Dirk Praetorius,
Michele Ruggeri

Wintersemester 2016/17
09.11.2016

Übungen zur Vorlesung
Einführung in das Programmieren für TM

Serie 6

Aufgabe 6.1. *Bubble-Sort* ist ein ineffizienter, aber kurzer Sortier-Algorithmus: Man vergleicht aufsteigend jedes Element eines Arrays x_j mit seinem Nachfolger x_{j+1} und - falls notwendig - vertauscht die beiden. Nach dem ersten Durchlauf muß zumindest das letzte Element bereits am richtigen Platz sein. Der nächste Durchlauf muß also nur noch bis zur vorletzten Stelle gehen, usw. Wie viele geschachtelte Schleifen braucht dieses Vorgehen? Schreiben Sie eine Funktion `bubblesort`, die ein gegebenes Array $x \in \mathbb{R}^n$ mittels Bubble-Sort aufsteigend sortiert, d.h. $x_1 \leq x_2 \leq \cdots \leq x_n$, und zurückgibt. Schreiben Sie ferner ein aufrufendes Hauptprogramm, das den Vektor x einliest und in sortierter Reihenfolge ausgibt. Bestimmen Sie den Aufwand Ihrer Funktion. Speichern Sie den Source-Code unter `bubblesort.c` in das Verzeichnis `serie06`.
Hinweis:

Aufgabe 6.2. Implementieren Sie den *Quicksort*-Algorithmus, um einen Vektor $x \in \mathbb{R}^n$ zu sortieren: *Quicksort* wählt willkürlich ein Pivotelement aus der zu sortierenden Liste x, z.B. x_1. Dann zerlegt man die Liste in zwei Teillisten $x^{(<)}$ und $x^{(\geq)}$ und das Pivotelement x_1: $x^{(<)}$ enthält dabei alle Elemente $< x_1$, $x^{(\geq)}$ enthält nur Elemente $\geq x_1$. $x^{(<)}$ und $x^{(\geq)}$ werden rekursiv sortiert. Anschließend wird das Ergebnis zusammengesetzt. Schreiben darüberhinaus ein `main`-Programm, in dem Sie den Vektor x und und die Länge n einlesen und Quicksort aufrufen. Testen Sie Ihren Code entsprechend! Wie groß ist der Aufwand ihrer Funktion? Speichern Sie den Source-Code unter `quicksort.c` in das Verzeichnis `serie06`.

Aufgabe 6.3. Schreiben Sie eine Funktion `merge`, die zwei aufsteigend sortierte Felder $a \in \mathbb{R}^m$ und $b \in \mathbb{R}^n$ so vereinigt, dass das resultierende Feld $c \in \mathbb{R}^{m+n}$ ebenfalls aufsteigend sortiert ist, z.B. soll $a = (1, 3, 3, 4, 7)$ und $b = (1, 2, 3, 8)$ als Ergebnis $c = (1, 1, 2, 3, 3, 3, 4, 7, 8)$ liefern. Dabei soll ausgenutzt werden, dass die Felder a und b bereits sortiert sind. Schreiben Sie die Funktion so, das neben dem Base-Pointer des Vektors c die Längen $m, n \in \mathbb{N}$ übergeben werden. Bei Übergabe gelte $c_j = a_j$ für $j = 0, \ldots, m-1$ und $c_j = b_{j-m}$ für $j = m, \ldots, m+n-1$, d.h. bei Eingabe gilt $c = (a, b)$. Der Vektor c soll dann geeignet überschrieben werden. Schreiben Sie ein aufrufendes Hauptprogramm, in dem $m, n \in \mathbb{N}$ sowie $a \in \mathbb{R}^m$ und $b \in \mathbb{R}^n$ eingelesen werden und $c \in \mathbb{R}^{m+n}$ ausgegeben wird. Testen Sie ihr Programm mit entsprechenden Beispielen! Speichern Sie den Source-Code unter `merge.m` in das Verzeichnis `serie06`.

Aufgabe 6.4. Schreiben Sie eine rekursive Funktion `mergesort`, die ein Feld a aufsteigend sortiert und das sortierte Feld zurückgibt. Gehen Sie dabei nach folgender Strategie vor:

- Hat a Länge ≤ 2, so wird das Feld a explizit sortiert.
- Hat a Länge > 2, halbiert man a in zwei Teilfelder b und c. Man ruft rekursiv `mergesort` für b und c auf und vereinigt die beiden sortierten Teilfelder wieder zu einem sortierten Feld.

Überlegen Sie sich, wie Sie die bereits sortierten Teilfelder b und c im zweiten Schritt vereinigen können! Machen Sie sich das gesamte Vorgehen anhand des Beispiels $a = (1, 3, 5, 2, 7, 1, 1, 3)$ klar. Schreiben Sie darüber hinaus ein Hauptprogramm, in dem Sie das Feld a und die Länge einlesen, mit `mergesort` und sortiert ausgeben. Testen Sie das Programm entsprechend! Speichern Sie den Source-Code unter `mergesort.c` in das Verzeichnis `serie06`.

Aufgabe 6.5. Schreiben Sie eine Funktion `void unique(double * x, int * n)`, die einen Vektor $x \in \mathbb{R}^n$ aufsteigend sortiert, doppelte Einträge streicht und den Vektor in gekürzter Form zurückgibt. Die Funktion soll also beispielsweise den Vektor $x = (4, 3, 5, 1, 4, 3, 4) \in \mathbb{R}^7$ durch den Vektor $x = (1, 3, 4, 5) \in \mathbb{R}^4$ überschreiben. Die Länge n der Vektoren ist dynamisch zu realisieren. Schreiben Sie ein aufrufendes Hauptprogramm, in dem n und $x \in \mathbb{R}^n$ eingelesen werden und das Ergebnis der Funktion `unique` ausgegeben wird. Testen Sie Ihre Implementierung mit passenden Beispielen! Speichern Sie den Source-Code unter `unique.c` in das Verzeichnis `serie06`.

Aufgabe 6.6. 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.

Aufgabe 6.7. Schreiben Sie eine Funktion `matrixvector` zur Berechnung des Matrix-Vektor-Produkts $Ax \in \mathbb{R}^n$, wobei die Matrix $A \in \mathbb{R}^{n \times n}$ spaltenweise gespeichert ist. Nutzen Sie zum anlegen und freigeben des Speichers ihre Matrix-Bibliothek aus Aufgabe 6.6. Speichern Sie den Source-Code unter `matrixvector.c` in das Verzeichnis `serie06`.

Aufgabe 6.8. Schreiben Sie eine Funktion `DynamicEratosthenes`, die für gegebene $n < m \in \mathbb{N}$ alle Primzahlen $n \leq p_i \leq m$ berechnet und als Vektor $(p_1, p_2, p_k, 0)$ zurückgibt . Legen Sie analog zur Funktion `eratosthenes` aus Übung 4 einen Vektor $(1, 2, \ldots, n)$ an und streichen Sie alle nicht-Primzahlen. Warum ist ratsam, ein zusätzliches Element 0 anzuhängen? Speichern Sie den Source-Code unter `DymanicEratosthenes.c` in das Verzeichnis `serie06`.