Universität der Bundeswehr
Fakultät für Informatik – Institut 2
Prof. Dr. Wolfgang Hommel
Michael Grabatin, Michael Steinke

WT 2020
Übungsblatt 6

Einführung in die Informatik II

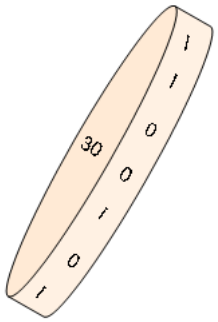
21. und 24.02.2019

1 Binare

In dieser Aufgabe beschäftigen wir uns mit einem Thema aus dem Bereich der Numismatik (=Münzkunde). Herkömmliche Automaten identifizieren Münzen zumeist anhand ihres Gewichtes und ihres Durchmessers. Diese Vorgehensweise macht es Fälschern natürlich sehr leicht. Nicht umsonst gibt es bei uns nur Münzen mit einem Maximalbetrag von 2 Euro.

Sie sind aus diesem Grunde beauftragt worden, einen Automaten zu programmieren, der sogenannte binarische Münzen identifizieren können soll. Binare sind Münzen, auf deren Rand ein Binärcode aufgeprägt ist, der den Wert der Münze eindeutig bestimmt. Eingelesen wird der Binärcode mittels eines Scanners ausgehend von einer nicht fest bestimmten Startposition.

Ihr Münz-Erkennungs-Algorithmus muss nun bei der Identifikation der zum Code gehörenden Münze berücksichtigen, dass Münzen sowohl gedreht als auch spiegelverkehrt eingeworfen werden können. Des Weiteren dürfen ungültige Münzen nicht akzeptiert werden.



Beispiel: Wir betrachten folgende 3 Münzen mit einer Codelänge von 4:

Münz-Code	Wert
0111	10
1001	20
0000	30

Da der Rand kein Anfang und kein Ende hat, haben die eingescannten Münzcodes `0111`, `1011`, `1101` und `1110` alle den selben Wert, nämlich 10.

Gegeben sind folgende Typdefinitionen:

```
1 type Bit = Boolean
2 val Zero = false
3 val One = true
4
5 //Muenzcode als Ringliste von Bits
6 class BitElem(var value : Bit = Zero, var next : BitElem = null)
```

a) Implementieren Sie zunächst die Funktionen, die eine Zeichenkette in einen Münzcode (also in eine Ringliste) umwandelt und umgekehrt einen Münzcode als String zurückgibt. Sie dürfen davon ausgehen, dass die Zeichenkette nur aus `0`en und `1`en besteht:

```
def code2String(code : BitElem) : String = {
  def bit2String(bit : Bit) = if (bit) "1" else "0"
  ...
}

def string2Code(code : String) : BitElem = {
  def char2Bit(c : Char) : Bit = if (c == '0') Zero else One
  ...
}
```

b) Implementieren Sie eine Funktion `equalsCoinCode`, die überprüft, ob zwei Münzcodes identisch sind. Ob zwei Münzcodes spiegelverkehrt zueinander sind und/oder durch Drehung auseinander hervorgehen, soll hierbei noch nicht betrachtet werden:

```
def equalsCoinCode(code1 : BitElem, code2 : BitElem) : Boolean = ...
```

c) Implementieren Sie nun eine Funktion, die zu einem übergebenen Münzcode den spiegelverkehrten Münzcode bestimmt:

```
def revCoinCode(code : BitElem) : BitElem = ...
```

d) Die Überprüfung auf Gleichheit zweier Münzcodes unter Berücksichtigung von „spiegelverkehrten“ sowie aus Drehung hervorgehenden Münzcodes kann auf verschiedene Art und Weise realisiert werden. Hier sollen zu vergleichende Münzcodes zunächst normiert werden. Eine geeignete Normierung ist die lexikographisch kleinste Bitrepräsentation.

Beispiel: Der Münzcode `1001` normiert ergäbe `0011`, der Münzcode `100110` ergäbe `001011` (hier spiegelverkehrt). Implementieren sie dazu eine Funktion `normCoinCode`, die einen Münzcode auf dieser Grundlage normiert.

```
def normCoinCode(code : BitElem) : BitElem = ...
```

Legen Sie dabei nur eine Kopie der originalen Liste an, wenn dies erforderlich ist.

Tipp: Implementieren Sie zunächst eine Hilfsfunktion, die lediglich die Drehung berücksichtigt. Im zweiten Schritt vergleichen Sie, ob sich der Münzcode lexikographisch weiter verkleinern lässt, wenn die Münze umgedreht wird.

e) Nun widmen wir uns der Anwendung. Es seien folgende Definitionen für die Münzen gegeben:

```
class Coin(val code : BitElem, val value : Int)
def createCoin(code : BitElem, value : Int) =
  new Coin(normCoinCode(code), value)

val coin10 = createCoin(string2Code("0111"), 10)
val coin20 = createCoin(string2Code("1001"), 20)
val coin30 = createCoin(string2Code("0000"), 30)

//Menge von Muenzen als einfach verkettete Liste mit Dummy
class CoinElem(val coin : Coin = null, var next : CoinElem = null)
def emptyCoinSet = new CoinElem
```

Implementieren Sie folgende Funktionen und Prozeduren. Dabei dürfen Sie annehmen, dass die Münzen mit `createCoin` erstellt wurden, die Münzcodes also bereits normiert sind.

```
1 //Muenzen in der Form "Muenze mit Wert von 10 (0111)" ausgeben
2 def printCoinSet(coinSet : CoinElem) : Unit ...
3
4 //Einer Muenzmenge eine Muenze hinzufuegen
5 def addCoin(coinset : CoinElem, coin : Coin) : Unit = ...
6
7 //Wert anhand des gelesenen Codes einer eingeworfenen Muenze erkennen
8 def getValue(coinset : CoinElem, code : BitElem) : Int = ...
```

Hinweis: Da der Zeiger auf eine Münzmenge immer erhalten bleiben soll, ist hier ein Dummy-Element erforderlich. Dieses Element ist bei sämtlichen Funktionen und Prozeduren zu berücksichtigen.

