Matthias Jauernig
Kurs: 03In1

19.11.03

# 7. Aufgabenserie zu den Grundlagen der Informatik

Abgabetermin: Mi, 26.11.03

## Zu 19.) C-Programm zur Bestimmung der vollkommenen Zahlen eines Bereichs

```c
/*  vollkomm.c -- Matthias Jauernig, 19.11.03
    Programm berechnet alle vollkommenen Zahlen eines Bereichs;
    eine Zahl heißt vollkommen, wenn sie sich aus der Summe aller
    ihrer Teiler ergibt                                              */

/* --- includes ------------------------------------------------------------ */
#include <stdio.h>
#include <stdlib.h>

////////////////////////////////////////////////////////////////////////////////
/* um die Summe einer vollkommenen Zahl mit geringem Speicherbedarf ausgeben zu können */
/* und als kleine Programmierübung speichern wir alle Summanden in einer einfachen     */
/* verketteten Liste, wobei jeder Knoten vom Typ "struct NODE" sein soll               */
struct NODE{
    unsigned long summand;
    struct NODE *next;
} *kopf, *ende, *akt; // wichtige Knoten, die global sein sollen

/* -- einfuegen() -- fügt ein neues Element (neuen Knoten) in die verkettete Liste ein */
void einfuegen(unsigned long zahl){
    struct NODE *neu=(struct NODE*)malloc(sizeof(struct NODE));
    neu->next=akt->next;
    akt->next=neu;
    neu->summand=zahl;
    akt=neu;
}

/* -- ausg_vollk() -- gibt die vollkommene Zahl als Summe aus ------------------- */
void ausg_vollk(unsigned long i){
    struct NODE *zgr=kopf->next;
    printf("  %lu = ", i);
    while(zgr->next!=zgr){
            printf("%lu", zgr->summand);
            zgr=zgr->next;
            if(zgr->next!=zgr)
                    printf("+");
    }
    printf("\n");
}

/* -- loesche_liste() -- gibt den Speicherplatz der einzelnen Knoten wieder frei ----- */
void loesche_liste(void){
    struct NODE *aux, *zgr=kopf->next;
    while(zgr->next!=zgr){
            aux=zgr->next;
            free(zgr);
            zgr=aux;
    }
    akt=kopf;
    kopf->next=ende;
}
////////////////////////////////////////////////////////////////////////////////
```

```c
/* --- main() ------------------------------------------------------------------------ */
int main(void){
    unsigned long zahl, summe, i, j;
    /* reserviere Speicher für die Knoten kopf & ende und setze Zeiger entsprechend    */
    kopf=(struct NODE*)malloc(sizeof(struct NODE));
    ende=(struct NODE*)malloc(sizeof(struct NODE));
    kopf->next=ende; kopf->summand=0;
    ende->next=ende; ende->summand=0;
    akt=kopf;

    printf( "Vollkommene Zahlen\n------------------\n\n");

    do{
            printf("Bis zu welcher Zahl soll ausgegeben werden?: ");
            scanf("%lu", &zahl);
    }while(zahl<2 && printf("-> Zahl muss groesser gleich 2 sein!\n\n"));

    printf("Alle vollkommenen Zahlen im Bereich [6...%lu] lauten:\n", zahl);

    /* geht jede Zahl von 6 bis zahl durch und prüft sie auf Vollkommenheit           */
    for(i=2; i<=zahl; ++i){
            summe=0;
            for(j=1; j<=i/2; ++j)
                    if(i%j==0){
                            summe+=j;
                            einfuegen(j);
                    }
            if(summe==i)
                    ausg_vollk(i);
            loesche_liste();
    }

    /* anfangs reservierte Knoten wieder freigeben*/
    free(kopf);
    free(ende);

    return 0;
}
```

→ Die Verwendung einer verketteten Liste, um die einzelnen Summanden ausgeben zu lassen, scheint das Programm zu überladen; ich habe diese Datenstruktur allerdings im Code belassen, da durch sie eine klarere Ausgabe erfolgt, jedoch auf Kosten der Lesbarkeit des Codes; auf Nachfrage per Mail an linux-related@web.de kann ich diese Datei auch gern zusenden!

