Fakultät IV - Elektrotechnik und Informatik
Einführung in die Programmierung WS 2015/16
Feldmann / Semmler / Lichtblau / Streibelt / Pujol / Rost

Aufgabenblatt 2

letzte Aktualisierung: 13. November, 14:27 Uhr

(f7944bc98a4503fe0963cef12caae51677cfba48)

Ausgabe: *Mittwoch, 11.11.2015*
Abgabe: *spätestens Freitag, 20.11.2015, 16:00*

Thema: *Laufzeitbestimmung, Insertionsort, Countsort*

Abgabemodalitäten

- Alle abzugebenden Quelltexte müssen ohne Warnungen und Fehler auf den Rechnern des tubIT/IRB mittels `gcc -std=c99 -Wall` kompilieren.
- Abgaben erfolgen prinzipiell immer in Gruppen à 2 Personen, welche in den Tutorien festgelegt wurden. Einzelabgaben sind explizit als solche gekennzeichnet.
- Die Abgabe erfolgt ausschließlich über SVN. Die finale Abgabe
  - für Gruppenabgaben erfolgt im Unterordner `Tutorien/t<xx>/Gruppen/g<xx>/Abgaben/Blatt<xx>/`
  - für Einzelabgaben erfolgt im Unterordner `Tutorien/t<xx>/Studierende/<tuBIT-Login>/Abgaben/Blatt<xx>/`

  wobei die Ordner von uns erstellt werden.
- Benutze für alle Abgaben von Programmcode das folgende Namensschema: `introprog_blatt0X_aufgabe0Y_Z.c`, wobei X durch die Blattnummer, Y durch die Aufgabe und Z durch die Unteraufgabe zu ersetzen ist.
  Beispiel: Aufgabe 1.2 wird zu: `introprog_blatt01_aufgabe01_2.c`
  Gib für jede Unteraufgabe maximal eine Quellcodedatei ab, es sei denn, die Aufgabenstellung erfordert explizit die Abgabe mehrerer Dateien pro Aufgabe.

  Benenne alle anderen Abgaben (Pseudocode, Textaufgaben) wie oben beschrieben. Die zugelassenen Abgabeformate sind PDF, ODT und Text (txt). Verwende auch hier eine Datei pro Aufgabe, nicht jedoch pro Unteraufgabe.

1. Aufgabe: Erläuterungen (unbewertet)

Die Laufzeit von Algorithmen wird in der Informatik zumeist 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. unabhängig davon 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 zwei Befehle zu zählen, da hier die Variable `a` implizit auf `0` gesetzt wird. 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:

```
1 int i = f();   //1 Befehl für die Zuweisung von i plus die Kosten von f
2 int j = f();   //1 Befehl für die Zuweisung von j plus die Kosten von f
3 int k = f();   //1 Befehl für die Zuweisung von k plus die 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 betrachte. Betrachten wir die folgende einfache `if`-Abfrage:

```
1 if(i == 0){
2    i = 1;
3 }
```

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 es im anderen Fall nur 1 Befehl ausgeführt wurde.

`while`-Schleifen

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

```
1 while(<Vergleich>){
2    <Körper>
3 }
```

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.

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

`for`-Schleifen

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

```
1 for(<Initialisierung>; <Vergleich>; <Zuweisung>){
2    <Körper>
3 }
```

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

```
1 <Initialisierung>
2 while(<Vergleich>){
3    <Körper>
4    <Zuweisung>
5 }
```

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.

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

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.

2. Aufgabe: Einstieg Laufzeitanalyse (1 Punkte)

In dieser Aufgaben sollst Du gemäß den Erläuterungen in Aufgabe 1 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 1). 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 `input_blatt02` definiert und liefert – auf recht komplizierte Weise[1] – den Wert 1 zurück.
Innerhalb der `main()`-Funktion der Vorgabe (siehe Listing 1) 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 – sowie 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.

Hinweis: 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.

Implementiere zunächst das Zählen der Befehle und beantworte anschließend folgende Fragen an Hand der Ausgabe des Programms:

- Ist das lineare, quadratische bzw. das kubische Wachstum der Funktionen `for_linear`, `for_quadratisch` bzw. `for_kubisch` klar zu erkennen?
- Hast Du immer die gleiche Ausgabe (bzgl. der Anzahl an ausgeführten Befehlen bzw. der Laufzeit) erhalten?
- Versuche folgende Frage zu beantworten: Welches Maß ist Deiner Meinung nach angemessener, um die Komplexität einer Funktion zu bewerten: Die gemessene Laufzeit oder die Anzahl an gezählten Befehlen? (Die Beantwortung dieser Frage ist optional.)

Check Deine Lösung als `introprog_blatt02_aufgabe02.c` im SVN (im Ordner `Tutorien/t<xx>/Gruppen/g<xx>/Abgaben/Blatt02/`) ein und geb Deine Antworten auf obige Fragen als `introprog_blatt02_aufgabe02.pdf`, `introprog_blatt02_aufgabe02.txt` oder `introprog_blatt02_aufgabe02.odt` wiederum im SVN ab.

Hinweis: Die Abgabe im SVN wird voraussichtlich ab Freitag, 13.11.2015 (morgens) möglich sein. Bitte beachtet diesbezügliche Nachrichten in ISIS.

Hinweis: Beachte, dass Du zum erfolgreichen Kompilieren des Programms zusätzlich die Quelldatei `input_blatt02.c` im Aufruf von `gcc` übergeben musst.

Hinweis: Beachte, dass Du den Code der `main`-Funktion weder anpassen musst noch 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.

[1] Sollte man die Funktion einfach durch den Wert 1 ersetzen, treten nicht die gewünschten Ergebnisse ein.

Listing 1: Vorgabe `introprog_blatt02_aufgabe02_vorgabe.c`

