LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

# Grundlagenpraktikum: Rechnerarchitektur

Schnelle Exponentiation von Matrizen (A326)
Projektaufgabe – Aufgabenbereich Algorithmik

## 1 Organisatorisches

Auf den folgenden Seiten finden Sie die Aufgabenstellung zu Ihrer Projektaufgabe für das Praktikum. Die Rahmenbedingungen für die Bearbeitung werden in der Praktikumsordnung festgesetzt, die Sie über die Praktikumshomepage[1] aufrufen können.

Wie in der Praktikumsordnung beschrieben, sind die Aufgaben relativ offen gestellt. Besprechen Sie diese innerhalb Ihrer Gruppe und konkretisieren Sie die Aufgabenstellung. Die Teile der Aufgabe, in denen C-Code anzufertigen ist, sind in C nach dem C17-Standard zu schreiben.

Der **Abgabetermin** ist **Sonntag 16. Juli 2023, 23:59 Uhr (CEST)**. Die Abgabe erfolgt per Git in das für Ihre Gruppe eingerichtete Projektrepository. Bitte beachten Sie die in der `README.md` angegebene Liste von abzugebenden Dateien.

Die **Abschlusspräsentationen** finden in der Zeit vom **21.08.2023 – 01.09.2023** statt. Weitere Informationen werden noch bekannt gegeben. Beachten Sie, dass die Folien für die Präsentation am obigen Abgabetermin im PDF-Format abzugeben sind und keine nachträglichen Änderungen akzeptiert werden können.

Bei Fragen/Unklarheiten in Bezug auf den Ablauf und die Aufgabenstellung wenden Sie sich bitte an Ihren Tutor.

Wir wünschen Ihnen viel Erfolg und Freude bei der Bearbeitung Ihrer Aufgabe!

Mit freundlichen Grüßen
Die Praktikumsleitung

[1] `https://gra.caps.in.tum.de`

# 2 Schnelle Exponentiation von Matrizen

## 2.1 Überblick

Die Algorithmik ist ein Teilgebiet der Theoretischen Informatik, welches wir hier unter verschiedenen praktischen Aspekten beleuchten: Meist geht es um eine konkrete Frage- oder Problemstellung, welche durch mathematische Methoden beantwortet oder gelöst werden kann. Sie werden im Zuge Ihrer Projektaufgabe ein Problem lösen und die Güte Ihrer Lösung wissenschaftlich bewerten.

## 2.2 Karazuba-Multiplikation

Wenngleich die Multiplikation zweier 64-Bit Zahlen auf einem 64-Bit Prozessor einfach und effizient in Hardware realisierbar ist, so ist dies bei größeren Zahlen nicht der trivialerweise der Fall. Eine effiziente Berechnung bei größeren Zahlen erfordert daher zusätzlichen Aufwand.
Die naive Multiplikation zweier Zahlen mit je $n$ Bits hat eine Laufzeitkomplexität von $\Theta(n^2)$, diese lässt sich mittels Divide-and-Conquer allerdings weiter reduzieren; beispielsweise mittels des Karazuba-Algorithmus [1] auf $\mathcal{O}(n^{1.59})$. Hierbei werden die Faktoren $x$ und $y$ in zwei gleich große Teile von je $m$ Bits aufgeteilt, $x = x_0 + 2^m x_1$ und $y = y_0 + 2^m y_1$. Die Berechnung des Produkts sieht dann wie folgt aus:

$$\begin{aligned} x \cdot y &= (x_0 + 2^m x_1)(y_0 + 2^m y_1) \\ &= x_0 y_0 + 2^m (x_0 y_1 + x_1 y_0) + 2^{2m} x_1 y_1 \\ &= x_0 y_0 + 2^m ((x_0 + x_1)(y_0 + y_1) - x_0 y_0 - x_1 y_1) + 2^{2m} x_1 y_1 \end{aligned}$$

Auf diese Weise wird die erforderliche Anzahl der Teil-Multiplikation von vier auf drei reduziert – es müssen lediglich die Multiplikationen $x_0 y_0$, $x_1 y_1$ und $(x_0 + x_1)(y_0 + y_1)$ berechnet werden. Durch eine rekursive Anwendung des Algorithmus bis $m$ effizient durch vorhandene CPU-Instruktionen berechenbar ist, lassen sich auch große Zahlen effizienter[2] multiplizieren.

