Programmieren 1 Übung: Arrays

Klaus Kusche

1.) Sieb des Eratosthenes

Schon die alten Griechen kannten folgende Methode,
um zu einer *Liste aller Primzahlen* bis zu einer gewissen Größe zu gelangen:

- Markiere einmal vorbeugend alle Zahlen als Primzahl.
- Geh die Zahlen von 2 aufwärts der Reihe nach durch.

  Wenn du auf eine markierte Zahl triffst, so ist sie eine Primzahl:

  - Gib sie aus.
  - Streiche die Markierung von allen *Vielfachen* der Zahl weg.

Hinweise:

- Dein Programm soll mit einer Zahl auf der Befehlszeile aufgerufen werden
  und die Primzahlen bis zu dieser Grenze ausgeben.
- Du darfst eine fix kodierte Konstante als oberes Limit für die einzugebende Grenze
  in dein Programm einbauen und dein Array mit dieser Konstante dimensionieren.
  Du solltest aber prüfen, ob die eingegebene Grenze auch unter diesem Limit liegt.
- Sowohl in Windows als auch in Linux ist das Speicherlimit für lokale Variablen
  viel niedriger (meist 1 bis 8 MB) als das für globale Daten (1 bis 3 GB).
  Wenn du dein Array groß machen möchtest, musst du es daher *global* machen.
- *Welcher Typ* eignet sich wohl am besten für ein solches Array von ja/nein-Werten?

Denksport:

- Was fallen dir für Optimierungsmöglichkeiten ein,
  um den Rechenaufwand und den Speicherbedarf zu reduzieren?

2.) Bubblesort (Sortieren durch Austauschen)

Sortieren gehört zu den wichtigsten Aufgaben des Computers.
Eine der einfachsten (aber auch langsamsten!) Methoden ist der "Bubblesort":

"Wandere immer wieder von links nach rechts Element für Element
durch die zu sortierenden Elemente.
Vergleiche dabei unmittelbar benachbarte Elemente und vertausche sie,
wenn sie in falscher Reihenfolge sind.
Wenn du hinten angekommen bist, fange wieder vorne an,
bis du einmal ohne eine einzige Vertauschung durch alle Elemente gegangen bist."

Dazu gibt es folgende kleine Verbesserung:

"Merke dir in jedem Durchgang die Position der letzten (hintersten) Vertauschung:
Du kannst im nächsten Durchgang an dieser Stelle mit dem Vergleichen aufhören,
weil alle Elemente dahinter schon stimmen.

Wenn es gar keine Vertauschung bis zu dieser Stelle gab,
oder wenn die letzte Vertauschung ganz vorne war, bist du fertig."

Hinweise:

- Implementiere das Sortieren in einer *eigenen Funktion*
  (frag mich betreffend *Übergabe eines Arrays als Parameter*).
- C bietet kein Konstrukt zum direkten Vertauschen zweier Werte:
  Zum Vertauschen brauchst du eine Hilfsvariable
  und drei Zuweisungen "im Kreis herum" (Im Fach-Jargon: "*Dreieckstausch*").
- Die zu sortierenden Zahlen sind auf der Befehlszeile angegeben,
  die sortierten Zahlen sollen der Reihe nach ausgegeben werden.
- Lege dein Array mit *variabler Dimensionierung* genau so groß an,
  wie du es brauchst!

3.) Pascal'sches Dreieck mit Array

Wir berechnen nochmals das Pascal'sche Dreieck, diesmal mit einem Array,
das die *Zahlen einer Zeile* enthält (d.h. der Array-Index ist gleich dem *k* von "*n* über *k*"):

- Für die *erste Zeile* enthält das Array am Anfang einmal 1 und sonst lauter 0.
- Für die Berechnung des Elementes **`a[i]`** in der neuen Zeile
  summieren wir die Elemente **`a[i-1]`** und **`a[i]`** der alten Zeile.

Zwei Tricks erleichtern die Sache wesentlich:

- Wenn man die Werte der neuen Zeile *von hinten nach vorne* (rechts nach links)
  berechnet, kann man sie direkt in der alten Zeile speichern.

  Würde man von links nach rechts arbeiten, bräuchte man 2 Arrays
  (eines für die alte und eines für die neue Zeile), weil man das alte **`a[i]`**
  noch zur Berechnung des neuen **`a[i+1]`** braucht und daher
  nicht mit dem neuen **`a[i]`** überschreiben darf.
- Für **`a[0]`** braucht *kein neuer Wert* berechnet werden: **`a[0]`** bleibt immer 1!

