RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
AACHEN

LEHR- UND FORSCHUNGSGEBIET INFORMATIK II
RWTH Aachen · D-52056 Aachen · GERMANY
http://www-i2.informatik.rwth-aachen.de/lufgi2

Prof. Dr. Jürgen Giesl
Darius Dlugosz, Thomas v. d. Maßen, Antje Nowack

Übung *Informatik I - Programmierung* - Blatt 7

(Lösungsvorschlag)

Aufgabe 1

Betrachten wir die Berechnung der n-ten Fibonacci-Zahl von „unten nach oben", d.h. wir beginnen mit $fib(0)$ (und $fib(1)$) und versuchen sukzessive die nächste Fibonacci-Zahl aus den bereits berechneten Werten zu bestimmen, bis wir die gesuchte n-te Fibonacci-Zahl erreicht haben. Nach Definition von $fib(n)$ für $n > 1$ brauchen wir jeweils zwei davor berechnete Werte, weshalb wir uns diese in zwei Variablen `a` und `b` merken. Zu Beginn ist `a = 0`, da $fib(0) = 0$ und `b = 1`, da $fib(1) = 1$. Der Wert von $fib(2)$ ist dann genau `b + a` und der von $fib(3)$ gleich $fib(2)$ + `b`. Damit $fib(2)$ bei der Berechnung von $fib(3)$ nicht nochmals bestimmt werden muss, merken wir uns diesen Wert ebenfalls, sobald er berechnet wurde. Dazu können wir die Variable `a` verwenden[1], da diese den Wert von $fib(0)$ enthält, den wir für weitere Berechnung nicht mehr benötigen. Für $fib(4)$ gilt dann $fib(3)$ + `a`. Auch hier sollten wir den Wert von $fib(3)$ bereits parat haben. Nach Berechnung von $fib(3)$ ist der Wert der Variable `b` der von $fib(1)$. Dieser wird jedoch nicht mehr benötigt, weshalb die Variable `b` den Wert von $fib(3)$ aufnehmen kann. Für $fib(4)$ rechnen wir dann `a + b` aus. Nach diesem Muster wird genauso für $fib(5)$, $fib(6)$, usw. verfahren.

Man braucht somit nur zwei Variablen und sorgt dafür, dass sie die beiden letzten Ergebnisse enthalten. Wenn die Variable `a` das Ergebnis von $fib(k)$ enthält, soll in `b` das Ergebnis von $fib(k-1)$ enthalten sein. Dann ist das Ergebnis von $fib(k+1)$ die Summe `a + b`. Das wird jetzt in der Variable `a` gespeichert, nachdem in `b` der Wert von `a` abgelegt wurde. Dann ist `a` $= fib(k+1)$ und `b` $= fib(k)$ und wir können mit der Berechnung fortfahren.

Um diese Idee in einen linearrekursiven Algorithmus umzusetzen, braucht man nun außer dem Parameter `n`, für dessen Wert die Fibonacci-Zahl berechnet wird (und zugleich angibt wie weit wir „nach oben" rechnen müssen), zwei weitere, die jeweils die Werte von `a` und `b` enthalten (zu Beginn ist `a = 0` und `b = 1`). Man könnte noch einen weiteren Parameter einführen, der mit jeder Berechnung (von `a + b`) um 1 erhöht wird und, falls dessen Wert gleich dem von `n` ist, das Ergebnis der Berechnung zurückgegeben wird (nicht-rekursiver Fall). Man kann jedoch stattdessen den Wert von `n` mit jeder neuen Berechnung um 1 erniedrigen, bis 0 erreicht ist (für `n < 0` ist das Ergebnis definitionsgemäß 0). Jetzt ist nur noch dafür zu sorgen, dass die Werte von `a` und `b` mit korrekten Werten an den rekursiven Aufruf übergeben werden. Wir nutzen ebenso die Tatsache, dass für `a = 0` und `b = 1` das Ergebnis von $fib(1)$ gleich `a + b` ist.