→ Vollkommene Zahlen im Bereich [2…100000] (Ausgabe obigen Programms):

6 = 1+2+3

28 = 1+2+4+7+14

496 = 1+2+4+8+16+31+62+124+248

8128 = 1+2+4+8+16+32+64+127+254+508+1016+2032+4064

**Zu 20.) Äquivalenzrelation**

(a) M={a,b,c,d,e,f}
R:={(a,a),(a,b),(c,f),(e,f),(d,d)}

Kleinste Äquivalenzrelation S, für die R⊆S gilt:
S := {(a,a), (b,b), (c,c), (d,d), (e,e), (f,f), (a,b), (b,a), (c,f), (f,c), (e,f), (f,e), (c,e), (e,c)}

Faktormenge M/S:
$[a]_S = \{a,b\} = [b]_S$
$[c]_S = \{c,e,f\} = [e]_S = [f]_S$
$[d]_S = \{d\}$

→ $M/S := \{[a]_S, [c]_S, [d]_S\}$

(b) zu zeigen: semantische Äquivalenz in der Menge der Booleschen Terme ist Äquivalenzrelation

Begriffsklärung: zwei boolesche Terme heißen zueinander semantisch äquivalent, wenn sie inhaltlich (also wertemäßig, nicht syntaktisch) gleich sind, d.h. dasselbe aussagen.
Die semantische Äquivalenz gilt somit bei allen Umformungsgesetzen boolescher Terme (Distributivgesetz, Kommutativgesetz, Assoziativitätsgesetz, Absorptionsgesetz, de Morgan'sche Gesetze), woraus folgt, dass alle Umformungsgesetze je eine Äquivalenzklasse bilden.

Festlegungen:
- es sei „=" die semantische Äquivalenz, wie man es auch bei Äquivalenzumformungen boolescher Terme verwendet;
- B sei die Menge der Booleschen Terme;
- bt1, bt2, bt3 ϵ B;

somit gilt für = über B:
1.) Reflexivität: bt1=bt1
gilt, da ein boolescher Term semantisch gleich bedeutend zu sich selbst ist;
2.) Symmetrie: bt1=bt2 → bt2=bt1
gilt, da die Umformungsregeln und somit auch die semantische Äquivalenz; kommutativ sind
3.) Transitivität: bt1=bt2 ∧ bt2=bt3 → bt1=bt3
gilt, da man auch schreiben könnte: bt1=bt2=bt3, woraus folgt, dass bt1=bt3; d.h. wenn ein Term bt1 semantisch äquivalent zu einem Term bt2 ist, so kann man bt1 für bt2 einsetzen und somit statt bt2=bt3 eben auch schreiben: bt1=bt3;

→ aus 1.)-3.) folgt: semantische Äquivalenz „=" ist Äquivalenzrelation über B;

Angabe von 2 Äquivalenzklassen:
1.) $[\neg(bt1 \vee bt2)]_= = \{ \neg(bt1 \vee bt2), \neg bt1 \wedge \neg bt2 \}$ ⇒ 1. de Morgan'sches Gesetz
2.) $[ bt1 \wedge bt2 ]_= = \{ bt1 \wedge bt2, bt2 \wedge bt1 \}$ ⇒ Kommutativgesetz bezüglich ∧