```c
#include <stdio.h>
#include "input_blatt02.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 WERTE[] = {5,6,7,8,9,10};
int LEN_WERTE = 6;
int LEN_ALGORITHMEN = 3;

int main(int argc, char *argv[]) {

    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");
    }

    //Ausgabe der Rückgabewerte, Anzahl ausgeführter Befehle sowie der gemessenen Laufzeiten (in Millisekunden)
    printf("%3s \t%28s \t%28s \t%28s\n", "","linear", "quadratisch", "kubisch ");
    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");

    for(int j = 0; j < LEN_WERTE; ++j)
    {
        printf("%3d \t ",WERTE[j]);
        for(int i = 0; i < LEN_ALGORITHMEN; ++i)
        {
            printf("%5ld %10ld %10.4f \t ", werte_array[i][j], befehle_array[i][j], laufzeit_array[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

3. Aufgabe: Laufzeitanalyse: Vergleich Count- und Insertionsort (2 Punkte)

In dieser Aufgabe sollst Du (empirisch) die Laufzeit Deines Insertion- und Countsort Implementierungen vergleichen. Ähnlich zu Aufgabe 2 erhälst Du eine Vorgabe, in welcher die komplette `main`-Funktion bereits vorgegeben ist. Du musst nur noch Deine Insertion- sowie Countsort Implementierungen (z.B. von Aufgabenblatt 1) einfügen und leicht anpassen. Du musst insgesamt vier verschiedene Funktionen schreiben bzw. anpassen:

1. `void count_sort_calculate_counts(int input_array[], int len, int count_array[], int* befehle)`
2. `void count_sort_write_output_array(int output_array[], int len, int count_array[], int* befehle)`
3. `void count_sort(int array[], int len, int* befehle)`
4. `void insertion_sort(int array[], int len, int* befehle)`

Hierbei sollen die Funktionen 1., 2., und 4. die gleiche Funktionalität wie auf dem Aufgabenblatt 1 haben, jedoch noch zusätzlich die ausgeführten Befehle mittels `int* befehle` zählen. Beachte zur Bestimmung der ausgeführten Befehle die Erläuterung aus Aufgabe 1, sowie die Hinweise von Aufgabe 2.
Die Funktion `count_sort` hat die gleiche Signatur, d.h. die gleichen Parameter und den gleichen Rückgabewert, wie die Funktion `insertion_sort`. Die Funktion `count_sort` soll hierbei die gesamte Funktionalität des Countsort-Algorithmus kapseln:

1. Erstelle zunächst mittels `malloc` ein Array zum Zählen der Häufigkeiten verschiedener Werte.
2. Rufe die Unterfunktionen `count_sort_calculate_counts` sowie `count_sort_write_output_array` so auf, dass das Ergebnis von Countsort (d.h. der Funktion `count_sort`) in das Eingabearray `int array[]` geschrieben wird.
3. Vergiss nicht den allozierten Speicher wieder frei zu geben.
4. Die Anzahl ausgeführter Befehle von Countsort erstreckt sich über insgesamt 3 Funktionen und muss dementsprechend auch zusammengezählt werden.

Die Vorgabe (siehe Listing 2) erzeugt für Count- und Insertion jeweils ein Array mittels `malloc`: `array_countsort` sowie `array_insertionsort`. Das Array `array_countsort` wird zunächst mittels des Funktionsaufrufs `fill_array_randomly(array_countsort, n, MAX_VALUE);` mit Zufallswerten beschrieben. Anschließend werden mittels `copy_array_elements(array_insertionsort, array_countsort, n);` die gleichen Werte auch in das Array `array_insertionsort` geschrieben werden.
Implementiere die vier Funktionen und werte die Laufzeit sowie die Anzahl der benötigten Befehle beider Algorithmen aus. Beantworte folgende Fragen:

- Welcher Algorithmus ist bei welchen Eingaben schneller?
- Inwieweit stehen die Anzahl der gezählten Befehle und die (empirisch gemessene) Laufzeit in Beziehung zueinander?
- Teste die Algorithmen für verschiedene Werte der Konstanten MAX_VALUE. Bei welchen Kombinationen von `MAX_VALUE` und Größen des Arrays n ist Insertionsort und wann Countsort vorzuziehen?

