Fakultät IV Elektrotechnik und Informatik
Einführung in die Programmierung WS 2019/2020
Manfred Hauswirth / Damien Foucard / Christoph Seifert

Aufgabenblatt 2

Ausgabe: *15.11.2019*
Abgabe: *26.11.2019 09:59*

Thema: *Debug-Techniken*

Abgabemodalitäten

1. Alle abzugebenden Quelltexte müssen ohne Warnungen und Fehler auf den Rechnern des ZE-CM/eecsIT mittels `clang -std=c11 -Wall -g` kompilieren.
2. Die Abgabe erfolgt ausschließlich über unser SVN im Abgaben-Ordner. Nur wenn ein Test in Osiris angezeigt wird ist sichergestellt, dass die Abgabe erfolgt ist.
3. Du kannst bis zur Abgabefrist beliebig oft neue Versionen abgeben.
4. Die Abgabe erfolgt in folgendem Unterordner:
   `introprog-wise1920/Studierende/<L>/<TU-Login>/Abgaben/Blatt<XX>`
   wobei `<L>` durch den ersten Buchstabe des TU-Logins und `<XX>` durch die Nummer des Aufgabenblattes zu ersetzen sind. Die Ordner werden automatisch angelegt sobald die Abgabe freigeschaltet wird.
5. Du darfst den Abgabeordner für das Blatt nicht selbst erstellen. Die Ordner erstellen wir kurz nach der Ausgabe der Aufgabe.
6. Um die Änderungen (z. B. neue Abgabeordner) vom Server abzuholen, musst Du den Befehl `svn update` im Wurzelverzeichnis (also im Verzeichnis `introprog-wise1920`) des Repositories ausführen.
7. Im Abgabeordner gelten einige restriktive Regeln. Dort ist nur das Einchecken von Dateien mit den in der Aufgabe vorgegebenen Namen erlaubt, ausserdem werden die Abgabefristen vom Server überwacht. Beachte eventuelle Fehlermeldungen beim SVN-Commit. Lade nur Dateien hoch, die Du selbst bearbeiten sollst, insbesondere also keine Vorgaben.
8. Der Dateiname bei Abgaben mit C-Code soll dem der Vorgaben entsprechen. Entferne ausschließlich `_vorgabe` aus dem Name. Bei Textaufgaben sind, wenn nicht anders angegeben, `.txt` Dateien zugelassen, die als Plaintext-Datei zu speichern sind (keine Word-Dateien umbenennen).
9. Es gibt einen Ordner `Arbeitsverzeichnis`, in dem Du Dateien für Dich ablegen kannst.
10. Die Ergebnisse der automatischen Tests kannst Du auf OSIRIS einsehen:
    https://osiris.ods.tu-berlin.de/

Hinweise Debugging

- Ziel dieses Blatts ist, Dir weitere Hilfen zum Debugging zu geben und Dich mit der Laufzeitanalyse vertraut zu machen.
- Aufgabe 1 baut auf Blatt 8 des C-Kurses zum Thema `gdb` und Debugging auf. Sie thematisiert Speicherfehler und stellt `valgrind` als nützliches Tool vor, mit dem Speicherfehler in C gefunden werden können. Ziel der Aufgabe ist es, vorzustellen, wie in C mit Hilfe von `printf`, `gdb` und `valgrind` die Funktionsweise von Programmen nachvollzogen und Fehler effektiv gefunden werden können. Es gibt für diese Aufgabe keine automatischen Tests und sie muss nicht im SVN abgegeben werden.
- Aufgabe 2 und 3 bilden eine praktische Einführung in das Thema Laufzeitanalyse. Nächste Woche wird auf Blatt 3, das Thema weiter vertieft und fortgeschrittene Themen der Laufzeitanalyse behandelt. Beachtet die Hinweise zur Abgabe bei Aufgabe 3!

