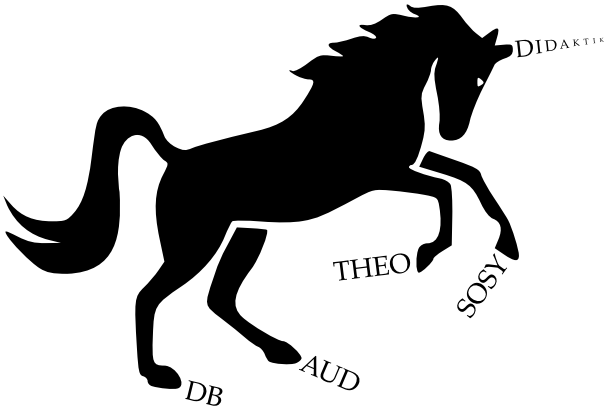
46115
Herbst
2017

Theoretische Informatik / Algorithmen / Datenstrukturen (nicht vertieft)

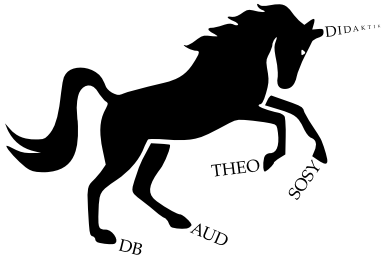
Aufgabenstellungen mit Lösungsvorschlägen



Die Bschlangaul-Sammlung

Hermine Bschlangaul and Friends

Aufgabenübersicht



Die Bschlangaul-Sammlung
Hermine Bschlangaul and Friends

Eine freie Aufgabensammlung mit Lösungen von Studierenden für Studierende zur Vorbereitung auf die 1. Staatsexamensprüfungen des Lehramts Informatik in Bayern.

Thema Nr. 2

Aufgabe 3 [Primzahl]

Die Methode `pKR` berechnet die n-te Primzahl ($n \geq 1$) kaskadenartig rekursiv und äußerst ineffizient:

```
static long pKR(int n) {
  long p = 2;
  if (n >= 2) {
    p = pKR(n - 1); // beginne die Suche bei der vorhergehenden Primzahl
    int i = 0;
    do {
      p++; // pruefe, ob die jeweils naechste Zahl prim ist, d.h. ...
      for (i = 1; i < n && p % pKR(i) != 0; i++) {
      } // pruefe, ob unter den kleineren Primzahlen ein Teiler ist
    } while (i != n); // ... bis nur noch 1 und p Teiler von p sind
  }
  return p;
}
```

Code-Beispiel auf Github ansehen: `src/main/java/org/bschlangaul/examen/examen_46115/jahr_2017/herbst/PrimzahlDP.java`

Überführen Sie `pKR` mittels *dynamischer Programmierung* (hier also *Memoization*) und mit möglichst *wenigen Änderungen* so in die *linear* rekursive Methode `pLR`, dass `pLR(n, new long[n + 1])` ebenfalls die n-te Primzahl ermittelt:

```
private long pLR(int n, long[] ps) {
  ps[1] = 2;
  // ...
}
```

Lösungsvorschlag

Lösungsvorschlag

> **Exkurs: Kaskadenartig rekursiv**
> Kaskadenförmige Rekursion bezeichnet den Fall, in dem mehrere rekursive Aufrufe nebeneinander stehen.

Lösungsvorschlag

> **Exkurs: Linear rekursiv**
> Die häufigste Rekursionsform ist die lineare Rekursion, bei der in jedem Fall der rekursiven Definition höchstens ein rekursiver Aufruf vorkommen darf.

```
static long pLR(int n, long[] ps) {
  ps[1] = 2;
  long p = 2;
  if (ps[n] != 0) // Fall die Primzahl bereits gefunden / berechnet wurde,
    return ps[n]; // gib die berechnet Primzahl zurück.
  if (n >= 2) {
```

```
      // der einzige rekursive Aufruf steht hier, damit die Methode linear rekursiv
      // ist.
      p = pLR(n - 1, ps);
      int i = 0;
      do {
        p++;
        // Hier wird auf das gespeicherte Feld zurückgegriffen.
        for (i = 1; i < n && p % ps[i] != 0; i++) {
        }
      } while (i != n);
    }
    ps[n] = p; // Die gesuchte Primzahl im Feld speichern.
    return p;
  }
```

