oder $g = 10$) zu arbeiten, werden die Zahlen zunächst bzgl. einer größeren Basis G dargestellt. Um nun zwei Zahlen in Basis G zu multiplizieren, sind mehrere Multiplikationen von Zahlen $< G$ zu erwarten. Diese werden rekursiv mit dem gleichen Algorithmus, aber bzgl. einer Basis $g \leq G' < G$ berechnet. Oft werden die anderen Basen $G, G', \ldots$ als Potenzen von g gewählt, sodass keine zusätzliche Rechenarbeit notwendig ist, um zwischen G-Ziffern und g-Ziffern umzurechnen. Beispiel: Die Ziffern zur Basis $G = 1000$ der Zahl $A = 123456789$ sind $a_2 = 123, a_1 = 456, a_0 = 789$.

Speziell für Karatsubas Methode wählt man G groß genug, dass die beiden Inputs A und B nur noch aus zwei Ziffern bestehen. Für Zahlen mit zwei Ziffern $A = a_1G + a_0, B = b_1G + b_0$ gilt:

$$AB = \underbrace{a_1b_2}_{=c_2} G^2 + (\underbrace{a_1b_0 + a_0b_1}_{=c_1})G + \underbrace{a_0b_0}_{=c_0}$$

In dieser Form aufgeschrieben, sehen wir genau die 2^2 Multiplikationen, die der klassische Algorithmus benötigt. Karatsuba sah, dass der mittlere Term auch gleich

$$c_1 = (a_1 + a_0)(b_1 + b_0) - a_1b_1 - a_0b_0 = (a_1 + b_1)(b_1 + b_0) - c_2 - c_0$$

ist. Somit kann man c_1 anstatt mit zwei Multiplikationen und einer Addition auch durch eine Multiplikation und zwei Subtraktionen berechnen. Additionen und Subtraktionen haben typischerweise annähernd die gleiche Laufzeit.

A3 Algorithmus (Karatsuba's algorithm)**:**

```
List<Digit> multiply(List<Digit> a, List<Digit> b) {
    int n := max(a.length, b.length);
    if(n <= CUTOFF){
        return slow_multiply(a,b); // algorithm A1
    }

    int k := ceil(n/2); // round up if n is odd

    low   := multiply(a[0..k-1], b[0..k-1]);
    high  := multiply(a[k..n-1], b[k..n-1]);

    mixed_a := add(a[0..k-1], a[k..n-1]);
    mixed_b := add(b[0..k-1], b[k..n-1]);
```

[4] russ. Mathematiker, 1937–2008

```
        product := multiply(mixed_a, mixed_b);
        middle := subtract(subtract(product, high), low);

        return add(low, (0,middle), (0,0,high));
    }
```

2.2: In der Praxis wählt man den Cutoff nicht so klein wie möglich, um möglichst viel mit der „besseren“ Karatsuba-Methode zu arbeiten, sondern ein kleines bisschen größer, da für sehr kleine Anzahlen von Ziffern die Schul-Methode in der Praxis doch schneller ist als die Karatsuba-Methode. Selbst wenn nur einer der beiden Faktoren wenig Ziffern hat, insbesondere wenn ein Faktor nur eine Ziffer hat, ist die Schul-Methode schneller.

Die Standard-Library von Java nutzt beispielsweise $g = 2^{32}$ als Basis, d.h. 32-bit Integer werden direkt per Hardware-Instruktion multipliziert, und $n = 80$ als Cutoff[5]: Sobald einer der Faktoren weniger als 80 Worte groß ist, wird mit der Schul-Methode multipliziert.
Die C-Bibliothek GMP (GNU Multiprecision Arithmetic library) definiert den Cutoff hardware-abhängig zwischen $n = 10$ und 34 Wörtern, z.B. $n = 26$ für ARM64 Apple's M1 Prozessor.

2.3: Man beachte, dass $(a_1+a_0)(b_1+b_0)$ nicht ein Produkt von zwei Zahlen $< G$, sondern $< 2G$ ist. Wir sind also ganz leicht über die Grenze gegangen, bis zu der wir gehen wollten. Man kann auch nicht einfach G etwas größer zu wählen, sodass die führenden Ziffern a_1 und b_1 kleiner werden, z.B. wird das mit $A = 199999$ nicht funktionieren: Egal, welche (sinnvolle) Potenz $G = 10^k$ wir wählen, die Ziffer a_0 wird immer 9..99 sein und somit bei Addition zum Overflow führen.

Da man i.d.R. nicht mit $g = 2$ arbeitet (sondern eher $g = 2^{32}$), benötigt man für die Summen a_1+a_0 und b_1+b_0 selbst im Extremfall nur eine weitere Ziffer. Man verliert also in der Praxis nicht wirklich viel. Insbesondere dann, wenn man die Karatsuba-Methode sowieso nur für die größeren Produkte einsetzt; ob man von 1000 zu 500 oder 501 Ziffern reduziert, ist nicht so furchtbar relevant.

Wenn man es unbedingt möchte, ist es aber möglich, stattdessen

$$c_1 = -(a_1 - a_0)(b_1 - b_0) + a_1b_1 + a_0b_0$$

zu verwenden, denn da $0 \leq a_0, a_1 < G$ gilt, wird auch stets $|a_1 - a_0| < G$ sein. Man zahlt dafür den Preis, ein bisschen mehr Buchhaltung mit den Vorzeichen betreiben zu müssen.

2.4 Proposition:
Algorithmus A3 ist korrekt, benötigt $O(n^{\log(3)/\log(2)})$ Multiplikationen, Additionen und Subtraktionen, sowie $O(n)$ Speicher.

[5] Diese Zahlen sind Implementierungsdetails, die sich von einer Java-Version zur nächsten auch mal ändern können. Ich habe sie im OpenJDK 21 geprüft. Wer selbst nachschauen will: Es geht um die Klasse `java.math.BigInteger` und ihre Methode `multiply(BigInteger)`.

Beweis. Die Abschätzungen ergeben sich direkt aus dem Mastertheorem (siehe Aufgabe 2.1), denn die Anzahl der Multiplikationen erfüllt die Rekursion $M(n) = 3M(\frac{n}{2})$, die Anzahl der Additionen/Subtraktionen $A(n) = 3A(\frac{n}{2}) + 2\frac{n}{2} + 3 \cdot 2n$.

Der Speicherbedarf ergibt sich daraus, dass wir für jede Stufe der Rekursion Speicher für `high`, `low`, `mixed_a`, `mixed_b`, `product`, `middle` sowie das Endergebnis benötigen, die jeweils n, n, $\frac{n}{2}+1$, $\frac{n}{2}+1$, $n+1$, n, $2n$ groß sind. Wir runden das zu $7n + O(1)$.[6]
Wenn wir sequentiell ausführen, also jeweils nur einen Zweig im Rekursionsbaum gleichzeitig im Speicher halten müssen, haben also demnach einen Speicherbedarf von

$$(7n + O(1)) + \left(7\frac{n}{2} + O(1)\right) + \left(7\frac{n}{4} + O(1)\right) + \ldots = 2 \cdot 7n + O(\log(n)) \qquad \square$$

2.5: Solange wir nur endlich viele Prozessoren haben und nicht maximal parallel arbeiten, benötigen wir auch nur ein konstantes Vielfaches der sequentiellen Speichermenge.
Wenn wir maximal parallelisieren und somit alle Zweige der Rekursion gleichzeitig im Speicher halten müssen, dann erfüllt der Speicherbedarf auch eine Rekursion wie $S(n) = 3S(\frac{n}{2}) + 5n$, womit sich dann ein Speicherbedarf von $O(n^{\log(3)/\log(2)})$ ergibt.
In der Praxis hat keine Maschine unendlich viele Prozessoren, sodass man niemals voll parallelisieren kann. Andererseits stoßen wir in der Praxis auch nicht auf Zahlen mit beliebig vielen Ziffern, sondern ausschließlich $\leq 2^{32}$ Ziffern (wenn die Zifferngröße $g = 2^{32}$ ist, dann wäre eine einzige solche Zahl schon $\approx 4\,\text{GB}$ groß!), sodass die Rekursionstiefe $\log(n)$ als praktisch konstant und ≤ 32 betrachtet werden kann und somit auch in der Nähe einer realistischen Anzahl an Prozessoren liegt.

[6] In der Tat kann man clever sein: Für `high`, `low` und `middle` benötigt man keinen zusätzlichen Speicher, wenn man `high`, low direkt im Ergebnis speichert und die Subtraktionen bzw. Addition jeweils in-place ausführt. Dann ist der konstante Faktor 5 statt 7.

