Dirk Praetorius,
Michele Ruggeri

Wintersemester 2017/18
30.11./01.12.2017

Übungen zur Vorlesung
Einführung in das Programmieren für TM

Serie 6

Aufgabe 6.1. Schreiben Sie eine Funktion `int anagram(char* firstStr, char* secondStr)`, die überprüft, ob ein Wort das Anagramm eines zweiten Wortes ist. Ein Anagramm eines Wortes ist eine Buchstabenfolge, die allein durch Permutation der Buchstaben gebildet wird. Beispielsweise ist "Ampel" ein Anagramm von "Palme".
Schreiben Sie ferner ein aufrufendes Hauptprogramm, welches zwei Wörter einliest und überprüft, ob die Wörter Anagramme sind. Falls dies der Fall ist, soll von der Funktion der Wert 1 zurückgegeben werden, sonst der Wert 0. Wie haben Sie Ihren Code auf Korrektheit getestet? Speichern Sie den Source-Code unter `anagram.c` in das Verzeichnis `serie06`.

Aufgabe 6.2. Zu gegebenen reellen Stützstellen $x_1 < \cdots < x_n$ und Funktionswerten $y_j \in \mathbb{R}$ garantiert die Lineare Algebra ein eindeutiges Polynom $p(t) = \sum_{j=1}^{n} a_j t^{j-1}$ vom Grad $n-1$ mit $p(x_j) = y_j$ für alle $j = 1, \dots, n$. Nun sei $t \in \mathbb{R}$ fixiert und $p(t)$ gesucht. Man kann $p(t)$ mit dem *Neville-Verfahren* berechnen, ohne zunächst den Koeffizientenvektor $a \in \mathbb{R}^n$ berechnen zu müssen: Dazu definiere man für $j, m \in \mathbb{N}$ mit $m \geq 2$ und $j + m \leq n + 1$ die Werte

$$\begin{aligned} p_{j,1} &:= y_j, \\ p_{j,m} &:= \frac{(t - x_j) p_{j+1,m-1} - (t - x_{j+m-1}) p_{j,m-1}}{x_{j+m-1} - x_j}. \end{aligned}$$

Es gilt dann $p(t) = p_{1,n}$. Schreiben Sie eine Funktion `neville`, die den Auswertungspunkt $t \in \mathbb{R}$ sowie die Vektoren $x, y \in \mathbb{R}^n$ übernimmt und $p(t)$ mittels Neville-Verfahren berechnet. Dazu berücksichtige man das folgende schematische Vorgehen

$$\begin{array}{ccccccccccccc}
y_1 & = & p_{1,1} & \longrightarrow & p_{1,2} & \longrightarrow & p_{1,3} & \longrightarrow & \cdots & \longrightarrow & p_{1,n} & = & p(t) \\
 & & & \nearrow & & \nearrow & & & & \nearrow & & & \\
y_2 & = & p_{2,1} & \longrightarrow & p_{2,2} & & & \nearrow & & & & & \\
 & & & \nearrow & & & & & & & & & \\
y_3 & = & p_{3,1} & \longrightarrow & \vdots & & & & & & & & \\
\vdots & & \vdots & & \vdots & \nearrow & & & & & & & \\
y_{n-1} & = & p_{n-1,1} & \longrightarrow & p_{n-1,2} & & & & & & & & \\
 & & & \nearrow & & & & & & & & & \\
y_n & = & p_{n,1} & & & & & & & & & &
\end{array}$$

Der mathematische Beweis für diesen Algorithmus folgt in der Vorlesung zur Numerischen Mathematik. Zunächst schreibe man die Funktion so, dass die Matrix $(p_{j,m})_{j,m=1}^{n}$ vollständig aufgebaut wird. Speichern Sie den Source-Code unter `neville.c` in das Verzeichnis `serie06`. Sie können den Code testen, indem Sie für ein bekanntes Polynom p als Funktionswerte $y_j = p(x_j)$ wählen.

Aufgabe 6.3. Man kann das Neville-Verfahren aus Aufgabe 6.2 so programmieren, dass zur Speicherung der Werte *keine* Matrix $(p_{j,m})_{j,m=1}^{n}$ aufgebaut wird, sondern die gegebenen y_j-Werte geeignet überschrieben werden. Dadurch wird kein weiterer Speicher benötigt. Man realisiere dieses Vorgehen in einer Funktion `neville2`. Wie haben Sie Ihren Code auf Korrektheit getestet? Speichern Sie den Source-Code unter `neville2.c` in das Verzeichnis `serie06`.

