

Einführung in die Informatik II

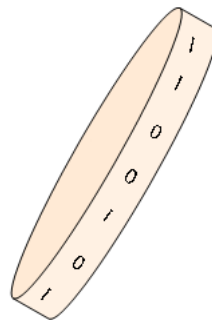
21. und 24.02.2019

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. Daher eignet es sich den Code als Ringliste darzustellen. Bei einer Ringliste zeigt der „next“ Zeiger des „letzten“ Elements wieder auf das erste Element. D. h. prinzipiell kann an jeder Stelle in der Liste eingestiegen werden und durch Folgen des “next“ Zeigers erreicht man, nachdem über alle Elemente der Liste iteriert wurde, wieder das Startelement.

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 0en und 1en besteht:

```
1 def code2String(code : BitElem) : String = {
2   def bit2String(bit : Bit) = if (bit) "1" else "0"
3   ...
4 }
5
6 def string2Code(code : String) : BitElem = {
7   def char2Bit(c : Char) : Bit = if (c == '0') Zero else One
8   ...
9 }
```

Mögliche Implementierungen lauten wie folgt:

```
1 def code2String(code : BitElem) : String = {
2   def bit2String(bit : Bit) = if (bit) "1" else "0"
3
4   // Zeiger auf das Startelement
5   val start = code
6   var p = start.next
7   var rtn : String = bit2String(start.value)
8
9   // Solange durch die Liste iterieren, bis wir wieder am
10  // Startelement sind. (Anstelle von start könnte hier auch
11  // direkt 'p != code' verwendet werden)
12  while (p != start) {
13    rtn += bit2String(p.value)
14    p = p.next
15  }
16  return rtn
17 }
18
19 def string2Code(code : String) : BitElem = {
20   def char2Bit(c : Char) : Bit = if (c == '0') Zero else One
21
22   var dummy = new BitElem
23   var p = dummy
24
25   for (b <- code) {
26     p.next = new BitElem(char2Bit(b))
27     p = p.next
28   }
29   p.next = dummy.next //Ring "schliessen"
30   return dummy.next
31 }
```

```
13 | }
```

- 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:

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

Hier muss Wert für Wert verglichen werden:

```
1 | def equalsCoinCode(code1 : BitElem, code2 : BitElem) : Boolean =  
  | {  
2 |   var p1 = code1; var p2 = code2  
3 |  
4 |   while (p1.value == p2.value) {  
5 |     p1 = p1.next  
6 |     p2 = p2.next  
7 |  
8 |     //Sind eine oder beide Listen zu Ende?  
9 |     if (p1 == code1) return p2 == code2  
10 |    else if (p2 == code2) return false  
11 |   }  
12 |   return false  
13 | }
```

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

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

Zur Umkehrung der Liste wird die Originalliste `code` Element für Element durchlaufen, bis der Anfang erreicht wird. Dabei wird Schritt für Schritte eine neue Liste `rtn` erstellt, indem jedes Element kopiert und **vorne** an `rtn` angehängt wird. Am Ende ist nur noch sicherzustellen, dass der Ring geschlossen ist.

```
1 | def revCoinCode(code : BitElem) : BitElem = {  
2 |   var rtn = new BitElem(code.value)  
3 |  
4 |   var pOrig = code.next  
5 |   var pRtn = rtn  
6 |  
7 |   while (pOrig != code) {  
8 |     pRtn = new BitElem(pOrig.value, pRtn)  
9 |     pOrig = pOrig.next  
10 |  }  
11 |  
12 |  rtn.next = pRtn //Ring "schliessen"  
13 |  return pRtn  
14 | }
```

- 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.

```
1 | 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.

Zunächst implementieren wir eine Funktion `lexSmaller`, die überprüft, ob eine gegebene Ringliste `code1` lexikographisch kleiner ist als eine zweite Ringliste `code2`. Obwohl wir hier von zwei gleich langen Listen ausgehen könnten, berücksichtigt die folgende Implementierung auch unterschiedlich lange Ringlisten. Die Liste `code1` gilt als kleiner, wenn ihre Elemente ein echtes Anfangsstück von `code2` bilden (und damit `code2` länger ist als `code1`):