Aufgaben

Aufgabe 2.1. – Mastertheorem
Beweise das *Mastertheorem für divide-and-conquer-Algorithmen*:
Gegeben sei eine Rekursion der Form

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

für Zahlen $a, b > 1$ und eine nichtnegative Funktion f. Definiere $c_0 := \frac{\log(a)}{\log(b)}$. Zeige:

a.) (leicht) Wenn $f \in O(n^c)$ mit $c < c_0$, dann ist $T \in \Theta(n^{c_0})$. Speziell für $f(n) := dn^c$ ist sogar $T(n) = k_0 n^{c_0} + kn^c$ mit geeigneten Konstanten k_0, k.

b.) (leicht) Wenn $f \in O(n^{c_0} \log(n)^k)$ mit $k \geq 0$, dann ist $T \in O(n^{c_0} \log(n)^{k+1})$.

c.) (mittel) Wenn $f \in o(n^c)$ mit $c > c_0$, dann ist $T \in o(n^c)$. Wenn zusätzlich $af(\frac{n}{b}) \leq kf(n)$ für eine Konstante $0 \leq k < 1$ und alle hinreichend großen n gilt, dann ist $T \in \Theta(f)$.

3 Level 2: $O(n^{1+\varepsilon})$ Toom-Cook

3.1: Der nächste große Durchbruch kam schnell nach Karatsuba: 1963 bereits beschrieb Toom[7] eine Verallgemeinerung von Karatsubas Algorithmus; 1966 beschrieb Cook[8] eine vereinfachte Version, die heutzutage Toom-Cook-Algorithmus genannt wird.

„Der“ Toom-Cook-Algorithmus ist tatsächlich eine Familie von Algorithmus, parametrisiert durch eine natürliche Zahl $k \geq 2$. In der Praxis werden jedoch einige wenige kleine Werte von k verwendet. Manchmal wird nur der Fall $k = 3$ als „der“ Toom-Cook-Algorithmus bezeichnet.

Die grundlegende Idee ist es, die Input-Zahlen nicht in 2, sondern k Teile zu zerlegen und die G-Ziffern der Inputs

$$A = a_{k-1}G^{k-1} + \ldots + a_1 G + a_0, \quad B = b_{k-1}G^{k-1} + \ldots + b_1 G + b_0$$

als Koeffizienten von zwei Polynomen

$$\alpha(X) = a_{k-1}X^{k-1} + \ldots + a_1 X + a_0, \quad \beta(X) = b_{k-1}X^{k-1} + \ldots + b_1 X + b_0$$

aufzufassen. Die Zahlen A und B sind Auswertungen dieser Produkte an der Stelle $X = G$; das Produkt $C := A \cdot B$ ist entsprechend die Auswertung des Polynoms $\gamma(X) := \alpha(X)\beta(X)$ an der Stelle $X = G$. Wenn wir die Koeffizienten von γ aus den Koeffizienten von α und β berechnen können, dann ist die Ermittlung von C sehr einfach.

Das Produkt $\alpha(X)\beta(X)$ mittels der Definition

$$\alpha(X)\beta(X) = \sum_{i,j} a_i b_j X^{i+j}$$

auszurechnen, würde uns direkt wieder zur Schulmultiplikation führen. Die entscheidende Einsicht ist aber, dass ein Polynom vom Grad d bereits eindeutig durch seine Werte an $d+1$ beliebigen Stützstellen festgelegt ist. Wir benötigen $2k-1$ Stützstellen für γ.
Wenn wir an Stelle der relativ großen Zahl G sehr kleine Stützstellen, z.B.

$$s \in \{ -k-1, \ldots, -1, 0, 1, \ldots, k-1 \}$$

verwenden, dann sind die Ergebnisse $\alpha(s)$ und $\beta(s)$ eher in der Größenordnung $const \cdot G$, nicht G^k.
Wir können also die Produkte $\alpha(s)\beta(s)$ mit einer kleineren Multiplikation bestimmen und so einen rekursiven Multiplikationsalgorithmus bauen, wenn wir es schaffen, aus den Stützstellen $\gamma(s)$ relativ schnell wieder die Koeffizient von γ zu bestimmen.

[7] Andrei Leonovich Toom, 1942–2022, russischer Mathematiker
[8] Stephen Arthur Cook, geb. 1939, amerikanisch-kanadischer Mathematiker

3.2 Lemma (Vandermonde-Matrix)**:**
Evaluation an q Stützstellen $s_0, s_1, \ldots, s_q \in \mathbb{R}$ ist eine lineare Abbildung $\mathbb{R}[X]_{\leq d} \to \mathbb{R}^{q+1}$

$$\begin{pmatrix} \alpha(s_0) \\ \alpha(s_1) \\ \vdots \\ \alpha(s_q) \end{pmatrix} = \underbrace{\begin{pmatrix} s_0^0 & s_0^1 & \cdots & s_0^d \\ s_1^0 & s_1^1 & \cdots & s_1^d \\ \vdots & \vdots & \ddots & \vdots \\ s_q^0 & s_q^1 & \cdots & s_q^d \end{pmatrix}}_{=:V \in \mathbb{R}^{(q+1)\times(d+1)}} \cdot \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_d \end{pmatrix}$$

Für $q = d$ ist die Matrix V invertierbar.

Beweis. Übung. □

A4 Algorithmus (Toom-Cook-k)**:**
Parameter: $k \in \mathbb{N}_{\geq 2}$, paarweise verschiedene Stützstellen $\{ s_\ell \mid 0 \leq \ell < 2k-1 \} \subset \mathbb{Q}$.
Vorberechnung:

- Die Evaluationsmatrix $\mathcal{E} \in \mathbb{Q}^{(2k-1)\times k}$ mit $\mathcal{E}_{\ell,m} = s_\ell^m$ für $0 \leq \ell < 2k-1, 0 \leq m < k$.
- Die Interpolationsmatrix $\mathcal{I} \in \mathbb{Q}^{(2k-1)\times(2k-1)}$ mit $(\mathcal{I}^{-1})_{\ell,m} = s_\ell^m$ mit $0 \leq \ell, m < 2k$.

```
List<Digit> multiply(List<Digit> a, List<Digit> b) {
    n := max(a.length, b.length);
    if(n < CUTOFF){
        return slow_multiplication(a,b); // algorithms A1 and/or A3
    }
    n_prime := ceil(n/k); // n' is the size of the individual chunks

    for(int m:=0; m<k; m++) {
        // not an actual computation, only a naming convention
        alpha[m] := a[m*n_prime .. (m+1)n_prime-1];
        beta[m]  := b[m*n_prime .. (m+1)n_prime-1];
    }

    // Step 1: evaluate alpha and beta at the chosen points
    eval_alpha := new List<>(2k-1); // each list entry is itself
    eval_beta  := new List<>(2k-1); // a list of n'+0(1) digits

    // Matrix multiplication E*alpha and E*beta
    for(int l:=0; l<2k-1; l++) {
        for(int m:=0; m<k; m++) {
            // multiplications with small, known constants
            // => no recursion necessary here
            eval_alpha[l] := add(eval_alpha[l],
                                 multiply(E[l][m], alpha[m]));
            eval_beta[l]  := add(eval_beta[l],
                                 multiply(E[l][m], beta[m]));
```

```
        }
    }

    // Step 2: pointwise multiplication

    // each of the entries if itself a list of 2n'+O(1) digits
    eval_gamma := new List<>(2k-1);

    for(int l:=0; l<2k-1; l++) {
        // recursion here
        eval_gamma[l] := multiply(eval_alpha[l], eval_beta[l]);
    }

    // Step 3: recover coefficients of gamma
    gamma := new List<Digit>(2k-1);

    // Matrix multiplication I*gamma
    for(int l:=0; l<2k-1; l++) {
        for(int m:=0; m<2k-1; m++) {
            // multiplication with constant rationals, i.e.
            // multiplication and division with small, known constants
            // => no recursion necessary here
            gamma[l] := add(gamma[l],
                            multiply(I[l][m], eval_gamma[m]));
        }
    }

    // Step 4: evaluate gamma(g^k)
    result := new List<Digit>[2n];
    for(int l=0: l<2k-1; l++) {
        result[l*n_prime...] := add(result[l*n_prime...], gamma[l]);
    }

    return result;
}
```

3.3 Proposition:
Der obige Algorithmus ist korrekt, hat Laufzeit $O(n^e)$ mit $e := \frac{\log(2k-1)}{\log(k)}$ und Speicherbedarf $O(n)$.

Beweis. Analog zum Beweis für den Karatsuba-Algorithmus folgt dies im Wesentlichen aus dem Mastertheorem, da wir die Rekursion $T(n) = (2k-1)T(\frac{n}{k})+O(n)$ für die Laufzeit haben. Dabei benutzen wir, dass Multiplizieren mit kleinen Konstanten Laufzeit $O(n)$ hat.
Wir benötigen Speicher für `eval_alpha`, `eval_beta`, `eval_gamma`, `gamma` und `result`. Das sind viermal $2n+O(1)$ und einmal $2n$ Speicher, die wir benötigen. Das $O(1)$ kommt

jeweils davon, dass wir bei den diversen Additionen Platz für einen Übertrag bereitstellen müssen.
Insgesamt kommen wir also auf $10n + O(1)$ pro Rekursionsstufe. Insgesamt benötigen wir also

$$(10n + O(1)) \left(1 + \frac{1}{k} + \frac{1}{k^2} + \dots\right) = 10n\frac{k}{k-1} + O(\log(n))$$

Speicher. □

3.4: Wir benötigen $O(k^2)$ viele Multiplikationen/Divisionen mit kleinen Konstanten. Die O-Konstante in $O(n)$ für die Multiplikationen/Divisionen hängt aber empfindlich davon ab, welche Konstanten das genau sind. Auch wenn es für die asymptotische Laufzeit nicht relevant ist, muss man sich also doch etwas Mühe geben bei der Wahl der Sützstellen.