2 Komplexitätsabschätzung mit O-Notation

In dieser Aufgabe sollen einige Algorithmen auf ihre Laufzeitkomplexität hin untersucht werden. Neben der theoretischen Betrachtung sollen auch experimentelle Messungen durchgeführt werden.

a) Betrachten Sie die folgenden Funktionen und ermitteln Sie deren durchschnittliche, worst-case und best-case Laufzeitkomplexität in O-Notation nur durch Betrachtung des gegebenen Codes: Geben Sie ebenfalls an wie der worst-case bzw. der best-case aussehen.

- Funktion 1

```
def head(a: Array[Int]): Int = {
    if(a.length == 0) return 0
    else return a(0)
}
```

- Funktion 2

```
def last(a: Array[Int]): Int = {
    var res = a(0)
    for(i <- 0 until a.length) {
        res = a(i)
    }
    return res
}
```

- Funktion 3

```
def selectionSort(list: Array[Int]): Unit = {
  def swap(list: Array[Int], i: Int, j: Int) = {
    var tmp = list(i)
    list(i) = list(j)
    list(j) = tmp
  }

  var i = 0
  while(i < (list.length - 1)) {
    var min = i
    var j = i + 1

    while (j < list.length) {
      if(list(j) < list(min)) {
        min = j
      }
      j += 1
    }

    swap(list, i, min)
    i += 1
  }
}
```

- Funktion 4

```
def quickSort(xs: Array[Int]): Array[Int] = {
    if (xs.length <= 1) xs
    else {
```

```
        val pivot = xs(xs.length / 2)
        Array.concat(
            quickSort(xs filter (pivot >)),
            xs filter (pivot ==),
            quickSort(xs filter (pivot <)))
    }
}
```

- Funktion 5

```
def bubbleSort(array: Array[Int]): Array[Int] = {
  var didSwap = false

  for(i <- 0 until array.length - 1)
    if(array(i+1) < array(i)){
      val temp = array(i)
      array(i) = array(i+1)
      array(i+1) = temp
      didSwap = true
    }

  // Repeat until we don't have anymore swaps
  if(didSwap)
    bubbleSort(array)
  else
    array
}
```

b) Testen Sie die Laufzeitkomplexität der gegebenen Algorithmen mit der oben gegebenen Zeitmessungsfunktion und tragen Sie ihre Messergebnisse in einen Graphen ein.

Zum Messen der Ausführungszeit einer beliebigen Funktion können die folgenden Funktionen verwendet werden. Die Funktion, deren Laufzeit gemessen werden soll, wird als Call-by-Name Parameter an die `measure` -Funktion übergeben. Dadurch wird sie erst dann ausgeführt, wenn sie zwischen den zwei Messpunkten ausgewertet wird. Ein zweiter Parameter kann gesetzt werden um bei der Ausgabe die gemessene Funktion zu identifizieren.

Damit bei Messungen von Funktionen deren Ausführung sehr schnell geht weniger Messfehler auftreten ist es eine gute Idee, diese öfters zu wiederholen und einen durchschnittlichen Messwert zu bestimmen. Dabei hilft die zweite Funktion `averageMeasure`. Dieser Funktion kann eine Vorbereitungsfunktion übergeben werden, deren Ausführung nicht gemessen wird, sowie die Funktion deren Ausführung mit den von der Vorbereitungsfunktion erstellten Daten gemessen wird. Der letzte Parameter gibt an wie viele Ausführungen gemittelt werden sollen.

```
def measure[A](f: => A, name: String = "f"): Long = {
    var t1 = System.nanoTime()
    f // f is passed as Call-by-Name and is run
      // here between the measurement points
    var t2 = System.nanoTime()
    var diff = (t2-t1)
    println(s"Execution of $name took ${(diff)/1000000000}s "+
     s"(${diff}ns).")
    return diff
}

def averageMeasure[A,B](prep: => A, test: A => B, c: Int): Long = {
    var sum = 0L
```

```
    for(i <- 1 to c) {
        var a = prep
        sum += measure(test(a))
    }
    println(s"Execution of $c runs took ${sum / c}ns on average")
    return sum / c
}
```

Ein Aufruf der Kombination kann zum Beispiel wie folgt aussehen:

```
averageMeasure(genRandArray(100000), bubbleSort, 3)
//> Execution of f took 16s (16383466120ns).
//> Execution of f took 16s (16322965376ns).
//> Execution of f took 16s (16345057231ns).
//> Execution of 3 runs took 16350496242 on average
// res45: Long = 16350496242L
```

Die folgende Funktion kann verwendet werden um ein Array der Länge c mit zufälligen Werten zu erstellen.

```
def genRandArray(c: Int): Array[Int] = {
    var r = new scala.util.Random
    var min = 0
    var max = 100
    return Array.tabulate(c)( _ => min + r.nextInt((max - min) + 1))
}
```