Wegberechnung im Gitter[1]

Betrachten Sie das folgende Gitter mit $m+1$ Zeilen und $n+1$ Spalten ($m \geq 1$ und $n \geq 1$): [2] geeksforgeeks [3]

Angenommen, Sie befinden sich zu Beginn am Punkt $(0,0)$ und wollen zum Punkt (m,n).

Für die Anzahl $A(i,j)$ aller verschiedenen Wege vom Punkt $(0,0)$ zum Punkt (i,j) lassen sich folgende drei Fälle unterscheiden (es geht jeweils um die kürzesten Wege ohne Umweg!):

- $1 \leq i \leq m$ und $j = 0$:

  Es gibt genau einen Weg von $(0,0)$ nach $(i,0)$ für $1 \leq i \leq m$.

- $i = 0$ und $1 \leq j \leq n$:

  Es gibt genau einen Weg von $(0,0)$ nach $(0,j)$ für $1 \leq j \leq n$.

- $1 \leq i \leq m$ und $1 \leq j \leq n$:

  auf dem Weg zu (i,j) muss als vorletzter Punkt entweder $(i-1,j)$ oder $(i,j-1)$ besucht worden sein.

Daraus ergibt sich folgende Rekursionsgleichung:

$$A(i,j) = \begin{cases} 1 & \text{falls } (1 \leq i \leq m \text{ und } j = 0) \text{ oder } (i = 0 \text{ und } 1 \leq j \leq n) \\ A(i-1,j) + A(i,j-1) & \text{falls } 1 \leq i \leq m \text{ und } 1 \leq j \leq n \end{cases}$$

Implementieren Sie die Java-Klasse `Gitter` mit der Methode

```java
public int berechneAnzahlWege(),
```

die ausgehend von der Rekursionsgleichung durch dynamische Programmierung die Anzahl aller Wege vom Punkt $(0,0)$ zum Punkt (m,n) berechnet. Die Überprüfung, ob $m \leq 1$ und $n \leq 1$ gilt, können Sie der Einfachheit halber weglassen.

```java

  public int berechneAnzahlWege() {
    int i, j;
    for (i = 1; i <= m; i++) {
      anzahlWege[i][0] = 1;
    }
    for (j = 1; j <= n; j++) {
      anzahlWege[0][j] = 1;
    }
    for (i = 1; i <= m; i++) {
      for (j = 1; j <= n; j++) {
        anzahlWege[i][j] = anzahlWege[i - 1][j] + anzahlWege[i][j - 1];
```

[1] *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen: Aufgabenblatt 3: Algorithmenmuster*, Seite 1, Dynamische Programmierung, Aufgabe 2.

[2] Quelle möglicherweise von `https://www.yumpu.com/de/document/read/17936760/ubungen-zum-prasenzmodul-algorithmen-und-datenstrukturen`

[3] `https://www.geeksforgeeks.org/count-possible-paths-top-left-bottom-right-nxm-matrix/`

```java
      }
    }
    return anzahlWege[m][n];
```

Code-Beispiel auf Github ansehen: `src/main/java/org/bschlangaul/aufgaben/aud/ab_3/Gitter.java`