Unsere Methode sieht somit folgendermaßen aus:

[1] oder auch die Variable `b`, wenn wir davor den Wert von `b` in `a` speichern

```
  static int fib(int n, int a, int b) {
    if (n <= 0) return a;
    else {
      int ax = a + b;
      int bx = a;
      n = n - 1;
      return fib(n, ax, bx);
    }
  }
```

Nach weiteren Überlegungen kann diese Methode in folgende überführt werden

```
  static int fib(int n, int a, int b) {
    if (n < 1) return a;
    else return fib(n - 1, a + b, a);
  }
```

Hierbei muss diese Methode immer mit dem Startwert `a = 0` (= fib(0)) und `b = 1` (= fib(1)) aufgerufen werden. Deshalb benutzen wir die obige Methode als eine Hilfsmethode und definieren:

```
  static int fib(int n) {
    return fib(n, 0, 1);
  }
```

Nun folgt das Fibonacci-Programm:

```
/**
 * Die Klasse Fibonacci enthaelt Methoden zur Berechnung der Fibonacci-Zahlen.
 *
 * @author Darius Dlugosz
 * Umgebung: JDK 1.3, Windows 2000
 * Erstellt: 3.12.2001
 */
public class Fibonacci {

  /**
   * Startet das Fibonacci-Programm.
   * Es wird vom Benutzer eine Zahl erwartet, fuer die die Fibonacci-Zahl
   * berechnet wird. Das Ergebnis der Berechnung wird auf dem Bildschirm
   * ausgegeben.
   */
  public static void main(String[] arguments) {
    System.out.print("Bitte Zahl eingeben: ");
```

```
    int x = IO.Eingabe();
    System.out.println(fib(x));
  }

  /**
   * Dieser (private!) Konstruktor verhindert die Erzeugung einer Instanz
   * (d.h eines Objektes) dieser Klasse. Dies ist erwuenscht, da ein Objekt
   * dieser Klasse nicht von Interesse ist.
   */
  private Fibonacci() {
  }

  /**
   * Die Methode berechnet die n-te Fibonacci-Zahl.
   *
   * @param n die Zahl, fuer die die zugehoerige Fibonacci-Zahl berechnet
   *          werden soll.
   *
   * @return die n-te Fibonacci-Zahl.
   */
  public static int fib(int n) {
    return fib(n, 0, 1);
  }

  /**
   * Eine Hilfsmethode, die mit linearem Aufwand, die n-te Fibonacci-Zahl
   * berechnet.
   *
   * @param n die Zahl, fuer die die zugehoerige Fibonacci-Zahl berechnet
   *          werden soll.
   * @param a soll immer mit 0 initialisiert werden.
   * @param b soll immer mit 1 initialisiert werden.
   *
   * @return die zu n zugehoerige Fibonacci-Zahl.
   */
  private static int fib(int n, int a, int b) {
    if (n < 1) return a;
    else return fib(n - 1, a + b, a);
  }

}
```

Aufgabe 2

Um aus einem nicht-endrekursiven Algorithmus einen (äquivalenten) endrekursiven zu erhalten, ist es notwendig die (Teil-) Ergebnisse in den Argumenten der Methode[2] einzusammeln und im nicht rekursiven Fall, diese Werte in die Berechnung des Endergebnisses einzubeziehen.

Bei der Transformation in eine iterative Version muss man die rekursiven Aufrufe mit einer Schleife simulieren. Hat man eine endrekursive Version, so lässt sich aus dieser leicht eine iterative angeben.[3] Hierbei muss die Berechnung, die in den Argumenten des rekursiven Aufrufs stattfindet, in dem Schleifenkörper durchgeführt werden.