Besonders einfach ist es, Polynome in $X = 0$ auszuwerten, weil dafür gar keine Rechnung notwendig ist, also wählt man 0 immer als eine der Stützstellen. Ebenfalls kostenlos ist die „Auswertung bei $X = \infty$“. Damit meint man für ein Polynom vom Grad d die Berechnung von $\lim_{x\to\infty} \frac{p(x)}{x^d}$, was genau den führenden Koeffizienten des Polynoms ergibt. In unserer Notation also a_{k-1}, b_{k-1} bzw. γ_{2k-2}. Man kann sich leicht überlegen, dass $\gamma(\infty) = \alpha(\infty)\beta(\infty)$ ebenfalls erfüllt ist, sodass unsere Rechnung trotzdem funktionieren.

3.5: Für $k = 2$ erhält man Karatsubas Algorithmus als Spezialfall, indem man die Stützstellen $0, 1, \infty$ wählt. Das ergibt nämlich genau die Produkte

$$\gamma(0) = \alpha(0)\beta(0) = a_0 b_0$$
$$\gamma(1) = \alpha(1)\beta(1) = (a_1 1^1 + a_0 1^0)(b_1 1^1 + b_0 1^0)$$
$$\gamma(\infty) = \alpha(\infty)\beta(\infty) = a_1 b_1$$

die wir für Karatsubas Algorithmus benutzt haben.

3.6: Für $k = 3$ erhalten wir einen Algorithmus in $O(n^{\log(5)/\log(3)}) \subseteq O(n^{1.465})$.

3.7 (Große k)**:** Für $k \to \infty$ ist $\frac{\log(2k-1)}{\log(k)} \searrow 1$. Durch hinreichend groß gewähltes k können wir also für jedes $\varepsilon > 0$ einen Multiplikationsalgorithmus in $O(n^{1+\varepsilon})$ hinschreiben.

In der Praxis verwendet man aber keine großen Werte für k, da der Overhead extrem schnell anwächst, während der Break-even-point, ab dem die theoretische asymptotische Laufzeit die tatsächliche Laufzeit bestimmt, viel zu schnell wächst: Wenn wir k so groß genug wählen wollen, damit $\frac{\log(2k-1)}{\log(k)} \approx \frac{\log(2k)}{\log(k)} \leq 1 + \frac{1}{e}$ erfüllt ist, dann müssen wir in der Tat $k \geq 2^e$ wählen.
Wenn wir das aber tun, dann ist der Overhead für einen einzigen Rekursionsschritt in der Größenordnung $Ck^2n = C2^{2e}n$. Für extrem große n wird das zwar dominiert von $C'n^{1+1/e}$, aber tatsächlich geschieht das erst bei $n \geq \left(\frac{C}{C'}\right)^e 2^{2e^2}$.
Selbst wenn wir nur $O(n^{1.1})$ erreichen wollen (also $e = 10$), dann müssen wir $k = 2^{10}$ und Zahlen mit vielen hundert Bit betrachten, bevor wir überhaupt eine *Chance* haben,

dass der Toom-Cook-k-Algorithmus sich tatsächlich auszahlt. Mehr noch: Wenn $k = 2^{10}$ ist, dann sind die Potenzen s_ℓ^m, die wir für die Vandermonde-Matrix ausrechnen müssen, schon im Bereich von bis zu $10 \cdot (2k-1) \approx 20000$ bits groß. Das stellt die Behauptung in Frage, wie „klein“ die kleinen Konstanten eigentlich sind, die da in den drei Matrix-Vektor-Multiplikationen vorkommen. Wir müssen also in Wahrheit eher zehntausende Bits, vielleicht sogar hunderttausende Bits große Zahlen betrachten, bevor sich dieser Algorithmus praktisch auszahlt.

3.8: Nichts desto trotz ist diese Überlegung für die Theorie interessant. Wenn man für jedes k einen geeigneten Cutoff vorberechnet, dann kann man alle Toom-Cook-Algorithmus zu einem gemeinsamen Algorithmus kombinieren, der für hinreichend große Zahlen Toom-Cook-k mit immer größer werdendem k verwendet. Dieser Algorithmus ist dann offenbar asymptotisch schneller als jeder einzelne Toom-Cook-Algorithmus, also in $o(n^{1+\varepsilon})$ für jedes $\varepsilon > 0$.

Es ist nicht genau bekannt, welche Komplexitätsklasse man mit diesem Ansatz erreichen kann, aber eine Implementierung von Knuth[9] erzielt eine Komplexität von $O(n2^{\sqrt{2\log_2(n)}}\log(n))$, aber es ist nicht bekannt, ob das die beste Implementierung ist.

3.9: Wegen des schnell zunehmenden Overheads werden nur sehr kleine k praktisch eingesetzt und selbst dann nur für sehr große n.

Die[10] Java Implementierung wählt z.B. $n = 240$ als Cutoff, d.h. erst wenn beide Faktoren mehr als 240 Worte groß sind, wird die Toom-Cook-3-Multiplikation gegenüber der Karatsuba-Methode bevorzugt. Ein höheres k wird gar nicht verwendet in der Implementierung.

GMP verwendet $k \leq 8$ in gewissen Grenzen und den (asymptotisch noch schnelleren) Schönhage-Strassen-Algorithmus darüber.

[9] Donald Ervin Knuth, geb. 1938, amerikanischer Mathematiker und Informatiker, Autor der „The Art of Computer Programming“ Buchreihe.

[10] Wieder OpenJDK 21 in diesem Fall; andere Implementierungen der Standardbibliothek könnten theoretisch davon abweichen

Aufgaben

Aufgabe 3.1. – Vandermonde-Determinante (mittel)
Beweise, dass die Vandermonde[11]-Matrix

$$V := \begin{pmatrix} s_0^0 & s_0^1 & \cdots & s_0^d \\ s_1^0 & s_1^1 & \cdots & s_1^d \\ \vdots & \vdots & \ddots & \vdots \\ s_d^0 & s_d^1 & \cdots & s_d^d \end{pmatrix}$$

die Determinante

$$\det(V) = \prod_{0 \le i < j \le d+1} (s_j - s_i)$$

hat. Insbesondere ist V genau dann invertierbar, wenn die s_i paarweise verschieden sind.

Aufgabe 3.2. (leicht)
Bestimme die Determinante der analogen Matrix für die Stützstellen $s_0, s_1, \ldots, s_{d-1}, \infty$.

Aufgabe 3.3.
Verallgemeinere den Toom-Cook-k-Algorithmus zum Toom-Cook-(k_1, k_2)-Algorithmus, der A in k_1 und B in k_2 kleinere Stücke zerlegt.

3.10: Der Toom-Cook-$(2, 3)$-Algorithmus wird manchmal auch als Toom-Cook-2.5 bezeichnet.

[11] Alexandre-Théophile Vandermonde, 1735–1796, französischer Mathematiker, Musiker und Chemiker

4 Level 3: Multiplizieren mittels Fourier-Transformation, Schönhage-Strassen & Co

4.1 Idee

4.1: Wie wir im Zusammenhang mit den Toom-Cook-Algorithmen gesehen haben, kann man Multiplikation großer Zahlen auf Multiplikation von ganzzahligen Polynomen mit kleinen Koeffizienten zurückführen. Und je höher wir den Grad der Polynome wählen, desto effizienter können wir sein, zumindest asymptotisch. Wir haben aber auch gesehen, dass das in Praxis nicht tauglich ist, weil der Overhead quadratisch mit dem Grad der Polynome zu wachsen scheint. Wir können also nicht einfach Grad=Anzahl der Ziffern wählen, weil wir sonst wieder bei einem $O(n^2)$-Algorithmus wären.

Ist das also das Ende? Nur, wenn Auswertung eines Polynoms vom Grad $< m$ in m Stützstellen sowie die Interpolation von m Werten zurück zu den Polynomkoeffizienten in $\Theta(m^2)$ sind. Aber wer sagt denn, dass quadratisch die bestmögliche Laufzeit ist?
In der Tat können wir sehr viel besser auswerten und interpolieren, da wir ja die völlig freie Wahl der Stützstellen haben. Es stellt sich heraus, dass Einheitswurzeln eine besonders eine effiziente Wahl sind, weil die Vandermonde-Matrix dann besonders viel Struktur hat, die man ausnutzen kann, um Rechnungen einzusparen. Das ist die Idee hinter der schnellen Fourier-Transformation (FFT).

4.2 Fast Fourier Transform

4.2 Definition (DFT)**:**
Gegeben eine endliche Folge von Koeffizienten $a := (a_0, a_1, \ldots, a_{n-1}) \in \mathbb{C}^n$ heißt die Folge $\hat{a} := (\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{n-1}) \in \mathbb{C}^n$ der Zahlen gegeben durch

$$\hat{a}_i := \sum_{j=0}^{n-1} a_i \zeta_n^{-ij}$$

die diskrete Fourier-Transformation, kurz DFT, von a, wobei $\zeta_n \in \mathbb{C}$ eine primitive n-te Einheitswurzel ist, d.h. $\zeta_n^n = 1$ und $\zeta_n^i \neq 1$ für $i < n$, beispielsweise $e^{2\pi i/n}$.

