Institut für Angewandte und
Numerische Mathematik

PD Dr. Mathias J. Krause
M.Sc. Stefan Karch
M.Sc. Mariia Sukhova

29.11.2023

Einstieg in die Informatik und Algorithmische Mathematik

Aufgabenblatt 7

Bearbeitungszeitraum: 11.12.2023 – 12.01.2024

Aufgabe 1 *Die Regel von Sarrus*

Jeder quadratischen Matrix $A \in \mathbb{R}^{n\times n}$ kann man eine reelle Zahl $\det(A)$ zuordnen, die man die *Determinante von* A nennt. Diese Zahl ist aus folgendem Grund wichtig: Man kann zeigen, dass ein lineares Gleichungssystem genau dann lösbar ist, wenn die Determinante der zugehörigen Systemmatrix von Null verschieden ist. Die Determinante einer 2×2–Matrix kann man beispielsweise wie folgt berechnen:

$$\det \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = a_{11}a_{22} - a_{21}a_{12}. \tag{1}$$

Auch für die Determinante einer 3×3–Matrix gibt es eine direkte Berechnungsformel. Dazu erweitert man die Matrix

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \tag{2}$$

zunächst zu einem 3×5–Tableau

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{11} & a_{12} \\ a_{21} & a_{22} & a_{23} & a_{21} & a_{22} \\ a_{31} & a_{32} & a_{33} & a_{31} & a_{32} \end{pmatrix}, \tag{3}$$

indem man die ersten zwei Spalten der Matrix A rechts an diesselbe anfügt. Anschließend berechnet man die Größen p_1, p_2, p_3 und n_1, n_2, n_3, indem man jeweils die Komponenten auf den ersten drei Diagonalen (für p_1, p_2, p_3) bzw. auf den ersten drei Gegendiagonalen (für n_1, n_2, n_3) des Tableaus miteinander multipliziert.

Es gilt also beispielsweise $p_1 = a_{11}a_{22}a_{33}$ und $n_2 = a_{11}a_{23}a_{32}$. Die Determinante der Matrix A ist dann durch

$$\det = p_1 + p_2 + p_3 - n_1 - n_2 - n_3 \tag{4}$$

gegeben. Dieses Berechnungsverfahren ist unter dem Namen „Regel von Sarrus“ bekannt. Für 4×4– und größere Matrizen gibt es keine einfachen Berechnungsformeln mehr. Schreiben Sie mit Java ein Programm, das die Determinante einer 3×3–Matrix nach der Regel von Sarrus berechnet. Gehen Sie dabei wie folgt vor:

- Erstellen Sie eine öffentliche Klasse mit dem Namen `Determinante`, die eine `main`–Methode enthält. Erzeugen Sie in dieser Methode ein zweidimensionales Feld namens `matrix` mit 3 Zeilen und 3 Spalten, welches Gleitkommazahlen speichert. Sie können dazu folgende Befehlszeile verwenden: `double[][] matrix = new double [3][3];` In dem Feld `matrix` soll die Matrix A gespeichert werden.

- Lesen Sie die Komponenten der Matrix A von der Konsole ein, und speichern Sie diese in den Feldkomponenten des Feldes `matrix` ab. Verwenden Sie dazu zwei geschachtelte `for`–Schleifen: Laufen Sie mit der äußeren Schleife die Zeilen und mit der inneren die Spalten der Matrix ab.

  Vorsicht! In der Mathematik ist es üblich, die Zeilen und Spalten einer Matrix mit 1 beginnend zu numerieren. Die erste Komponente der Matrix wird daher mit a_{11} bezeichnet. In Java werden die Komponenten eines Feldes jedoch mit 0 beginnend numeriert. Die erste Feldkomponente ist daher `matrix[0][0]`. Achten Sie darauf, wenn Sie auf die einzelnen Feldkomponenten zugreifen.