Aufgabe 1 Benutzung von valgrind zum Finden von Speicherfehlern: Student vs. Gauß (unbewertet)

Ein Student traut der Gaußschen Summenformel trotz aller induktiven Beweise nicht und bezweifelt, dass die Summe $1+2+\cdots+N$ wirklich gleich $(N+1)\cdot N/2$ ist. Um die Formel zu überprüfen, hat besagter Student ein recht komplexes Programm geschrieben, welches die Summe der ersten N natürlichen Zahlen aufaddiert und das Ergebnis anschließend mit dem Wert der Gaußschen Summenformel vergleicht.

Der Student hat sich Folgendes ausgedacht:

- Er alloziert dynamisch ein Array `array` der Länge N und schreibt in das Array die Werte 1,2, …, N. Die Zahl i steht somit im Array am Index $i-1$.
- Die Zahlen im Array werden anschließend aufsummiert, indem über das Array `array` – von hinten nach vorne – iteriert wird. Befindet sich der Algorithmus am Index i, wird der Wert `array[`i`]` durch die Summe von `array[`i`]` und `array[`$i+1$`]` ersetzt.
- Somit steht das Ergebnis[1] am Ende in `array[0]`.

Der Student ist sich sicher, dass sein Code korrekt arbeitet und meint somit Gauß widerlegen zu können: Bei manchen Werten unterscheidet sich nämlich sein Ergebnis von dem von Gauß 'vorhergesagten'.

1. Führt das Programm für verschiedene Eingaben aus. Bei welchen Eingaben erhält der Student ein anderes Ergebnis als Gauß?
2. Betrachtet den Code des Studenten. Versucht, die Funktionsweise des Programms nachzuvollziehen, und deckt etwaige Fehler auf.
3. Macht Vorschläge, wie die Fehler beseitigt werden können, und verbessert, falls möglich, den Code, so dass dieser besser lesbar / besser strukturiert / einfacher wird.

[1] Der Student war an dieser Stelle stolz darauf, dass keine weitere Variable benötigt wird.

Bei der Suche nach Fehlern kann das Programm `valgrind` sehr hilfreich sein. Es deckt folgende Fehler im Zugriff auf Speicher auf, welche teilweise das Programm zum Abstürzen bringen können:

- Wird auf Speicherbereiche geschrieben, welche nicht vom Benutzer mittels `malloc` reserviert wurden, gibt `valgrind` 'INVALID WRITE' Fehlermeldungen aus und verweist auch auf die entsprechende Code-Zeile, in welcher der Zugriff erfolgte.
- Wird aus Speicherbereichen gelesen, welche nicht vom Benutzer mittels `malloc` reserviert wurden, gibt `valgrind` 'INVALID READ' Fehlermeldungen aus und verweist auch auf die entsprechende Code-Zeile, in welcher der Zugriff erfolgte.
- Wird mittels `malloc` allozierter Speicher nicht wieder freigegeben, so kann auch dies durch `valgrind` kenntlich gemacht werden. In der Ausgabe von `valgrind` finden sich dementsprechend Informationen zu 'MEMORY LEAKs' (Deutsch: undichtem, bzw. leckgeschlagenem Speicher).

Um `valgrind` zu benutzen, geht wie folgt vor:

- Kompiliert euer C-Programm zunächst mit den 'Debug-Flags' und übergebt `clang` das `-g` flag: `clang -std=c11 -Wall -g introprog_valgrind_debugging.c input_valgrind_debug.c -o introprog_valgrind_debugging`
- Ruft nun `valgrind` wie folgt auf: `valgrind ./introprog_valgrind_debugging`
- Betrachtet die Ausgabe von Valgrind (für die ursprüngliche Version) und versucht in der Ausgabe die verschiedenen Fehlerarten sowie deren Ursprungsort zu lokalisieren.
- Die Optionen `--leak-check` bzw. `--verbose` können benutzt werden, um die Menge an ausgegebenen Informationen zu kontrollieren. Im Allgemeinen genügt jedoch der Aufruf ohne diese Parameter.

