Matthias Jauernig
Kurs: 03In1

10.01.04
linux-related.de

12. Aufgabenserie zu den Grundlagen der Informatik
oder auch: ein „Programmier+mehr“-Beleg

Abgabetermin: Mi, 14.01.04

Zu 34.) *Schlange (queue)*

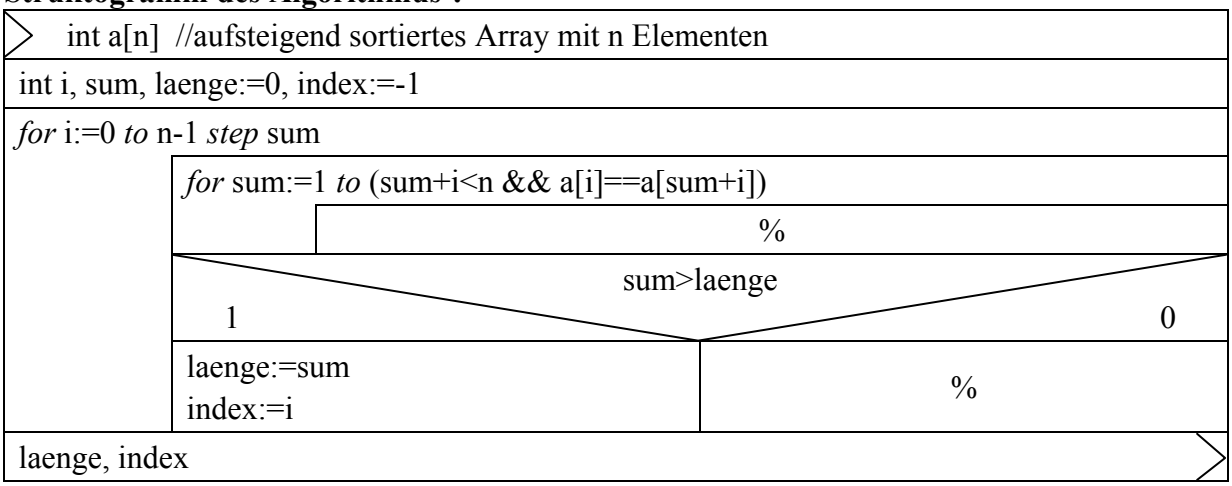
C-Programm und Beispielausgabe:

```
/*      queue.c -- Matthias Jauernig, 08.01.04                                  */
/*      Implementation einer rudimentären Schlange mit ihren Funktionen         */

#include <stdio.h>
#include <stdlib.h>

/* -- Struktur für ein Queue-Element deklarieren ---------------------------------- */
typedef struct QUEUE {
        double elem;
        struct QUEUE *next;
} queue;

/* -- empty() -- legt eine leere queue an ----------------------------------------- */
inline queue *empty(void){  return NULL;  }

/* - isempty() - prüft, ob eine queue leer ist und gibt entsprechenden Wert zurück  */
inline char isempty(queue *qe){  return qe?0:1;  }

/* -- head() -- gibt den Wert des Kopfes der queue zurück ------------------------- */
double head(queue *qe){
        if(isempty(qe)){
                printf("!! head() -- Schlange ist leer!\n");
                exit(1);
        }else
                return qe->elem;
}

/* -- tail() -- löscht den Kopf, liefert Adresse des nächsten queue-Elements ------- */
queue *tail(queue *qe){
        if(isempty(qe)){
                printf("!! tail() -- Schlange ist leer!\n");
                exit(1);
        }else{
                queue *h=qe->next;
                free(qe);
                return h;
        }
}

/* -- append() -- hängt ein Element an die queue an, gibt den aktuellen Kopf zurück */
queue *append(double zahl, queue *qe){
        queue *tmp=qe, *h=(queue *)malloc(sizeof(queue));
        h->elem=zahl;
        h->next=NULL;
        if(isempty(tmp))
                return h;
        while(tmp->next)
                tmp=tmp->next;
        tmp->next=h;
        return qe;
}

/* -- freigeben() -- gibt den Speicherplatz der queue wieder frei ----------------- */
```

```
inline void freigeben(queue *qe){  while(qe)  qe=tail(qe);  }

/* ---- main() ------------------------------------------------------------------ */
int main(void){
        queue *qe=empty(); //lege eine neue queue an, qe zeigt stets auf deren Kopf
        double zahl;
        // Testen einiger Aufrufe, um die Korrektheit der Funktionen zu prüfen
        printf( "Testlauf für die Schlange\n"
                "-------------------------\n\n");
        zahl=18.74536;
        printf("append(%g)...\n", zahl);
        qe=append(zahl, qe);
        printf("head()=%g\n", head(qe));
        printf("tail()...\n");
        qe=tail(qe);
        printf("Schlange ist %sleer!\n", isempty(qe) ? "" : "nicht ");
        zahl=1234.56789;
        printf("append(%g)...\n", zahl);
        qe=append(zahl, qe);
        zahl=-19.2837465;
        printf("append(%g)...\n", zahl);
        qe=append(zahl, qe);
        printf("head()=%g\n", head(qe));
        printf("tail()...\n");
        qe=tail(qe);
        printf("head()=%g\n", head(qe));
        printf("Schlange ist %sleer!\n", isempty(qe) ? "" : "nicht ");

        freigeben(qe);
        return 0;
}
/* Ausgabe für die Testaufrufe in main()
Testlauf für die Schlange
-------------------------

append(18.7454)...
head()=18.7454
tail()...
Schlange ist leer!
append(1234.57)...
append(-19.2837)...
head()=1234.57
tail()...
head()=-19.2837
Schlange ist nicht leer!
*/
```