## 2.3 Schnelle Exponentiation

Der Algorithmus ermöglicht die effiziente Potenzierung für einen ganzzahligen Exponenten. Die Beschreibung ist hier für natürliche Zahlen gegeben, der Algorithmus kann aber ohne Probleme beispielsweise auch auf Matrizen angewandt werden. Das folgende Beispiel soll die generelle Funktionsweise verdeutlichen:
Angenommen, Sie wollen $A = 7^{11}$ berechnen. Zur Berechnung suchen Sie zunächst die kleinste Zahl $z$, für die $2^z > 11$ gilt; in unserem Fall findet man $z = 4$. Die Zahl $z - 1$ entspricht der Zahl der Quadrierungen, die Sie für die Eingabebasis 7 ausführen müssen. Tabelle 1 zeigt die einzelnen Schritte zur Berechnung der Werte $a_i$.
Anschließend zerlegt man den Exponenten in seine Binärdarstellung $11 = 1011_b$ und liest die Stellen gleich 1 gesetzten Bits ab. Im konkreten Fall wären das die Bits $0, 1$ und

[2] Es gibt auch effizientere Verfahren mit geringerer asymptotischer Laufzeit, siehe z.B. [2].

| $i$ | $a_i$ |
|---|---|
| 0 | $7 = 7$ |
| 1 | $7^2 = 49$ |
| 2 | $49^2 = 2401$ |
| 3 | $2401^2 = 5764801$ |

Tabelle 1: Quadrierungen

3 (die unterste Stelle ist das "nullte" Bit). Die abgelesenen Zahlen entsprechen den $a_i$ aus Tabelle 1, die miteinander multipliziert werden müssen, um das Endergebnis $A$ zu erhalten:

$$\begin{aligned} A &= a_0 \cdot a_1 \cdot a_3 \\ &= 7 \cdot 49 \cdot 5764801 \\ &= 1977326743 \\ &= 7^{11} \end{aligned}$$

Damit ist der Algorithmus zur schnellen Exponentiation grob umrissen.

### 2.4 Funktionsweise

Eine Methode, die Konstante $\sqrt{2}$ zu berechnen, liefert die Rechenvorschrift:

$$\begin{pmatrix} 0 & 1 \\ 1 & 2 \end{pmatrix}^n = \begin{pmatrix} x_{n-1} & x_n \\ x_n & x_{n+1} \end{pmatrix} \quad \Rightarrow \quad \lim_{n\to\infty} 1 + \frac{x_n}{x_{n+1}} = \sqrt{2} \tag{1}$$

Durch Potenzieren der Matrix lassen sich so für ein beliebiges $n$ lassen sich $x_{n-1}$, $x_n$ und $x_{n+1}$ bestimmen, sodass $1 + \frac{x_n}{x_{n+1}}$ gegen $\sqrt{2}$ konvergiert. Sie sollen diese Formel für ein beliebiges $n$ mittels des Algorithmus der schnellen Exponentiation umsetzen und damit $\sqrt{2}$ *beliebig genau* berechnen.

### 2.5 Aufgabenstellungen

Ihre Aufgaben lassen sich in die Bereiche Konzeption (theoretisch) und Implementierung (praktisch) aufteilen. Sie können (müssen aber nicht) dies bei der Verteilung der Aufgaben innerhalb Ihrer Arbeitsgruppe ausnutzen. Antworten auf konzeptionelle Fragen sollten an den passenden Stellen in Ihrer Ausarbeitung in angemessenem Umfang erscheinen. Entscheiden Sie nach eigenem Ermessen, ob Sie im Rahmen Ihres Abschlussvortrags auch auf konzeptionelle Fragen eingehen. Die Antworten auf die Implementierungsaufgaben werden durch Ihren Code reflektiert.
**Wichtig:** Stellen Sie sicher, dass Ihre Implementierung für beliebige $n$ korrekt funktioniert, also insbesondere auch für Werte $x_n$, die mehr als 64 Bits in der Darstellung erfordern!

### 2.5.1 Theoretischer Teil

- Erklären Sie, warum die schnelle Exponentiation funktioniert.
- Übertragen Sie die schnelle Exponentiation auf Matrizen. Machen Sie sich die Funktionsweise anhand eines einfachen Beispiels klar.
- Überlegen Sie, wie Sie eine Multiplikation und Addition von ganzen Zahlen beliebiger Genauigkeit effizient realisieren können.
- Leiten Sie ein effizientes Verfahren für die Division her, beispielweise das *Newton–Raphson*-Verfahren. Warum sind Fließkommazahlen nach IEEE-754 *nicht* geeignet?
- Definieren Sie eine Datenstruktur `struct bignum`, um Ganzzahlen beliebiger Größe zu speichern, sowie ein Format, um das Berechnungsergebnis als Fixkommazahl zu speichern.

### 2.5.2 Praktischer Teil

- Implementieren Sie im Rahmenprogramm I/O-Operationen in C, mit welchen der Benutzer die Konstante in wählbarer Genauigkeit berechnen und wahlweise in dezimaler oder hexadezimaler Darstellung ausgeben lassen kann.
- Implementieren Sie in der Datei mit Ihrem C-Code die Funktion:

  ```
  struct bignum sqrt2(size_t s)
  ```

  Diese soll $1+\frac{x_n}{x_{n+1}} \approx \sqrt{2}$ als Fixpunkt-Zahl mit $s$ binären Nachkommastellen berechnen. Dabei soll $n$ so gewählt werden, dass das Ergebnis auf $s$ binäre Nachkommastellen genau ist.