Listing 1: introprog_valgrind_debugging.c

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

#include "input_valgrind_debug.h"

//
//                      W A R N U N G
// Dieser Code dient als Vorlage, wie man ein C Programm nicht
// schreiben sollte und dient nur zu Debug-Zwecken.
//

// Alloziert ein Integer Array der Größe *size
int* allocate_array(int* size)
{
    int* result = (int*)malloc(sizeof(int) * (*size));
    return result;
}

// Initialisiere das array: Am Index i wird die Zahl i+1
// eingetragen
void initialize_array(int array[], int size)
{
    for (int i = 0; i <= size; ++i) {
        array[i-1] = i;
    }
}

// Berechnet die Summe der Elemente des Arrays, indem das Array
// von hinten durchgegangen wird: Der Wert des Arrays an der
// Stelle i-1 berechnet sich immer als Summe des Werts an der
// Stelle i-1 und des Werts an der Stelle i. Somit steht nach
// der Ausführung die Summe  aller Zahlen in array[0].

void compute_sum_and_place_in_first_elem(int array[], int* size)
{
    /*
    Mit diesem alten Code gab es Probleme; müsste aber auch
    korrekt sein

    int* ptr_to_elem = array + (*size);
    while (ptr_to_elem >= array)
    {
        (*(ptr_to_elem-1)) = (*(ptr_to_elem-1)) + (*(ptr_to_elem));
        ptr_to_elem--;
    }
    */

    for (int i = *size; i >= 0; --i) {
        array[i-1] = array[i-1] + array[i];
    }

    // Die Summe steht jetzt in array[0]
}

// Gibt den Speicher des Arrays sowie den Pointer N wieder frei.
void free_memory(int array[], int* N)
{
    // Gib den Speicher für das array und für N wieder frei.
    return; // in dieser Funktion gab es Fehler
```

```
    free(array);
    free(N);
}

// Hat Gauß wirklich recht? Ist die Summe 1,2,...,N wirklich (
// N+1)*N 2? Dieses Programm berechnet die Summe der Zahlen 1
// bis N. N kann dem Programm als Parameter übergeben werden.
// Wird kein Wert übergeben, wird der Wert N über die
// Kommandozeile abgefragt.

int main(int argc, char* argv[])
{
    int* N = malloc(sizeof(int));
    if (argc == 1) {
        // Falls das Programm ohne Parameter aufgerufen wurde,
        // lies den Wert N über die Kommandozeile ein
        while (read_number_from_stdin(N) != 1);
    }
    else if (argc == 2) {
        if (read_number_from_string(argv[1], N) == 0) {
            printf("Kommandozeilen Parameter %s konnte nicht in
                ↪ eine Zahl konvertiert werden!\n", argv[1]);
            return -1;
        }
    }
    else {
        printf("Es wird maximal ein numerischer Parameter (optional
            ↪ ) erwartet. Ausführung des Programms wird
            ↪ abgebrochen!\n");
        return -1;
    }

    // Erstelle ein Array der Größe N
    int* array = allocate_array(N);

    // Schreibe die Zahlen 1,2,...,N in das Array
    initialize_array(array, *N);
    // Berechne die Summe des Arrays
    compute_sum_and_place_in_first_elem(array,N);
    int result = array[0];
    int result_gauss = ((*N+1)*(*N)/2);
    if (result == result_gauss) {
        printf("\nVielleicht hatte Gauß ja doch recht...\nDie Summe
            ↪  der %d natürlichen Zahlen ist %d und somit gleich
            ↪ (%d+1)*%d/2.\n\n", *N, result, *N,*N);
    }
    else {
        printf("\nGauß hatte unrecht!\nDie Summe der %d natürlichen
            ↪  Zahlen ist %d und somit UNGLEICH (%d+1)*%d/2\n\n",
            ↪ *N, result, *N,*N);
    }