(a) In `foo` wird mit jedem rekursiven Aufruf eine 1 zum Ergebnis des rekursiven Aufrufs addiert. Im nicht rekursiven Fall wird `y` ausgegeben. Man kann also das Aufaddieren in einem zusätzlichen Parameter `z` bewerkstelligen und am Ende (nicht-rekursiver Fall) den Wert von `z` zu `y` addieren. Diese Überlegung führt zu folgender Methode

```
public static int foo(int x, int y, int z) {
  if (x == 0) return y + z;
  else return foo(x-1, y, z+1);
}
```

Hierbei muss für einen (initialen) Aufruf von `foo(x, y, z)` mit konkreten Werten für `x` und `y`, `z` mit 0 initialisiert werden und wir definieren

```
public static int foo(int x, int y) {
  return foo(x, y, 0);
}
```

Die Methode `foo(x, y, z)` wird nur als Hilfsmethode verwendet und sollte deshalb in einem Programm als `private` definiert werden.

Bei genauer Betrachtung erkennt man, dass die 1 direkt zu `y` addiert werden kann, weshalb die Akkumulator-Variable `z` nicht benötigt wird. Somit kann anstelle der Hilfsmethode `foo(x, y, z)`, die endrekursive Version wie folgt angegeben werden

```
public static int foo(int x, int y) {
  if (x == 0) return y;
  else return foo(x-1, y+1);
}
```

[2] mit Methode ist hier eine Java-Methode gemeint

[3] Die Transformation von der endrekursiven in eine iterative Version ist ohne weiteres gestattet, da die endrekursive Variante äquivalent zu der Originalversion ist.

Nun kann `foo` in eine iterative Version überführt werden. Die endrekursive Variante von `foo(x, y)` ruft sich solange rekursiv auf, solange `x` $\neq$ `0` ist, wobei `x-1` und `y+1` in den Argumenten ausgeführt wird. Zum Schluss wird `y` zurückgegeben. Man erhält somit folgende iterative Version:

```
public static int foo(int x, int y) {
  if (x == 0) return y;
  else
    while (x != 0) {
      --x;
      ++y;
    }
  return y;
}
```

Man kann die Fallunterscheidung `if (x == 0)...` auslassen, da diese in der while-Bedingung implizit enthalten ist und bei Nicht-Erfüllbarkeit der while-Bedingung gleich zu Beginn der Ausführung von `foo(x, y)`, ebenfalls `y` zurückgegeben wird. Die iterative Version lautet somit

```
public static int foo(int x, int y) {
  while (x != 0) {
    --x;
    ++y;
  }
  return y;
}
```

(Wie man erkennt, berechnet `foo(x, y)` die Addition von `x` und `y`, für `x` $\geq$ `0`.)

(b) In `boo` ist der erste rekursive Aufruf (Fall `x % 2 = 0`, wobei `x` $\neq$ `0`) bereits endrekursiv. Im Falle `x % 2` $\neq$ `0`[4] wird zum Ergebnis des rekursiven Aufrufs der Wert von `y` addiert. Man kann das Aufaddieren (wie in Teilaufgabe (a)) in einem zusätzlichen Parameter von `boo` durchführen und im nicht-rekursiven Fall den Wert des zusätzlichen Argumentes zurückliefern. In dem ersten rekursiven Aufruf bleibt der Wert der Akkumulator-Variable unverändert. Die endrekursive Version hat damit die folgende Form

```
public static int boo(int x, int y) {
   return boo(x, y, 0);
}
```

[4] wobei `x` $\neq$ `0`

```
private static int boo(int x, int y, int z) {
  if (x == 0) return z;
  else {
    if ((x % 2) == 0) return boo(x/2, 2*y, z);
    else return boo((x-1)/2, 2*y, z+y);
  }
}
```

Daraus ergibt sich folgende iterative Version:

```
public static int boo(int x, int y) {
   return boo(x, y, 0);
}

private static int boo(int x, int y, int z) {
  while (x != 0) {
    if ((x % 2) == 0) {
      x = x/2;
      y = 2*y;
    }
    else {
      x = (x-1)/2;
      z = z+y;
      y = 2*y;
    }
  }
  return z;
}
```

Die Fallunterscheidung `if (x == 0)...` wurde ausgelassen, da sie in der while-Bedingung implizit enthalten ist und für `x = 0` gleich zu Beginn der Ausführung, ebenfalls `z` zurückgegeben wird.