Das umgekehrte Problem, aus $\hat{a}$ wieder a zu rekonstruieren, heißt entsprechend inverse DFT.

4.3: Die DFT ist allgemein für unstrukturierte Folgen von Zahlen definiert, weil sie in vielen verschiedenen Zusammenhängen auftritt, in denen diese Zahlenfolgen unterschiedliche Bedeutung und Herkunft haben.
Für unser Ziel ist es aber zweckmäßig, die Zahlen a als Koeffizienten eines Polynoms

$$\alpha(X) := a_0 + a_1 X + \ldots + a_{n-1} X^{n-1} \in \mathbb{C}[X]$$

aufzufassen. Dann ist nämlich $\hat{a}_i = \alpha(\zeta_n^{-i})$.

4.4 Lemma:
Es stellt sich heraus, dass die iDFT im Wesentlichen genauso aussieht wie die DFT selbst. Mit den Bezeichnungen von oben gilt:

$$a_i = \frac{1}{n} \sum_{k=0}^{n-1} \hat{a}_k \zeta_n^{ik}$$

Beweis. Definitionsgeschubse. □

4.5: Eben weil es so viele verschiedene Arten gibt, in denen die DFT auftreten kann, gibt es verschiedene Konventionen, die Definition zu schreiben, die aber im Wesentlichen äquivalent sind.
So kann man z.B. das andere Vorzeichen wählen und stattdessen α in ζ_n^{+i} auszuwerten. Da $\{1, \zeta_n, \zeta_n^2, \dots\} = \{1, \zeta_n^{-1}, \zeta_n^{-2}, \dots\}$ ist, führt das nur dazu, dass die Komponenten von $\hat{a}$ in einer anderen Reihenfolge auftreten, aber ansonsten die gleichen Werte haben.
Manchmal wird der Faktor $\frac{1}{n}$ auch für die DFT anstelle der iDFT verwendet. Beides zusammen führt im Wesentlichen dazu, dass das, was wir jetzt als iDFT bezeichnen, zur DFT erklärt wird und umgekehrt. Da Hin- und Rückrichtung aber sowieso zum Verwechseln ähnlich ist, macht das auch keinen großen Unterschied mehr, welche von den beiden Operationen man als DFT und welche man als iDFT bezeichent.
Manchmal wird der Faktor einfach gleichmäßig verteilt, sodass in beiden Formeln ein $\frac{1}{\sqrt{n}}$ vorkommt.

4.6 Satz (Gauss, Cooley-Tukey uvm.)**:**
Ist n eine Zweierpotenz, so ist möglich, die DFT eines Vektors $a \in \mathbb{C}^n$ mit $n \log(n)$ komplexen Additionen und Multiplikationen und $n + O(1)$ Speicherbedarf zu berechnen.

4.7: Dieser Algorithmus wurde mehrfach wiederentdeckt. Die zugrundeliegende Idee war bereits Gauss bekannt, aber Gauss fand sie nicht wichtig genug, um sie zu veröffentlichen. Der Algorithmus wurde nach dem zweiten Weltkrieg mehrfach wiederentdeckt, aber auch dann ein paar Mal ohne die Erkenntnis, dass die geringe Komplexität etwas Besonderes und Veröffentlichungswürdiges ist.

4.8: Die Annahme, dass n eine Zweierpotenz ist, ist in der Tat überflüssig. Sie ist nur für den konkreten Algorithmus nötig, den wir jetzt besprechen wollen. Es gibt andere Algorithmen, die für andere n funktionieren und ebenfalls in $O(n \log(n))$ sind.

Beweis. Wir werden die DFT als Auswertung des Polynoms α in den n Punkten

$$\zeta_n^0, \zeta_n^1, \zeta_n^2 \dots, \zeta_n^{n-1}$$

betrachten.
Die erste nützliche Beobachtung ist, dass wir α hier in Wirklichkeit als Element von $\mathbb{C}[X]/(X^n - 1)$ auffassen können, da die Auswertung in n-ten Einheitswurzeln eh keinen Unterschied zwischen $X^n - 1$ und 0 feststellen kann, da ja $\zeta^n - 1 = 0$ ist. Der Ring

$\mathbb{C}[X]/(X^n - 1)$ wird nun mittels des chinesischen Restsatzes rekursiv in kleinere Ringe (jeweils mit halber Dimension) zerlegt:
Dabei benutzen wir in jedem Schritt, dass $X^{2k} - z^2 = (X^k - z)(X^k + z)$ gilt und das eine Zerlegung in paarweise teilerfremde Faktoren ist (wenn $z \neq 0$), sodass wir den chinesischen Restsatz tatsächlich anwenden können.
Da n eine Zweierpotenz, ζ_n eine n-te Einheitswurzel und $1 = \zeta_n^n$ ist, kann man per Induktion zeigen, dass man in jedem Schritt Faktoren der Form $X^k - \zeta_n^{ak}$ erhält.

Am Beispiel von $n = 8$ sieht das so aus wie in 1 dargestellt.
Man beachte, dass wir am Ende die Reduktionen auf die eindimensionalen Quotienten $\mathbb{C}[X]/(X - \zeta_n^i)$ erhalten, also die Auswertung im Punkt ζ_n^i, also genau die i-te Komponente der DFT.

Die Laufzeit dieses Algorithmus kommt daher, dass die Reduktion von $\mathbb{C}[X]/(X^{2k} - z^2)$ zu $\mathbb{C}[X]/(X^k - z)$ mit k komplexen Multiplikationen und Additionen erledigt werden kann:
Wir schreiben eine Restklasse $[f] \in \mathbb{C}[X]/(X^{2k} - z^2)$ als Polynom vom Grad $\leq 2k - 1$:

$$f \equiv f_0 + f_1 X + f_2 X^2 + \dots f_{2k-1} X^{2k-1} \mod X^{2k} - z^2$$

und gruppieren jetzt nach Koeffizienten kleiner/größergleich k:

$$\begin{aligned} f \equiv &(f_0 &&+ f_1 X + \dots + f_{k-1} X^{k-1}) + \\ &(f_k &&+ f_{k+1} X + \dots + f_{2k-1} X^{k-1}) X^k \mod X^{2k} - z^2 \end{aligned}$$

Da wir dieses Polynom nun modulo $X^k - z$ reduzieren wollen, können wir X^k durch die Konstante z ersetzen:

$$f \equiv (f_0 + f_k z) + (f_1 + f_{k+1} z) X^1 + (f_2 + f_{k+2} z) X^2 + \dots (f_k + f_{k-1} z) X^{k-1} \mod X^k - z$$

Das sind also genau k komplexe Multiplikationen und k Additionen, die wie für diese Reduktion benötigen.

In jedem Reduktionsschritt benötigen wir das zweimal: Wir reduzieren von $\mathbb{C}[X]/(X^{2k} - z^2) = \mathbb{C}[X]/(X^{2k} - (-z)^2)$ zu $\mathbb{C}[X]/(X^k - z)$ und zu $\mathbb{C}[X]/(X^k - (-z))$. Das machen wir für mehrere k, sodass wir am Ende bei genau n Additionen bzw. Multiplikationen in jeder Ebene der Rekursion landen.

Dass der Speicherbedarf $O(n)$ ist, folgt daraus, dass wir in jedem Reduktionsschritt die Anzahl der Faktoren verdoppeln, aber jeder Faktor nur noch halb so groß ist. Im Beispiel $n = 8$ beginnen wir etwa mit einem Vektor der Länge 8, reduzieren auf zwei Vektoren der Länge 4, reduzieren auf vier Vektoren der Länge 2 und schließlich acht Vektoren der Länge 1. Für jede neue Ebene der Rekursion benötigen wir nur Speicher für genau n neue komplexe Zahlen.
Wenn wir die n Zahlen der vorherigen Level wegwerfen, nachdem wir sie nicht mehr benötigen, kommen wir mit Speicher für $2n$ Zahlen aus. Wenn wir die Formel genau angucken, sehen wir, dass wir immer die i-ten Koeffizienten der beiden Reduktionen aus dem i-ten und $(i + k/2)$-ten Koeffizient des Inputs berechnen. Das heißt, dass wir den Input einfach direkt mit dem Ergebnis überschreiben können und mit $n + O(1)$ Speicher auskommen. □

