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

WT 2020
Übungsblatt 2

Einführung in die Informatik II

24. und 27.01.2020

1 Permutationen – Ordnung

Wir bezeichnen eine einstufige Reihung P mit n Elementen als *Permutation*, wenn jeder der Indizes $0, 1, ..., n-1$ von P genau einmal als Wert in P vorkommt.

a) Vervollständigen Sie die folgende Funktion, die mit Hilfe einer charakteristischen Funktion cfm prüft, ob der Parameter P in diesem Sinn eine Permutation ist.

```
1 def isPerm(p : Array[Int]) : Boolean = {
2   val n = p.length
3   var cfm = Array.fill(n)(false)
4   //...
5 }
```

Zwei mögliche Ansätze sind im folgenden gezeigt:

Lösungsvorschlag 1:

```
1  //Bestimmen, ob p eine Permutation ist
2  def isPerm1(p : Array[Int]) : Boolean = {
3    val n = p.length
4    var cfm = Array.fill(n)(false)
5
6    var cnt = 0
7    //Elemente von p durchlaufen, solange sie zwischen 0 und n - 1
8    //liegen und noch nicht aufgetreten sind
9    for (i <- p; if i >= 0 && i < n && !cfm(i)) {
10     cnt = cnt + 1
11     cfm(i) = true
12   }
13
14   //Alle n Elemente vorhanden?
15   return cnt == n
16 }
```

Lösungsvorschlag 2:

```
1 //Bestimmen, ob p eine Permutation ist
2 def isPerm2(p : Array[Int]) : Boolean = {
3   val n = p.length
4   var cfm = Array.fill(n)(false)
5
6   //Elemente von p durchlaufen, solange sie zwischen 0 und n - 1
7   //liegen
```

```
8     for (i <- p; if i >= 0 && i < n) {
9       cfm(i) = true
10    }
11
12    //Alle Elemente vorhanden?
13    return cfm.foldLeft(true)(_ & _)
14  }
```

b) Geben Sie die lexikographisch größte Permutation der Zahlen $0, 1, ..., 8$ an.

Die lexikographisch größte Permutation ist

0	1	2	3	4	5	6	7	8
8	7	6	5	4	3	2	1	0

.

c) Erläutern Sie, warum die Permutationen

0	1	2	3	4	5	6	7	8
0	3	5	1	8	4	7	6	2

und

0	1	2	3	4	5	6	7	8
0	3	5	1	8	6	2	4	7

lexikographisch unmittelbar direkt aufeinander folgen.

Offensichtlich ist die erste Permutation lexikographisch kleiner als die zweite, da sich sich in den ersten fünf Stellen (Indizes 0 bis 4) nicht unterscheiden und an sechster Stelle (Index 5) 4 kleiner als 6 ist.

Es bleibt zu zeigen, dass es keine Permuation gibt, die lexikographisch zwischen den vorhandenen Permutationen liegt. Nimmt man an, es gäbe eine solche, dann müsste auch diese mindestens in den ersten fünf Stellen übereinstimmen, d.h. die gesuchte Permutation dürfte sich maximal in den letzten vier Stellen (Indizes 5 bis 8) unterscheiden. Da die letzten drei Stellen der ersten Permutation absteigend sortiert sind $(7, 6, 2)$, wird durch eine Umsortierung der letzten drei Stellen nur eine lexikographisch niedrigere Permutation entstehen. Daher muss zwangsläufig die Stelle bei Index 5 verändert, genaugenommen geringstmöglich erhöht werden. Als mögliche Werte kommen nur Werte in Frage, die rechts von Index 5 stehen. Der geringsmögliche ist der Wert 6. Ordnet man die verbleibenden Werte $(2, 4, 7)$ so an, dass die Permutation lexikographisch möglichst klein wird, dann erhält man die zweite Permutation. Sie folgt also tatsächlich lexikographisch unmittelbar der ersten.