    // Gib den Speicher wieder frei
    free_memory(array, N);
}
```

Aufgabe 2 Erläuterungen (unbewertet)

Aufgabe 2.1 Zählweise

Die Laufzeit von Algorithmen wird in der Informatik oft auch theoretisch untersucht. Zu diesem Zweck wird der Pseudocode bzgl. der Anzahl ausgeführter Befehle hin analysiert. Auf diesem Aufgabenblatt steht die Analyse der Laufzeit in C-Programmen im Vordergrund. Wir gehen im Folgenden daher detailliert auf die Bestimmung der Anzahl ausgeführter "Befehle" in C-Programmen ein. In den Aufgaben 2 und 3 soll die hier vorgestellte "Zählweise" dann zur empirischen Analyse von einfachen `for`-Schleifen und danach Count- und Insertionsort erprobt werden.

Wir treffen die folgenden Vereinbarungen:

Deklaration von Variablen/Allokation von Speicher

Die Deklaration einer Variablen ohne Zuweisung ist nicht als die Ausführung eines Befehls zu verstehen, da lediglich (statisch) Speicher reserviert wird. Somit wird `int a;` nicht als die Ausführung eines Befehls gezählt.

Andererseits wird der Aufruf von `malloc` zur Speicherreservierung als die Ausführung eines Befehls gezählt, da hier "aktiv" Speicher alloziert wird.

Definition von Variablen/Zuweisung von Werten

Die Definition von Variablen, z. B. mittels `int i = 0;` bzw. die Zuweisung eines Wertes ist genau so als ein Befehl zu zählen wie `i = 0;`. Hierbei ist die Komplexität des Ausdrucks auf der rechten Seite der Zuweisung unerheblich. Somit ist auch `i = a*a*a*a*a + 57 % 312;` als ein Befehl zu zählen.

Die Definition mehrerer Variablen im Sinne von `int a,b = 5;` ist als ein Befehl zu zählen, da die Variable `a` nicht initialisiert wird (d.h., keinen Wert zugewiesen bekommt). Weiterhin sind verkettete Zuweisungen der Form `a = b = c = 42;` jeweils als einzelne Befehle – in diesem Fall 3 Stück – zu zählen.

Funktionsaufrufe/Rückgabe von Werten aus Funktionen

Sei die Funktion `int f(){ return 0; }` gegeben. In dieser Funktion wird genau ein Befehl ausgeführt, nämlich die Rückgabe des Werts `0`: die Rückgabe eines Wertes wird als Ausführung eines Befehls gezählt. Funktionsaufrufe werden jedoch nicht als die Ausführung eines Befehls gezählt.

Für das folgende Snippet werden daher nur die Ausführung von 6 Befehlen veranschlagt:

```
int i = f(); // 1 Befehl für die Zuweisung von i plus Kosten von f
int j = f(); // 1 Befehl für die Zuweisung von j plus Kosten von f
int k = f(); // 1 Befehl für die Zuweisung von k plus Kosten von f
```

`if`-Abfragen/Vergleiche bzw. Bedingungen

Der Vergleich von Werten, welche eine Auswirkung auf die Programmausführung hat, ist als die Ausführung eines Befehls zu betrachten. Betrachten wir die folgende einfache `if`-Abfrage:

```
if(i == 0){
  i = 1;
}
```

In dieser `if`-Abfrage ist der Vergleich `i==0` als die Ausführung eines Befehls zu zählen, da in Abhängigkeit des Vergleichs der Programmfluss geändert wird. Die Ausführung der Zuweisung `i = 1;` ist hingegen nur zu zählen, falls die Bedingung `i == 0` wahr war. Im Falle, dass `i == 0` gegolten hat, werden somit 2 Befehle gezählt, während im anderen Fall nur 1 Befehl ausgeführt wurde.

`while`-Schleifen

Eine `while`-Schleife hat in C die folgende Struktur:

```
while(<Vergleich>){
  <Körper>
}
```

Hierbei bezeichnet `<Vergleich>` einen logischen Ausdruck, welcher entweder den Wert 0 (d.h. falsch) oder 1 (d.h. wahr) annimmt. Gemäß der Vereinbarung, dass programmflusssteuernde Vergleiche als Befehl gezählt werden, ist jede Ausführung des Vergleichs auch als Befehl zu zählen.

Für das folgende Snippet werden daher insgesamt 8 Befehle gezählt, da auch der Vergleich `0 > 0`, durch welchen die Schleife verlassen wird, gezählt wird.