- Erzeugen Sie ein zweites zweidimensionales Feld mit dem Namen `tableau` vom Gleitkommatyp mit 3 Zeilen und 5 Spalten. Speichern Sie in diesem Feld die Komponenten des Tableaus gemäß (3). Kopieren Sie dazu die Werte aus dem Feld `matrix` an in die entsprechenden Komponenten des Feldes `tableau`

- Erzeugen Sie zwei eindimensionale Felder mit den Namen `p` und `n` vom Gleitkommatyp der Länge 3. Berechnen Sie die Größen p_1, p_2, p_3 und n_1, n_2, n_3 und speichern Sie diese in den entsprechenden Feldkomponenten ab. Berechnen Sie anschließend die Determinante der Matrix A gemäß (4), und geben Sie das Ergebnis auf der Konsole aus.

- Testen Sie Ihr Programm mit den folgenden Beispielen:

$$\det \begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix} = 4, \qquad \det \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = 0.$$

Aufgabe 2 (Pflichtaufgabe) *Spiel des Lebens*

Für das „Spiel des Lebens“ soll eine Matrix mit $n \times n$ ($n = 10$) Zellen programmiert werden. Eine Zelle kann sich entweder im Zustand `LEBT` oder `TOT` befinden. Als Nachbarn einer Zelle bezeichnet man alle Zellen, die links, rechts, oberhalb, unterhalb oder diagonal versetzt zu dieser Zelle liegen. (**Vorsicht**: am Rand existieren nicht alle diese Zellen.) Eine Zelle im Innern hat also acht, Randzellen haben fünf und Eckzellen haben drei Nachbarzellen.
Die Zellen sind zeilenweise wie folgt indiziert:

$$\begin{matrix} (1,1) & (1,2) & \dots & (1,n) \\ (2,1) & (2,2) & \dots & (2,n) \\ \vdots & \vdots & & \vdots \\ (n,1) & (n,2) & \dots & (n,n) \end{matrix}$$

Beispielsweise hat die Eckzelle $(1, 1)$ die Nachbarn $(1, 2)$, $(2, 1)$ und $(2, 2)$.
Ausgehend von einer Anfangsgeneration (der Anfangszustand `TOT` oder `LEBT` soll zufällig bestimmt werden) wird eine neue Zellgeneration nach folgenden Regeln erzeugt:

- Eine Zelle wird (unabhängig von ihrem derzeitigen Zustand) in der nächsten Generation `TOT` sein, wenn sie in der jetzigen Generation weniger als zwei (Vereinsamung) oder mehr als drei (Überbevölkerung) lebende Nachbarn besitzt.
- Eine Zelle mit genau zwei lebenden Nachbarn ändert ihren Zustand nicht.
- Eine Zelle mit genau drei lebenden Nachbarn wird sich in der nächsten Generation im Zustand `LEBT` befinden.

Hinweis: Man darf erst dann die alte Generation ändern, wenn alle Entscheidungen für die neue Generation getroffen sind.
Das dazu passende Java-Programm soll wie folgt programmiert werden:

- Erstellen Sie eine Klasse `Zellen` mit einer Methode `genesis`. In dieser Methode soll eine boolsche Matrix `zellen` erzeugt werden, welche den Zustand der Zellkolonie repräsentiert. Dabei steht der boolsche Wert **`true`** für den Zustand `LEBT` und **`false`** für `TOT`. Diese Matrix wird zum Schluss zurückgegeben – der Rückgabetyp ist also **`boolean`**`[][]`. Als Argument erhält die Methode die ganzzahlige Dimension `n` der Matrix sowie die Anzahl der anfangs lebendigen Zellen `lebendig`, welche zufällig auf die Matrix verteilt werden. Es müssen also folgende Schritte durchgeführt werden:
  - Erstellen Sie eine $n \times n$ Matrix `zellen` vom Typ **`boolean`**`[][]`.
  - Erzeugen Sie eine **`while`**– Schleife, die `lebendig`-mal durchlaufen soll. Dazu eignet sich die Definition einer Zählvariablen `zaehl`.
  - Erzeugen Sie in dieser Schleife zufällige Indizes `strasse` und `hausnummer`, welche die Koordinaten in der Matrix `zellen` beschreiben.
    Mit dem Ausdruck `(`**`int`**`)(Math.random()*N)` erhalten Sie eine zufällige ganze Zahl im Bereich $\{0, ..., N-1\}$.
  - Prüfen Sie zusätzlich in der Schleife, ob die Zelle, die durch `strasse` und `hausnummer` beschrieben wird, tot ist. In diesem Fall wird `zaehl` um eins erhöht und dieser Zelle der Wert **`true`** zugewiesen. So können Sie mitzählen, wie viele Zellen bisher aktiviert wurden.
- Schreiben Sie eine Methode `anzNachbarn`, welche bestimmen soll, wie viele lebendige Nachbarn eine bestimmte Zelle besitzt. Übergeben Sie dazu die boolsche Zellmatrix `zellen`, sowie die Koordinaten `strasse` und `hausnummer` der Zelle als Argument. Achten Sie beim Zählen darauf, dass Randzellen weniger Nachbarn besitzen und die Methode nur auf die gültigen Bereiche der Matrix zugreift. Weiterhin zählt eine Zelle selbst *nicht* als einer ihrer Nachbarn. Geben Sie anschließend die Anzahl der lebendigen Nachbarzellen zurück.
- Schreiben Sie eine Methode `neueGeneration` in der aus einer gegebenen Zellmatrix `zellen` die Matrix der nachfolgenden Generation `nextGeneration` berechnet und zurückgegeben werden soll. Tragen Sie die neuen Zustände in einer neu erstellten boolschen Matrix ein. Nutzen Sie für jede Koordinate die Methode `anzNachbarn` um zu wissen, wie der neue Zustand der gegebenen Zelle lauten muss.

- Schreiben Sie eine Methode `ausgabe` ohne Rückgabewert. Diese soll eine übergebene boolsche Zellmatrix anschaulich auf der Konsole ausgeben. Beispielsweise können Sie lebendige Zellen durch ein Sternchen (`*`) und tote Zellen durch ein Leerzeichen veranschaulichen.
- Schreiben Sie die `main`-Methode der Klasse. Hier soll eine entsprechende Kolonie `zellen` erstellt und ihre Entwicklung über 10 Generationen simuliert werden.
  Lesen Sie hierzu die Anzahl der anfangs lebendigen Zellen ein. Verwenden Sie die Methode `genesis` um die Anfangsgeneration zu berechnen. Für jede Generation soll die neue Matrix `zellen` mit Hilfe der Methode `neueGeneration` ermittelt werden. Geben Sie die aktuelle Zellmatrix auf dem Bildschirm aus. Erstellen Sie eine geeignete **`for`**– Schleife um die Kolonie für 10 Generationen zu beobachten.

Aufgabe 3 *Lösung eines linearen Gleichungssystems (Crout)*

Die Lösung eines linearen Gleichungssystems $Ax = b$ mit $A \in \mathbb{R}^{n \times n}$ und $x, b \in \mathbb{R}^n$, also

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \dots & + & a_{2n}x_n & = & b_2 \\ \vdots & & \vdots & & & & \vdots & & \vdots \\ a_{n1}x_1 & + & a_{n2}x_2 & + & \dots & + & a_{nn}x_n & = & b_n \end{array}$$

kann nach dem Gaußschen Eliminationsverfahren bestimmt werden. Will man das Gleichungssystem für verschiedene rechte Seiten lösen, so ist dieses Verfahren zu aufwendig, da es für jede rechte Seite komplett durchgeführt werden muss. Günstiger ist für dieses Problem das Verfahren von *Crout*:

Man zerlegt die Koeffizientenmatrix $A = (a_{ik})$ zunächst in eine untere und eine obere Dreiecksmatrix L und U, für die gilt

$$Ax = LUx = b.$$

Die Lösung von $Ax = b$ erhält man dann durch Auflösen der beiden gestaffelten Systeme

$$Ly = b \qquad \text{und} \qquad Ux = y.$$

Dieses Verfahren soll nun in einem Java-Programm implementiert werden. Gehen Sie dazu folgendermaßen vor:

- Erstellen Sie eine öffentliche Klasse `Crout` und schreiben Sie zwei statische Methoden `read_matrix` und `read_vector` für die zeilenweise Eingabe von Gleitkommazahlen in eine quadratische Matrix `A` und in einen eindimensionalen Vektor `b`. Die Dimension `n` wird dabei jeweils als ganzzahlige Variable an die Methode übergeben. Innerhalb der Methoden soll eine Matrix der Dimension $n \times n$ bzw. ein Feld der Dimension n erstellt, vom Benutzer mit Werten gefüllt und dann zurückgegeben werden.
- Schreiben Sie eine statische Methode `lu` ohne Rückgabewert, welche die LU-Zerlegung einer $n \times n$-Matrix A gemäß dem folgenden Algorithmus durchführt:

$$\left.\begin{array}{lcll} l_{ii} & := & 1 & \\ u_{ik} & := & a_{ik} - \sum_{j=1}^{i-1} l_{ij}u_{jk}, & k = i\,, \ldots,\, n \\ l_{ki} & := & (a_{ki} - \sum_{j=1}^{i-1} l_{kj}u_{ji})/u_{ii}, & k = i+1,\, \ldots,\, n \end{array}\right\} i = 1,\, \ldots,\, n$$

$$L = (l_{i,j})_{1 \le i,j \le n}, U = (u_{i,j})_{1 \le i,j \le n}$$

`A`, `L`, `U` sind zweidimensionale Matrizen die Gleitkommazahlen speichern und sollen dabei als Argumente vom Typ an die Methode `lu` übergeben werden. Die Dimension n erhalten Sie über `A.length` (Da eine Matrix ein Feld von Feldern ist, erhält man mit diesem Ausdruck genau genommen die Anzahl der Zeilen von A). Innerhalb von `lu` soll davon ausgegangen werden, dass `A` bereits mit Werten gefüllt und die übergebenen `L` und `U` bereits Felder der Dimension $n \times n$ sind, in die die passenden Werte nur noch eingetragen werden müssen.

Hinweis: Da Java Felder immer referenziert ("veränderbar") an Methoden übergibt, benötigen wir keinen Rückgabewert: Ruft man die Methode `lu` beispielsweise im `main`-Programm auf, so wirken sich die hier gemachten Änderungen an `L` und `U` auch dort aus. Achtung, für Argumente mit "einfachen" Datentypen wie **`int`** oder **`double`** gilt das nicht!

- Schreiben Sie eine Funktion `LU_solve` zur Lösung des linearen Gleichungssystems nach erfolgter LU-Zerlegung. Dabei muss zunächst das Gleichungssystems $Ly = b$ mit der unteren Dreiecksmatrix L durch eine Berechnung der Einträge

$$y_i = (b_i - \sum_{j=1}^{i-1} l_{ij}y_j)/l_{ii}, \quad i = 1 \ldots n.$$

gelöst werden. Danach werden die Einträge des Lösungsvektors x durch Lösen des Gleichungssystems $Ux = y$ mit der oberen Dreiecksmatrix U durch

$$x_i = (y_i - \sum_{j=i+1}^{n} u_{ij}x_j)/u_{ii}, \quad i = n \ldots 1$$

bestimmt. Die Matrizen `L`, `U` sowie der Vektor `b` sollen dabei als ein- bzw zweidimensionale Argumente vom Gleitkommatyp an `LU_solve` übergeben werden. Erstellen Sie die Felder `y` und `x` von passender Länge. Der Ergebnisvektor `x` soll zum Schluss als eindimensionales Feld zurückgegeben werden.