Von diesen Überlegungen ausgehend, lässt sich zu einer beliebigen (lexikographisch nicht größten) Permutation P die lexikographisch unmittelbar nächste Permutation Q wie folgt bestimmen:

1. Man beginnt von rechts nach links und bestimmt die Stelle k, so dass $P(k+1) > P(k)$.
   Hinweis: Die Folge $P(k+1), ..., P(n-1)$ ist dann absteigend sortiert.
2. Man bestimme den Index $l > k$, so dass für alle $m > k$ mit $m \neq l$ gilt: $P(k) < P(l) < P(m)$.
   Hinweis: $P(l)$ ist dann der kleinste Wert, der rechts von k steht und größer als $P(k)$ ist.
3. Nun sortiert man die Folge $P(m)$ mit $m \geq k$ und $m \neq l$ aufsteigend und erhält die Folge $Q(k+1), ..., Q(n-1)$. Zusammen mit $Q(k) = P(l)$ und $Q(i) = P(i)$ für alle $0 \leq i < k$ ergibt sich die Permutation Q als Folge $Q(0), ..., Q(n-1)$.
   Hinweis: Da die Folge $P(m)$ mit $m > k$ bereits absteigend sortiert ist und diese Sortierung beibehalten wird, wenn man $P(l)$ mit $P(k)$ vertauscht, kann die notwendige Sortierung (nach der erfolgten Vertauschung) auf einfache Weise durch Umkehren des Indexbereiches $k+1$ bis $n-1$ erreicht werden.

d) Geben Sie zu

0	1	2	3	4	5	6	7	8
0	3	2	1	5	8	7	6	4

die lexikographisch nächste Permutation an.

Nehmen wir uns obige Überlegungen zu Nutze:

1. Für $k = 4$ gilt $P(k+1) = 8 > 4 = P(k)$. Insbesondere existiert kein $m > k$ für welches gilt $P(m+1) > P(m)$. Die Folge $P(5), ..., P(8)$ ist zudem absteigend sortiert.
2. Mit $l = 7$ ist der kleinste Wert $P(l) = 6$ gefunden, der größer ist als $P(k) = 5$.
3. Vertauscht man nun $P(l) = 6$ und $P(k) = 5$ und kehrt die Folge der Werte mit den Indizes $k+1 = 5$ bis $n-1 = 8$ um, dann erhält man die lexikographisch nächstgrößere Folge.

$$\left|\begin{array}{ccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 0 & 3 & 2 & 1 & 6 & 4 & 5 & 7 & 8 \end{array}\right|.$$

e) Entwickeln Sie ein Scala-Programmstück, welches zu einer im Array P gegebenen Permutation im Array Q die nächstgrößere Permutation herstellt.

Unter Berücksichtigung der Überlegungen aus der Teilaufgabe c lässt sich in Scala folgende Implementierung realisieren:

```
1  def nextPerm(p : Array[Int], q : Array[Int]) : Boolean = {
2    val n = p.length
3
4    var last = -1
5    var k = n - 1
6
7    //Von rechts beginnend Index k bestimmen, so dass gilt
8    //Fuer alle m > k: p(m) > p(k)
9    //Fuer spaetere Sortierung sicherstellen,
10   //dass fuer alle m > k gilt q(m)=p(m)
11   while (k >= 0 && p(k) > last) {
12     last = p(k)
13     q(k) = p(k)
14     k = k - 1
15   }
16
17   //Test auf lexikographisch groesste Permutation
18   if (k >= 0) {
19     //Index l bestimmen, so dass fuer alle m > k mit
20     //m != l p(m) > p(l) > p(k)
21     var l = k
22     while (l + 1 < n && p(l + 1) > p(k)) {
23       l += 1
24     }
25
26     //Indizes 0 bis k - 1 bleiben identisch
27     for (i <- 0 to k - 1) {
28       q(i) = p(i)
29     }
30
31     //Stelle k und l miteinander vertauschen
32     q(k) = p(l)
33     q(l) = p(k)
34
35     //Bereich in q zwischen k + 1 und n - 1 invertieren
36     for (i <- 0 to (n - 1 - k) / 2 - 1) {
```

