Stefan Kurtz **Universität Hamburg**
Mathias von Behren **Zentrum für Bioinformatik**

Programmierung in der Bioinformatik
Wintersemester 2015
Übungen zur Vorlesung: Ausgabe am 26.10.2015

Punktevergabe:

- Aufgabe 3.1: 2 Punkt
- Aufgabe 3.2: 2 Punkte
- Aufgabe 3.3: 3 Punkte
- Aufgabe 3.4: 3 Punkte

Aufgabe 3.1 Präzedenzregeln

1. Werten Sie unter Berücksichtigung der Operator-Präzedenzregeln aus der Vorlesung die folgende Ausdrücke aus. Die verwendeten Variablen sind dabei jeweils wie folgt deklariert und initialisiert: `int a = 5, b = 2, c = 8, d = 11, e;`
   - `e = b = c = d`
   - `e = 3 * c++ - 6`
   - `d += a + c / b`
2. Vereinfachen Sie unter Berücksichtigung der Operator-Präzedenzregeln aus der Vorlesung die folgenden Ausdrücke soweit wie möglich, indem Sie Klammern streichen, ohne die Bedeutung des Ausdrucks zu verändern:
   - `(((a && b) || c) != d) ? (3+a) : (c)`
   - `(((a + b) / c) == (d - a))`
   - `a = b ? (c) : (3 + (d << 1))`

Aufgabe 3.2 In dieser Aufgabe sollen Sie die Regeln, die Sie in der Vorlesung zu gutem und lesbarem Code gelernt haben, auf das Programm in der Datei

`exponentialCalculation_obfuscated.c` anwenden.

Hier sind die wichtigsten Regeln:

- Fassen Sie ähnliche Funktionalitäten in Funktionen zusammen.
- Verwenden Sie aussagekräftige Variablen und Funktionsnamen.
- Verwenden Sie Konstanten mit aussagekräftigen Namen.
- Vermeiden Sie globale Variablen.
- Dokumentieren Sie Funktionen, aber vermeiden Sie Redundanz.
- Vermeiden Sie Funktionen mit Seiteneffekten.

Begründen Sie jede Änderung, die Sie durchführen.

Aufgabe 3.3 Die Quersumme einer Zahl ist die Summe der Ziffern aus der diese Zahl besteht. Implementieren Sie ein C-Programm `quersumme.x`, welches eine positive ganze Zahl als Parameter übergeben bekommt und mittels einer Funktion `quersumme` deren Quersumme berechnet und diese dann ausgibt.

Beachten Sie, dass Sie kein interaktives Programm schreiben sollen. Die Zahl wird als Parameter übergeben und Sie können über das Array `argv` darauf zugreifen.

Aufgabe 3.4 Im binären Zahlensystem wird eine natürliche Zahl als Summe von Zweierpotenzen dargestellt. Man schreibt diese Summe dann als Bitvektor, d.h. als eine Folge der Ziffern 0 und 1. Nehmen wir an, wir wollen alle Zahlen zwischen 0 und $2^{32} - 1$ darstellen. Dann benötigen wir 32 Bits. Die Bitvektoren haben also die Länge 32. Z.B. kann man die Dezimalzahl 42 durch den Bitvektor `00000000 00000000 00000000 00101010` darstellen, denn es gilt:

$$1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 32 + 0 + 8 + 0 + 2 + 0 = 42$$

In der Darstellung des Bitvektors wurde zur besseren Lesbarkeit nach jeweils 8 Bits ein Leerzeichen eingefügt.

Wir setzen die folgende Definition voraus:

```
#define NUMOFBITS 32    /* number of bits in bitvector */
```

Implementieren Sie die folgenden C-Funktionen zur Umwandlung von Dezimalzahlen in Bitvektoren.

```
void decimal2bitvector(char *bitvector, unsigned int n)
unsigned int bitvector2decimal(const char *bitvector)
```

Die C-Funktion `decimal2bitvector()` soll jede nicht negative Dezimalzahl (übergeben als zweiter Parameter) verarbeiten können. Der berechnete Bitvektor wird im Speicherbereich, auf den `bitvector` zeigt, abgelegt. Die Funktion `bitvector2decimal` soll den Bitvektor in einen Dezimalwert vom Typ **`unsigned int`** verwandeln und als **`return`**-Wert liefern. Beachten Sie, dass bei einem Bitvektor, der nicht nur aus den Zeichen `0` und `1` besteht, oder der zu lang ist, eine entsprechende Fehlermeldung ausgegeben wird und die Funktion abbricht.

Da `bitvector` ein String ist, sollte er mit `\0` enden. Wenn das 33. Zeichen des Vectors nicht `\0` ist, so ist der Vector zu lang.

Testen Sie Ihre Funktionen, indem Sie jede Zahl n, $0 \leq n \leq 2^{24} - 1$, in einen Bitvektor verwandeln, und diesen dann wieder in eine Dezimalzahl m. Wenn Ihre Implementierung korrekt ist, dann müssen natürlich n und m identisch sein.

Die Lösungen zu diesen Aufgaben werden am 09.11.2015 besprochen.