Aufgabe 6.4. Schreiben Sie eine Bibliothek zur Verwaltung von *spaltenweise* gespeicherten $m \times n$-Matrizen. Implementieren Sie die folgenden Funktionen

- `double* mallocmatrix(int m, int n)`
  Allokieren von Speicher für eine spaltenweise gespeicherte $m \times n$- Matrix.

- `double* freematrix(double* matrix)`
  Freigeben des allokierten Speichers einer Matrix.
- `double* reallocmatrix(double* matrix, int m, int n, int mNew, int nNew)`
  Reallokieren und initialisieren von neuen Einträgen.

Speichern Sie die Funktionssignaturen in das Header-File `dynamicmatrix.h`. Schreiben Sie auch entsprechende Kommentare zu den Funktionen in das Header-File. In die Datei `dynamicmatrix.c` kommt dann die Implementierung der Funktionen. Verwenden Sie dynamische Arrays. Wie haben Sie Ihren Code auf Korrektheit getestet?

Aufgabe 6.5. Erweitern Sie die Bibliothek aus Aufgabe 6.4 um folgende Funktionalitäten

- `void printmatrix(double* matrix, int m, int n)`
  Gibt eine spaltenweise gespeicherte $m \times n$-Matrix als Matrix am Bildschirm aus. Die 2×3-Matrix `double matrix[6]={1,2,3,4,5,6}` soll wie folgt ausgegeben werden:

```
1 3 5
2 4 6
```

- `double* scanmatrix(int m, int n)`
  Allokiert Speicher für eine Matrix und liest die Koeffizienten der Matrix von der Tastatur ein.
- `double* cutOffRowJ(double* matrix, int m, int n, int j)`
  Schneidet die j-te Zeile aus einer $m \times n$-Matrix heraus.
- `double* cutOffColK(double* matrix, int m, int n, int k)`
  Schneidet die k-te Spalte aus einer $m \times n$-Matrix heraus.

Verwenden Sie dynamische Arrays. Wie haben Sie Ihren Code auf Korrektheit getestet?

Aufgabe 6.6. Für eine Matrix $A \in \mathbb{R}^{m \times n}$ ist die Zeilensummennorm durch

$$\|A\| = \max_{j=1,\ldots,m} \sum_{k=1}^{n} |A_{jk}|$$

gegeben. Schreiben Sie eine Funktion `zeilensummennorm`, die die Zeilensummennorm einer spaltenweise gespeicherten Matrix A berechnet und zurückgibt. Schreiben Sie ein aufrufendes Hauptprogramm, in dem A eingelesen und $\|A\|$ ausgegeben wird. Wie haben Sie Ihren Code auf Korrektheit getestet? Welchen Aufwand hat Ihre Funktion? Falls die Funktion für $n = m = 10^4$ eine Laufzeit von 0.1 Sekunden hat, welche Laufzeit erwarten Sie für $n = m = 3 \cdot 10^5$? Speichern Sie den Source-Code unter `zeilensummennorm.c` in das Verzeichnis `serie06`.

Aufgabe 6.7. Schreiben Sie Funktionen `int countValueInRow(int** matrix, int m, int n, int val, int row)` und `int countValueInColumn(int** matrix, int m, int n, int val, int col)`, die zu einer gegebenen $m \times n$-Matrix die Anzahl der Einträge in der gegebenen Zeile/Spalte zurückliefern, die mit `val` übereinstimmen. Wie haben Sie Ihren Code auf Korrektheit getestet? Speichern Sie den Source-Code unter `countValueInRowCol.c` in das Verzeichnis `serie06`.

Aufgabe 6.8. Schreiben Sie ein Programm, welches das bekannte Spiel *Tic Tac Toe* realisiert. Ihr Hauptprogramm könnte z.B. so aussehen:

```
int player=1;
int winner;
int** playboard = newPlayboard();
resetPlayboard(playboard);
while(!isGameFinished(playboard)){ // Solange noch kein Sieger klar & Felder frei.
    makeMove(playboard,player);  // Lese Spielzug von der Tastatur ein
                                 // und speichere ihn ins Spielfeld.
```

```
    printPlayboard(playboard);   // Gib die neue Spielsituation am Bildschirm aus.
    player=changePlayer(player); // Wechsle den Spieler.
}
winner=getWinner(playboard);
printWinnerMessage(winner);
delPlayboard(playboard);
```

Die Funktion `makeMove` sollte dabei den Spieler solange nach einem Spielzug fragen, bis dieser gültig ist. (Felder dürfen nicht überschrieben werden!) Für die Funktion `getWinner` können Sie unter anderem Aufgabe 6.7 verwenden. Die Funktion `isGameFinished` kann wiederum die Funktion `getWinner` verwenden. Speichern Sie den Source-Code unter `tictactoe.c` in das Verzeichnis `serie06`.