Technische Universität München
Institut für Informatik
Dr. rer. nat. Tobias Weinzierl
Kaveh Rahnema

März 2013

Kompaktkurs Einführung in die Programmierung

Übungsblatt 5: Funktionen

Lernziele

- Mehrfache (verschachtelte) Schleifen.
- "Modularisierung" über Funktionen.
- Einstieg in Rekursion.

1) Primzahlzwillinge (P)

Im Gegensatz zum Satz von Euklid, nachdem bekannt ist, dass es unendlich viele Primzahlen gibt, ist bis heute unbekannt ob es auch unendlich viele Primzahlzwillinge gibt.

i) Schreiben Sie eine Funktion `ist_prim(int n)` die überprüft ob eine Zahl `n` Primzahl ist oder nicht und dies entsprechend zurückgibt.

ii) Primzahlzwillinge sind Primzahlen, deren Differenzbetrag 2 und somit der kleinst mögliche ist (z.B. 3 und 5 oder 11 und 13). Realisieren Sie eine Funktion `primzahl_zwillinge()`, die die ersten n Primzahlzwillinge ermittelt und druckt. Benutzen Sie hierfür die in Teilaufgabe i) implementierte Funktion `ist_prim(int n)`. Für $n = 10$ ergibt sich folgende Ausgabe:

```
3 5
5 7
11 13
17 19
29 31
41 43
59 61
71 73
101 103
107 109
```

Hinweis: Sie sollen hierbei keinen eigenen komplizierten Algorithmus zur Berechnung der Primzahlen entwickeln. Eine einfache Variante die korrekte Ergebnisse liefert ist völlig ausreichend.

2) Tannenwald drucken (P)

Schreiben Sie ein Programm `tannenwald.cpp` welches auf die Konsole einen Wald bestehend aus Tannen druckt. Der Wald soll quadratische Größe besitzen – es gibt also genausoviel Tannen in jeder Tannenreihe wie Tannenreihen selbst. Die Anzahl der Tannenreihen kann über einen Parameter `baum_reihen` gesteuert werden.

Jede Tanne hat eine gewisse Höhe (z.B. Parameter `hoehe`) und besteht aus einem Dreiecksmuster welches pro Höhenstufe ein um eine Dreiecksreihe größeres Dreieck darstellt:

```
  *            *            *
               *            *
              ***          ***
                            *         ...
                           ***
                          *****
hoehe 1      hoehe 2      hoehe 3    ...
```

Hinweise:

- Ihre Waldfunktion kann somit die Signatur
  `void wald(int baum_reihen, int hoehe)` besitzen.
- Sie sollten Schritt für Schritt vorgehen. Versuchen Sie zunächst nur eine Tanne pro Reihe zu drucken. Dabei kann es hilfreich sein vorerst alle Dreiecke einer Tanne nur linksbündig und noch nicht symmetrisch aufzubauen. Beispiel eines Waldes der Größe 2 mit Tannen der Höhe 3:

```
  *
  *
  ***
  *
  ***
  *****

  *
  *
  ***
  *
  ***
  *****
```

- Anschließend können Sie versuchen mehrere Tannen nebeneinander zu drucken. Beispiel:

```
  *           *
  *           *
  ***         ***
  *           *
  ***         ***
  *****       *****

  *           *
  *           *
  ***         ***
  *           *
  ***         ***
  *****       *****
```

- Abschließend können Sie sich Gedanken über Abstände zwischen den Tannen und deren Symmetrisierung machen.

Folgend ist als Endresultat ein Wald der Größe 3 abgebildet (3 Baumreihen mit jeweils 3 Tannen) in dem die Tannen eine Höhe von 4 besitzen:

```
   *          *          *
   *          *          *
  ***        ***        ***
   *          *          *
  ***        ***        ***
 *****      *****      *****
   *          *          *
  ***        ***        ***
 *****      *****      *****
*******    *******    *******

   *          *          *
   *          *          *
  ***        ***        ***
   *          *          *
  ***        ***        ***
 *****      *****      *****
   *          *          *
  ***        ***        ***
 *****      *****      *****
*******    *******    *******

   *          *          *
   *          *          *
  ***        ***        ***
   *          *          *
  ***        ***        ***
 *****      *****      *****
   *          *          *
  ***        ***        ***
 *****      *****      *****
*******    *******    *******
```

4) Größter gemeinsamer Teiler (Rekursion) (P)

Schreiben Sie eine Funktion zur Berechnung des größten gemeinsamen Teilers `ggT(a,b)` zweier `int`-Zahlen a und b ($a, b \neq 0$) mittels des *euklidischen Algorithmus*. Eine rekursive Lösung lässt sich realisieren über:

- Was ist, wenn a ein Teiler von b ist?
- Wenn nicht, können Sie $\text{ggT}(a, b) = \text{ggT}(b \bmod a, a)$ verwenden.