Die komplette Java-Klasse

```java
import org.bschlangaul.helfer.Farbe;
import org.bschlangaul.helfer.Konsole;

/**
 * <a href="https://www.studon.fau.de/file2521908_download.html">Angabe:
↪  AB_3 Greedy_DP_Backtracking.pdf</a>
 * <a href="https://www.studon.fau.de/file2521907_download.html">Lösung:
↪  AB_3 Greedy_DP_Backtracking_Lsg.pdf</a>
 */
public class Gitter {

  /**
   * m + 1: Anzahl der Zeilen
   */
  private int m;

  /**
   * n + 1: Anzahl der Spalten
   */
  private int n;

  /**
   * anzahlWege[i][j]: Anzahl der Wege vom Punkt (0,0) zum Punkt (i,j)
   */
  private int anzahlWege[][];

  public Gitter(int m, int n) {
    this.m = m;
    this.n = n;
    anzahlWege = new int[m + 1][n + 1];
  }

  public int berechneAnzahlWege() {
    int i, j;
    for (i = 1; i <= m; i++) {
      anzahlWege[i][0] = 1;
    }
    for (j = 1; j <= n; j++) {
      anzahlWege[0][j] = 1;
    }
    for (i = 1; i <= m; i++) {
      for (j = 1; j <= n; j++) {
        anzahlWege[i][j] = anzahlWege[i - 1][j] + anzahlWege[i][j - 1];
      }
    }
    return anzahlWege[m][n];
  }

  /**
   * Zeige die Lösung in der Konsole.
   */
```

```java
  public void zeigeLoesung() {
    System.out.println(String.format("Anzahl der Wege von %sx%s: %s",
      Farbe.gelb(m), Farbe.gelb(n), Farbe.grün(berechneAnzahlWege())));
    System.out.println(Farbe.rot("Gitter:"));
    Konsole.zeige2DIntFeld(anzahlWege);
    System.out.println();
  }

  public static void main(String args[]) {
    new Gitter(2, 2).zeigeLoesung();
    new Gitter(3, 3).zeigeLoesung();
    new Gitter(4, 4).zeigeLoesung();
    new Gitter(5, 5).zeigeLoesung();
  }
}
```

Code-Beispiel auf Github ansehen: `src/main/java/org/bschlangaul/aufgaben/aud/ab_3/Gitter.java`

Text-Ausgabe

```
Anzahl der Wege von 2x2: 6
Gitter:
 x 0 1 2
 0 0 1 1
 1 1 2 3
 2 1 3 6

Anzahl der Wege von 3x3: 20
Gitter:
  x  0  1  2  3
  0  0  1  1  1
  1  1  2  3  4
  2  1  3  6 10
  3  1  4 10 20

Anzahl der Wege von 4x4: 70
Gitter:
  x  0  1  2  3  4
  0  0  1  1  1  1
  1  1  2  3  4  5
  2  1  3  6 10 15
  3  1  4 10 20 35
  4  1  5 15 35 70

Anzahl der Wege von 5x5: 252
Gitter:
   x   0   1   2   3   4   5
   0   0   1   1   1   1   1
   1   1   2   3   4   5   6
   2   1   3   6  10  15  21
   3   1   4  10  20  35  56
   4   1   5  15  35  70 126
   5   1   6  21  56 126 252
```

Test-Datei

```java
import static org.junit.Assert.*;
import org.junit.Test;
```

```java
public class GitterTest {
  @Test
  public void zweiMailZwei() {
    Gitter gitter = new Gitter(2, 2);
    assertEquals(6, gitter.berechneAnzahlWege());
  }

  @Test
  public void zehnMalZwanzig() {
    Gitter gitter = new Gitter(10, 20);
    assertEquals(30045015, gitter.berechneAnzahlWege());
  }
}
```

Code-Beispiel auf Github ansehen: `src/test/java/org/bschlangaul/aufgaben/aud/ab_3/GitterTest.java`

Aufgabe 7: Dynamische Programmierung[4]

Mittels Dynamischer Programmierung (auch Memoization genannt) kann man insbesondere rekursive Lösungen auf Kosten des Speicherbedarf beschleunigen, indem man Zwischenergebnisse „abspeichert" und bei (wiederkehrendem) Bedarf „abruft", ohne sie erneut berechnen zu müssen.[5]

Gegeben sei folgende geschachtelt-rekursive Funktion für $n, m \geq 0$:

$$a(n,m) = \begin{cases} n + \lfloor \frac{n}{2} \rfloor & \text{falls } m = 0 \\ a(1, m-1), & \text{falls } n = 0 \land m \neq 0 \\ a(n + \lfloor \sqrt{a(n-1,m)} \rfloor, m-1), & \textit{sonst} \end{cases}$$

(a) Implementieren Sie die obige Funktion `a(n,m)` zunächst ohne weitere Optimierungen als Prozedur/Methode in einer Programmiersprache Ihrer Wahl.

```java
public static long a(int n, int m) {
  if (m == 0) {
    return n + (n / 2);
  } else if (n == 0 && m != 0) {
    return a(1, m - 1);
  } else {
    return a(n + ((int) Math.sqrt(a(n - 1, m))), m - 1);
  }
}
```

