Praktikum 4: Mehrdimensionale Arrays und Rekursion

Empfohlener Bearbeitungszeitraum: 25.11.2024 – 08.12.2024

Terminübersicht

Zeit	Montag	Dienstag	Mittwoch	Donnerstag	Freitag
11 – 13			INF 2 (04.12) INF 3 (27.11)		
13 – 15	INF 1 (02.12)				
15 – 17	WINF 1 (25.11) WINF 2 (02.12)				ITTI (29.11)

Aufgabe 1: Kreuzprodukt

Durch das Kreuzprodukt zweier Vektoren $\vec{a}, \vec{b} \in \mathbb{R}^3$ wird ein dritter Vektor $\vec{c} \in \mathbb{R}^3$ berechnet, welcher senkrecht zu den Vektoren $\vec{a}, \vec{b}$ ist. Das Kreuzprodukt ist mathematisch wie folgt definiert:

$$\vec{a} \times \vec{b} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{pmatrix}$$

Beispiel:

$$\vec{a} \times \vec{b} = \begin{pmatrix} 4 \\ -1 \\ 3 \end{pmatrix} \times \begin{pmatrix} -12 \\ 8 \\ 2 \end{pmatrix} = \begin{pmatrix} (-1 * 2) - (3 * 8) \\ (3 * -12) - (4 * 2) \\ (4 * 8) - (-1 * -12) \end{pmatrix} = \begin{pmatrix} -26 \\ -44 \\ 20 \end{pmatrix}$$

Implementieren Sie eine Funktion `void cross_product(double a[], double b[], double result[])`, welche für zwei gegebene Vektoren $\vec{a}, \vec{b}$, repräsentiert als Double-Arrays, das Kreuzprodukt $\vec{a} \times \vec{b}$ berechnet und das Ergebniss in das Double-Array `result` schreibt. Das Ergebniss der Berechnung soll dann in der `main`-Funktion ausgegeben werden.

Nutzen Sie folgenden Programmrahmen:

```c
#include <stdio.h>
void cross_product(double a[], double b[], double result[]){
  // YOUR CODE HERE
}

int main() {
  double a[3] = {4, -1, 3};
  double b[3] = {-12, 8, 2};
  double c[3];
  cross_product(a,b,c);
  // YOUR CODE HERE
  return 0;
}
```

Aufgabe 2: Addition von Matritzen

Matrixaddition ist eine additive Verknüfung zweier Matritzen $A, B \in \mathbb{R}^{m \times n}$ mit m Zeilen und n Spalten. Für die resultierenden Summenmatrix $C \in \mathbb{R}^{m \times n}$ werden die Elemente durch die komponentenweise Addition der

jeweils entsprechenden Einträge der beiden Ausgangsmatritzen berechnet:

$$A = \begin{bmatrix} a_{11} & \dots & a_{1_n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & \dots & b_{1_n} \\ \vdots & \ddots & \vdots \\ b_{m1} & \dots & b_{mn} \end{bmatrix} \quad A + B = \begin{bmatrix} a_{11} + b_{11} & \dots & a_{1n} + b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & \dots & a_{mn} + b_{mn} \end{bmatrix}$$

Beispiel:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad B = \begin{bmatrix} 5 & 3 & 1 \\ 2 & 7 & 9 \\ 4 & 6 & 8 \end{bmatrix} \quad A + B = \begin{bmatrix} 6 & 5 & 4 \\ 6 & 12 & 15 \\ 11 & 14 & 17 \end{bmatrix}$$

Schreiben Sie eine Funktion `void matrix_sum(double a[3][3], double b[3][3], double result[3][3])`, das zwei Matrizen $A, B \in \mathbb{R}^{3\times3}$, repräsentiert durch 2-dimensionale Double-Arrays, addiert und das Ergebniss in das 2-dimensionale Double-Array `result` schreibt. Die das Ergebniss der Berechnung soll anschließend in der `main`-Funktion ausgegeben werden.

Nutzen Sie folgenden Programmrahmen:

```c
#include <stdio.h>
void matrix_sum(double a[3][3], double b[3][3], double result[3][3]){
  // YOUR CODE HERE
}

int main() {
  double a[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
  double b[3][3] = {{5, 3, 1}, {2, 7, 9}, {4, 6, 8}};
  double c[3][3];
  matrix_sum(a,b,c);
  // YOUR CODE HERE
  return 0;
}
```

Zusatzaufgabe Implementieren Sie die Skalarmultiplikation $n * A$ mit $n \in \mathbb{R}$ und $A \in \mathbb{R}^{n\times m}$. Dabei wird n mit jedem Element in der Matrix multipliziert. Definieren Sie sich dafür eine extra Funktion. Berechnen Sie $A + 1.23 * B$ für die obigen Matritzen und geben Sie das Ergebniss aus.