```
int i = 3;     //Zuweisung = 1 Befehl
while(i > 0){ //Vergleiche: 3 > 0, 2 > 0, 1 > 0, 0 > 0: 4 Befehle
  i = i-1;     //Verringerung des Werts genau i Mal: 3 Befehle
}
```

`for`-Schleifen

Eine `for`-Schleife hat in C die folgende Struktur:

```
for(<Initialisierung>; <Vergleich>; <Zuweisung>){
  <Körper>
}
```

`for`-Schleifen dienen der kompakten Schreibweise und sind *semantisch äquivalent* zu `while`-Schleifen der Form:

```
<Initialisierung>
while(<Vergleich>){
  <Körper>
  <Zuweisung>
}
```

Gemäß dieser Äquivalenz wird auch die Anzahl an ausgeführten Befehlen gezählt. Die folgende `for`-Schleife ist äquivalent zur oben betrachteten `while`-Schleife – ohne einen `<Körper>` zu besitzen – und es werden somit auch insgesamt 8 Befehle zur Ausführung benötigt.

```
for(int i = 3; i > 0; i--){
  //leer
}
```

Wird der `<Körper>` einer `for`-Schleife insgesamt n mal ausgeführt, so werden im Allgemeinen also

$$\underbrace{1}_{\text{für die Initialisierung}} + \underbrace{n+1}_{\text{für die Vergleiche}} + \underbrace{\sum \texttt{<Körper>}}_{\text{für die Ausführung des Körpers}} + \underbrace{n}_{\text{für die Zuweisung (meist Inkrementierung / Dekrementierung)}}$$

viele Befehle ausgeführt. Hierbei bezeichnet $\sum$ `<Körper>` die Summe der Befehle, welche insgesamt bei den n Durchführungen der `for`-Schleife ausgeführt worden sind.

Hinweis: Obige Formel trifft natürlich nur auf `for`-Schleifen zu, in welchen alle Komponenten der `for`-Schleife benutzt werden: Wird die Initialisierung nicht benötigt, so wird dafür auch kein Befehl gezählt.

Aufgabe 2.2 Lösungsformat

Neben der Abgabe von Code ist auf diesem Aufgabenblatt auch die Beantwortung von Multiple-Choice-Fragen gefordert. Die Beantwortung der Fragen erfolgt auch via SVN, indem ihr ein Textformular wie folgt ausfüllt:

- Editiere die Datei *ausschließlich* mit einem Editor wie Kate, Gedit, vi(m) oder emacs.
- Editiere die Datei *keinesfalls* mit einem Textverarbeitungssoftware wie Word oder LibreOffice.

- Ersetze neben den jeweils von Dir gewählten Antworten das leere Kästchen [ ] durch ein [X].
- Füge am Ende der Frage optional einen Kommentartext anstelle des Platzhalters ein.
- Halte das vorgegebene Format unbedingt ein!
- *Nimm sonst keine weiteren Änderungen am Text vor!*

Demnach wäre zur Beantwortung der entsprechenden Frage der vorgegebene Text

```
=== FRAGE 1 ===
Was ist die Antwort auf die Frage nach dem Leben, dem Universum und
    ↪  dem ganzen Rest?

ANTWORT:

        [ ] 4711
        [ ] 23
        [ ] 42

KOMMENTAR: <kein Kommentar>
=== ENDE FRAGE 1 ===
```

beispielsweise so anzupassen:

```
=== FRAGE 1 ===
Was ist die Antwort auf die Frage nach dem Leben, dem Universum und
    ↪  dem ganzen Rest?

