Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Programmieren 2

Prof. Dr. Larissa Putzar &
Thorben Ortmann

Rekursion, Komplexität und Nebenläufigkeit

Aufgabe 2

1. Erstellen Sie eine Klasse *RecursionAlgorithms.*
2. Fügen Sie je eine statische Methode für die Berechnung der Fakultätsfunktion *factorial(n)*, der Exponentialfunktion *power(a,b)* und der Fibonacci-Folge *fib(n)* hinzu. Implementieren Sie die Methoden noch nicht, sondern schreiben Sie nur ihre Signatur.
3. Schreiben Sie Testfälle für verschiedene Eingabewerte für alle Methoden.
4. Implementieren Sie alle Methoden, sodass Ihre Testfälle nun „grün werden“.

Aufgabe 3

Geben Sie begründet die Komplexität der Exponentialfunktion in O-Notation an.

Aufgabe 4

Fügen Sie Ihrer Klasse *RecursionAlgorithms* eine statische Methode namens *fibConcurrent()* hinzu. Diese nutzt einen *ForkJoinPool* und *RecursiveTasks* zur Berechnung der Fibonacci-Folge.

(Sie können Ihre bisherigen Testfälle für *RecursionAlgorithms.fib()* hier wiederverwenden, da Eingabe und Ausgabe mit der sequentiellen Ausführung überein stimmen sollen.)

Welcher Schwellwert zum Abbruch der Rekursion im *FibonacciTask* ist sinnvoll?

Messen Sie die Ausführungszeit für die sequentielle Ausführung von *RecursionAlgorithms.fib()* und *RecursionAlgorithms.fibConcurrent()* mit verschiedenen n. Variieren Sie bei *fibConcurrent()* zudem auch den Schwellwert. Zeitmessungen können Sie etwa wie folgt durchführen:

```
var start = Instant.now();
RecursionAlgorithms.fib(42);
var finish = Instant.now();
System.out.println("Elapsed Time: " + Duration.between(start, finish).toMillis());
```

Aufgabe 5

Durch Nebenläufigkeit kann die Berechnungsdauer eines Algorithmus erheblich gesenkt werden. Allerdings ist es noch besser einen Algorithmus an sich „schlauer“ bzw. weniger komplex zu machen.

So gibt es einen Algorithmus für die Exponentialfunktion mit logarithmischer Komplexität O(log n). Dieser nennt sich *schnelle Exponentiation*. Recherchieren Sie den Algorithmus und implementieren Sie ihn als *RecursionAlgorithms.powerFast(a, b).* Sie können Ihre Testfälle für *RecursionAlgorithms.power()* wiederverwenden.

Aufgabe 6

Durch Nebenläufigkeit kann die Berechnungsdauer eines Algorithmus erheblich gesenkt werden. Allerdings ist es noch besser einen Algorithmus an sich „schlauer“ bzw. weniger komplex zu machen.

So gibt es auch für die Berechnung der Fibonacci-Folge deutlich weniger komplexe Algorithmen. Eine Lösung in linearer Komplexität O(n) ist möglich, wenn man die Berechnung von „unten“ nach „oben“ durchführt und sich die Zwischenergebnisse „merkt“. Implementieren Sie diesen Algorithmus als *fibSmart(n). fibSmart(n)* benutzt eine Unterfunktion *subFib(n, f1, f2)*, wobei beim ersten Aufruf *n* gleich dem *n* aus *fibSmart(n)* ist und *f1* und *f2* die ersten beiden Fibonacci-Zahlen sind. *subFib* ruft sich dann bis zum Abbruch immer wieder selbst und addiert bei jedem Schritt *f1* und *f2*.