Zu beachten ist, dass die Zuweisung `z = z+y` vor der Zuweisung `y = 2*y` ausgeführt wird, da in `y = 2*y` der Wert von `y` verändert wird.

Man kann die Hilfsmethode noch etwas kürzen. Die Zuweisung `y = 2*y` kann nach der if-Anweisung durchgeführt werden[5] [6], da diese Zuweisung in beiden Zweigen der if-Anweisung

[5] Ist das eine Optimierung? Wird das Programm dadurch effizienter?

[6] !!! Wenn man es eine Optimierung nennen will, dann nur weil der Code eine Zeile kürzer wurde; die Zuweisung wird in beiden Fällen gleich oft ausgeführt.

ausgeführt wird. Man erhält somit[7] [8] [9] [10]

```
public static int boo(int x, int y, int z) {
  while (x != 0) {
    if ((x % 2) == 0) {
      x = x/2;
    }
    else {
      x = (x-1)/2;
      z = z+y;
    }
    y = 2*y;
  }
  return z;
}
```

(c) Nach genauerer Betrachtung der Methode (ähnlich wie in Aufgabe 1) erhält man z. B. die folgende endrekursive Version:

```
public static int rham(int n)  {
  return rham(n, 1, 0);
}

private static int rham(int n, int a, int b) {
  if (n == 1) return a + b;
  else if (n > 1) {
    if ((n % 2) == 0) return rham(n/2, a+b, b);
    else return rham((n-1)/2, a, a+b);
  }
  else return 0;
}
```

Die iterative Version ist:

```
public static int rham(int n)  {
  return rham(n, 1, 0);
```

[7] Überlegen Sie, ob eine Optimierung der Hilfsmethode möglich ist. Gilt `x/2 = (x-1)/2` für alle int-Werte von `x`, falls `x % 2` $\neq$ `0`? Terminiert `boo` für jedes `x`? Kann man das ausnutzen?

[8] Was berechnet `boo(x, y)` für `x` $\geq$ `0`?

[9] !!! Für `x % 2` $\neq$ `0` gilt `x/2 = (x-1)/2` (Ganzahldivision), falls `x` $\geq$ `0`; für `x` $<$ `0` gilt das nicht. `boo` terminiert für `x` $<$ `0` nicht. Man könnte also denken, dass man in der Hilfsmethode den Ausdruck `(x-1)/2` durch `x/2` ersetzen und dies dann nach if-Anweisung verschieben kann. (Das wäre eine Optimierung, da man keine Subtraktion mehr durchführt). Dann terminiert aber die Hilfsmethode z. B. für -1, da $-1/2 = 0$.

[10] !!! `boo(x, y)` berechnet das Produkt von `x` und `y` für `x` $\geq$ `0`.

```
    }

    private static int rham(int n, int a, int b) {
      if (n < 1) return 0;
      if (n == 1) return a + b;
      while (n > 1) {
        if (n%2 == 0) {
          a = a + b;
          n = n/2;
        }
        else {
          b = a + b;
          n = (n-1)/2;
        }
      }
      return a + b;
    }
```