Zu 35.) ***Plateau-Problem***

- **Struktogramm des Algorithmus':**



- **Betrachtung der Laufzeitkomplexität:**

- Die äußere Schleife wird maximal n-mal aufgerufen, nämlich genau dann, wenn sum immer 1 ist. Dies ist quantitativ betrachtet dann der Fall, wenn im aufsteigend sortierten Feld a für beliebige i,j gilt: $a_i \neq a_j$.
  Wenn stets gilt: sum=1, so wird die innere Schleife stets gerade einmal durchlaufen, d.h. die Laufzeitkomplexität ist linear.
- Die äußere Schleife wird maximal einmal aufgerufen, wenn sum=n ist. Dies ist quantitativ betrachtet dann der Fall, wenn im aufsteigend sortierten Feld a für beliebige i,j gilt: $a_i=a_j$
  Wenn gilt: sum=n, so wird die innere Schleife n-mal durchlaufen, d.h. die Laufzeitkomplexität ist ebenfalls linear.
- Im allgemeinen Fall ist ersichtlich: jedes Element a_i ($0 \leq i \leq n$) wird von den beiden Schleifen insgesamt nur einmal betrachtet. Wird die innere Schleife bis zum Element a_i durchlaufen, so bezieht sich der nächste Aufruf der äußeren Schleife auf a_{i+1}, der darauf folgende Aufruf der inneren Schleife auf a_{i+2} usw.

$\Rightarrow$ Daraus folgt: o.a. Algorithmus besitzt eine asymptotische Laufzeitkomplexität von O(n).

- **Beweis der Korrektheit des Algorithmus':**
  - Hierzu möchte ich zwei Schleifeninvarianten iv1 und iv2 definieren:
    - (iv1) - innere Schleife:
      ist die Laufvariable der äußeren Schleife i, so gibt sum beim Durchlauf der inneren Schleife die Plateaulänge des aktuellen Plateaus bis zum Element a_{sum+i} an
    - (iv2) - äußere Schleife:
      laenge und index geben stets das längste Plateau wieder und i ist stets der Index des nächsten zu betrachtenden Plateaus
  - Behauptungen:
    (B1) iv1 ist vor dem Ausführen der inneren Schleife richtig
    (B2) iv1 ist nach Ausführen der inneren Schleife richtig
    (B3) die innere Schleife terminiert
    (B4) iv2 ist vor dem Ausführen der äußeren Schleife richtig
    (B5) iv2 ist nach Ausführen der äußeren Schleife richtig
    (B6) die äußere Schleife terminiert
  - Beweis:
    - (B1): vor Ausführen der inneren Schleife gilt stets: sum:=1; da die Plateaulänge eines Elements 1 beträgt, ist iv1 in diesem Fall richtig
    - (B2): wenn $a_i=a_{sum+i}$, die beiden Elemente also gleich sind, so gehören sie zu demselben Plateau, sum wird also um 1 erhöht und beim nächsten Durchlauf wird das nächste Element von a auf Gleichheit mit a_i geprüft; nach Ausführen der inneren Schleife gibt sum also die Plateaulänge des aktuellen Plateaus mit der Zahl a_i an, iv1 ist also richtig
    - (B3): die innere Schleife terminiert entweder, wenn das letzte Element im Feld a betrachtet wurde oder wenn $a_i \neq a_{sum+i}$, die Länge des aktuellen Plateaus also berechnet wurde; nach spätestens n Aufrufen bei n Feldelementen ist dies der Fall
    - (B4): vor Ausführen der äußeren Schleife ist noch kein Plateau ermittelt, daher gilt: laenge=0 und index=-1; zudem wird i=0 gesetzt, es wird also das erste Element im Feld betrachtet, iv2 ist damit richtig
    - (B5): beim Ausführen der äußeren Schleife wird durch die innere Schleife mit sum die Länge des aktuellen Plateaus berechnet (siehe B2); ist diese nun größer der zur Zeit maximalen Plateaulänge, so wird die neue maximale Länge mit laenge=sum gespeichert, zudem wird index=i, also auf den Anfang des aktuellen Plateaus gesetzt, somit sind die neuen Daten des längsten Plateaus gespeichert; weiter wird i nun auf den Anfang des nächsten Plateaus gesetzt; dies geschieht, indem i um die Länge des aktuellen Plateaus (sum) erhöht wird, also i=i+sum; somit ist iv2 richtig
    - (B6): die äußere Schleife und somit der ganze Algorithmus terminieren, wenn die Bedingung i<n nicht mehr erfüllt ist; dies ist der Fall, wenn alle n Elemente des Feldes a betrachtet wurden, die beiden Schleifen zusammen also n-mal durchlaufen wurden, was durch stetige Erhöhung von i nach maximal n äußeren Schleifenaufrufen erfüllt ist