Code-Beispiel auf Github ansehen:
`src/main/java/org/bschlangaul/examen/examen_46115/jahr_2016/herbst/DynamischeProgrammierung.java`

(b) Geben Sie nun eine DP-Implementierung der Funktion `a(n,m)` an, die `a(n,m)` für $0 \geq n \geq 100000$ und $0 \geq m \geq 25$ höchstens einmal gemäß obiger rekursiver Definition berechnet. Beachten Sie, dass Ihre Prozedur trotzdem auch weiterhin mit $n > 100000$ und $m > 25$ aufgerufen werden können soll.

[4] *Staatsexamen 46115 Theoretische Informatik / Algorithmen / Datenstrukturen (nicht vertieft) 2016 Herbst*, Thema 2 Aufgabe 4.

[5] *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen: Präsenz- und Aufgabenblatt Wiederholung*.

```java
  static long[][] tmp = new long[100001][26];

  public static long aDp(int n, int m) {
    if (n <= 100000 && m <= 25 && tmp[n][m] != -1) {
      return tmp[n][m];
    } else {
      long merker;
      if (m == 0) {
        merker = n + (n / 2);
      } else if (n == 0 && m != 0) {
        merker = aDp(1, m - 1);
      } else {
        merker = aDp(n + ((int) Math.sqrt(aDp(n - 1, m))), m - 1);
      }
      if (n <= 100000 && m <= 25) {
        tmp[n][m] = merker;
      }
      return merker;
    }
  }
```

Code-Beispiel auf Github ansehen:
`src/main/java/org/bschlangaul/examen/examen_46115/jahr_2016/herbst/DynamischeProgrammierung.java`

Kompletter Code

```java
public class DynamischeProgrammierung {
  public static long a(int n, int m) {
    if (m == 0) {
      return n + (n / 2);
    } else if (n == 0 && m != 0) {
      return a(1, m - 1);
    } else {
      return a(n + ((int) Math.sqrt(a(n - 1, m))), m - 1);
    }
  }

  static long[][] tmp = new long[100001][26];

  public static long aDp(int n, int m) {
    if (n <= 100000 && m <= 25 && tmp[n][m] != -1) {
      return tmp[n][m];
    } else {
      long merker;
      if (m == 0) {
        merker = n + (n / 2);
      } else if (n == 0 && m != 0) {
        merker = aDp(1, m - 1);
      } else {
        merker = aDp(n + ((int) Math.sqrt(aDp(n - 1, m))), m - 1);
      }
      if (n <= 100000 && m <= 25) {
        tmp[n][m] = merker;
      }
      return merker;
    }
  }

  public static void main(String[] args) {
```

```java
    for (int i = 0; i < 100001; i++) {
      for (int j = 0; j < 26; j++) {
        tmp[i][j] = -1;
      }
    }
    System.out.println("schnell mit DP: " + aDp(7,7));
    System.out.println("langsam ohne DP: " + a(7,7));
  }
}
```

Code-Beispiel auf Github ansehen: src/main/java/org/bschlangaul/examen/examen_46115/jahr_2016/herbst/DynamischeProgrammierung.java

Aufgabe 3[6]

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

```java
  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:

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

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

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

[6] **46115:2017:09**.