Aufgabe 3

Die Beschreibung des Programms (und der Ideen) sind in den Kommentaren des Programms gegeben.

```
 /**
  * QueensSolver loest das m-beschraenkte n-Damen Problem.
  * Es werden alle moegliche Aufstellungen der Damen berechnet.
  * Die Berechnung erfolgt mit Hilfe von Backtracking.
  *
  * @author Darius Dlugosz
  * Umgebung: JDK 1.3, Windows 2000
  * Erstellt: 4.12.2001
  * Letzte Aenderung: 5.12.2001
  */
public class QueensSolver {

  /** Seitenlaenge des Schachbretts */
  private static int length;

  /** Anzahl der Damen */
  private static int queens;

  /** Reichweite jeder Dame */
  private static int range;
```

```
/** Anzahl der berechneten Loesungen */
private static int solutions;

/**
 * Koordinaten (Positionen) der Damen auf dem Schachbrett
 *
 * position[i]  enthaelt die Koordinaten (Zeile, Spalte) der i-ten Dame
 * position[i][0] = line    bedeutet, dass die i-te Dame in der Zeile line
 *                          gesetzt wird
 * position[i][1] = column  bedeutet, dass die i-te Dame in der Spalte column
 *                          gesetzt wird
 */
private static int [][] position; // das ist kein Schachbrett

/**
 * Startet das QueensSolver-Programm.
 * Es werden die Seitenlaenge des Schachbretts, Anzahl der Damen und
 * ihre Reichweite erfragt und anschliessend jede moegliche
 * Aufstellung der Damen berechnet und auf dem Bildschirm ausgegeben.
 */
public static void main(String[] arguments) {
  System.out.print("Bitte die Länge des Schachbretts eingeben: ");
  length = IO.Eingabe();
  System.out.print("Bitte die Anzahl der Damen eingeben: ");
  queens = IO.Eingabe();
  System.out.print("Bitte die Reichweite jeder Dame eingeben: ");
  range = IO.Eingabe();

    // waren die Eingaben sinnvoll?
  if (length > 0 && queens > 0 && range > -1) {

    // erzeuge position-Array, dessen Laenge gleich Anzahl der Damen ist und
    // jedes Feld des Arrays zwei Zahlen (die Koordinaten) aufnehmen kann
    position = new int [queens][2];

    // beginne die Suche mit der ersten Dame
    // (0 ist der erste Index eines Arrays, d.h. der ersten Dame, queen-1 ist
    // somit der Index der letzten Dame)
    search(0);
  }

  if (solutions == 0) System.out.println("Keine gültige Aufstellung möglich.");
}
```

```
/**
 * Dieser (private!) Konstruktor verhindert die Erzeugung einer Instanz
 * (d.h eines Objektes) dieser Klasse. Dies ist erwuenscht, da alle Methoden
 * (bis auf "main") statisch definiert sind.
 */
private QueensSolver() {
}

/**
 * Sucht nach allen moeglichen Aufstellungen der Damen queenNr, queenNr+1, ...,
 * bis Damenanzahl (d.h. die Positionen der Damen mit Index kleiner queenNr
 * werden nicht geaendert). Wird eine gueltige Aufstellung der Damen gefunden,
 * werden die Positionen aller Damen auf dem Bildschirm ausgegeben. Gleichzeitig
 * wird die Anzahl der bereits gefundenen Loesungen gespeichert.
 *
 * Wird die Methode mit dem Index der ersten Dame aufgerufen, so wird nach
 * Aufstellungen aller Damen gesucht.
 *
 * @param queenNr Nummer der Dame mit der die Suche (nach geeigneten Positionen)
 *        beginnen soll
 */
private static void search(int queenNr) {
  /*
   * Vorgehensweise: Setze die Dame queenNr auf das naechste freie Feld des
   * Schachbretts. (Die Felder sind in einer bestimmten Weise geordnet; siehe
   * die Methode getNextPosition.) Das naechste freie Feld ist das erste Feld
   * des Schachbretts, falls noch keine Dame aufgestellt wurde oder das (in der