Code-Beispiel auf Github ansehen: `src/main/java/org/bschlangaul/examen/examen_46115/jahr_2017/herbst/PrimzahlDP.java`

Der komplette Quellcode

```
/**
 * Berechne die n-te Primzahl.
 *
 * Eine Primzahl ist eine natürliche Zahl, die größer als 1 und ausschließlich
 * durch sich selbst und durch 1 teilbar ist.
 *
 * <ul>
 * <li>1. Primzahl: 2
 * <li>2. Primzahl: 3
 * <li>3. Primzahl: 5
 * <li>4. Primzahl: 7
 * <li>5. Primzahl: 11
 * <li>6. Primzahl: 13
 * <li>7. Primzahl: 17
 * <li>8. Primzahl: 19
 * <li>9. Primzahl: 23
 * <li>10. Primzahl: 29
 * </ul>
 */
public class PrimzahlDP {

  /**
   * Die Methode pKR berechnet die n-te Primzahl ({@code n >= 1}) Kaskadenartig
↪  Rekursiv.
   *
   * @param n Die Nummer (n-te) der gesuchten Primzahl. Die Primzahl 2 ist die
   *          erste Primzahl. Die Primzahl 3 ist die zweite Primzahl etc.
   *
   * @return Die gesuchte n-te Primzahl.
   */
  static long pKR(int n) {
    long p = 2;
    if (n >= 2) {
      p = pKR(n - 1); // beginne die Suche bei der vorhergehenden Primzahl
      int i = 0;
```

```
    do {
      p++; // pruefe, ob die jeweils naechste Zahl prim ist, d.h. ...
      for (i = 1; i < n && p % pKR(i) != 0; i++) {
      } // pruefe, ob unter den kleineren Primzahlen ein Teiler ist
    } while (i != n); // ... bis nur noch 1 und p Teiler von p sind
  }
  return p;
}

/**
 * Die Methode pLR berechnet die n-te Primzahl ({@code n >= 1}) Linear Rekursiv.
 *
 * @param n  Die Nummer (n-te) der gesuchten Primzahl. Die Primzahl 2 ist die
 *           erste Primzahl. Die Primzahl 3 ist die zweite Primzahl etc.
 * @param ps Primzahl Speicher. Muss mit n + 1 initialisert werden.
 *
 * @return Die gesuchte n-te Primzahl.
 */
static long pLR(int n, long[] ps) {
  ps[1] = 2;
  long p = 2;
  if (ps[n] != 0) // Fall die Primzahl bereits gefunden / berechnet wurde,
    return ps[n]; // gib die berechnet Primzahl zurück.
  if (n >= 2) {
    // der einzige rekursive Aufruf steht hier, damit die Methode linear rekursiv
    // ist.
    p = pLR(n - 1, ps);
    int i = 0;
    do {
      p++;
      // Hier wird auf das gespeicherte Feld zurückgegriffen.
      for (i = 1; i < n && p % ps[i] != 0; i++) {
      }
    } while (i != n);
  }
  ps[n] = p; // Die gesuchte Primzahl im Feld speichern.
  return p;
}

static void debug(int n) {

  ↪ System.out.println(String.format("%d. Primzahl: %d (kaskadenartig rekursiv berechnet)",
  ↪ n, pKR(n)));
  System.out.println(String.format("%d. Primzahl: %d (linear rekursiv berechnet)", n,
  ↪ pLR(n, new long[n + 1])));
}

public static void main(String[] args) {
  System.out.println(pKR(10));
  System.out.println(pLR(10, new long[11]));

  for (int i = 1; i <= 10; i++) {
    debug(i);
  }
```

```
    }
}
```