ANTWORT:

        [ ] 4711
        [X] 23
        [ ] 42

KOMMENTAR: Illuminatus! von Robert Anton Wilson ist eh besser als
    ↪ der Hitchhiker`s guide.
=== ENDE FRAGE 1 ===
```

Hinweis: Beachtet, dass es für das Einchecken der Dateien auch Tests in OSIRIS gibt. Diese beantworten zwar nicht die Frage, ob die Antworten richtig sind, jedoch wird überprüft, ob das Format eingehalten wurde.

Aufgabe 3 Einstieg Laufzeitanalyse (10 Punkte)

Aufgabe 3.1 Befehlszähler (8 von 10 Punkten)

In dieser Aufgabe sollst Du gemäß der Erläuterungen in Aufgabe 2 die Anzahl an Befehlen für den Körper verschiedener Funktionen bestimmen. Konkret handelt es sich um die Funktionen `for_linear`, `for_quadratisch` und `for_kubisch` der Programmvorgabe (siehe Listing 2). Diesen Funktionen werden zwei Parameter übergeben: ein ganzzahliger Wert `int n` sowie ein Pointer auf den Befehlszähler `int * befehle`. In Abhängigkeit des Parameters `n` wird durch die Verschachtelung von `for`-Schleifen genau n, n^2 oder n^3 mal die Zeile `sum += get_value_one();` ausgeführt. Die Funktion `get_value_one()` ist in der Datei `introprog_complexity_steps_input.c` definiert und liefert – auf recht komplizierte Weise[2] – den Wert 1 zurück.

Innerhalb der `main()`-Funktion der Vorgabe (siehe Listing 2) werden die drei verschiedenen Funktionen nacheinander für verschiedene Werte n, welche im Array `int WERTE[]` definiert sind, ausgeführt. Dabei werden sowohl die (empirisch gemessene) Laufzeit, der Rückgabewert der jeweiligen Funktion – d.h. n, n^2 oder n^3 – als auch die Anzahl der Befehle in entsprechenden Arrays abgelegt. Es ist Deine Aufgabe, in jeder der Funktionen die Anzahl an ausgeführten Befehlen an der Stelle, auf die der Pointer `int * befehle` zeigt, zu speichern und somit die gezählte Anzahl an Befehlen zurückzugeben.

Aufgabe und Abgabemodalitäten:

Implementiere zunächst das Zählen der Befehle und checke Deine Lösung als `introprog_complexity_steps_intro.c` im SVN
(im Ordner `introprog-wise1920/Studierende/<L>/<TU-Login>/Abgaben/Blatt02`, wobei <L> durch den ersten Buchstabe des TU-Logins zu ersetzen ist)
ein.

Hinweise:

- Inkrementiere den Befehlszähler jeweils nur um 1; zähle also explizit jede Ausführung eines Befehls einfach. Weiterhin musst Du jede Inkrementierung des Befehlszählers mittels eines Kommentars kurz (sic!) begründen. Sollte dies nicht befolgen, erhälst Du im Zweifelsfall keine Punkte.
- Beachte, dass Du zum erfolgreichen Kompilieren des Programms zusätzlich die Quelldatei `introprog_complexity_steps_input.c` im Aufruf von `clang` übergeben musst.
- Beachte, dass Du den Code der `main`-Funktion weder anpassen musst noch anpassen sollst. Natürlich steht es Dir offen, die Werte des Arrays `int WERTE[]` anzupassen. Dies kann z. B. nützlich sein, falls die Ausführung des Programms zu lange dauert.

[2] Sollte man in der Funktion einfach direkt den Wert 1 zurückgeben, können Laufzeitunterschiede bei unseren Eingaben von n nicht richtig gemessen werden.

Listing 2: Vorgabe `introprog_complexity_steps_intro_vorgabe.c`

