Aufgaben zu Rekursion und Dynamische Programmierung anhand der Fibonacci-Zahlen

Gegeben seien die folgenden Formeln zur Berechnung der *ersten* Fibonacci-Zahlen:

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

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

> **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: $\sum_{k=1}^{n} a_k$. d. h. wir 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*]

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) {
```

```
16  public static long fibNaive(int n) {
17    if (n <= 2) {
18      return 1;
19    }
20    return fibNaive(n - 1) + fibNaive(n - 2);
21  }
22
23  public static long sosNaive(int n) {
24    if (n <= 1) {
25      return fibNaive(n);
26    }
27    return fibNaive(n) * fibNaive(n) + sosNaive(n - 1);
28  }
```

(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) {
```

```
30  public static long fibDP(int n) {
31    // Nachschauen, ob die Fibonacci-Zahl bereits berechnet wurde.
32    if (fib[n] != 0) {
33      return fib[n];
34    }
35    // Die Fibonacci-Zahl neu berechnen.
36    if (n <= 2) {
37      fib[n] = 1;
38    } else {
39      fib[n] = fibDP(n - 1) + fibDP(n - 2);
40    }
41    return fib[n];
42  }
43
44  public static long sosDP(int n) {
45    // Nachschauen, ob die Quadratsumme bereits berechnet wurde.
46    if (sos[n] != 0) {
47      return sos[n];
48    }
49    // Die Quadratsumme neu berechnen.
50    if (n <= 1) {
51      sos[n] = fibDP(n);
52    } else {
```

```
53          long tmp = fibDP(n);
54          sos[n] = tmp * tmp + sosDP(n - 1);
55        }
56        return sos[n];
57      }
```

(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:

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

bzw.

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

```
59      public static long fibIter(int n) {
60        long a = 1;
61        long b = 1;
62        for (int i = 2; i < n; i++) {
63          long tmp = a + b;
64          b = a;
65          a = tmp;
66        }
67        return a;
68      }
69
70      public static long sosIter(int n) {
71        long a = 1;
72        long b = 0;
73        long sosSum = 1;
74        for (int i = 2; i <= n; i++) {
75          long tmp = a + b;
76          b = a;
77          a = tmp;
78          sosSum += a * a;
79        }
80        return sosSum;
81      }
```