Fakultät IV Elektrotechnik und Informatik
Einführung in die Programmierung WS 2021/2022
Manfred Hauswirth / Damien Foucard / Christoph Seifert / Uwe Kuehn

Programmierblatt 3

Ausgabe: *18.11.2021 12:00*
Abgabe: *30.11.2021 08:00*

Thema: *Laufzeitanalyse*

Abgabemodalitäten

1. Alle abzugebenden Quelltexte müssen ohne Warnungen und Fehler auf Deinem Rechner mit dem Befehl `clang -std=c11 -Wall -g` kompilieren.
2. Die Abgabe für den Quellcode erfolgt ausschließlich über unser Git im entsprechenden Branch. Nur wenn ein Ergebnis im ISIS-Kurs angezeigt wird, ist sichergestellt, dass die Abgabe erfolgt ist. Die Abgabe ist bestanden, wenn Du an Deinem Test einen grünen Haken siehst.
3. Du kannst bis zur Abgabefrist beliebig oft neue Versionen abgeben. Lies Dir die Hinweise der Tests genau durch, denn diese helfen Dir Deine Abgabe zu korrigieren.
Bitte beachte, dass ausschließlich die letzte Abgabe gewertet wird.
4. Die Abgabe erfolgt, sofern nicht anders angegeben, in folgendem Branch: `iprg-b<xx>-a<yy>`, wobei `<xx>` durch die zweistellige Nummer des Aufgabenblattes und `<yy>` durch die entsprechende Nummer der Aufgabe zu ersetzen sind.
5. Gib für jede Aufgabe die Quellcodedatei(en) gemäß der Vorgabe ab. Im ISIS-Kurs werden zum Teil Vorgabedateien bereitgestellt. Nutze diese zur Lösung der Aufgaben.
6. Die Abgabefristen werden vom Server überwacht. Versuche Deine Abgabe so früh wie möglich zu bearbeiten. Du minimierst so auch das Risiko, die Abgabefrist auf Grund von „technischen Schwierigkeiten“ zu versäumen. Eine Programmieraufgabe gilt als bestanden, wenn alle bewerteten Teilaufgaben bestanden sind.

Erläuterung: 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 der Aufgabe soll die hier vorgestellte “Zählweise” dann zur empirischen Analyse von einfachen `for`-Schleifen und im nächsten Aufgabenblatt für 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 drei – 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:

```c
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:

```c
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 ein Befehl gezählt werden, ist jede Ausführung des Vergleichs auch als ein 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 1 Einstieg Laufzeitanalyse: Befehlszähler (bewertet)

In dieser Aufgabe sollst Du gemäß der vorherigen Erläuterungen 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 Vorgabe. 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[1] – den Wert 1 zurück.

Innerhalb der `main()`-Funktion 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.

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 begründen. Sollte dies nicht befolgen, erhältst Du im Zweifelsfall keine Punkte.

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

- 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.

Nutze zur Lösung der Aufgabe die Vorgaben aus unserem ISIS-Kurs. Füge Deine Lösung als Datei `introprog_complexity_steps_intro.c` im entsprechenden Abgabebranch in Dein persönliches Repository ein und übertrage die Lösung an die Abgabeplattform.

Aufgabe 2 Laufzeitanalyse – Vergleich Count- und Insertionsort (bewertet)

In dieser Aufgabe sollst Du (empirisch) die Laufzeit Deiner Insertion- und Countsort-Implementierungen vergleichen. Ähnlich zur vorigen Aufgabe erhälst Du eine Vorgabe, in welcher die komplette `main`-Funktion bereits vorgegeben ist. Du musst nur noch Deine Insertion- sowie Countsort- Implementierungen 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[], unsigned int* befehle)`
2. `void count_sort_write_output_array(int output_array[], int len, int count_array[], unsigned int* befehle)`
3. `void count_sort(int array[], int len, unsigned int* befehle)`
4. `void insertion_sort(int array[], int len, unsigned int* befehle)`

Hierbei sollen die Funktionen 1, 2, und 4 die gleiche Funktionalität wie auf den Aufgabenblättern 1 und 2 haben, jedoch noch zusätzlich die ausgeführten Befehle mittels `int* befehle` zählen. Beachte zur Bestimmung der ausgeführten Befehle die Erläuterung und Hinweise aus Aufgabe 1.

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 ISIS-Kurs) erzeugt für Count- und Insertionsort 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.

Aufgabe und Abgabemodalitäten:

- Implementiere die vier Funktionen und werte die Laufzeit sowie die Anzahl der benötigten Befehle beider Algorithmen aus.
- Teste die Algorithmen für verschiedene Werte der Konstanten `MAX_VALUE`. Finde Kombinationen aus `MAX_VALUE` und der Größe des Arrays n, bei denen jeweils entweder Insertionsort oder Countsort vorzuziehen ist.

Hinweise:

- 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[]` oder die Konstante `MAX_VALUE` anzupassen. Dies kann z. B. nützlich sein, falls die Ausführung des Programms zu lange dauert.
- Du musst die main-Funktion keinerlei editieren.
- Bitte kommentiere Deinen Code genügend, um Klarkheit zu schaffen. Jedoch nicht zu viel (bspw. in jeder einzelnen Zeile), denn das würde Deinen Code wiederum unübersichtlicher machen.

Nutze zur Lösung der Aufgabe die Vorgaben aus unserem ISIS-Kurs. Füge Deine Lösung als Datei `introprog_complexity_steps_sorting.c` im entsprechenden Abgabebranch in Dein persönliches Repository ein und übertrage die Lösung an die Abgabeplattform.