```c
#include <stdio.h>
#include "introprog_complexity_steps_input.h"

long for_linear(int n, int* befehle)
{
    // TODO: Die Befehle müssen richtig gezählt werden

    long sum = 0;
    for(int i = 1; i <= n; ++i) {
        // Zähle die folgende Zeile als genau ein Befehl!
        sum += get_value_one();
    }
    return sum;
}

long for_quadratisch(int n, int* befehle)
{
    // TODO: Die Befehle müssen richtig gezählt werden

    long sum = 0;
    for(int i = 1; i <= n; ++i) {
        for(int j = 1; j <= n; ++j) {
            // Zähle die folgende Zeile als genau ein Befehl!
            sum += get_value_one();
        }
    }
    return sum;
}

long for_kubisch(int n, int* befehle)
{
    // TODO: Die Befehle müssen richtig gezählt werden

    long sum = 0;
    for(int i = 1; i <= n; ++i) {
        for(int j = 1; j <= n; ++j) {
            for(int k = 1; k <= n; ++k) {
                // Zähle die folgende Zeile als genau ein
                // Befehl!
                sum += get_value_one();
            }
        }
    }
    return sum;
}


int main(int argc, char *argv[])
{
    const int WERTE[] = {5,6,7,8,9,10};
    const int LEN_WERTE = 6;
    const int LEN_ALGORITHMEN = 3;

    long befehle_array[LEN_ALGORITHMEN][LEN_WERTE];
    long werte_array[LEN_ALGORITHMEN][LEN_WERTE];
    double laufzeit_array[LEN_ALGORITHMEN][LEN_WERTE];

    for(int j = 0; j < LEN_WERTE; ++j) {
        int n = WERTE[j];
        for(int i = 0; i < LEN_ALGORITHMEN; ++i) {
            printf("Starte Algorithmus %d mit Wert %d\n",
                (i+1), n);
            int anzahl_befehle = 0;
            int wert = 0;

            // Starte den Timer
            start_timer();

            // Aufruf der entsprechenden Funktion
            if(i==0) {
                wert = for_linear(n, &anzahl_befehle);
            }
            else if(i==1) {
                wert = for_quadratisch(n, &anzahl_befehle);
            }
            else if(i==2) {
                wert = for_kubisch(n, &anzahl_befehle);
            }

            // Speichere Laufzeit, Rückgabewert und Anzahl
            // ausgeführter Befehle ab
            laufzeit_array[i][j] = end_timer();
            werte_array[i][j] = wert;
            befehle_array[i][j] = anzahl_befehle;
        }
        printf("\n");
    }
```

```
88
89      // Ausgabe der Rückgabewerte, Anzahl ausgeführter Befehle
90      // sowie der gemessenen Laufzeiten (in Millisekunden)
91      printf("%3s \t%-28s \t%-28s \t%-28s\n", "","linear", "
            ↪ quadratisch", "kubisch");
92      printf("%3s \t %5s %10s %10s\t %5s %10s %10s\t %5s %10s %10s\n"
            ↪ , "n","Wert","Befehle","Laufzeit","Wert","Befehle","
            ↪ Laufzeit","Wert","Befehle","Laufzeit");
93
94      for(int j = 0; j < LEN_WERTE; ++j) {
95          printf("%3d \t ",WERTE[j]);
96          for(int i = 0; i < LEN_ALGORITHMEN; ++i) {
97              printf("%5ld %10ld %10.4f \t ", werte_array[i][j],
                    ↪ befehle_array[i][j], laufzeit_array[i][j]);
98          }
99          printf("\n");
100     }
101
102     return 0;
103 }
```

Aufgabe 3.2 Multiple-Choice Laufzeit I (2 von 10 Punkten)

Im SVN findest Du unter
`Aufgaben/Blatt02/Vorgaben/introprog_complexity_steps_intro_vorgabe.txt`
eine ausfüllbare Textdatei mit den folgenden Fragen:

1. Ist das lineare, quadratische bzw. das kubische Wachstum der Funktionen `for_linear`, `for_quadratisch` bzw. `for_kubisch` klar zu erkennen?
2. Bleibt die Anzahl der ausgeführten Befehle bei mehrfachem Aufruf des Programms gleich?

Aufgabe und Abgabemodalitäten:

Kreuze in der Datei (unter Beachtung von sämtlichen Hinweisen und Vorgaben) die korrekte Antwort an und checke die modifizierte Datei im Abgabeordner für dieses Blatt als `introprog_complexity_steps_intro.txt` im SVN ein.