```
1 | def lexSmaller (code1 : BitElem, code2 : BitElem) : Boolean = {
2 |   var p1 = code1
3 |   var p2 = code2
4 |
5 |   while (p1.value == p2.value) {
6 |     p1 = p1.next
7 |     p2 = p2.next
8 |
9 |     if (p1 == code1) return p2 != code2 //true wenn code2
        laenger
10 |    else if (p2 == code2) return false  //code1 ist laenger
11 |   }
12 |   return p1.value < p2.value
13 | }
```

Als nächstes implementieren wir eine Funktion `normCoinCodeSimple`, die ohne Berücksichtigung der Spiegelverkehrtheit die kleinste lexikographische Darstellung eines gegebenen Codes zurückliefert. Der naive Ansatz hierbei wäre, die Liste immer um ein Element weiterzudrehen und anschließend zu überprüfen, ob die „neue“ Liste „kleiner“ ist. Wenn man sich allerdings überlegt, dass die kleinstmögliche Darstellung (sofern sie nicht nur aus 0en oder nur aus 1en besteht) in jedem Fall mit einer 0 beginnen und mit einer 1 enden muss, lässt sich eine gewisse Vereinfachung erreichen:

- Man sucht die erste 0 unmittelbar nach einer 1
- Gibt es keine, dann ist die Darstellung bereits kleinstmöglich (da sie nur aus 0en oder nur aus 1en besteht).
- Man hält die vorerst kleinste Darstellung fest.
- Solange man nicht den Beginn der Ringliste überschreitet, sucht man immer die nächste 0 nach einer 1 und überprüft, ob dadurch eine Verbesserung erreicht wird.
- Zum Schluss gibt man die zuletzt gefundene beste Darstellung zurück.

In Scala kann dies wie folgt aussehen:

```
1 | def normCoinCodeSimple (code : BitElem) : BitElem = {
2 |   var p = code
3 |
4 |   //Naechste 0 finden, sofern vorhanden
5 |   //Vorbedingung: p zeigt auf eine 1
```

```

6  def nextZero() : Boolean = {
7      p = p.next
8      while (p != code && p.value == One) p = p.next
9      return p != code //bei false keine 0 mehr
10 }
11
12 //Erste 1 finden sofern vorhanden
13 while (p.value != One) {
14     p = p.next
15     if (p == code) return code
16 }
17
18 //Erste 0 nach der 1 finden wenn vorhanden
19 if (!nextZero()) return code
20
21 //Gegenwaertig beste Darstellung
22 var best = p
23
24 p = p.next
25
26 //Code enthaelt nun mind eine 1 und eine 0
27 //Versuchen eine bessere Dastellung zu finden
28 while (p != code) {
29     //naechste 1 finden
30     while (p != code && p.value != One) p = p.next
31
32     //naechste 0 nach der 1 finden
33     if (p != code && nextZero()) {
34         //Verbesserung?
35         if (lexSmaller(p, best)) best = p
36     }
37 }
38
39 return best
40 }

```

Hinweis: Im beigelegten Quelltext ist noch eine Variante dieser Funktion `normCoinCodeSimple2` vorhanden. Hierbei wird ausgenutzt, dass sich die Münzcodes als Bitcodierungen auffassen und sich somit in eine Ganzzahl überführen lassen und dass die lexikographisch kleinstmögliche Darstellung auch die kleinstmögliche Ganzzahl liefert. Dabei werden mehrere bitweise Operationen (Shift (<<), Negation (~), Oder (|), Und (&)) auf Ganzzahlen durchgeführt.

Abschließend lässt sich nun die Funktion `normCoinCodeSimple` implementieren:

```

1  def normCoinCode(code : BitElem) : BitElem = {
2      var best1 = normCoinCodeSimple(code)
3      var best2 = normCoinCodeSimple(revCoinCode(code))
4
5      return if (lexSmaller(best1, best2)) best1 else best2
6  }

```

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

```

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

```

```

4
5 val coin10 = createCoin(string2Code("0111"), 10)
6 val coin20 = createCoin(string2Code("1001"), 20)
7 val coin30 = createCoin(string2Code("0000"), 30)
8
9 //Menge von Muenzen als einfach verkettete Liste mit Dummy
10 class CoinElem(val coin : Coin = null, var next : CoinElem = null)
11 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.