- Schreiben Sie eine **`void`**-Funktion `print_vector` für die Ausgabe eines eines eindimensionalen Gleitkomma-Vektors `b` auf der Konsole. Wie oben erhalten Sie die Länge eines Feldes `b` über den Ausdruck `b.length`.

- Schreiben Sie nun die `main`-Methode

  (a) Zunachst soll eine zweidimensionale Gleitkomma `A` deklariert werden. Lesen Sie daraufhin die Dimension `n` ein und nutzen Sie die Methode `read_matrix` um die Matrix `A` mit Elementen zu belegen.

  (b) Definieren Sie sich **`double`**`[][]`-Variablen `L` und `U` denen Sie wie in der Methode `read_matrix` mithilfe der **`new`**-Operation leere $n \times n$-Felder zuweisen. Rufen Sie die Methode `lu` auf um `L` und `U` zu füllen.

(c) Definieren Sie sich einen eindimensionales Feld `b`, welches Gleitkommazahlen speichert. In einer Schleife soll nun jeweils mittels `read_vector` ein Vektor `b` eingelesen und das entsprechende Ergebnis von `LU_solve` mittels `print_vector` ausgegeben werden. Fragen Sie den Benutzer nach jedem Schleifendurchlauf, ob ein weiteres `b` eingegeben oder das Programm beendet werden soll (beispielsweise kann bei der Eingabe einer Zahl eine 0 für Abbruch stehen, eine eingegebene 1 führt zu einer weiteren Eingabe).

Beachten Sie, dass die aufwändige LU-Zerlegung bei diesem Verfahren nur einmal durchgeführt werden muss. Verwenden Sie zum Testen des Programms die Systeme

$$\begin{pmatrix} 1 & 3 & 9 & 6 \\ 3 & 7 & 2 & 3 \\ 1 & 1 & 1 & 4 \\ 8 & 3 & 9 & 1 \end{pmatrix} \cdot x = \begin{pmatrix} 37 \\ 40 \\ 13 \\ 16 \end{pmatrix} \quad , \quad \begin{pmatrix} 1 & 3 & 9 & 6 \\ 3 & 7 & 2 & 3 \\ 1 & 1 & 1 & 4 \\ 8 & 3 & 9 & 1 \end{pmatrix} \cdot x = \begin{pmatrix} 15 \\ 56 \\ 23 \\ 10 \end{pmatrix}$$

Näherungsweise Lösungen: $x = \begin{pmatrix} -1.538 \\ 5.154 \\ 1.2 \\ 2.046 \end{pmatrix}$ und $x = \begin{pmatrix} 2.517 \\ 6.021 \\ -3.636 \\ 4.524 \end{pmatrix}$

Hinweis:

- Dieses Verfahren ist nicht für alle quadratischen Matrizen geeignet, da solch eine LU-Zerlegung nicht existieren muss! Eine hinreichende Bedingung für die Existenz ist beispielsweise, dass die Determinanten von $\begin{pmatrix} a_{1,1} \end{pmatrix}$, $\begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix}$, $\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}$, ... alle ungleich Null sind.

Aufgabe 4 (Weihnachtsaufgabe) *Der Weihnachtsmann ist unterwegs...*

Am 24. Dezember hat der Weihnachtsmann viel zu tun. Mit seinem Schlitten muss er in jedem Dorf von Schornstein zu Schornstein fliegen, um Geschenke für die Kinder zu verteilen. Zum Glück muss er jeden Schornstein in einem Dorf nur genau einmal besuchen. Dennoch stellt sich für ihn die Frage, in welcher Reihenfolge er die Schornsteine besuchen sollte, um einen möglichst geringen Flugweg zurückzulegen.
Diese Fragestellung ist unter dem Begriff *TSP* (*Travelling Santa-Claus Problem**) bekannt. Bei einem TSP sind n Zielpunkte mit den Koordinaten (x_0, y_0), (x_1, y_1), ..., $(x_{n-1}, y_{n-1}) \in \mathbb{R} \times \mathbb{R}$ gegeben. Ziel ist es, Routen mit minimaler Weglänge zu finden, bei der jeder Zielpunkt genau einmal erreicht wird. Ausgangs- und Endpunkt jeder Route ist dabei (o.b.d.A.) der erste Zielpunkt mit den Koordinaten (x_0, y_0). Eine Route kann man als Permutation der endlichen Folge $(1, 2, \ldots, n-1)$ darstellen. Für n Zielpunkte ergeben sich daher $(n-1)!$ verschiedene Routen. Schreiben Sie ein Java-Programm, welches dem Weihnachtsmann helfen kann, eine optimale Flugroute zu finden. Gehen Sie dabei folgendermaßen vor.