## Zu 21.) „Das Kaffee-Kannen-Problem“

*C-Programm, welches den Ablauf simuliert:*

```c
/*  kaffee_kanne.c -- Matthias Jauernig, 19.11.03
    Programm simuliert das sogenannte "Kaffee-Kannen-Problem":
    es werden pro Durchgang 2 Bohnen gezogen; sind sie gleich, so wird eine
    schwarze Bohne zurückgelegt, ansonsten eine weiße...                        */

/* --- includes ------------------------------------------------------------- */
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

/* --- main() --------------------------------------------------------------- */
int main(void){
    unsigned long s, w, gez;
    char gleich[2], i;
    srand(time(NULL));
    printf(  "Kaffee-Kannen-Problem\n"
             "----------------------\n\n");
    do{
             printf("Wie viele weiße Bohnen sollen sich in der Kanne befinden?:    ");
             scanf("%lu", &w);
             printf("Wie viele schwarze Bohnen sollen sich in der Kanne befinden?: ");
             scanf("%lu", &s);
    }while((s+w)<2 && printf("-> Mehr als zwei Bohnen!\n\n"));

    /* führe solange aus, wie sich mehr als 1 Bohne in der Kanne befinden       */
    do{
             /* ziehe 2 Bohnen*/
             for(i=1; i<=2; i++){
                      /* wähle aus allen Bohnen eine aus*/
                      gez=rand()%(s+w)+1;
                      /* wenn sie schwarz ist, dann nimm eine schwarze heraus*/
                      if(gez<=s){
                               gleich[i-1]=0;
                               s--;
                      }
                      /* ist sie weiß, dann nimm eine weiße heraus*/
                      else{
                               gleich[i-1]=1;
                               w--;
                      }
             }

             /* sind die gezogenen gleich, so lege eine schwarze zurück*/
             if(gleich[0]==gleich[1])
                      s++;
             /* sind die gezogenen unterschiedlich, so lege eine weiße zurück*/
             else
                      w++;
    }while(s+w>=2);

    printf("Die letzte Bohne in der Kanne ist: %s...\n", (w==1)?"weiß":"schwarz");
    return 0;
}
```

→ Ausgaben mehrerer Testläufe:

| | |
|---|---|
| Wie viele weiße Bohnen sollen sich in der Kanne befinden?: | 19 |
| Wie viele schwarze Bohnen sollen sich in der Kanne befinden?: | 183 |
| Die letzte Bohne in der Kanne ist: | weiß… |
| Wie viele weiße Bohnen sollen sich in der Kanne befinden?: | 131 |
| Wie viele schwarze Bohnen sollen sich in der Kanne befinden?: | 18 |
| Die letzte Bohne in der Kanne ist: | weiß… |

| | |
|---|---|
| Wie viele weiße Bohnen sollen sich in der Kanne befinden?: | 1234 |
| Wie viele schwarze Bohnen sollen sich in der Kanne befinden?: | 123 |
| Die letzte Bohne in der Kanne ist: | schwarz… |
| Wie viele weiße Bohnen sollen sich in der Kanne befinden?: | 12 |
| Wie viele schwarze Bohnen sollen sich in der Kanne befinden?: | 1422 |
| Die letzte Bohne in der Kanne ist: | schwarz… |

→ Dies bringt mich zu folgender Behauptung:

Die zuletzt übrig bleibende Bohne ist genau dann *weiß*, wenn die Zahl der anfänglich in der Kanne enthaltenen *weißen* Bohnen *ungerade* ist und
die zuletzt übrig bleibende Bohne ist genau dann *schwarz*, wenn die Zahl der anfänglich in der Kanne enthaltenen *weißen* Bohnen *gerade* ist.

→ Beweis:

- Festsetzungen: s = Anzahl der schwarzen Bohnen, w = Anzahl der weißen Bohnen;
  % = Modulo-Operation
- Eine Änderung von w und s findet nach obigem Programm nur in der unteren do-while-Schleife statt; zum Beweis ziehe ich deshalb eine Schleifeninvariante hinzu:
  - Definition der Schleifeninvariante: **w % 2 = x, x=1 ∨x=0**

- *Fall 1: x=1, d.h. w ist ungerade:*
  - Behauptungen:
    (B1) w ist vor Eintritt in die Schleife ungerade
    (B2) w ist nach Durchlauf des Schleifenkörpers ungerade
    (B3) der Algorithmus terminiert
  - Beweis der Behauptungen:
    - Zu (B1): der Beweis dieser Behauptung folgt direkt aus der Definition von Fall 1: da x=1, ist w%2 = 1, d.h. w ist ungerade
    - Zu (B2): zum Beweis dieser Behauptung muss wieder eine Fallunterscheidung durchgeführt werden:
      (1) die 2 zufällig gezogenen Bohnen haben dieselbe Farbe:
        (1.1) die 2 Bohnen sind schwarz:
        in dem Fall wird s wie folgt geändert: s-2; der Wert von w ändert sich nicht; da die Bohnen gleich sind, wird eine schwarze zurückgelegt, der Wert von w bleibt allerdings erhalten;
        da sich w nicht geändert hat, ist ***w%2=1***
        (1.2) die 2 Bohnen sind weiß:
        in diesem Fall wird eine schwarze Bohne zurückgelegt und 2 weiße wurden entnommen; d.h. der Wert von w wird um 2 verringert: w-2; da w ungerade ist, so ist auch w-2 ungerade: ***(w-2)%2=1***
      (2) die 2 zufällig gezogenen Bohnen haben unterschiedliche Farben:
      es wurde eine schwarze und eine weiße Bohne entnommen: s-1, w-1;
      da die 2 Bohnen nicht dieselbe Farbe haben, wird die weiße Bohne zurückgelegt, sodass sich der Wert von w nach dem Schleifendurchlauf nicht geändert hat; da w vor der Schleife ungerade war, folgt daraus, dass ***w%2=1*** auch nach der Schleife
      → da somit alle Fälle der Änderung von s und w betrachtet wurden, folgt, dass w unter (B1) auch nach einem Schleifendurchlauf noch ungerade ist