```java
  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

```java
/**
 * 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;
```

```java
    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`

Aufgaben zu Rekursion und Dynamische Programmierung anhand der Fibonacci-Zahlen[7]

Gegeben[8] seien die folgenden Formeln zur Berechnung der *ersten* Fibonacci-Zahlen[9]:

$$\text{fib}_n = \begin{cases} 1 & \text{falls } n \leq 2 \\ \text{fib}_{n-1} + \text{fib}_{n-2} & \text{sonst} \end{cases}$$

sowie der Partialsumme der Fibonacci-Quadrate:

$$\text{sos}_n = \begin{cases} \text{fib}_n & \text{falls } n = 1 \\ \text{fib}_n^2 + \text{sos}_{n-1} & \text{sonst} \end{cases}$$

Sie dürfen im Folgenden annehmen, dass die Methoden nur mit $1 \leq n \leq 46$ aufgerufen werden, so dass der Datentyp `long` zur Darstellung aller Werte ausreicht.

> **Exkurs: Fibonacci-Folge**
> Die Fibonacci-Folge beginnt zweimal mit der Zahl 1. Im Anschluss ergibt jeweils die Summe zweier aufeinanderfolgender Zahlen die unmittelbar danach folgende Zahl: 1, 1, 2, 3, 5, 8, 13[a]
>
> [a] *Wikipedia-Artikel „Fibonacci-Folge“*.

> **Exkurs: Partialsumme**
> Unter der n-ten Partialsumme s_n einer Zahlenfolge a_n versteht man die Summe der Folgenglieder von a_1 bis a_n. Die immer weiter fortgesetzte Partialsumme einer (unendlichen) Zahlenfolge nennt man eine (unendliche) Reihe. [a] Partialsummen sind das Bindeglied zwischen Summen und Reihen. Gegeben sei die Reihe $\sum_{k=1}^{\infty} a_k$. Die n-te Partialsumme dieser Reihe lautet: $\sum_{k=1}^{n} a_k$. ðwir summieren unsere Reihe nur bis zum Endindex n. [b]
>
> [a] `https://www.lernhelfer.de/schuelerlexikon/mathematik/artikel/folgen-partialsummen`
> [b] `https://www.massmatics.de/merkzettel/index.php#!164:Partialsummen`

[`sos` steht für *Summe of Squares*]

[7] *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen: Präsenzübung 3: Algorithmenmuster*, Seite 1.
[8] *Staatsexamen 66115 Theoretische Informatik / Algorithmen (vertieft) 2017 Frühjahr*, Thema 1 Aufgabe 3 Seite 5.
[9] *Wikipedia-Artikel „Fibonacci-Folge“*.

n	fib_n	fib_n^2		$\sum_{k=1}^{n} \text{fib}^k$
1	1	1	1	1
2	1	1	$1+1$	2
3	2	4	$1+1+4$	6
4	3	9	$1+1+4+9$	15
5	5	25	$1+1+4+9+25$	40
6	8	64	$1+1+4+9+25+64$	104
7	13	169	$1+1+4+9+25+64+169$	273
8	21	441	$1+1+4+9+25+64+169+441$	714
9	34	1156	$1+1+4+9+25+64+169+441+1156$	1870
10	55	3025	$1+1+4+9+25+64+169+441+1156+3025$	4895

(a) Implementieren Sie die obigen Formeln zunächst rekursiv (ohne Schleifenkonstrukte wie `for` oder `while`) und ohne weitere Optimierungen („naiv") in Java als:

```
long fibNaive (int n) {
```

bzw.

```
long sosNaive (int n) {
```

```
public static long fibNaive(int n) {
  if (n <= 2) {
    return 1;
  }
  return fibNaive(n - 1) + fibNaive(n - 2);
}

public static long sosNaive(int n) {
  if (n <= 1) {
    return fibNaive(n);
  }
  return fibNaive(n) * fibNaive(n) + sosNaive(n - 1);
}
```

Code-Beispiel auf Github ansehen:
`src/main/java/org/bschlangaul/examen/examen_66115/jahr_2017/fruehjahr/Fibonacci.java`

(b) Offensichtlich ist die naive Umsetzung extrem ineffizient, da viele Zwischenergebnisse wiederholt rekursiv ausgewertet werden müssen. Die Dynamische Programmierung (DP) erlaubt es Ihnen, die Laufzeit auf Kosten des Speicherbedarfs zu reduzieren, indem Sie alle einmal berechneten Zwischenergebnisse speichern und bei erneutem Bedarf „direkt abrufen". Implementieren Sie obige Formeln nun rekursiv aber mittels DP in Java als:

```
long fibDP (int n) {
```

bzw.