### 2.5.3 Rahmenprogramm

Ihr Rahmenprogramm muss bei einem Aufruf die folgenden Optionen entgegennehmen und verarbeiten können. Wenn möglich soll das Programm sinnvolle Standardwerte definieren, sodass nicht immer alle Optionen gesetzt werden müssen.

- `-V<Zahl>` — Die Implementierung, die verwendet werden soll. Hierbei soll mit `-V 0` Ihre Hauptimplementierung verwendet werden. Wenn diese Option nicht gesetzt wird, soll ebenfalls die Hauptimplementierung ausgeführt werden.
- `-B<Zahl>` — Falls gesetzt, wird die Laufzeit der angegebenen Implementierung gemessen und ausgegeben. Das *optionale* Argument dieser Option gibt die Anzahl an Wiederholungen des Funktionsaufrufs an.
- `-d<Zahl>` — Ausgabe von $n$ dezimalen Nachkommastellen. Hat Vorrang vor `-h`
- `-h<Zahl>` — Ausgabe von $n$ hexadezimalen Nachkommastellen

- `-h` — Eine Beschreibung aller Optionen des Programms und Verwendungsbeispiele werden ausgegeben und das Programm danach beendet.
- `--help` — Eine Beschreibung aller Optionen des Programms und Verwendungsbeispiele werden ausgegeben und das Programm danach beendet.

Sie dürfen weitere Optionen implementieren, beispielsweise um vordefinierte Testfälle zu verwenden. Ihr Programm muss jedoch nur unter Verwendung der oben genannten Optionen verwendbar sein. Beachten Sie ebenfalls, dass Ihr Rahmenprogramm etwaige Randfälle korrekt abfangen muss und im Falle eines Fehlers mit einer aussagekräftigen Fehlermeldung auf `stderr` und einer kurzen Erläuterung zur Benutzung terminieren sollte.

### 2.6 Allgemeine Bewertungshinweise

Beachten Sie grundsätzlich alle in der Praktikumsordnung angegebenen Hinweise. Die folgende Liste konkretisiert einige der Bewertungspunkte:

- Stellen Sie unbedingt sicher, dass *sowohl* Ihre Implementierung *als auch* Ihre Ausarbeitung auf der Referenzplattform des Praktikums (`lxhalle`) kompilieren und vollständig korrekt bzw. funktionsfähig sind.
- Die Implementierung soll mit GCC/GNU as kompilieren. Verwenden Sie keinen Inline-Assembler. Achten Sie darauf, dass Ihr Programm keine x87-FPU- oder MMX-Instruktionen und SSE-Erweiterungen nur bis SSE4.2 verwendet. Andere ISA-Erweiterungen (z.B. AVX, BMI1) dürfen Sie nur benutzen, sofern Ihre Implementierung auch auf Prozessoren ohne derartige Erweiterungen lauffähig ist.
- Sie dürfen die angegebenen Funktionssignaturen (nur dann) ändern, wenn Sie dies (in Ihrer Ausarbeitung) begründen.
- Verwenden Sie die angegebenen Funktionsnamen für Ihre Hauptimplementierung. Falls Sie mehrere Implementierungen schreiben, legen wir Ihnen nahe, für die Benennung der alternativen Implementierungen mit dem Suffix „`_V1`“, „`_V2`“ etc. zu arbeiten.
- Denken Sie daran, das Laufzeitverhalten Ihres Codes zu testen (Sichere Programmierung, Performanz) und behandeln Sie *alle möglichen Eingaben*, auch Randfälle. Ziehen Sie ggf. alternative Implementierungen als Vergleich heran.
- Eingabedateien, welche Sie generieren, um Ihre Implementierungen zu testen, sollten mit abgegeben werden; größere Eingaben sollten stattdessen stark komprimiert oder (bevorzugt) über ein abgegebenes Skript generierbar sein.
- Stellen Sie Performanz-Ergebnisse nach Möglichkeit grafisch dar.
- Vermeiden Sie unscharfe Grafiken und Screenshots von Code.

- Geben Sie die Folien für Ihre Abschlusspräsentation im PDF-Format ab. Achten Sie auf hinreichenden Kontrast (schwarzer Text auf weißem Grund!) und eine angemessene Schriftgröße. Verwenden Sie 16:9 als Folien-Format.

## Literatur

[1] Anatolii Alekseevich Karatsuba und Yu P. Ofman. „Multiplication of many-digital numbers by automatic computers“. In: *Doklady Akademii Nauk*. Bd. 145. 2. Russian Academy of Sciences. 1962, S. 293–294.

[2] Donald E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 3. Aufl. Addison-Wesley Professional, 2014.