Die Ausgabe kann wie folgt implementiert werden (Dummy-Element wird übergangen):

```

1 def printCoinSet(coinSet : CoinElem) : Unit = {
2   var p = coinSet.next
3
4   while (p != null) {
5     val coin = p.coin
6     println("Muenze mit Wert von " + coin.value
7       + " (" + code2String(coin.code) + ")")
8     p = p.next
9   }
10 }

```

Zum Hinzufügen: Da eine Menge verwendet wird, jedes Element daher nur einmal vorhanden sein soll, ist die Menge zu durchlaufen und zu überprüfen, ob das Element bereits existiert. Wurde das Element beim Durchlaufen bis zum Schluss nicht angetroffen, dann kann das Element neu hinzugefügt werden:

```

1 def addCoin(coinset : CoinElem, coin : Coin) : Unit = {
2   val code = coin.code
3   var p = coinset
4
5   //Muenzcode bereits vorhanden?
6   while (p.next != null) {
7     p = p.next
8     if (equalsCoinCode(p.coin.code, coin.code)) return
9   }
10
11   //Es gibt keines => hinten anfüegen
12   p.next = new CoinElem(coin)
13 }

```

Zur Bestimmung des Wertes: Die Menge wird durchsucht, ob der übergebene Münzcode gefunden wird.

Ist dies der Fall, wird der entsprechende Geldwert zurückgegeben. Anderenfalls wird 0 zurückgeliefert:

```
1 def getValue(coinset : CoinElem, code : BitElem) : Int = {  
2   var p = coinset.next  
3   var nc = normCoinCode(code)  
4   while (p != null) {  
5     if (equalsCoinCode(nc, p.coin.code)) return p.coin.value  
6     p = p.next  
7   }  
8   return 0  
9 }
```

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

```
1 | def head(a: Array[Int]): Int = {  
2 |     if(a.length == 0) return 0  
3 |     else return a(0)  
4 | }
```

Die `head`-Funktion hat immer eine konstante Laufzeit: $O(1)$

- Funktion 2

```
1 | def last(a: Array[Int]): Int = {  
2 |     var res = a(0)  
3 |     for(i <- 0 until a.length) {  
4 |         res = a(i)  
5 |     }  
6 |     return res  
7 | }
```

Die `last`-Funktion hat immer eine lineare Laufzeit: $O(n)$

- Funktion 3

```
1 | def selectionSort(list: Array[Int]): Unit = {  
2 |     def swap(list: Array[Int], i: Int, j: Int) = {  
3 |         var tmp = list(i)  
4 |         list(i) = list(j)  
5 |         list(j) = tmp  
6 |     }  
7 |  
8 |     var i = 0  
9 |     while(i < (list.length - 1)) {  
10 |         var min = i  
11 |         var j = i + 1  
12 |  
13 |         while (j < list.length) {  
14 |             if(list(j) < list(min)) {  
15 |                 min = j  
16 |             }  
17 |             j += 1  
18 |         }  
19 |  
20 |         swap(list, i, min)  
21 |         i += 1  
22 |     }  
23 | }
```


Selection Sort hat immer eine quadratische Laufzeit: $O(n^2)$

• Funktion 4

```
1 def quickSort(xs: Array[Int]): Array[Int] = {
2   if (xs.length <= 1) xs
3   else {
4     val pivot = xs(xs.length / 2)
5     Array.concat(
6       quickSort(xs filter (pivot >)),
7       xs filter (pivot ==),
8       quickSort(xs filter (pivot <)))
9   }
10 }
```

Quick Sort hat (in dieser Implementierung) immer eine überlogarithmische Laufzeit: $O(n \log(n))$

• Funktion 5

```
1 def bubbleSort(array: Array[Int]): Array[Int] = {
2   var didSwap = false
3
4   for(i <- 0 until array.length - 1)
5     if(array(i+1) < array(i)) {
6       val temp = array(i)
7       array(i) = array(i+1)
8       array(i+1) = temp
9       didSwap = true
10    }
11
12    // Repeat until we don't have anymore swaps
13    if(didSwap)
14      bubbleSort(array)
15    else
16      array
17 }
```

Bubble Sort hat eine quadratische durchschnittliche und worst-case Laufzeit: $O(n^2)$

Die best-case Laufzeitkomplexität $O(n)$ hat Bubble Sort auf einer bereits sortierten Liste.