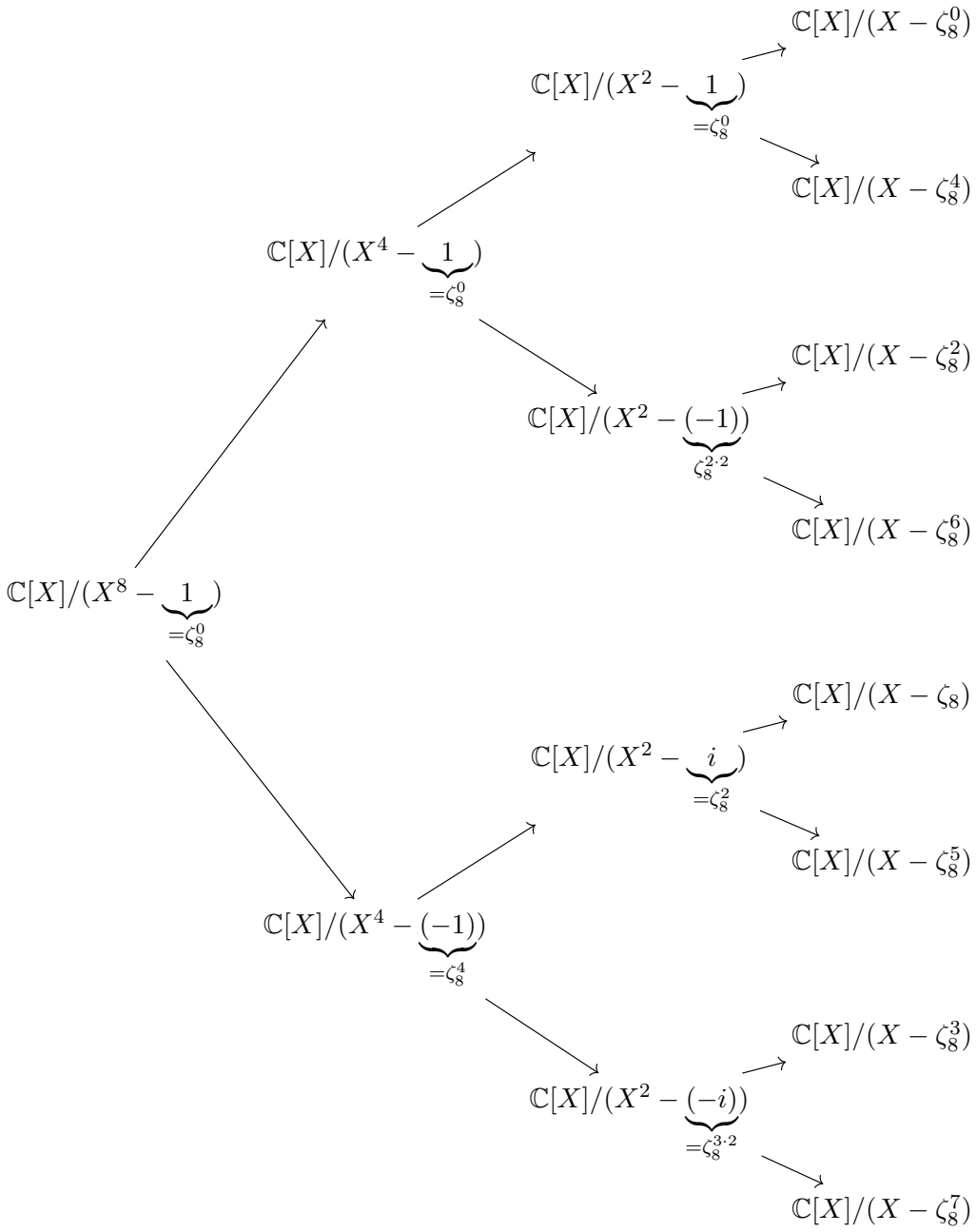

Abbildung 1: FFT am Beispiel $n = 8$

4.3 Der Schönhage-Strassen-Algorithmus

4.9 Korollar:
Die Multiplikation zweier Zahlen $A, B \in \mathbb{Z}$ mit n Ziffern kann mit $O(n \log(n))$ arithmetischen Operationen auf komplexen Zahlen und Speicher für $O(n)$ komplexe Zahlen berechnet werden.

Beweis. Mittels FFT ist es jetzt klar wie es geht: Wir nutzen die Ziffern von A und B als Koeffizienten von Polynomen

$$\alpha(X) = a_0 + a_1 X^1 + \dots a_{n-1} X^{n-1}$$
$$\beta(X) = b_0 + b_1 X^1 + \dots b_{n-1} X^{n-1}$$

Berechnen $\alpha(\zeta_{2n}^i)$ und $\beta(\zeta_{2n}^i)$ für alle $i = 0, 1, \dots, 2n-1$ in $O(n \log(n))$ mittels FFT. $\gamma(\zeta_n^i) = \alpha(\zeta_n^i) \cdot \beta(\zeta_n^i)$ auszurechnen, kostet noch einmal $2n$ Multiplikationen. Die inverse Fourier-Transformation wird ebenfalls mit FFT in $O(n \log(n))$ ausgerechnet.
Damit haben wir die Koeffizienten von $\gamma(X)$. Man beachte, dass die Koeffizienten von γ exakt ganzzahlig sind, auch wenn die Zwischenergebnisse komplexe Zahlen waren!
Durch Einsetzen von unserer Zahlenbasis $X = g$ erhalten wir das Produkt $A \cdot B = C = \gamma(g)$. Das geht in $O(n)$ elementaren Operationen auf den Ziffern. □

4.10: Haben wir damit das Problem gelöst? Nicht ganz, denn niemand sagt uns, dass „arithmetische Operationen auf komplexen Zahlen“ genauso billig sind wie Operationen auf einzelnen Ziffern, was ja eigentlich unsere Maßeinheit ist.

Da wir a priori wissen, dass die Koeffizienten $\gamma_0, \gamma_1, \dots, \gamma_{2n-1}$, die wir berechnen wollen, ganze Zahlen sind, können wir das Rechnen mit unendlicher Präzision durch eine Rechnung mit hinreichend langen Festkomma-Zahlen ersetzen. Die Länge der Festkomma-Zahlen muss so bestimmt werden, dass das Endergebnis einen Fehler $< \frac{1}{2}$ hat, sodass wir durch simples Runden zur nächsten ganzen Zahl das Ergebnis bekommen.
Sobald wir uns erst einmal auf diese Länge festgelegt haben, ist das Rechnen mit Festkomma-Zahlen wieder nur die Addition und Multiplikation von ganzen Zahlen dieser Länge. Solange wir feststellen, dass die notwendige Anzahl an Ziffern für die FFT und iFFT der Länge n signifikant kleiner als n ist, können wir so einen rekursiven Multiplikationsalgorithmus konstruieren.

Schönhage[12] und Strassen[13] haben genau solche eine Analyse durchgeführt und sind zu folgendem Ergebnis gekommen.

4.11 Satz (Schönhage-Strassen I)**:**
Multiplikation von zwei n-Bit Zahlen ist in $O(n \log(n)^{1+\varepsilon})$ elementaren arithmetischen Operationen auf Ziffern möglich. Präziser kann man für jedes $k \in \mathbb{N}$ die Komplexität

$$O(n \cdot \log(n) \cdot \log(\log(n)) \cdots (\log^k(n))^2)$$

erreichen.

[12] Arnold Schönhage, dt. Mathematiker, geb. 1934
[13] Volker Strassen, dt. Mathematiker, geb. 1936

4.12: Der FFT-Algorithmus, den wir betrachtet haben, ist nicht fundamental davon abhängig, dass mit komplexen Zahlen gerechnet wird. Man kann sich leicht davon überzeugen, dass man $\mathbb{C}$ durch jeden kommutativen Ring R ersetzen kann, der

a.) eine primitive $2n$-te Einheitswurzel ζ enthält.

b.) ein multiplikatives Inverses von n enthält.

Wenn wir eine Zweierpotenz $n = 2^\ell$ verwenden wollen, ist das z.B. vom Restklassenring $\mathbb{Z}/(2^{2^\ell}+1)$ erfüllt, denn $\zeta := 2$ erfüllt per Konstruktion $\zeta^{2^\ell} = -1$ und ist somit eine $2^{2\ell}$-te primitive Einheitswurzel. Die Restklassen repräsentiert man natürlicherweise durch ganze Zahlen zwischen 0 und 2^{2^ℓ}, sodass man keine komplexe Arithmetik benötigt, sondern nur ganzzahlige Arithmetik[14].
Natürlich ist $2^{2^\ell} = 2^n$ wieder in der Größenordnung, in der unsere Inputs A und B waren, denn die hatten ja n Ziffern. Wenn wir $g = 2^{32}$ verwenden, dann hat 2^n zwar trotzdem weniger Bits als g^n, aber nur um einen konstanten Faktor weniger, sodass wir diesen Ansatz rekursiv anwenden müssen, wenn wir Arithmetik in $\mathbb{Z}/(2^{2^\ell}+1)$ verwenden wollen.
Wenn wir sowieso rekursiv arbeiten, dann hindert uns aber niemand daran, wie bei den vorherigen Algorithmen eine passende Zwischengröße G zu wählen, sodass unsere Rekursionsschritte mit einem deutlich kleineren n (und damit einem deutlich kleineren ℓ) zufrieden sind.

All diese Überlegungen haben Schönhage und Strassen auch durchgeführt und haben so ihr vorheriges Ergebnis wie folgt verbessert:

4.13 Satz (Schönhage-Strassen II)**:**
Multiplikation von zwei n-Bit Zahlen ist in $O(n\log(n)\cdot\log(\log(n)))$ elementaren arithmetischen Operationen auf Ziffern möglich.