Es reicht eine kurze Begründung jeweils aus. Belege Deine Aussagen mit der Ausgabe Deines Programms. Checkt Deine Lösung als `introprog_blatt02_aufgabe03.c` im SVN (im Ordner `Tutorien/t<xx>/Gruppen/g<xx>/Abgaben/Blatt02/`) ein und geb Deine Antworten auf obige Fragen als `introprog_blatt02_aufgabe03.pdf`, `introprog_blatt02_aufgabe03.txt` oder `introprog_blatt02_aufgabe03.odt` wiederum im SVN ab.

Hinweis: Die Abgabe im SVN wird voraussichtlich ab Freitag, 13.11.2015 (morgens) möglich sein. Bitte beachtet diesbezügliche Nachrichten in ISIS.

Hinweis: Beachte, dass Du zum erfolgreichen Kompilieren des Programms zusätzlich die Quelldatei `input_blatt02.c` im Aufruf von `gcc` übergeben musst.

Hinweis: Beachte, dass Du den Code der `main`-Funktion weder anpassen musst noch sollst. Natürlich steht es Dir offen die Werte des Arrays `int WERTE[]` oder die Konstante `MAX_VALUE` anzupassen. Dies kann z.B. nützlich sein, falls die Ausführung des Programms zu lange dauert.

Listing 2: Vorgabe `introprog_blatt02_aufgabe03_vorgabe.c`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "input_blatt02.h"
4
5  int MAX_VALUE = 5000000;
6
7  void count_sort_calculate_counts(int input_array[], int len, int count_array
       ↪ [], int* befehle) {
8      //muss implementiert werden
9  }
10
11 void count_sort_write_output_array(int output_array[], int len, int
       ↪ count_array[], int* befehle) {
12     //muss implementiert werden
13 }
14
15 void count_sort(int array[], int len, int* befehle) {
16     //muss implementiert werden
17 }
18
19
20 void insertion_sort(int array[], int len, int* befehle) {
21     //muss implementiert werden
22 }
23
24 int WERTE[] = {10000,20000,30000,40000,50000};
25 int LEN_WERTE = 5;
26 int LEN_ALGORITHMEN = 2;
27
28 int main(int argc, char *argv[]) {
29
30     long befehle_array[LEN_ALGORITHMEN][LEN_WERTE];
31     double laufzeit_array[LEN_ALGORITHMEN][LEN_WERTE];
32
33     for(int j = 0; j < LEN_WERTE; ++j)
34     {
35         int n = WERTE[j];
36
```

```
37          //lege Arrays der Länge n an
38          int* array_countsort = malloc(sizeof(int) * n);
39          int* array_insertionsort = malloc(sizeof(int) * n);
40
41          //fülle array_countsort mit Zufallswerten ..
42          fill_array_randomly(array_countsort, n, MAX_VALUE);
43          //.. und kopiere die erzeugten Werte in das Array array_insertionsort
44          copy_array_elements(array_insertionsort, array_countsort, n);
45
46          //teste ob beide Arrays auch wirklich die gleichen Werte enthalten
47          if(!check_equality_of_arrays(array_countsort, array_insertionsort, n)
               ↪ )
48          {
49              printf("Die Eingaben für beide Algorithmen müssen für die
                   ↪ Vergleichbarkeit gleich sein!\n");
50              return -1;
51          }
52
53          for(int i = 0; i < LEN_ALGORITHMEN; ++i)
54          {
55              int anzahl_befehle = 0;
56
57              start_timer();
58
59              //Aufruf der entsprechenden Sortieralgorithmen
60              if(i==0)
61              {
62                      count_sort(array_countsort, n, &anzahl_befehle);
63              }
64              else if(i==1)
65              {
66                      insertion_sort(array_insertionsort, n, &anzahl_befehle);
67              }
68
69              //speichere die Laufzeit sowie die Anzahl benötigter Befehle
70              laufzeit_array[i][j] = end_timer();
71              befehle_array[i][j] = anzahl_befehle;
72          }
73
74          //teste ob die Ausgabe beider Algorithmen gleich ist
75          if(!check_equality_of_arrays(array_countsort, array_insertionsort, n)
               ↪ )
76          {
77              printf("Die Arrays sind nicht gleich. Eines muss (falsch)
                   ↪ sortiert worden sein!\n");
78              return -1;
79          }
80
81          //gib den Speicherplatz wieder frei
82          free(array_countsort);
83          free(array_insertionsort);
84      }
85
86      //Ausgabe der Anzahl ausgeführter Befehle sowie der gemessenen Laufzeiten
```



```
               ↪  (in Millisekunden)
87      printf("Parameter MAX_VALUE hat den Wert %d\n", MAX_VALUE);
88      printf("\t %32s          %32s \n", "Countsort","Insertionsort");
89      printf("%8s \t %16s %16s \t %16s %16s \n", "n","Befehle", "Laufzeit","
               ↪ Befehle.","Laufzeit");
90
91      for(int j = 0; j < LEN_WERTE; ++j)
92      {
93          printf("%8d \t ",WERTE[j]);
94          for(int i = 0; i < LEN_ALGORITHMEN; ++i)
95          {
96              printf("%16ld %16.4f \t ",  befehle_array[i][j], laufzeit_array[i
                       ↪ ][j]);
97          }
98          printf("\n");
99      }
100
101     return 0;
102 }
```