E. Array-Aufgaben

E.1. Array erzeugen *

Schreiben Sie ein Java-Programm, das ein Array mit `n` ganzen Zahlen erzeugt. Die Anzahl der Elemente `n` soll vom Benutzer eingegeben werden. Das Array soll folgenden Inhalt haben:

n	n-1	...	4	3	2	1

Programmieren Sie auch eine Funktion void printArray(int[] array), die das Feld auf der Konsole ausgibt. Diese Funktion soll Integer-Arrays beliebiger Länge drucken können.

E.2. Suche im Array *

Erweitern Sie das Programm um eine Suchfunktion in einer Methode `int`

```
searchIndex(int[] a, int value)
```

Diese Methode soll das Array und den zu suchenden Wert als Parameter bekommen, und den Index des Elements liefern, das den Suchwert enthält. Denken Sie auch an den Fall, dass der Wert nicht im Array enthalten ist. Welches Ergebnis liefert die Suche dann?

Testen Sie das Programm durch evtl. wiederholte Eingabe von Suchschlüsseln.

E.3. Vertauschen von Werten *

Schreiben Sie eine Methode `void`

```
swap(int[] a, int i, int j)
```

die zwei Werte in einem Integer-Array vertauscht. Die Funktion wird mit einem Array und den Indizes der beiden Werte als Parameter aufgerufen. Denken Sie auch an Fehlerfälle bei ungültigen Parametern!

Beispiel (das Array sei über die Variable a deklariert):

Vorher:

4	2	10	3	-5	0	17

```
swap(a, 3, 5)
```

4	2	10	0	-5	3	17

E.4. Umkehren eines Arrays *

Implementieren Sie eine Methode `void`

`reverseArray(int[] a)`

die die Reihenfolge der Werte umkehrt. Benutzen Sie die Methode `swap(...)` aus der vorigen Aufgabe!

Beispiel:

4	2	10	3	-5	0	17

`reverse(a)`

17	0	-5	3	10	2	4

E.5. Kopieren eines Arrays *

Wie Sie wissen, sind Arrays durch Zuweisung von Referenzen nicht wirklich kopierbar., sondern Sie erhalten nur ein Alias. Schreiben Sie daher eine Methode

`int[] cloneArray(int[] a)`

die eine 1:1-Kopie des Arrays herstellt und als Funktionsergebnis liefert.

Überlegen Sie sich, wie man testen könnte, ob die Methode korrekt arbeitet.

E.6. Maximum/Minimum bestimmen *

Schreiben Sie zwei Methoden

`int maximum(int[] a)`

`int minimum(int[] a)`

die den größten bzw. kleinsten Wert eines Arrays ganzer Zahlen liefert. Implementieren Sie zusätzlich zwei Methoden `int maxIndex(int[] a)` `int minIndex(int[] a)`

die an Stelle der Werte jeweils den Index des größten bzw. kleinsten Elements liefern:

4	2	10	3	-5	0	17

`maximum(a) → 17`

`minimum(a) → -5`

`maxIndex(a) → 6`

`minIndex(a) → 4`

E.7. Selection-Sort **

Das Sortierverfahren Selection-Sort arbeitet nach folgendem Prinzip:

1. Der Index des kleinsten Wertes wird bestimmt.
2. Dieses kleinste Element wird mit dem ersten Element vertauscht (d.h. das erste Element ist nun bereits das kleinste).
3. Danach wird, beginnend mit dem 2, Element, wiederum das Minimum bestimmt und dieses Element mit dem 2. vertauscht (jetzt sind bereits zwei Elemente in der richtigen Reihenfolge)
4. Die Minimum-Suche und das Vertauschen wird so lange wiederholt, bis das vorletzte Element erreicht wurde (dieses kann dann ggfs. mit dem letzten Element vertauscht werden)
5. Danach ist das Array vollständig sortiert

Programmieren Sie das Verfahren Selection-Sort so, dass das Array aufsteigend sortiert wird.

Beispiel:

4	2	10	3	-5	0	17

-5	2	10	3	4	0	17

-5	0	10	3	4	2	17

-5	0	2	3	4	10	17

-5	0	2	3	4	10	17

...ab jetzt keine Änderung mehr, aber das Verfahren läuft noch weiter bis zum vorletzten Element. Das Ergebnis ist ein sortiertes Array:

-5	0	2	3	4	10	17

E.8. Binäre Suche **

Wenn die Elemente eines Arrays sortiert sind, kann zur Suche eines Elements die binäre Suche verwendet werden. Das Verfahren beginnt mit der Suche in der Mitte des Arrays. Ist dort der gesuchte Wert, so kann der Index als Ergebnis geliefert werden. Ist der gesuchte Wert kleiner

als das mittlere Element, so wird die Suche in der Mitte des linken Teils fortgesetzt. Ist der gesuchte Wert größer als das mittlere Element, geht die Suche im rechten Teil weiter.

In jedem Schritt wird so die Größe des zu durchsuchenden Bereichs halbiert. Daher ist das Verfahren im Gegensatz zur linearen Suche, die für n Elemente maximal n Schritte benötigt, bereits nach höchstens $\log_2(n)$ Schritten fertig.

Implementieren Sie das Verfahren in einer Methode `int`

```
binarySearchIndex(int[] a, int value)
```

die den Index des gesuchten Wertes `key` im Array `a` liefert. Falls `value` nicht in `a` vorkommt, soll als Fehleranzeige der ungültige Index -1 ausgegeben werden.

