Programmierkurs IPK WS22/23
Dr. Dominic Kempf
IWR, Universität Heidelberg

Aufgabenblatt 2
Abgabedatum 11. November 2022, 12 Uhr

Aufgabenblatt 2

Allgemeine Hinweise:

- Ab diesem Blatt ist eine Abgabe in Dreiergruppen verpflichtend.
- Alle drei Gruppenmitglieder müssen im gleichen Tutorium sein.
- Bei der Abgabe muss zusätzlich eine Datei votierpunkte.txt mitgeschickt werden, die für jede Aufgabe angibt, wer dafür Punkte bekommen soll (ein, zwei oder drei Namen angeben).

Aufgabe 1: Collatz-Vermutung (4 Punkte)

Schreiben Sie eine Funktion `void collatz(int number)`, die folgendes tut:

- Gib `number` auf dem Bildschirm aus.
- Falls `number` gerade ist, teile die Zahl durch 2.
- Andernfalls multipliziere die Zahl mit 3 und addiere 1.
- Wiederhole diese Schritte, bis einer der folgenden Zahlenwerte erreicht wird: $1, 0, -1, -5$ oder -17.
- Gib abschließend dieses Ergebnis aus.

Hinweis:
Um herauszufinden, ob eine Zahl x gerade ist, testen Sie, ob $(x \mod 2) = 0$. Hierfür gibt es in C++ den Operator `x % y`, der den Rest der Ganzzahl-Division von x durch y berechnet:

```
1 int x = 23 / 5; // 4 (runden nach 0)
2 int y = 23 % 5; // 3 (vorzeichenbehafteter Divisionsrest)
```

Rufen Sie die obige Funktion aus der `main`-Funktion auf, wobei Sie den ersten Wert für `number` von der Tastatur einlesen. Auf diese Weise können Sie die entstehenden Zahlenfolgen für verschiedene Startwerte untersuchen. Warum kann der Wert 0 nur auf eine Weise erreicht werden? Und was haben alle Zahlen gemeinsam, die zum Wert 1 führen? Informieren Sie sich unter https://en.wikipedia.org/wiki/Collatz_conjecture über die mathematische Vermutung hinter diesen Zahlenfolgen.

Aufgabe 2: Fibonacci-Folge (4 Punkte)

Jedes Element der Fibonacci-Folge wird durch Addition der beiden vorherigen Folgen-Elemente gebildet, wobei die ersten beiden Elemente durch 0 und 1 gegeben sind:

$$\begin{aligned} f_1 &= 0, \\ f_2 &= 1, \\ f_n &= f_{n-2} + f_{n-1}. \end{aligned}$$

Damit ergibt sich als Beginn der Folge:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

(a) Implementieren Sie die Berechnung der Folgen-Elemente in einem C++-Programm.

i. Schreiben sie eine Funktion `int fibonacci(int number)`, welche f_N berechnet. Lesen Sie N von der Kommandozeile ein und geben sie f_N aus.

ii. Erweitern Sie Ihr Programm so, dass es nacheinander f_n für alle Werte von 0 bis N auf der Kommandozeile ausgibt.

iii. Probieren Sie Ihr Programm für verschiedene Werte von N aus. Was passiert, wenn Sie N groß werden lassen? Haben Sie eine Erklärung hierfür?

Hinweis: Sie können die Geschwindigkeit Ihres Programms verbessern, indem Sie beim Kompilieren die Option `-O3` angeben. Dadurch analysiert der Compiler Ihr Programm und erzeugt ein optimiertes Programm, das normalerweise deutlich schneller ist.

(b) Im folgenden geht es darum, das Programm aus dem ersten Teil zu verbessern.

Anmerkung: Es kann natürlich sein, dass Sie den vorherigen Aufgabenteil schon so implementiert haben, dass hier gar keine Probleme auftreten. In diesem Fall sind Sie bereits fertig.

i. Sorgen Sie dafür, dass Ihr Programm auch für grössere N korrekt funktioniert. Schauen Sie sich hierzu die Folien zu Variablentypen an.

ii. Die Laufzeit des Programms steigt für ungefähr $N = 40$ sehr stark an. Woran liegt das? Schreiben Sie eine alternative Version des Programms, die dieses Problem umgeht und die Folgenglieder bis mindestens $N = 90$ ohne nennenswerte Verzögerung ausgeben kann.

Aufgabe 3: Positive Potenzen von ganzen Zahlen (4 Punkte)

In dieser Aufgabe sollen Sie positive Potenzen $n \in \mathbb{N}_0$ von ganzen Zahlen $q \in \mathbb{Z}$ berechnen:

$$q^n = \prod_{i=1}^{n} q = \underbrace{q \cdot q \cdots q}_{n-\text{mal}}, \qquad q^0 = 1.$$

(a) Schreiben Sie eine Funktion `int iterative(int q, int n)`, die q^n mit Hilfe einer Schleife berechnet.

(b) Schreiben Sie eine Funktion `int recursive(int q, int n)`, die q^n berechnet, indem sie sich wiederholt selbst mit anderen Argumenten aufruft. Hier müssen Sie eine geeignete Abbruchbedingung finden, bei der die Funktion sich nicht mehr weiter selbst aufruft.

(c) Eine naive Implementierung obiger Funktionen muss $n-1$ Multiplikationen durchführen. Können Sie eine bessere Implementierung finden? Sie können die verbesserte Version entweder iterativ oder rekursiv implementieren, was immer Ihnen einfacher erscheint. Tipp: $q^{2n} = (q^n)^2$.

Hinweise:

- Ihr Hauptprogramm soll q und n von der Kommandozeile einlesen und das Ergebnis ausgeben. Überprüfen Sie, ob die Eingabe gültig ist. Bei einem Fehler schreiben Sie eine Meldung nach `std::cerr` und geben `1` zurück.