4.14: Beliebige Zahlen n sind natürlich keine Zweierpotenzen. Wir können zwar o.B.d.A. annehmen, dass n eine Zweierpotenz ist, indem wir den Zahlen A und B ggf. einfach ein paar Nullen voranstellen, aber im Extremfall verdoppelt das die Länge der Zahlen.
Für die asymptotische Komplexität ist ein Faktor 2 nicht relevant, für die Praxis aber schon, da der Schönhage-Strassen-Algorithmus ja sowieso erst eingesetzt wird, wenn die Inputs sehr groß sind. [15]
Wenn aber alleine unsere Inputs schon Kilo- und Megabytes groß sind, dann ist selbst ein konstanter Faktor im Speicherbedarf störend.

4.4 $O(n\log(n))$

4.15: Natürlich stellt sich die unmittelbare Frage, wie weit man dieses Spiel treiben kann und was das tatsächliche Optimum für Multiplikation ist.
Schönhage & Strassen stellten folgende Vermutung auf

[14] Man beachte, dass wegen $2^{2^\ell} \equiv -1$ die notwendigen Divisionen-mit-Rest durch Additionen, Subtraktionen und Bitshifts realisiert werden können
[15] Cutoff in GMP liegt bei mehreren tausend Wörtern abhängig von der verwendeten Hardware

4.16 Vermutung (Schönhage-Strassen)**:**
Multiplikation von n-Bit Zahlen ist in $\Theta(n \log(n))$ möglich.

4.17 Satz (Harvey-Hoeven, 2019)**:**
Die Schönhage-Strassen-Vermutung ist wahr.

4.18: Der Beweis ist komplex. Er kombiniert geschickt verschiedene Ideen, um den FFT-Ansatz zu optimieren.

4.19: Eine grundlegende Idee ist, dass eine FFT der Länge $n_1 n_2$ durch eine $n_1 \times n_2$ zweidimensionale FFT (oder noch mehr Faktoren) zu ersetzen. Eine zweidimensionale FT von (a_{ij}) ist definiert durch

$$\hat{a}_{ij} := \sum_{k=0}^{n_1-1} \sum_{l=0}^{n_2-1} a_{kl} \zeta_{n_1}^{-ki} \zeta_{n_2}^{-lj}$$

Analog sind multidimensionale FTs definiert.[16] Der eindimensionale FFT-Algorithmus überträgt sich in natürlicher Weise auf den multidimensionalen Fall.

Für unseren Ansatz, FFT zu benutzen um Polynome zu multiplizieren, bedeutet das, dass man nicht nur das g in

$$A = a_0 + a_1 g + a_2 g^2 + \dots$$

durch eine Unbekannte X ersetzt, sondern dass man zusätzlich g^{n_1} durch eine weitere Unbekannte Y ersetzt. Jeder Exponenten $e < n_1 n_2$ kann eindeutig als $e' + e'' n_1$ mit $e' < n_1, e'' < n_2$ geschrieben werden und so wird g^e durch $X^{e'} Y^{e''}$ ersetzt. So erhält man statt eines Polynoms in einer Variablen ein Polynom in zwei (oder noch mehr) Variablen, dessen Exponenten durch n_1 bzw. n_2 beschränkt sind.

Natürlich ist nicht jede Zahl n nichttrivial als $n_1 n_2$ faktorisierbar. Da wir aber jederzeit A und B eine Null voranstellen können, können wir jederzeit zu $n+1$ übergehen und so eine zusammengesetzte Zahl erhalten, wenn n zufällig eine Primzahl war. Wenn wir uns wenige (!) Nullen mehr erlauben, können wir $n+k$ so wählen, dass möglichst viele möglichst kleine Faktoren vorkommen und somit maximal zerlegt werden kann.

Eine zweidimensionale $n_1 \times n_2$ FFT benötigt zwar trotzdem $O(n \log(n))$ Operationen, aber da wir nur n_1-te und n_2-te Einheitswurzeln benötigen anstatt n-te Einheitswurzeln, können wir mit deutlich kleineren Restklassenringen arbeiten und einschränken, wie oft der zu findende Multiplikationsalgorithmus rekursiv angewendet werden muss, bis die auftretenden Koeffizienten klein genug sind.

[16] Wer's kennt: Abstrakt formuliert heißt das nur, dass die $n_1 \times n_2$-FT das Tensorprodukt der n_1-FT und der n_2-FT ist. Aus dem Grund übertragen sich alle Algorithem für 1D-Fouriertransformationen auch auf multidimensionale FTs.

4.20: Speziell für Primzahlen gibt es außerdem „Rader's trick“, der es erlaubt eine FFT der Länge p auf eine FFT der Länge $p-1$ zurückzuführen. Dies ermöglicht weitere Freiheitsgrade bei der Wahl möglichst gut teilbarer Zahlen.

4.21: Wenn man in der Situation ist, eine multidimensionale $n_1 \times n_2 \times \cdots \times n_d$ FFT mit Zweierpotenzen $n_1 \leq n_2 \leq \ldots \leq n_d$ durchzuführen, dann kann man in jeder Dimension außer der letzten die FFT in $O(n_i)$ statt $O(n_i \log(n_i))$ durchführen, indem man anstatt einer komplexen Zahl (oder einem Element eines Restklassenrings) als Einheitswurzel die Unbekannte $X_d \in R[X_d]/(X_d^{n_d}-1)$ verwendet, die ja ebenfalls eine n_d-te Einheitswurzel ist. Das Multiplizieren mit einer Unbekannten ist in Polynomringen völlig gratis, da es nur die Koeffizienten herum permutiert.
Wenn wir die n_i alle in der Größenordnung $n^{1/d}$ wählen können, dann reduziert das den Aufwand für die FFTs von $\approx n\log(n)$ zu $\approx \frac{n\log(n)}{d}$.

4.22: Harvey und Hoeven konnten zeigen, dass es immer genügend viele Zahlen „in der Nähe von n“ gibt, dass man genügend schnell zu einer Zahl mit sehr vielen sehr kleinen Faktoren kommen kann und diese Faktoren klein genug sind.

4.23: Die Details sind sehr alle technisch und der Algorithmus, der dabei herauskommt, ist so komplex und hat so viel Overhead, dass er nur auf dem Papier interessant ist. Der Overhead ist so gigantisch, dass kein praktisches Problem groß genug ist, damit der Harvey-Hoeven-Algorithmus den Schönhage-Strassen-Algorithmus tatsächlich schlägt.

Harvey und Hoeven selber schätzen, dass man Zahlen mit $\approx 10^{14}$ Ziffern benötigt, bevor sich ihr Algorithmus auszahlt.

Während Schönhage-Strassen tatsächlich in Bibliotheken wie GMP implementiert ist und auch praktisch verwendet wird, ist mir (zum jetzigen Zeitpunkt) keine einzige Implementierung von Harvey-Hoeven bekannt.

99 Anhang: Gruppen & Ringe

99.1 Gruppen

99.1 Definition (Gruppen)**:**
Eine Gruppe $(G, *)$ besteht aus

- einer Menge G und
- einer Abbildung $* : G \times G, (g, h) \mapsto g \cdot h$, genannt Multiplikation,

die die „Gruppen-Axiome“ erfüllen:

(G1) Assoziativität: $\forall x, y, z \in G : x * (y * z) = (x * y) * z$.

(G2) Neutrales Element: $\exists e \in G \forall x \in G : x * e = e * x = x$.

(G3) Inverse Elemente: $\forall x \in G \exists x' \in G : x * x' = e = x' * x$.

Die Gruppe heißt abelsch[17] oder kommutativ, falls zusätzlich dazu die folgende Bedingung erfüllt ist:

(G4) Kommutativit: $\forall x, y, \in G : x * y = y * x$.

Der Kürze halber schreibt man oft nur G statt $(G, *)$, falls klar ist, welche Operation $*$ gemeint ist.

99.2: Es gibt zwei Klassen von Gruppen. Die einen sind Gruppen, bei denen man traditionell eine multiplikative Schreibweise wählt, d.h. der Name der Gruppenverknüpfung $*$ erinnert an ein Multiplikationssymbol, etwa $*$, $\cdot$, $\otimes$, $\circ$ etc. Oder sogar, wie bei der gewöhnlichen Multiplikation: Man lässt das Symbol ganz weg und schreibt nur noch xy. Praktisch alle nichtabelschen Gruppen schreibt man multiplikativ. In dieser Schreibweise nennt man das (es gibt wirklich nur eines, siehe Lemma 99.5.a) neutrale Element dann 1 statt e und das (ebenfalls eindeutige, siehe Lemma 99.5.b) von x nennt man x^{-1} statt x'.
Die zweite Klasse sind Gruppen, die man traditionell additiv schreibt, d.h. der Name der Gruppenverknüpfung erinnert an ein Additionssymbol, etwa $+$ oder $\oplus$. Dies wird fast ausschließlich bei abelschen Gruppen angewandt. In dieser Schreibweise nennt man das neutrale Element dann 0 statt e und das inverse Element von x nennt man $-x$ statt x'. Es gibt aber natürlich keinen inhaltlichen Unterschied zwischen diesen Schreibweisen. Die Wahl, wie wir etwas aufschreiben, ist ja nur eine Frage der Ästhetik, sie hat keine inhaltlichen Konsequenzen.