E.9. Lottozahlen **

Schreiben Sie ein Java-Programm, das 6 Lottozahlen aus dem Bereich [1, 49] ausgibt. Die Zahlen sollen natürlich zufällig gewählt werden und keine der 6 Zahlen darf doppelt vorkommen.

Tipp: Benutzen Sie ein Array mit 49 Einträgen und "mischen" Sie die Zahlen!

E.10. Matrizen *

Schreiben Sie eine Java-Methode

```
double[][] createMatrix(int lines, int columns)
```

die eine Matrix mit entsprechend vielen Zeilen und Spalten erzeugt. Die Matrix soll Zufallswerte aus dem Intervall [-100, 100] enthalten. Zur Ausgabe der Daten wird eine Methode

```
void printMatrix(double[][] m)
```

benötigt. Diese Methode soll die Daten zeilenweise ausgeben.

E.11. Matrizen-Operationen **

Ergänzen Sie in der Klasse aus der vorigen Aufgabe folgende Matrizen-Operationen. Die Operationen sollen die Einaben nicht verändern, sondern immer eine neue Matrix mit dem Ergebnis liefern. Prüfen Sie bei der Addition und der Multiplikation, ob die Dimensionen der Matrizen zueinander passen. Geben Sie anderenfalls eine Fehlermeldung aus! **Multiplikation einer Matrix mit Skalar *** `double[][] multMatrix(double[][] m, double x)`

Addition zweier Matrizen * `double[][]`

`addMatrix(double[][] a, double[][] b)`

Bedingung: Beide Matrizen müssen die gleiche Zeilen- und Spaltenzahl haben.

Multiplikation zweier Matrizen ** `double[][] multMatrix(double[][] a, double[][] b)`

Bedingung: Wenn `a` die Dimension `m × n` hat, muss `b` die Dimension `n × p` haben. Das Ergebnis `c` hat die Dimension `m × p`. Die einzelnen Komponenten von c bilden sich gemäß der Formel

$$c_{ij} = a_{i1} \cdot b_{1j} + a_{i2} \cdot b_{2j} + \ldots\ldots + a_{in} \cdot b_{mj} \text{ mit } 1 \leq i \leq m \text{ und } 1 \leq j \leq p$$

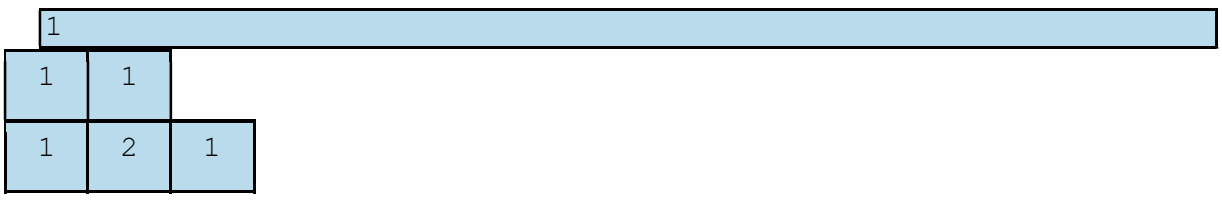
E.12. Pascal'sches Dreieck **

Das so genannte "Pascal'sche Dreieck" ist rekursiv definiert: Jede Zeile `z≥1` enthält genau `z` ganze Zahlen. Die erste und die letzte Zahl einer Zeile ist immer `1`. Die weiteren Zahlen (ab Zeile 3) ergeben sich aus der Summe der beiden Zahlen, die in der Zeile `z-1` über dem gesuchten Wert stehen:

```
                1
             1     1
          1     2    1
       1    3      3    1
    1    4     6     4    1
  1    5   10     10    5    1
1    6   15    20    15    6    1
...
```

Beispiel: Die 3. Zahl der 6. Zeile (10) ergibt sich aus der Summe der 2. (4) und 3. Zahl (6) der 5. Zeile.

Schreiben Sie ein Programm, das die ersten `n` Zeilen (`n` eingebbar) des Pascal'schen Dreiecks in einem 2-dimensionalen Integer-Array berechnet und ausgibt. Die einzelnen Zeilen sind dabei unterschiedlich lang und wie folgt anzuordnen:



1	3	3	1			
1	4	6	4	1		
1	5	10	10	5	1	
1	6	15	20	15	6	1

E.13. Sieb des Eratosthenes **

Das Sieb des Eratosthenes ist ein Verfahren zur Bestimmung von Primzahlen. Das Verfahren berechnet die Primzahlen bis zu einer wahlfreien Obergrenze `n`:

1. Schreibe die Zahlen von `1` bis `n` in einer Liste auf
2. Streiche die `1`
3. Für jede Zahl `x` von `2` bis `n`: Streiche alle Vielfachen von `x` aus der Liste. Beginne mit dem Streichen bei `2x`
4. Alle Zahlen, die nach dem Streichen übrig bleiben, sind Primzahlen.

Lassen Sie die Obergrenze `n` vom Benutzer eingeben. Führen Sie danach das Verfahren durch und geben Sie alle gefundenen Primzahlen aus!

Leichte Beschleunigung des Verfahrens: Sie können mit dem Streichen au*ören, wenn `x` größer als die Quadratwurzel von `n` ist.