```
long sosDP (int n) {
```

```java
    public static long fibDP(int n) {
      // Nachschauen, ob die Fibonacci-Zahl bereits berechnet wurde.
      if (fib[n] != 0) {
        return fib[n];
      }
      // Die Fibonacci-Zahl neu berechnen.
      if (n <= 2) {
        fib[n] = 1;
      } else {
        fib[n] = fibDP(n - 1) + fibDP(n - 2);
      }
      return fib[n];
    }

    public static long sosDP(int n) {
      // Nachschauen, ob die Quadratsumme bereits berechnet wurde.
      if (sos[n] != 0) {
        return sos[n];
      }
      // Die Quadratsumme neu berechnen.
      if (n <= 1) {
        sos[n] = fibDP(n);
      } else {
        long tmp = fibDP(n);
        sos[n] = tmp * tmp + sosDP(n - 1);
      }
      return sos[n];
    }
```

Code-Beispiel auf Github ansehen:
`src/main/java/org/bschlangaul/examen/examen_66115/jahr_2017/fruehjahr/Fibonacci.java`

(c) Am „einfachsten“ und bzgl. Laufzeit [in $\mathcal{O}(n)$] sowie Speicherbedarf [in $\mathcal{O}(1)$] am effizientesten ist sicherlich eine iterative Implementierung der beiden Formeln. Geben Sie eine solche in Java an als:

```java
long fibIter (int n) {
```

bzw.

```java
long sosIter (int n) {
```

```java
    public static long fibIter(int n) {
      long a = 1;
      long b = 1;
      for (int i = 2; i < n; i++) {
        long tmp = a + b;
        b = a;
        a = tmp;
      }
      return a;
    }

    public static long sosIter(int n) {
      long a = 1;
      long b = 0;
      long sosSum = 1;
      for (int i = 2; i <= n; i++) {
        long tmp = a + b;
        b = a;
        a = tmp;
```

```java
      sosSum += a * a;
    }
    return sosSum;
  }
```

Code-Beispiel auf Github ansehen:
src/main/java/org/bschlangaul/examen/examen_66115/jahr_2017/fruehjahr/Fibonacci.java

Kompletter Code

```java
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

/**
 * <a href="https://www.studon.fau.de/file2861014_download.html">Angabe:
 * PUE_AUD_3.pdf</a>
 * <a href="https://www.studon.fau.de/file2893062_download.html">Lösung:
 * PUE_AUD_3_Lsg.pdf</a>
 */
public class Fibonacci {
  static long[] fib = new long[47];
  static long[] sos = new long[47];

  public static long fibNaive(int n) {
    if (n <= 2) {
      return 1;
    }
    return fibNaive(n - 1) + fibNaive(n - 2);
  }

  public static long sosNaive(int n) {
    if (n <= 1) {
      return fibNaive(n);
    }
    return fibNaive(n) * fibNaive(n) + sosNaive(n - 1);
  }

  public static long fibDP(int n) {
    // Nachschauen, ob die Fibonacci-Zahl bereits berechnet wurde.
    if (fib[n] != 0) {
      return fib[n];
    }
    // Die Fibonacci-Zahl neu berechnen.
    if (n <= 2) {
      fib[n] = 1;
    } else {
      fib[n] = fibDP(n - 1) + fibDP(n - 2);
    }
    return fib[n];
  }

  public static long sosDP(int n) {
    // Nachschauen, ob die Quadratsumme bereits berechnet wurde.
    if (sos[n] != 0) {
      return sos[n];
    }
    // Die Quadratsumme neu berechnen.
    if (n <= 1) {
      sos[n] = fibDP(n);
```

```java
      } else {
        long tmp = fibDP(n);
        sos[n] = tmp * tmp + sosDP(n - 1);
      }
      return sos[n];
    }

    public static long fibIter(int n) {
      long a = 1;
      long b = 1;
      for (int i = 2; i < n; i++) {
        long tmp = a + b;
        b = a;
        a = tmp;
      }
      return a;
    }

    public static long sosIter(int n) {
      long a = 1;
      long b = 0;
      long sosSum = 1;
      for (int i = 2; i <= n; i++) {
        long tmp = a + b;
        b = a;
        a = tmp;
        sosSum += a * a;
      }
      return sosSum;
    }

    public static String fixColumn(long n) {
      return (n + "     ").substring(0, 5);
    }