- **C-Programm und Beispielausgabe:**

```
/*      plateau.c -- Matthias Jauernig, 09.01.03                                */
/*      Programm liefert einen Algorithmus zur Lösung des "Plateau-Problems"    */

#include <stdio.h>

/* -- Struktur, um die Länge und den Index eines max. Plateaus abzuspeichern ------ */
typedef struct {
        int laenge;
        int index;
} plateau;

/* - plateau_alg() - sucht den Index und die Länge des längsten Plateaus in Array a */
plateau plateau_alg(int *a, int n){
        plateau erg={0,-1}; //Rückgabestruktur mit Anfangswerten definieren
        int i, sum;
        //schaue der Reihe nach alle Plateaus an und greife deren Länge ab
        for(i=0; i<n; i+=sum){
                for(sum=1; sum+i<n && a[i]==a[sum+i]; sum++);//Länge des akt. Plateaus
                if(sum>erg.laenge){//neues längstes Plateau gefunden -> Werte speichern
                        erg.laenge=sum;
                        erg.index=i;
                }
        }
        return erg;
}

/* ---- main() ------------------------------------------------------------------ */
int main(void){
        plateau laengstes;
        int a[]={1,2,3,3,4,4,4,5,6,6,6,6,6,7,8,9,9,9,9,9,10};//Beispielarray
        int elem=sizeof(a)/sizeof(a[0]);//Elementeanzahl im Array bestimmen
        int i;

        printf( "Plateau-Problem\n"
                "---------------\n\n");
        printf("Ausgabe des Arrays:\n");
        printf("  ");
        for(i=0; i<elem; i++)
                printf("%d, ", a[i]);
        printf("\b\b \n\n");

        laengstes=plateau_alg(a,elem);
        printf( "Laenge des laengsten Plateaus: %d\n", laengstes.laenge);
        printf( "Index des ersten laengsten Plateaus: %d (Zahl %d)\n\n",
                laengstes.index, a[laengstes.index]);

        return 0;
}

/* Ausgabe mit dem Beispielarray aus main()
Plateau-Problem
---------------

Ausgabe des Arrays:
  1, 2, 3, 3, 4, 4, 4, 5, 6, 6, 6, 6, 6, 7, 8, 9, 9, 9, 9, 9, 10

Laenge des laengsten Plateaus: 5
Index des ersten laengsten Plateaus: 8 (Zahl 6)
*/
```

Zu 36.) ***Welfare-Crook-Problem***

- **Pseudo-Code mit der Dijkstra'schen while-Anweisung:**
  (ohne Prüfung auf Überlauf, das vorausgesetzt, dass mind. 1 Element gleich)

```
Eingabe dreier aufsteigend sortierter Arrays a,b,c mit l1,l2,l3 Elementen;
int i, j, k, gef=0;
for(i=0, j=0, k=0; !gef; i++){
        do    //d-while-Anweisung
              □ b[j]<a[i] → j++;
              □ c[k]<a[i] → k++;
        od
        wenn a[i]==b[j] und a[i]==c[k], dann speichere a[i] und Indizes ab
        und setze gef=1, sodass die for-Schleife terminiert
}
Ausgabe der Indizes und des kleinsten gemeinsamen Eintrags, dann beenden
```