- Erstellen Sie eine öffentliche Klasse namens `Weihnachtsmann`. Definieren Sie für diese Klasse eine Klassenmethode namens `fakultaet` mit einem formalen ganzzahligen Parameter. Die Methode soll für eine übergebene Zahl $n \in \mathbb{N}$ die Fakultät $n!$ *rekursiv* berechnen und als ganzzahligen Wert zurückgeben.

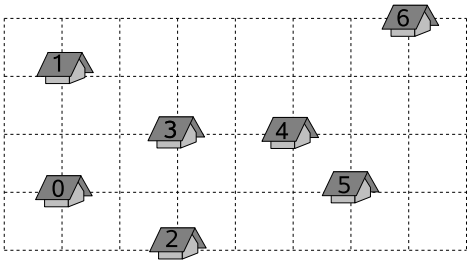
- Definieren Sie für die Klasse `Weihnachtsmann` die nachfolgende Klassenmethode

```
static int[] permutation(int[] route, int index) {
    int[] neueRoute = new int [route.length];
    System.arraycopy(route, 0, neueRoute, 0, route.length);
    int fakul = fakultaet(neueRoute.length-1);
    for (int i = 0; i < neueRoute.length - 1; i++) {
        int k = (index / fakul) % (neueRoute.length - i);
        int z = neueRoute[i + k];
        for (int j = i + k; j > i; j--) {
            neueRoute[j] = neueRoute[j - 1];
        }
        neueRoute[i] = z;
        fakul /= (neueRoute.length - (i + 1));
    }
    return neueRoute;
}
```

  Die Methode gibt die p-te Permutation ($0 \leq p < m!$) einer Folge von m natürlichen Zahlen zurück. Die Folge wird dabei über den Parameter `route`, der Index p über den Parameter `index` übergeben.

- Definieren Sie eine Klassenmethode namens `weglaenge` mit einem formalen Parameter vom Typ **`int`**`[]`, über den eine Route (d.h. eine Permutation von $(1, 2, \ldots, n-1)$) an die Methode übergeben werden kann, sowie zwei weiteren Parametern vom Typ **`double`**`[]`. Die Methode soll die Gesamtweglänge der übergebenen Route berechnen und als Wert vom Typ **`double`** zurückgeben. Die Koordinaten der Zielpunkte sollen über die letzten zwei Parameter an die Methode übergeben werden. Bedenken Sie, dass jede Route am ersten Zielpunkt mit den Koordinaten (x_0, y_0) beginnt und endet.

- Erstellen Sie nun die `main`-Methode Ihres Programms. Lesen Sie in dieser zunächst die Anzahl der Zielpunkte von der Konsole ein. Lesen Sie anschließend die x-Koordinaten $x_0, x_1, \ldots, x_{n-1}$ sowie die y-Koordinate $y_0, y_1, \ldots, y_{n-1}$ aller Zielpunkte von der Konsole ein, und speichern Sie diese in zwei geeigneten Feldern ab. Bestimmen Sie alle möglichen Routen und berechnen Sie für jede Route die Weglänge. Geben Sie die Routen mit der minimalen Weglänge auf der Konsole aus. Testen Sie Ihr Programm für das folgende Dorf:



* Tatsächlich steht TSP für *Travelling Salesman Problem*. Frohe Weihnachten!