Code-Beispiel auf Github ansehen: `src/main/java/org/bschlangaul/examen/examen_46115/jahr_2017/herbst/PrimzahlDP.java`

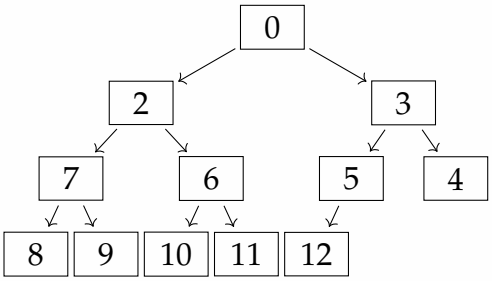
Aufgabe 6: [Halden - Heaps]

Gegeben sei folgende Feld-Einbettung (Array-Darstellung) einer Min-Halde:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
0	2	3	7	6	5	4	8	9	10	11	12

(a) Stellen Sie die Halde graphisch als (links-vollständigen) Baum dar.

Lösungsvorschlag



(b) Entfernen Sie das kleinste Element (die Wurzel 0) aus der obigen initialen Halde, stellen Sie die Haldeneigenschaft wieder her und geben Sie nur das Endergebnis in Felddarstellung an.

Lösungsvorschlag

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	6	3	7	10	5	4	8	9	12	11

(c) Fügen Sie nun den Wert 1 in die obige initiale Halde ein, stellen Sie die Haldeneigenschaft wieder her und geben Sie nur das Endergebnis in Felddarstellung an.

Lösungsvorschlag

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
0	2	1	7	6	3	4	8	9	10	11	12	5

Aufgabe 7

(a) Führen Sie *„Sortieren durch Einfügen“* lexikographisch aufsteigend und *in-situ* (*in-place*) so in einem Schreibtischlauf auf folgendem Feld (Array) aus, dass gleiche Elemente ihre relative Abfolge jederzeit beibehalten (also dass z. B. A_1 stets vor A_2 im Feld steht). Jede Zeile stellt den Zustand des Feldes dar, nachdem das jeweils nächste Element in die Endposition verschoben wurde. Der bereits sortierte Teilbereich steht vor |||. Gleiche Elemente tragen zwecks Unterscheidung ihre *„Objektidentität“* als Index (z. B. `"A1".equals("A2")` aber `"A1" != "A2"`)

L	A_1	B_1	F	A_2	B_2

Lösungsvorschlag

L	A_1	B_1	F	A_2	B_2
A_1	L	B_1	F	A_2	B_2
A_1	B_1	L	F	A_2	B_2
A_1	B_1	F	L	A_2	B_2
A_1	A_2	B_1	F	L	B_2
A_1	A_2	B_1	B_2	F	L

(b) Ergänzen Sie die folgende Methode so, dass sie die Zeichenketten im Feld `a` lexikographisch aufsteigend durch Einfügen sortiert. Sie muss zum vorangehenden Ablauf passen, ðsie muss *iterativ* sowie *in-situ* (*in-place*) arbeiten und die relative Reihenfolge gleicher Elemente jederzeit beibehalten. Sie dürfen davon ausgehen, dass kein Eintrag im Feld null ist.

```
void sortierenDurchEinfuegen(String[] a) {
  // Hilfsvariable:
  String tmp;
}
```

Lösungsvorschlag

```
static void sortierenDurchEinfuegen(String[] a) {
  // Hilfsvariable:
  String tmp;
  for (int i = 1; i < a.length; i++) {
    tmp = a[i];
    int j = i;
    while (j > 0 && a[j - 1].compareTo(tmp) >= 1) {
      a[j] = a[j - 1];
      j = j - 1;
    }
    a[j] = tmp;
  }
```

Code-Beispiel auf Github ansehen: `src/main/java/org/bschlangaul/examen/examen_46115/jahr_2017/herbst/InsertionSort.java`