Aufgabe 3: Darstellung eines 2D-Arrays durch ein 1D-Array

Im Vergleich zu Zugriffe and Elemente in 1-dimensionalen Arrays können die Zugriffe auf die Elemente in 2-dimensionalen Arrays langsamer sein. Ein Grund dafür ist, dass der benötigte Speicherplatzt für ein 2-dimensionales Array nicht zwangsläufig am Stück im verfügbaren Speicher allokiert wird.

Ein gängiger Trick ist es daher bei ressourcen-kritischen Anwendungen, z.B., Simulationen und Computer Spiele, ein 2-dimensionales Array `array[n][m]`, mit `n` Zeilen und `m` Spalten, durch ein 1-dimensionalen Array `array[n*m]` zu „emulieren“. Die Herausforderung besteht nun dabei die zusammengesetzten Indicies `i` und `j` in `array[i][j]` auf einen neuen Index `x` in `array[x]` abzubilden. Dabei können nur die Informationen über `n`, `m`, `i` und `j` genutzt werden.

Gegeben Sei folgende Matrix $A = \mathbb{Z}^{3\times5}$ gegeben:

$$A = \begin{pmatrix} -1 & 12 & -3 & 2 & 1 \\ 0 & 0 & 1 & 2 & 3 \\ 5 & 5 & -5 & -6 & 6 \end{pmatrix}$$

Schreiben Sie ein Programm und gehen Sie dabei wie folgt vor:

a) Speichern Sie die Matrix A in einem 1D-Array.

b) Überlegen Sie sich wie Sie in dem 1D-Array ein beliebiges Matrixelement $a_{ij} \in A$ für $i = 1, ..., n$ und $j = 1, ..., m$ adressieren können und geben Sie sich im Terminal das Element $a_{23} \in A$ aus. Beachten Sie die 0-Indizierung bei Arrays in C.

c) Schreiben Sie sich eine Funktion, die das 1D-Array im Terminal als Matrix ausgibt.

d) Verändern Sie den Wert von a_{21} zu 5.

e) Modifizieren Sie die Matrix A so, dass gilt: $\forall i, j : j >= i \implies a_{ij} = 0$. Benutzen Sie dafür Schleifen.

f) Geben Sie sich die Matrix A aus. Sie sollte wie folgt aussehen:

$$A = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 \\ 5 & 5 & 0 & 0 & 0 \end{pmatrix}$$

g) (SCHWER) Überlegen Sie sich wie 3D-Array `a[z][y][x]` durch ein 1D-Array `a[n]` repräsentiert werden kann. Geben Sie eine Formel an um von den Indizies x, y, z den entsprechenden Index n zu berechnen.

Aufgabe 4: Conway's Game of Life

Conway's Game of Life ist ein zellulärer Automat, der vom britischen Mathematiker John Conway entwickelt wurde.

Das Spielfeld ist ein Raster von Zellen, die entweder lebendig oder tot sind. Die Entwicklung der Zellen erfolgt nach den folgenden Regeln:

1.) Eine lebende Zelle mit weniger als zwei lebenden Nachbarn stirbt an Einsamkeit.

2.) Eine lebende Zelle mit zwei oder drei lebenden Nachbarn bleibt am Leben.

3.) Eine lebende Zelle mit mehr als drei lebenden Nachbarn stirbt an Überbevölkerung.

4.) Eine tote Zelle mit genau drei lebenden Nachbarn wird lebendig, durch Reproduktion.

Für die Berechnung der Zellen für den Zeitpunkt $t + 1$, werden die o.g. Regeln auf alle Zellen zum Zeitpunkt t angewandt. Dabei werden die Zellen zum Zeitpunkt t nicht überschrieben, sondern es wird ein neues Spielfeld erzeugt und mit den neuen Zellen gefüllt. Ein Beispiel für den Ablauf des Automaten ist in Abbildung 1 dargestellt.

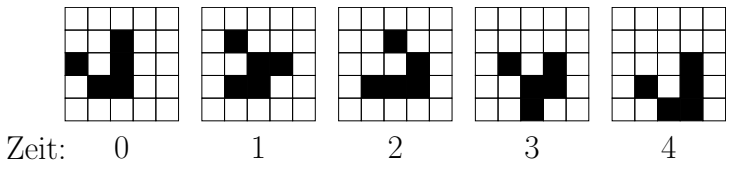

Abbildung 1: Darstellung des zellulären Automaten mit 5x5 Zellen über 5 Zeitschritte.