- **C-Programm mit Beispielausgabe:**

```
/*      welfare_crook.c -- Matthias Jauernig, 10.01.04                          */
/*      Programm simuliert das Welfare-Crook-Problem, nach dem der kleinste     */
/*      gemeinsame Eintrag dreier Arrays gefunden werden soll                   */


#include <stdio.h>

/*- Struktur zum Speichern des gemeinsamen Eintrags sowie der Indizes deklarieren - */
typedef struct{
        int kge;        // kge=(k)leinster (g)emeinsamer (E)intrag
        int i1;         // a[i1]=kge, b[i2]=kge, c[i3]=kge;
        int i2;
        int i3;
} rg_struct;

/* -- welfare_crook() -- ermittelt den kleinsten gemeinsamen Eintrag der 3 Arrays - */
rg_struct welfare_crook(int *a, int *b, int *c, int l1, int l2, int l3){
        int i, j, k, gef=0;
        rg_struct erg={-1,-1,-1,-1};
        //solange kein gemeinsamer Eintrag gefunden (gef=0) und eines alle drei
        //Arrays ihre Grenze nicht überschritten haben...
        for(i=0, j=0, k=0; !gef && i<l1 && j<l2 && k<l3; i++){
                while(b[j]<a[i]) j++;//geht zu nächstem relevanten Index j von b
                while(c[k]<a[i]) k++;//geht zu nächstem relevanten Index k von c

                if(a[i]==b[j] && a[i]==c[k]){//wenn drei Elemente gleich -> abspeichern
                        erg.kge=a[i];
                        erg.i1=i;
                        erg.i2=j;
                        erg.i3=k;
                        gef=1;//Abbruchbedingung erfüllen
                }
        }
        return erg;
}

/* ---- main() ------------------------------------------------------------- */
int main(void){
        int a[]={1,3,4,7,8,11,13,15,16,17,20};//Beispielarrays initialisieren
        int b[]={2,5,13,15,16,18,20};
        int c[]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
        int l1=sizeof(a)/sizeof(a[0]);//Anzahl der Elemente der Arrays a,b,c bestimmen
        int l2=sizeof(b)/sizeof(b[0]);
        int l3=sizeof(c)/sizeof(c[0]);
        int i;
        rg_struct erg=welfare_crook(a,b,c,l1,l2,l3);

        printf( "\nWelfare-Crook-Problem\n"
                "---------------------\n\n");
        printf("Gegeben seien folgende drei Arrays:\n");
```

```
        printf(" a[]:  ");
        for(i=0; i<l1; i++)
                printf("%d, ", a[i]);
        printf("\b\b \n b[]:  ");
        for(i=0; i<l2; i++)
                printf("%d, ", b[i]);
        printf("\b\b \n c[]:  ");
        for(i=0; i<l3; i++)
                printf("%d, ", c[i]);
        printf("\b\b \n\n");

        printf("Lösung des Welfare-Crook-Problems:\n");
        if(erg.i1==-1) printf("Kein gemeinsamer Eintrag vorhanden!\n\n");
        else{
                printf( " kleinster gemeinsamer Eintrag: %d\n", erg.kge);
                printf( " Stellen dessen Vorkommens: a[%d], b[%d], c[%d]\n\n",
                        erg.i1, erg.i2, erg.i3);
        }

        return 0;
}
/* Ausgabe für die Beispielarrays:
Welfare-Crook-Problem
---------------------

Gegeben seien folgende drei Arrays:
 a[]:  1, 3, 4, 7, 8, 11, 13, 15, 16, 17, 20
 b[]:  2, 5, 13, 15, 16, 18, 20
 c[]:  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

Lösung des Welfare-Crook-Problems:
 kleinster gemeinsamer Eintrag: 13
 Stellen dessen Vorkommens: a[6], b[2], c[12]
*/
```

- **Betrachtung der Laufzeitkomplexität des Programms:**
  Es reicht, die Laufzeit der Funktion welfare_crook() zu betrachten:
  Diese ist von den Längen der Arrays a,b und c abhängig, die hier mit m, n und k gekennzeichnet seien (im Programm: l1, l2, l3).
  Betrachtung des ungünstigsten Falls: dieser tritt auf, wenn jeweils die letzten Elemente der drei Arrays den ersten kleinsten gemeinsamen Eintrag bilden.
  Dann gilt:
  - Die äußere Schleife läuft von i=0 bis i=m-1, also m-mal.
  - Der Schleifenkörper der ersten inneren Schleife wird während der gesamten m Durchläufe der äußeren Schleife genau n-mal ausgeführt.
  - Der Körper der zweiten inneren Schleife wird während der gesamten m Durchläufe der äußeren Schleife genau k-mal ausgeführt.

  $\Rightarrow$ Daraus ergibt sich eine asymptotische Laufzeitkomplexität von O(n)+O(m)+O(k), also O(max{n,m,k}), genau wie in der Aufgabenstellung gefordert.