   * Ordung) naechste Feld nach dem Feld, auf dem die Dame queenNr-1 gesetzt
   * wurde. Kann die Dame an dieser Position stehen bleiben, dann suche (nach
   * dem gleichen Prinzip, rekursiv) eine Position fuer die naechste Dame, usw.
   * bis die letzte Damen platziert wurde, sonst setzte die Dame auf das
   * naechste Feld und wiederhole den Vorgang. Wird eine Loesung (fuer alle
   * Damen) gefunden, wird dann fuer die letzte Dame die naechste gueltige
   * Position gesucht (beginnend mit dem Feld, auf dem sie zuletzt stand),
   * dann fuer die vorletzte Dame (natuerlich wird dann auch die letzte Dame
   * noch mal entsprechend platziert), dann fuer die vor-vorletze usw. bis wir
   * fuer die Dame queenNr die naechste moegliche Position erneut suchen. Dieser
   * Prozess wird solange wiederholt bis alle Positionen (fuer Dame queenNr)
   * betrachtet wurden.
   */

    // falls die Damenanzahl erreicht wurde (d.h. alle Damen wurden platziert),
    // erhoehe den Loesungsazaehler um 1 und gebe die Positionen der Damen aus
  if (queenNr == queens) {
```

```
    ++solutions;
    print();
  }
  else {
    int [] start = {1, 1};
      // wenn eine Dame bereits platziert wurde, kann die naechste nur auf einem
      // Feld ab der Position der zuletzt platzierten Dame aufgestellt werden.
      // Die Felder davor wurden bereits von queenNr-1 durchprobiert
    if (queenNr != 0) start = getNextPosition(position[queenNr-1]);
    for (int [] pos = start; pos[0] <= length && pos[1] <= length;
          pos = getNextPosition(pos)) {
      position[queenNr] = pos;
        // wird die Dame auf ihrer aktuellen Position nicht geschlagen, dann
        // suche eine Position fuer die naechste Dame, sonst setzte sie auf das
        // naechste Feld (for-Schleife) und teste erneut.
      if (test(queenNr)) search(queenNr+1);
        // teste auch die naechste Position (die in for-Schleife) fuer die
        // aktuelle Dame
    }
  }
}

/**
 * Ueberprueft, ob die aktuelle Position der gegebenen Dame, in der
 * (horinzontalen, vertikalen oder diagonalen) Reichweite einer bereits
 * aufgestellten Dame liegt.
 *
 * @param queenNr Nummer der Dame, fuer die die Ueberpruefung stattfindet
 *
 * @return true, falls die Dame queen von keiner der bereits aufgestellten Damen
 *         geschlagen wird;
 *         false, sonst (eine der bereits aufgestellten Damen schlaegt die Dame
 *         queenNr)
 */
private static boolean test(int queenNr) {
  for (int q = 0 ; q < queenNr; q++)
    int distance = getDistance(position[queenNr], position[q]);
    if (distance != -1 && distance <= range) return false;
  }
  return true;
}

/**
 * Liefert den Abstand zwischen zwei Positionen des Schachbretts zurueck.
```

```
 * Abstand ist die Anzahl der Felder, die man von einer Position gehen muss,
 * um einer andere Position zu erreichen, falls beide Positionen auf der
 * gleichen horizontalen, vertikalen oder diagonalen Linie liegen. Dies
 * entspricht der Anzahl der dazwischen liegenden Felder + 1. Fuer zwei
 * gleiche Positionen ist der Abstand 0. Fuer zwei benachbarte Felder ist der
 * Abstand 1, usw.
 *
 * @param pos1 eine Position des Schachbretts
 * @param pos2 eine weitere Position des Schachbretts
 *
 * @return Anzahl der Felder die von pos1 zu pos2 liegen, falls pos1 und pos2
 *         sich auf der gleichen horizontalen, vertikalen oder diagonalen Linie
 *         befinden;
 *         -1, sonst (pos1 und pos2 koennen nicht durch eine horizontale,