Implementieren Sie Conway's Game of Life unter Verwendung des vorgegebenen Programmrahmens `conway.c`. Folgende Aufgaben sind dabei zu erledigen:

a) Implementieren Sie die Prozedur `init_field`, die den Zustand aller Zellen auf `DEAD` setzt.

b) Implementieren Sie die Prozedur `print_field`, die das Feld auf der Standardausgabe ausgibt. Achten Sie darauf, dass die x-Dimension für Zeilen steht, und die y-Dimension für Spalten.

c) Implementieren Sie die Funktion `next_cell_state`, die die Koordinaten einer Zelle und ein Feld mit den Zuständen aller Zellen übergeben bekommt und den neuen Zustand der Zelle zurückgibt. Das Feld soll als Torus implementiert werden. Nutzen Sie für den Wrap-Around beim Torus die Funktion `torus`. Achten Sie darauf, den Zustand der Zelle in der Funktion `next_cell_state` nicht zu verändern.

d) (SCHWER) Es gibt in Conway's Game of Life sogenannte oszillierende Objekte. Hierbei handelt es sich um Objekte, die sich nach einem bestimmten Schema periodisch verändern, d. h. nach einer endlichen Anzahl von Generationen wieder den Ausgangszustand erreichen. Erzeugen Sie ein Programm, das systematisch alle möglichen Startzustände in einem kleinen Bereich erzeugt und durch Ausführen experimentell prüft, ob es oszillierende Objekte sind. Geben Sie diese Objekte dann aus. Beschränken Sie die Anzahl der zu berechnenden Zeitschritte auf 10. Der kleinste, nicht triviale, Oszillator ist auf einem Feld 3x3 zu finden. Ebenso sollte ein Oszillator auf einem 10x10 Feld zu finden sein. Wie viele mögliche Startzustände gibt es auf einem $n \times n$ Feld? ´

Aufgabe 5: Rekursion

a) Gegeben sei folgende Funktion:

```c
void collatz(int n) {
  printf("%d ", n);
  if (n == 1) {
    printf("\n");
    return;
  }
  if (n % 2 == 0) {
    collatz(n / 2);
  } else {
    collatz(3 * n + 1);
  }
  return;
}
```

Lösen Sie folgende Aufgaben:

(a) Geben Sie die Ausgabe des Programmes für $n = 8$ an!

(b) Stellen Sie die Funktion als Struktogramm da!

(c) Schreiben Sie eine äquivalente Funktion, welche anstatt von Rekursion nun Schleifen verwendet.

(d) Vergleichen Sie die iterative und rekursive Funktion bzgl. der erwarteten Rechenzeit für große n. Begründen Sie die Antwort!

b) Schreiben Sie eine rekursive Funktion `void printNumbers(int n)`, welche die Zahlen von 1 bis inklusive `n` ausgibt. 6 Punkte

Beispiel: Ausgabe für `printNumbers(4)`

```
1
2
3
4
```

(Bonus) Aufgabe 6: Textgenerierung mit Grammatiken

Gegeben sei folgende Grammatik für Zuweisungen:

⟨*Zuweisungen*⟩ ::= ⟨*Zuweisung*⟩ | ⟨*Zuweisung*⟩ ⟨*Zuweisungen*⟩

⟨*Zuweisung*⟩ ::= ⟨*Variable*⟩ ':=' ⟨*Ausdruck*⟩ ';\n'

⟨*Ausdruck*⟩ ::= ⟨*Variable*⟩ | ⟨*Konstante*⟩ | ⟨*Ausdruck*⟩ '+' ⟨*Ausdruck*⟩

⟨*Variable*⟩ ::= 'a' | 'b'

⟨*Konstante*⟩ ::= '0' | '1'

Schreiben Sie ein Programm, das zufällige Zuweisungen gemäß dieser Grammatik erzeugt.

a) Schreiben Sie zunächst eine Funktion `choice`, die eine ganze Zahl n als Parameter entgegennimmt und eine zufällige ganze Zahl aus dem Intervall $[0, n-1]$ zurückgibt. Alle Zahlen im Intervall sollen mit annähernd gleicher Wahrscheinlichkeit gewählt werden.

b) Schreiben Sie für jedes Nichtterminal eine Prozedur, die eine Zeichenkette im Terminal ausgibt, die der Definition des Nichtterminals entspricht. Für die zufällige Wahl zwischen verschiedenen Alternativen (mit |) können Sie die Funktion `choice` verwenden.

c) Verwenden Sie die Prozedur für das Nichtterminal ⟨*Zuweisungen*⟩ als Einstiegspunkt in der `main`-Funktion.