99.3 Beispiel: a.) $G = \mathbb{Z}$ mit $x * y = x + y$. Es ist $e = 0$, $x' = -x$. Diese Gruppe ist abelsch. Analog sind auch $(\mathbb{Q}, +)$, $(\mathbb{R}, +)$ und $(\mathbb{C}, +)$ abelsche Gruppen.

b.) $G = \mathbb{R}_{>0} = \{ x \in \mathbb{R} \mid x > 0 \}$ mit $x * y = xy$ (Multiplikation). Es ist $e = 1$, $x' = \frac{1}{x}$.

[17] Niels Hendrik Abel, norwegischer Mathematiker, 1802–1829

c.) Die triviale Gruppe $G = \{1\}$, mit $1 * 1 = 1$. Neutrales Element 1, und $1' = 1$.

d.) Die Menge $G = \{+1, -1\}$, mit Multiplikation. Es ist $e = 1$, $x' = \frac{1}{x} = x$. Dies ist eine endliche Gruppe: $|G| = 2$.

e.) Die Matrizengruppen $GL_n(K)$, $SL_n(K)$, $O_n(\mathbb{R})$, $SO_n(\mathbb{R})$, $U_n(\mathbb{C})$, $SU_n(\mathbb{C})$, ... sind wichtig für Geometrie und Physik:

$O_n(\mathbb{R})$ beschreibt Drehungen und Spiegelungen im n-dimensionalen Raum; $SO_n(\mathbb{R})$ beschreibt nur die Drehungen. $U_1(\mathbb{C})$ z.B. beschreibt die Phasenverschiebung zwischen zwei rotierenden Systemen; $SU_2(\mathbb{C})$ ist z.B. für die Beschreibung der elektromagnetischen Kraft zuständig und ist u.A. für das quantenphysikalische Phänomen des Spins verantwortlich; $SU_3(\mathbb{C})$ spielt in der Beschreibung der starken Kernkraft eine Rolle.

Andere Gruppen wie z.B. $O_{3,1}(\mathbb{R})$ kommen in der Relativitätstheorie vor (Sie erkennen drei Raum- und eine Zeitdimension). Diese Gruppe beschreibt sogenannte „Lorentz-Boosts".

Diese Gruppen sind fast alle nichtabelsch und unendlich.

99.4 Beispiel: a.) $(\mathbb{N}, +)$ ist hingegen keine Gruppe. (G1) und (G2) sind zwar erfüllt, aber (G3) nicht, denn nicht *jedes* Element hat ein inverses Element. 0 hat eines, 1 aber nicht, denn es gibt keine natürliche Zahl, die $n + 1 = 1 + n = 0$ erfüllt. (Es gibt eine ganze Zahl, die das erfüllt, aber keine natürliche)

b.) Aus ähnlichen Gründen ist auch $(\mathbb{R}, \cdot)$ keine Gruppe. Das Element $0 \in \mathbb{R}$ ist nicht invertierbar, denn keine reelle Zahl erfüllt $x \cdot 0 = 1$.

99.5 Lemma (Eindeutigkeit des neutralen Elements und der inversen Elemente)**:**
Sei $(G, *)$ eine Gruppe.

a.) Es gibt in G genau ein neutrales Element.

b.) Zu jedem $x \in G$ gibt es genau ein Inverses $x' \in G$. Dieses nennt man daher x^{-1}.

Beweis. Übung. □

99.6 Lemma und Definition (Potenzschreibweise)**:**
Zusätzlich zur Schreibweise g^{-1} für inverse Elemente führen wir allgemeiner Potenzen mit ganzzahligen Exponenten für alle Gruppenelemente ein.
Ist $(G, \cdot)$ eine (multiplikativ notierte) Gruppe und $g \in G$, so definieren wir g^k, indem wir

$$g^0 := 1 \quad g^{k+1} := g^k \cdot g \quad \text{und} \quad g^{k-1} := g^k \cdot g^{-1}$$

für alle $k \in \mathbb{Z}$ festlegen. Mit dieser Bezeichnung gilt dann:

a.) $\forall g \in G \forall n, m \in \mathbb{Z} : g^{n+m} = g^n \cdot g^m$

b.) $\forall g \in G \forall n, m \in \mathbb{Z} : g^{nm} = (g^n)^m$

Falls $(G, \cdot)$ eine abelsche Gruppe ist, dann gilt außerdem

c.) $\forall g_1, g_2 \in G \forall n \in \mathbb{Z} : (g_1 \cdot g_2)^n = g_1^n \cdot g_2^n$

Beweis. Übung. □

99.7: In additiv geschriebenen Gruppen benutzt man normalerweise nicht die Potenzschreibweise g^n, sondern die Schreibweise ng. Dann schreiben sich die drei Potenzgesetze aber ebenso wiedererkennbar einfach als

$$(n+m)g = ng + mg, \quad (nm)g = n(mg), \quad n(g_1 + g_2) = ng_1 + ng_2$$

99.2 Ringe

99.8 Definition:
Ein Ring $R = (R, +, \cdot, 0, 1)$ besteht aus einer Menge R zusammen mit Abbildungen $+ : R \times R \to R$, $(x, y) \mapsto x + y$, genannt „Addition“, und $\cdot : R \times R \to R$, $(x, y) \mapsto x \cdot y = xy$, genannt „Multiplikation“, sowie zwei Elementen $0, 1 \in R$, welche die folgenden Bedingungen erfüllen:

(R1) $(R, +, 0)$ ist eine abelsche Gruppe. Das neutrale Element 0 nennen wir Nullelement des Rings.

(R1) Assoziativität der Multiplikation:

$$\forall x, y, z \in R : (xy)z = x(yz)$$

(R3) Neutrales Element der Multiplikation:

$$\exists 1_R \in R \forall x \in R : x \cdot 1 = x = 1 \cdot x$$

Dieses Element nennen wir Einselement des Rings.

(R4) Distributivgesetze:

$$\forall x, y, z \in R : x(y + z) = xy + xz \wedge (x + y)z = xz + yz$$

Der Ring heißt kommutativ, falls zusätzlich

(R5) Kommutativität: $\forall x, y \in R : xy = yx$.

gilt.
Der Kürze halber schreibt man oft nur R statt $(R, +, \cdot, 0, 1)$, wenn aus dem Kontext klar ist, welche Addition und Multiplikation gemeint sind.

99.9 Beispiel: a.) $\mathbb{Z}$, $\mathbb{Q}$ und $\mathbb{R}$ sind Ringe bezüglich der üblichen Addition und Multiplikation.

b.) Der Nullring $R = \{0\}$ mit $0 + 0 = 0 \cdot 0 = 0$ ist ein Ring, mit $1_R = 0_R = 0$. In der Tat ist das der einzige Ring, in dem $0_R = 1_R$ gilt (Übung).

c.) Ist R ein Ring und $n \in \mathbb{N}$, so ist die Menge $R^{n\times n}$ aller $n \times n$-Matrizen über R ein Ring bezüglich Matrixaddition und -multiplikation (s. unten).

d.) Die Menge $R[X]$ aller Polynome mit Koeffizienten aus R ist ein Ring zusammen mit der üblichen Addition:

$$\sum_{i=0}^{n} a_i X^i + \sum_{i=0}^{n} b_i X^i := \sum_{i=0}^{n} (a_i + b_i) X^i$$

$$(\sum_{i=0}^{n} a_i X^i)(\sum_{j=0}^{m} b_j X^j) := \sum_{k=0}^{n+m} (\sum_{\substack{i,j \\ i+j=k}} a_i b_j) X^k$$

Ist R kommutativ, dann ist auch $R[X]$ ein kommutativer Ring.

99.10 Lemma:
Sei R ein Ring. Dann:

a.) $\forall x \in R : 0 \cdot x = x \cdot 0 = 0$.

b.) $\forall x, y \in R : x \cdot (-y) = (-x) \cdot y = -(xy)$.

c.) $(-1)^2 = 1$.

99.11 Lemma und Definition (Ideale und Quotienten)**:**
Es sei R ein Ring. Eine Teilmenge $I \subseteq R$ heißt Ideal, wenn

(I1) I ist eine Untergruppe von $(R, +)$, d.h.

$$0 \in I \wedge \forall x, y \in I : x + y \in I \wedge -x \in I$$

(I2) I ist unter Multiplikation mit beliebigen Elementen von R abgeschlossen:

$$\forall x \in I, y \in R : xy, yx \in I$$

Ist nun $I \trianglelefteq R$ ein Ideal, dann gilt:

a.) Die Relation

$$x \equiv_I y : \Longleftrightarrow\ x - y \in I$$

ist eine Äquivalenzrelation auf R

b.) Der Quotientenring R/I ist definiert als die Menge der Äquivalenzklassen zusammen mit folgender Addition und Multiplikation

$$[x] +_{R/I} [y] := [x +_R y], \quad [x] \cdot_{R/I} [y] := [x \cdot_R y]$$

Dies ist tatsächlich ein Ring.