Zum besseren Verständnis für die Arbeitsweise Ihrer Funktion ist es hilfreich, als Seiteneffekt eine Ausgabe der Aktualparameter (zu Beginn der Funktion) und des Ergebnisses (am Ende) auf den Bildschirm zu erzeugen.

Besser lesbar wird die Ausgabe, wenn Sie jede Druckzeile entsprechend der Rekursionstiefe einrücken. Dazu können Sie der Funktion einen weiteren Parameter spendieren, der angibt, wie weit eingerückt wird. Dieser Parameter kann als Obergrenze für eine `for`-Schleife dienen und sollte beim rekursiven Funktionsaufruf um eins erhöht werden.

Die Funktion wird also etwa folgende Signatur besitzen:

```
int ggT(int a, int b, int tiefe)
    /* und hier folgt der Rumpf */
```

Die Ausgabe, die durch den Aufruf `ggT(72,30,1)` erzeugt wird, sollte dann etwa so aussehen:

```
|  ggT(72,30) aufgerufen
|  |  ggT(30,72) aufgerufen
|  |  |  ggT(12,30) aufgerufen
|  |  |  |  ggT(6,12) aufgerufen
|  |  |  |  ggT(6,12) fertig; Ergebnis ist 6
|  |  |  ggT(12,30) fertig; Ergebnis ist 6
|  |  ggT(30,72) fertig; Ergebnis ist 6
|  ggT(72,30) fertig; Ergebnis ist 6
Der ggT von 72 und 30 ist: 6
```

5) Quicksort vs. Mergesort (★)

Ziel dieser Aufgabe ist es Arrays (Felder) bestehend aus ganzen Zahlen zu sortieren.

i) Realisieren Sie zunächst eine Funktion zum Ausgeben von `int` Arrays, sowie eine Funktion `Fill_Array(...)` die ein Array der Größe `SIZE` mit Zufallswerten füllt.

ii) Implementieren Sie nun eine Funktion `Is_Sorted(...)`, die überprüft ob ein Array sortiert vorliegt oder nicht und dies dem Nutzer auf der Konsole anzeigt. Diese Funktion sollte nach jedem Ihrer Sortierversuche aufgerufen werden um sicherzugehen, dass Ihre (nachfolgenden) Algorithmen korrekt funktionieren.

iii) **Quicksort**: arbeitet nach dem "Divide-and-Conquer" Prinzip:

- Wähle ein Pivotelement aus dem Array. (erstes Element, oder mittleres Element).
- Sortiere das Array so, dass Elemente mit kleinerem Wert vor und Elemente mit größerem Wert nach dem Pivotelement zu liegen kommen. Elemente mit gleichem Wert können auf beiden Seiten auftreten. Nach dieser Partitionierung befindet sich das Pivotelement in seiner finalen Position.
- Sortiere nun die Teilarrays auf beiden Seiten durch rekursive Aufrufe von Quicksort.
- Die sortierten Hälften können jetzt verschmolzen werden.

Machen Sie sich (durch weitere Quellen) mit Quicksort vertraut und implementieren Sie `Quicksort(...)` nach obigem Schema.

iv) **Mergesort**: erzeugt eine sortierte Folge durch Verschmelzen sortierter Teilstücke. Mergesort arbeitet ebenfalls nach dem "Divide-and-Conquer" Prinzip:

- Hat das Array die Länge 0 oder 1, dann ist das Array bereits sortiert. Anderenfalls:

- Teile das Array in 2 Arrays (ungefähr) halber Länge.
- Sortiere jedes dieser Teilarrays durch einen rekursiven Aufruf von Mergesort.
- Die sortierten Hälften können jetzt durch Aufruf der Prozedur `Merge(...)` verschmolzen werden.

Machen Sie sich (durch weitere Quellen) mit Mergesort vertraut und implementieren Sie `Mergesort(...)` nach obigem Schema.

v) Realisieren Sie eine Zeitmessung die Ihnen anzeigt wie lange Ihre Mergesort und/oder Quicksort Sortiertung benötigt. Setzen Sie nun ihr Zufallsarray auf die Größe `SIZE=10000000` (oder größer) und messen Sie die Zeit die Ihre beiden Algorithmen zum Sortieren benötigen. Können Sie Unterschiede feststellen?
Hinweis: Sie könnten z.B. `clock()` aus `time.h` benutzen.