    public static void invokeStaticMethod(String methodeName)
        throws NoSuchMethodException, IllegalAccessException, InvocationTargetException {
      Method method = Fibonacci.class.getMethod(methodeName, int.class);
      System.out.println("\n" + methodeName);
      for (int i = 1; i <= 10; i++) {
        System.out.print(fixColumn((long) method.invoke(null, i)));
      }
    }

    public static void main(String args[])
        throws NoSuchMethodException, IllegalAccessException, InvocationTargetException {
      invokeStaticMethod("fibNaive");
      invokeStaticMethod("sosNaive");
      invokeStaticMethod("fibDP");
      invokeStaticMethod("sosDP");
      invokeStaticMethod("fibIter");
      invokeStaticMethod("sosIter");
      System.out.println();
    }
  }
```

Code-Beispiel auf Github ansehen: `src/main/java/org/bschlangaul/examen/examen_66115/jahr_2017/fruehjahr/Fibonacci.java`

Aufgabe 4[10]

Das GUTSCHEIN-Problem ist gegeben durch eine Folge $w_1, \ldots, w_n$ von Warenwerten (wobei $w \in N_0$ für $i = 1, \ldots, n$) und einem Gutscheinbetrag $G \in \mathbb{N}_0$.

Da Gutscheine nicht in Bargeld ausgezahlt werden können, ist die Frage, ob man eine Teilfolge der Waren findet, sodass man genau den Gutschein ausnutzt. Formal ist dies die Frage, ob es eine Menge von Indizes I mit $I \subseteq \{1, \ldots, n\}$ gibt, sodass $\sum_{i \in I} w_i = G$

(a) Sei $w_1 = 10, w_2 = 30, w_3 = 40, w_4 = 20, w_5 = 15$ eine Folge von Warenwerten.

(i) Geben Sie einen Gutscheinbetrag $40 < G < 115$ an, sodass die GUTSCHEIN-Instanz eine Lösung hat. Geben Sie auch die lösende Menge $I \subseteq \{1, 2, 3, 4, 5\}$ von Indizes an.

> 50
> $I = \{1, 3\}$

(ii) Geben Sie einen Gutscheinbetrag G mit $40 < G < 115$ an, sodass die GUTSCHEIN-Instanz keine Lösung hat.

> 51

(b) Sei *table* eine $(n \times (G+1))$-Tabelle mit Einträgen table$[i,k]$, für $1 < i < n$ und $0 \leq k \leq G$, sodass

$$\text{table}[i,k] = \begin{cases} \textbf{true} & \text{falls es } I \subseteq \{1, \ldots, n\} \text{ mit } \sum_{i \in I} w_i = G \text{ gibt} \\ \textbf{false} & \text{sonst} \end{cases}$$

Geben Sie einen Algorithmus in Pseudo-Code oder Java an, der die Tabelle *table* mit *dynamischer Programmierung* in Worst-Case-Laufzeit $\mathcal{O}(n \times G)$ erzeugt. Begründen Sie die Korrektheit und die Laufzeit Ihres Algorithmus. Welcher Eintrag in *table* löst das GUTSCHEIN-Problem?

```java
// import java.util.Arrays;

/**
 * https://www.geeksforgeeks.org/subset-gutscheinBetrag-problem-dp-25/
 */
public class Gutschein {
  /**
   * @param gutscheinBetrag Das GUTSCHEIN-Betrag von 0, 1, ...
   *
   * @param warenWerte      Das GUTSCHEIN-Problem ist gegeben durch
→  eine Folge w1,
   *                       ..., wn von Warenwerten.
   *
   * @return Wahr, wenn der Gutscheinbetrag vollständig in Warenwerten
→  eingelöst
   *         werden kann, falsch wenn der Betrag nicht vollständig
→  eingelöst
```

[10] **66115:2020:09**.