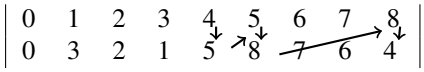
```
37          val h = q(k + 1 + i)
38          q(k + 1 + i) = q(n - 1 - i)
39          q(n - 1 - i) = h
40        }
41        return true
42
43      }
44      else return false
45    }
```

f) Testen Sie das Programmstück ebenso wie das Ergebnis von Teilaufgabe *a* mit geeigneten Daten am eigenen Rechner.

2 Permutation – Zyklus

Sei P eine Permutation der Länge n und sei i einer der Indizes $0, 1, ..., n-1$ von P. Durch wiederholte Anwendung von P auf i erhält man den *Zyklus* von i in P.

Beispiel: In der Permutation aus Aufgabe 1d findet man zu 4 den Zyklus $(4, 5, 8)$, weil P 4 auf 5, 5 auf 8 und 8 auf 4 abbildet.



a) Geben Sie zu allen Permutationen aus der Angabe von Aufgabe 1 jeweils alle verschiedenen Zyklen an (der Zyklus von 8 ist offenbar gleich dem Zyklus von 4 bzw. von 5).

Die Zyklen der drei Permutationen (ohne Indizes):

Permutation	**Zyklen**
(0, 3, 5, 1, 8, 4, 7, 6, 2)	(0), (1, 3), (2, 5, 4, 8), (6, 7)
(0, 3, 5, 1, 8, 6, 2, 4, 7)	(0), (1, 3), (2, 5, 6), (4, 8, 7)
(0, 3, 2, 1, 5, 8, 7, 6, 4)	(0), (1, 3), (2), (4, 5, 8), (6, 7)

b) Gegeben seien die Typvereinbarungen

```
1 type Perm = Array[Int]
2 type Cycles = List[List[Int]]
```

Dann berechnet die folgende Funktion zu einer gegebenen Permutation alle seine Zyklen:

```
1  def cyclesOf(p : Perm) : Cycles = {
2    val n = p.length
3    var cs : Cycles = List()
4    var cfm = Array.fill(n)(false)
5    for (i <- 0 to n - 1) {
6      if (!cfm(i)) {
7        cfm(i) = true
8        var cycle = List(i)
9        var j = p(i)
10       while (j != i) {
11         cfm(j) = true
12         cycle = cycle ::: List(j)
13         j = p(j)
14       }
15       cs = cs ::: List(cycle)
16     }
17   }
18   return cs
19 }
```

Die folgende Prozedur gibt die Zyklen einer Permutation aus:

```
1 def printCycles(p : Perm) : Unit = {
2   for (c <- cyclesOf(p)) println(c.mkString("(", ", ", ")"))
3 }
```

Was gibt `printCycles(P)` für alle Permutationen aus der Angabe von Aufgabe 1 aus?

Siehe Teilaufgabe a).

c) Aus den mit `cyclesOf(P)` berechneten Zyklen lässt sich die Permutation P zurückgewinnen. Vervollständigen Sie dazu folgende Scala-Funktion:

```
1 def permByCycles(cs : Cycles) : Perm = {
2   //groessten enthaltenen Zahlenwert in cs bestimmen
3   def maxEl(cs: Cycles): Int = {
4   ...
5   }
6   var p = Array.fill(maxEl(cs) + 1)(0)
7   ...
8   return p
9 }
```

Eine mögliche Implementierung:

```
1  def permByCycles(cs : Cycles) : Perm = {
2    //groessten Zahlenwert in cs
3    def maxEl(cs: Cycles): Int = {
4      cs.flatten.reduceLeft(math.max(_, _))
5    }
6
7    var p = Array.fill(maxEl(cs) + 1)(0)
8
9    for (c <- cs) {
10     var i = c.head
11     for (j <- c.tail) {
12       p(i) = j
13       i = j
14     }
15     p(i) = c.head
16   }
17
18   return p
19 }
```