- Zu (B3): der Algorithmus terminiert, da in jedem der im Beweis zu (B2) aufgeführten Fälle mindestens eine weiße oder eine schwarze Bohne entnommen wird; „worst-case"-Szenario wäre, dass bei jedem Schleifendurchlauf immer dasselbe geschieht, was allerdings nicht sein kann, wie folgende Fälle zeigen:
  - Es werden immer 2 schwarze Bohnen gezogen: nach endlich vielen Schritten würden sich dabei keine schwarzen Bohnen mehr in der Kanne befinden, woraus folgt, dass 2 weiße Bohnen entnommen werden. Dadurch wird zwar wieder eine schwarze Bohne zurückgelegt, w wurde allerdings um 2 gegen Null geführt, wodurch die Gesamtanzahl der Bohnen vermindert wurde.
  - Es werden immer 2 weiße Bohnen gezogen: nach endlich vielen Schritten würden dabei keine weißen Bohnen mehr in der Kanne sein, woraus folgt, dass nur noch schwarze Bohnen gezogen werden. Da hierbei immer um 1 dekrementiert wird, terminiert der Algorithmus, wenn s=1.
  - Es werden immer eine weiße und schwarze Bohne gezogen: dabei wird s jedes Mal um 1 dekrementiert, sodass sich nach endlich vielen Schritten nur noch weiße Bohnen in der Kanne befinden. Die Folge daraus findet man unter Anstrich 1.

  s und w werden also in endlich vielen Schritten zum Wert 0 hin dekrementiert, wodurch die Bedingung „s+w>=2" nicht mehr erfüllt ist und der Algorithmus terminiert;

→ Aus dem Beweis von (B2) folgt, dass nach jedem Schleifendurchlauf <u>w ungerade</u> bleibt. Da w und s immer weiter gegen 0 streben, bleibt als logische Schlussfolgerung, dass, wenn sich nur noch eine Bohne in der Kanne befindet, diese <u>Bohne weiß</u> sein muss (da 1 ungerade ist und w den Wert 0 nicht annehmen kann).

- *Fall 2: x=0, d.h. w ist gerade:* ! Beweis wird analog zu x=1 geführt, daher sehr abgekürzt !
  - Behauptungen:
    (B1) w ist vor Eintritt in die Schleife gerade
    (B2) w ist nach Ausführung der Schleife gerade
    (B3) der Algorithmus terminiert
  - Beweis der Behauptungen:
    - Zu (B1): w%2 = 0, d.h. w ist gerade
    - Zu (B2): Fallunterscheidung:
      (1) 2 Bohnen haben dieselbe Farbe:
        (1.1) schwarz: s-1, w → *w%2=0*
        (1.2) weiß: s+1, w-2 → *(w-2)%2=0*, da w%2=0
      *(2)* 2 Bohnen haben unterschiedliche Farben: s-1, w → *w%2=0*
      → w ist unter (B1) auch nach einem Schleifendurchlauf noch gerade
    - Zu (B3): siehe Fall 1

→ Aus dem Beweis von (B2) folgt, dass nach jedem Schleifendurchlauf <u>w gerade</u> bleibt. Da w und s immer weiter gegen 0 streben, bleibt als logische Schlussfolgerung, dass, wenn sich nur noch eine Bohne in der Kanne befindet, diese <u>Bohne schwarz</u> sein muss (da 1 ungerade ist und w diesen Wert nicht annehmen kann).