```java
   *         werden kann.
   */
  public static boolean gutscheinDP(int gutscheinBetrag, int warenWerte[]) {
    // Der Eintrag in der Tabelle tabelle[i][k] ist wahr,
    // wenn es eine Teilsumme der
    // warenWerte[0..i-1] gibt, die gleich k ist.
    int n = warenWerte.length;
    boolean tabelle[][] = new boolean[n + 1][gutscheinBetrag + 1];

    // Wenn der Gutschein-Betrag größer als 0 ist und es keine
    // Warenwerte (n = 0) gibt, kann der Gutschein nicht eingelöst
    // werden.
    for (int k = 1; k <= gutscheinBetrag; k++)
      tabelle[0][k] = false;

    // Ist der Gutscheinbetrag 0, dann kann er immer eingelöst werden.
    for (int i = 0; i <= n; i++)
      tabelle[i][0] = true;

    for (int i = 1; i <= n; i++) {
      for (int k = 1; k <= gutscheinBetrag; k++) {
        tabelle[i][k] = tabelle[i - 1][k];
        if (k >= warenWerte[i - 1])
          tabelle[i][k] = tabelle[i][k] || tabelle[i - 1][k - warenWerte[i - 1]];
      }
    }
    // System.out.println(Arrays.deepToString(tabelle));
    return tabelle[n][gutscheinBetrag];
  }

  public static void main(String[] args) {
    System.out.println(gutscheinDP(10, new int[] { 10, 30, 40, 20, 15 }));
    System.out.println(gutscheinDP(3, new int[] { 1, 2, 3 }));
  }
}
```

Code-Beispiel auf Github ansehen: `src/main/java/org/bschlangaul/examen/examen_66115/jahr_2020/herbst/Gutschein.java`

Die äußere for-Schleife läuft n mal und die innere for-Schleife G mal. Der letzte Eintrag in der Tabelle, also der Wert in der Zelle: `tabelle[warenWerte.length][gutscheinBetrag]`.

```java
import static org.junit.Assert.*;
import org.junit.Test;

public class GutscheinTest {

  int[] warenWerte = new int[] { 10, 30, 40, 20, 15 };

  private void assertEingelöst(int gutscheinBetrag, int[] warenWerte) {
    assertEquals(Gutschein.gutscheinDP(gutscheinBetrag, warenWerte), true);
  }

  private void assertNichtEingelöst(int gutscheinBetrag, int[] warenWerte) {
    assertEquals(Gutschein.gutscheinDP(gutscheinBetrag, warenWerte), false);
  }
```

```java

  @Test
  public void eingelöst() {
    assertEingelöst(0, warenWerte);
    assertEingelöst(10, warenWerte);
    assertEingelöst(100, warenWerte);
    assertEingelöst(115, warenWerte);
    assertEingelöst(15, warenWerte);
    assertEingelöst(20, warenWerte);
    assertEingelöst(30, warenWerte);
    assertEingelöst(40, warenWerte);
    assertEingelöst(60, warenWerte);
    assertEingelöst(70, warenWerte);
  }

  @Test
  public void nichtEingelöst() {
    assertNichtEingelöst(11, warenWerte);
    assertNichtEingelöst(31, warenWerte);
    assertNichtEingelöst(41, warenWerte);
    assertNichtEingelöst(21, warenWerte);
    assertNichtEingelöst(16, warenWerte);
    assertNichtEingelöst(999, warenWerte);
  }
}
```

Code-Beispiel auf Github ansehen: `src/test/java/org/bschlangaul/examen/examen_66115/jahr_2020/herbst/GutscheinTest.java`

- `gutscheinDP(3, new int[] { 1, 2, 3 }));`: wahr (w)

```
[
  [w, f, f, f],
  [w, w, f, f],
  [w, w, w, w],
  [w, w, w, w]
]
```

- `gutscheinDP(7, new int[] { 1, 2, 3 });`: falsch (f)

```
[
  [w, f, f, f, f, f, f, f],
  [w, w, f, f, f, f, f, f],
  [w, w, w, w, f, f, f, f],
  [w, w, w, w, w, w, w, f]
]
```

- `gutscheinDP(10, new int[] { 10, 30, 40, 20, 15 });`: wahr (w)

```
[
  [w, f, f, f, f, f, f, f, f, f, f],
  [w, f, f, f, f, f, f, f, f, f, w],
  [w, f, f, f, f, f, f, f, f, f, w],
  [w, f, f, f, f, f, f, f, f, f, w],
  [w, f, f, f, f, f, f, f, f, f, w],
  [w, f, f, f, f, f, f, f, f, f, w]
]
```