Hinweise:

- Dein Programm wird mit einer Zahl aufgerufen:
  Der *Anzahl der Zeilen*, die auszugeben sind.
- Das Hauptprogramm sollte nur die Eingabe prüfen:
  Die eigentliche Berechnung sollte in eine *eigene Funktion* ausgelagert werden.
  Auch die Ausgabe soll eine separate Funktion werden, die
  von der Berechnungsfunktion nach jeder Berechnung einer Zeile aufgerufen wird.
- Wenn die Berechnung in einer eigenen Funktion liegt,
  kann man dort auch gleich das Array in der richtigen (variablen) Größe anlegen.
- Es geht mir diesmal nur um die Funktion, nicht um die schöne Ausgabe.

4.) Simulation des Nagelbrettes

Wir wollen den Versuch mit dem Nagelbrett simulieren:

- Man lässt k Kugeln der Reihe nach mittig von oben auf r Reihen pyramidenförmig angeordneter, in jeder Reihe gegeneinander versetzter Nägel fallen:
  Die 1. Reihe enthält einen Nagel, die zweite zwei, die letzte r Nägel.
- Unter der untersten Nagelreihe sind (r+1) Fächer: Eines links vom ersten Nagel, eines rechts vom letzten Nagel, und je eines zwischen zwei Nägeln.

  Man zählt, wie viele Kugeln in jedem der (r+1) Fächer landen, wobei die Chance in jeder Reihe 50:50 ist, dass die Kugel links oder rechts am Nagel vorbeifällt.

Hinweise:

- Abstraktion des Problems:
  - Wir simulieren die k Kugeln einzeln nacheinander,
    d.h. wir lassen in einer Schleife *k Mal eine einzelne Kugel* fallen.
  - Für jede Kugel müssen wir *r Mal nacheinander* zufällig 0 (für links) oder 1 (für rechts) *summieren*, wodurch sich eine Zahl n zwischen 0 und r ergibt.

    Dabei ist die Reihenfolge der 0 und 1 egal, es kommt nur auf die Anzahl an: Alle Wege mit x Mal links und y Mal rechts landen im selben Fach, egal, ob die Kugel zuerst nach links und dann nach rechts fällt oder umgekehrt.
  - Dieses n ist die *Nummer des Faches*,
    dessen Zählerstand wir für diese Kugel um 1 erhöhen müssen.
  - Am Anfang sind alle Fächer *leer*,
    am Ende geben wir die *Zählerstände* aller Fächer aus.
- Das Programm wird mit 2 Zahlen r (Reihen) und k (Kugeln) aufgerufen.
- Es gilt dasselbe wie oben: Das Hauptprogramm macht nur die Eingabe (und deren Prüfung), eine Funktion die Berechnung (mit einem Array der richtigen Größe!).
- Erinnerung: Berechnung von Zufallszahlen:
  - **`rand()`** liefert dir bei jedem Aufruf eine neue, pseudo-zufällige ganze Zahl zwischen 0 und irgendeinem (systemabhängigen) Maximum.
    (Wie berechnet man daraus eine zufällige 0 oder 1 ?)
  - Der einmalige Aufruf von **`srand(n)`** initialisiert die Zahlenfolge, die **`rand()`** liefert: Für fixes **`n`** (oder ohne **`srand`**) liefert **`rand()`** bei jedem Programmlauf *dieselbe* Zahlenfolge, bei verschiedenem **`n`** jedesmal eine andere.
  - Eine Initialisierung mit **`time()`** (Sekunden seit 1.1.1970) garantiert daher, dass Programmaufrufe im Abstand von mehr als 1 Sekunde verschiedene Simulationsergebnisse liefern: **`srand(time(NULL));`**
  - **`rand`** und **`srand`** kommen aus **`stdlib.h`**, time aus **`time.h`**.

Zusatzaufgabe:

- So wirklich schön ist die Simulation erst mit einem *Balkendiagramm als Ausgabe*.

  Vertikale Balken sind ziemlich aufwändig, aber horizontale sollten machbar sein.
- Passe deine Balken an Bildschirmbreite und Ergebnis an: Die Skalierung sollte so sein, dass *der längste Balken gerade die Bildschirmbreite* (80) bzw. -Höhe (24) füllt.