- 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.

```

1 | def measure[A] (f: => A, name: String = "f"): Long = {
2 |     var t1 = System.nanoTime()
3 |     f // f is passed as Call-by-Name and is run
4 |     // here between the measurement points
5 |     var t2 = System.nanoTime()
6 |     var diff = (t2-t1)
7 |     println(s"Execution of $name took ${(diff)/1000000000}s "+
8 |         s"${diff}ns.")
9 |     return diff
10 | }
11 |
12 | def averageMeasure[A,B] (prep: => A, test: A => B, c: Int): Long = {
13 |     var sum = 0L
14 |     for(i <- 1 to c) {
15 |         var a = prep
16 |         sum += measure(test(a))
17 |     }
18 |     println(s"Execution of $c runs took ${sum / c}ns on average")
19 |     return sum / c
20 | }

```

Ein Aufruf der Kombination kann zum Beispiel wie folgt aussehen:

```

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

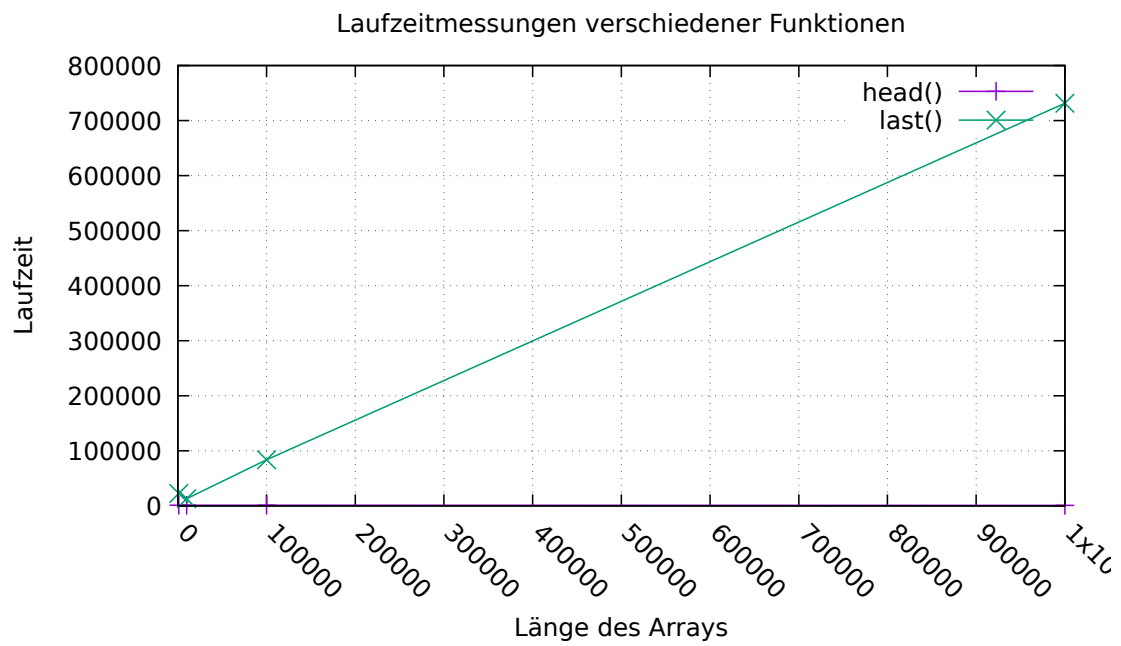
```

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

```

1 | def genRandArray(c: Int): Array[Int] = {
2 |     var r = new scala.util.Random
3 |     var min = 0
4 |     var max = 100
5 |     return Array.tabulate(c) ( _ => min + r.nextInt((max - min) + 1))
6 | }

```



Bitte beachten Sie, dass in der folgenden Grafik beide Achsen logarithmisch angetragen sind.