3 Damenproblem

Die folgenden gegebenen Permutationen stellen Lösungen des Damenproblems, wie in der Vorlesung (Kap. 1, Folie 54) definiert, dar.

```
1 val P = Array(7, 3, 0, 2, 5, 1, 6, 4)
2 val Q = Array(5, 2, 4, 6, 0, 3, 1, 7)
```

a) Die Funktion `spiegle` spiegelt eine beliebige Permutation an der Hauptdiagonalen. Wir betrachten Aufrufe der folgenden **fehlerhaften** Implementierung:

```
1 def spiegleF(p: Perm): Perm = {
2   var res = Array.fill(8)(0)
3   for (i <- 0 to 7) res(P(i)) = i
4   return res
5 }
```

Der Aufruf `spiegleF(P)` ergibt die (korrekte) Antwort `Array(2, 5, 3, 1, 7, 4, 6, 0)`. Das gleiche Ergebnis liefern aber auch die Aufrufe `spiegleF(spiegleF(P))` und `spiegleF(Q)`. Warum?

> Der Parameter `p` (*kleingeschrieben*) wird in Funktion `spiegleF` gar nicht verwendet. Stattdessen nutzt die Funktion die Konstante `P` (*großgeschrieben*), welche im Block 1 definiert ist. Daher entspricht jeder Aufruf von `spiegleF(...)` (egal mit welchem Argument) einem Aufruf der Methode `spiegle` mit der Konstanten `P` als übergebenem Argument.

b) Mit folgendem beispielhaften Aufruf wollen wir feststellen, ob die Permutation P lexikographisch kleiner ist als die Permutation Q.

```
1 lexKleiner(P, Q)
```

Ergänzen Sie die folgende Definition entsprechend!

```
1 def lexKleiner(p : Perm, q : Perm) : Boolean = ...
```

> Ein mögliche Implementierung:
>
> ```
> 1 def lexKleiner(p : Perm, q : Perm) : Boolean = {
> 2   val n = p.length
> 3   var l = 0;
> 4   while (l < n &&  p(l) == q(l)) l += 1
> 5
> 6   return l < n && p(l) < q(l)
> 7 }
> ```

c) Eine Lösung P des Damenpromblems gelte als „neu“, wenn sie sich **nicht** durch Spiegelungen und/oder Drehungen in eine lexikographisch kleinere Permutation transformieren lässt. Vervollständigen Sie in diesem Sinn die folgende Definition!

```
1 def drehe(p: Perm): Perm = {
2     var Erg = Array.fill(8)(0)
3     for(i <- (0 to 7)) Erg(p(i)) = p.length - 1 - i
4     return Erg
5 }
6
7 def istNeu(p : Perm) : Boolean = ...
```

Hinweis: Aus der Vorlesung bekannte Prozeduren und Funktionen können verwendet werden.

Dazu dreht man die Lösung P dreimal. Nach jeder Drehung wird überprüft, ob die neue Lösung oder die Spiegelung davon lexikographisch kleiner ist. Ist dies der Fall, ist P keine neue Lösung. Eine mögliche Implementierung ist wie folgt:

```
1  def istNeu(p: Perm): Boolean = {
2    var tmp = p
3    for (r <- 0 to 3) {
4      if (lexKleiner(tmp, p) || lexKleiner(spiegle(tmp), p)) {
5        return false
6      }
7      tmp = drehe(tmp)
8    }
9    return true
10 }
```

Hinweis: In dieser Implementierung wird zuallererst geprüft, ob die übergebene Lösung zu sich selbst lexikographisch kleiner ist. Außerdem wird, sofern die Lösung nicht „neu“ ist, immer eine vierte Rotation durchgeführt. Durch entsprechende `if`-Anweisungen lassen sich diese beiden unnötigen Operationen noch unterbinden.