 *                    vertikale oder diagonale Linie des Schachbretts verbunden
 *                    werden).
 */
private static int getDistance(int [] pos1, int [] pos2) {
  // Zeilenabstand (Math.abs(x) berechnet Betrag von x)
  int lineDistance = Math.abs(pos1[0] - pos2[0]);

  // Spaltenabstand
  int columnDistance = Math.abs(pos1[1] - pos2[1]);

  /* Falls die Positionen die gleiche Zeilennummer haben, so ist der Abstand
   * die Anzahl der Spalten dazwischen. Falls die Positionen die gleiche
   * Spaltennummer haben, dann ist der Abstand die Anzahl der Zeilen dazwischen.
   * Falls die Positionen auf derselben diagonalen Linie liegen, dann ist der
   * Abstand gleich dem Zeilenabstand. (oder auch Spaltenabstand).
   */
  if (pos1[0] == pos2[0]) return columnDistance;
  else if (pos1[1] == pos2[1]) return lineDistance;
  else if (lineDistance == columnDistance) return lineDistance;
  else return -1;
}

/**
 * Liefert ausgehend von der gegebenen Position die naechste Position des
 * Schachbretts zurueck. Die Positionen werden als Koordinaten (Zeile, Spalte)
 * verarbeitet. Die Felder (Positionen) einer Zeile werden immer von links nach
 * rechts angeordnet/gezaehlt, d.h. der Nachfolger eines Feldes (x, y) ist
 * (x, y+1) fuer y < Spaltenanzahl. Fuer das letzte Feld einer Zeile x ist der
 * Nachfolger das erste Feld der naechsten Zeile, also das Feld (x+1, 1) fuer
 * (x, Spaltenanzahl).
```

```
 *
 * Ein Schachbrett a x a kann somit z. B. in folgender Form dargestellt werden:
 * (hier sind die Felder die Koordinaten (Zeile, Spalte))
 *
 *     (1, 1) (1, 2) (1, 3) ... (1, a)
 *     (2, 1) (2, 2)    ...     (2, a)
 *     (3, 1)        ...        (3, a)
 *       .      .                 .
 *       .            .           .
 *       .                  .     .
 *     (a, 1)        ...        (a, a) <- die letzte Position des Schachbretts
 *
 *  Die erste Position ist (1, 1), die zweite (1, 2), die dritte (1, 3),
 *  die a-te (1, a), die (a+1)-te (2, 1), usw.
 *
 *  Beispiele:
 *  Bsp. 1: fuer (1, 1)   ist (1, 2) die naechste Position, wobei 2 <= a
 *  Bsp. 2: fuer (2, a-1) ist (2, a) die naechste Position, wobei 2 <= a
 *  Bsp. 3: fuer (4, a)   ist (5, 1) die naechste Position, wobei 5 <= a
 *  Bsp. 4: fuer (a, a)   ist (a+1, 1) die naechste Position, wobei dies keine
 *          gueltige Position des Schachbretts ist
 *
 * @param pos eine gueltige Position (Zeile, Spalte) des Schachbretts
 *
 * @return naechste Position des Schachbretts, falls pos nicht die letzte
 *         Position auf dem Schachbrett ist;
 *         (a + 1, 1), falls pos die letzte Position auf dem Schachbretts ist
 *                     (d.h. pos=(a, a))
 *
 * Achtung: es wird nicht ueberprueft, ob die gegebene Position eine gueltige
 * Position des Schachbretts ist
 */
private static int [] getNextPosition(int [] pos) {
  int [] nextPosition = {length + 1, 1};

  if (pos[1] == length) {
    if (pos[0] < length) nextPosition[0] = pos[0] + 1;
  }
  else {
    nextPosition[0] = pos[0];
    nextPosition[1] = pos[1] + 1;
  }
  return nextPosition;
}
```

```
  /**
   * Gibt nacheinander die Positionen der Damen in der Form "(Zeile, Spalte)"
   * auf dem Bildschirm aus.
   */
  private static void print() {
    System.out.println("Lösung " + solutions + ":");
    for (int q = 0; q < queens; q++) {
      System.out.println("(" + position[q][0] + ", " + position[q][1] + ")");
    }
    System.out.println();
  }

}
```