99.12 Beispiel:
In $R = \mathbb{Z}$ sind die Ideale genau die Teilmengen $n\mathbb{Z} := \{ nk \mid k \in \mathbb{Z} \}$ (Übung). Die Quotienten $\mathbb{Z}/n\mathbb{Z}$ nennt man auch Restklassenringe.

99.13 Satz (Chinesischer Restsatz)**:**
Es sei R ein Ring und $I, J \subseteq R$ zwei Ideale. Die natürliche Abbildung

$$R \to R/I \times R/J, x \mapsto ([r]_{R/I}, [y]_{R/J})$$

ist

a.) injektiv genau dann, wenn $I \cap J = \{ 0 \}$, und

b.) surjektiv genau dann, wenn $I+J = R$ oder äquivalent, wenn $\exists i \in I, j \in J : i+j = 1$.

99.2.1 Matrizen

99.14 Definition:
Es sei R ein Ring und $n, m \in \mathbb{N}$. Eine $n \times m$-Matrix mit Einträgen aus R ist eine rechteckige Anordnung von nm Elementen von R in n Zeilen und m Spalten. Im Deutschen Sprachgebrauch ist es üblich, runde Klammern um eine Matrix zu schreiben.
Die Menge aller $n \times m$-Matrizen mit Einträgen aus R bezeichnen wir mit $R^{n \times m}$.
Ist A eine solche Matrix, so bezeichnet man mit A_{ij} den Eintrag an der Stelle (i, j), d.h. in der $i$ten Zeile und der $j$ten Spalte.
Eine Matrix heißt quadratisch, wenn die Anzahl der Zeilen und der Spalten gleich sind.

99.15 Beispiel:
Hier ist eine (3×2)-Matrix mit Einträgen aus $\mathbb{R}$: $\begin{pmatrix} 1 & 2 \\ 4 & -7 \\ 0,5 & \pi \end{pmatrix}$.

99.16 Beispiel:
Eine 1×0-Matrix: ().

99.17 Beispiel:
Für $A = \begin{pmatrix} 1 & 3 & 7 & 4 \\ 3 & 1 & 2 & 9 \\ 8 & 0 & 7 & 3 \end{pmatrix}$ ist $A_{23} = 2$.

99.18 Definition (Matrixaddition und -multiplikation)**:**
Für Matrizen $A, B \in R^{m \times n}$ wird die Summe $A + B \in R^{m \times n}$ definiert durch

$$(A + B)_{ij} := A_{ij} + B_{ij}$$

für alle $1 \leq i \leq m, 1 \leq j \leq n$.

Sind $A \in R^{m \times n}$ und $B \in R^{n \times p}$ Matrizen, so wird das Produkt $AB \in R^{m \times p}$ definiert durch

$$(AB)_{ik} := A_{i1}B_{1k} + A_{i2}B_{2k} + A_{i3}B_{3k} + \cdots + A_{in}B_{nk} = \sum_{j=1}^{n} A_{ij}B_{jk}$$

für alle $1 \leq i \leq m, 1 \leq j \leq n$.

Ist $A \in R^{m \times n}$ eine Matrix und $\lambda \in R$ ein Skalar, so ist $\lambda A \in R^{m \times n}$ definiert durch

$$(\lambda A)_{ij} := \lambda A_{ij}$$

Die Nullmatrix $0 = 0_{n \times m} \in R^{n \times m}$ ist definiert durch

$$(0_{n \times m})_{ij} := 0$$

Die Einheitsmatrix $1_{n \times n}$ ist die $n \times n$-Matrix, deren Eintrag an der Stelle (i, j) genau

$$\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{sonst} \end{cases}$$

ist.

99.19 Beispiel:
$\begin{pmatrix} 1 & 2 & 4 \\ 2 & 5 & 7 \end{pmatrix} + \begin{pmatrix} 0 & 3 & 1 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 5 & 5 \\ 3 & 6 & 8 \end{pmatrix}$.

99.20 Beispiel:

$$\begin{pmatrix} 1 & 0 \\ 3 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \end{pmatrix} = \begin{pmatrix} 1 \cdot 0 + 0 \cdot 1 & 1 \cdot 2 + 0 \cdot 3 & 1 \cdot 4 + 0 \cdot 5 \\ 3 \cdot 0 + 1 \cdot 1 & 3 \cdot 2 + 1 \cdot 3 & 3 \cdot 4 + 1 \cdot 5 \end{pmatrix} = \begin{pmatrix} 0 & 2 & 4 \\ 1 & 9 & 17 \end{pmatrix}$$

99.21: Beachten Sie: das Produkt AB ist nur dann definiert, wenn A die gleiche Anzahl von Spalten hat, wie B Zeilen hat.

99.22 Lemma (Ringeigenschaften)**:**
Matrizenringe $R^{n\times n}$ sind wirklich Ringe. Allgemeiner gilt selbst für rechteckige Matrizen:

a.) $(R^{n\times m}, +, 0)$ ist eine abelsche Gruppe.

b.) Matrixmultiplikation ist assoziativ:

$$\forall A \in R^{n\times m}, B \in R^{m\times p}, C \in R^{p\times q} : (AB)C = A(BC)$$

c.) Matrizenmultiplikation ist distributiv:

$$\forall A \in R^{n\times m}, B, C \in R^{m\times p} : A(B + C) = AB + AC$$

$$\forall A; B \in R^{n\times m}, C \in R^{m\times p} : (A + B)C = AC + BC$$

d.) Einheitsmatrizen sind Einselemente:

$$\forall A \in R^{n\times m} : 1_{n\times n} \cdot A = A = A \cdot 1_{m\times m}$$

e.) Multiplikation mit Skalaren ist assoziativ:

$$\forall A \in R^{n\times m}, B \in R^{m\times p} \forall \lambda \in R : (\lambda A)B = \lambda(AB)$$

sowie, wenn R kommutativ ist

$$\forall A \in R^{n\times m}, B \in R^{m\times p} \forall \lambda \in R : \lambda(AB) = A(\lambda B)$$

Beweis. Übung. □

99.23 Beispiel:
Schon $R^{2\times 2}$ und alle größeren Matrizenringe sind nicht kommutativ, selbst wenn R noch kommutativ war, denn

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \quad \text{aber} \quad \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}.$$

Aufgaben

Aufgabe 99.1. – Gruppen
Beweise alle Behauptungen in Abschnitt

Aufgabe 99.2. – Ringe
Beweise alle Behauptungen in Abschnitt

Aufgabe 99.3. – Quotienten von Polynomringen I
Es sei R ein kommutativer Ring. Betrachte den Polynomring $R[X]$ in einer Unbekannten und ein normiertes Polynom $p(X) = X^n + p_{n-1}X^{n-1} + \ldots + p_1 x + p_0 \in R[X]$ vom Grad n. Betrachte das von p erzeugte Hauptideal $(p) := pR[X] := \{ q \cdot p \mid q \in R[X] \}$.

a.) Jede Restklasse von $R[X]/(p)$ enthält genau ein Polynom vom Grad $\leq n-1$. Dieses kann auch effizient berechnet werden: Ist ein beliebiges Polynom

$$f = f_0 + f_1 X + f_2 X^2 + \ldots f_m X^m$$

gegeben, so kann das Polynom $\tilde{f}$ vom Grad $< n$ mit $\tilde{f} \equiv f \mod p$ in $O(nm)$ ausgerechnet werden.

b.) Etwas abstrakter: Die additive Gruppe von $R[X]/(p)$ ist via

$$(a_0, a_1, \ldots, a_{n-1}) \mapsto [a_0 + a_1 X + \ldots + a_{n-1} X^{n-1}]$$

zu R^n isomorph.

c.) Wenn $n = 1$, also $p(X) = X - p_0$ ist, dann ist das in der Tat ein Isomorphismus von Ringen. Die Umkehrabbildung ist die Auswertungsabbildung $R[X]/(X - p_0) \to R, [f] \mapsto f(p_0)$.

Aufgabe 99.4. – Ideale vs. Teilbarkeit
Historisch sind Ideale als „ideale Zahlen“ eingeführt worden, um bestimmte Eigenschaften der Teilbarkeitsrelation besser zu verstehen. Dies beruht auf folgenden zu zeigenden Eigenschaften.
Es sei R ein kommutativer Ring, $p, q \in R$ zwei beliebige Elemente. Zeige:

a.) $(p) \subseteq (q) \iff q | p$

b.) Gibt es ein $r \in R$ mit $(p) + (q) = (r)$, dann ist r ein größter gemeinsamer Teiler von p und q.

c.) Gibt es ein $r \in R$ mit $(p) \cap (q) = (r)$, dann ist r ein kleinstes gemeinsames Vielfaches von p und q.

Aufgabe 99.5.
Zeige: $\mathbb{Z}$ ist ein Hauptidealring, d.h. jedes Ideal $I \trianglelefteq \mathbb{Z}$ ist von der Form $I = (k)$ für ein $k \in \mathbb{Z}$.

Aufgabe 99.6.
Es sei $\mathbb{K}$ ein Körper. Zeige, dass $\mathbb{K}[X]$ auch ein Hauptidealring ist.