Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
Arbeitsgruppe Datenbanksysteme

Sommersemester 2021
18. 04. 2021

Prof. Dr. Bernhard Seeger
Nikolaus Glombiewski M.Sc.

Übungen zur Vorlesung
Algorithmen und Datenstrukturen

Abgabe: 25. 04. 2021,
bis **spätestens** 23:59 Uhr
über die ILIAS Plattform

Übung 1

Aufgabe 1.1: Goldbachsche Vermutung (2+2+2) (6 Punkte)

Die Goldbachsche Vermutung ist eine (bisher unbewiesene) Aussage aus der Zahlentheorie. Sie besagt, dass jede gerade Zahl größer als 2 durch die Summe zweier Primzahlen repräsentiert werden kann. Gezeigt wurde diese Vermutung nur für alle geraden Zahlen bis $4 \cdot 10^{18}$.

a) Bestimmen Sie, analog zur Vorlesung (ggT), die Vor- und Nachbedingungen für einen Algorithmus, der eine gerade Zahl z in zwei Primzahlen n und m zerlegt.

b) Implementieren Sie die Vor- und Nachbedingungen als boolesche-Methoden in der gegebenen Java Klasse `Goldbach`.

c) Implementieren Sie einen entsprechenden Algorithmus in der Klasse `Goldbach`. Der Algorithmus soll beide Primzahlen als `Pair` zurückgeben. Verwenden Sie für die Eingabe einen Datentyp, der ganze Zahlen bis $4 \cdot 10^{18}$ repräsentieren kann.

Aufgabe 1.2: Invarianten (2+2) (4 Punkte)

Gegeben ist folgender Algorithmus, der die Potenz X^N für $N \in \mathbb{N}$ berechnet. Dabei ist `div` die ganzzahlige Division und `mod` der Rest der Division.

```
1  {x == X AND n == N}
2  p = 1;
3  if x ≠ 0 then
4      while n ≠ 0 do
5          if n mod 2 == 1 then
6              p = p · x;
7          n = n div 2;
8          x = x · x;
9  else
10     p = 0;
11 {p == X^N}
```

a) Durchlaufen Sie den Algorithmus für folgende Werte: $X = 2$, $N =$7. Geben Sie die Werte von x, n, p nach jedem Schleifendurchlauf an.

b) Finden Sie eine geeignete Schleifeninvariante. Begründen Sie Ihre Lösung.

Aufgabe 1.3: Binäre Suche (1+1+1+2) **(5 Punkte)**

In dieser Aufgabe sollen Sie die binäre Suche auf einem generischen, aufsteigend sortierten Array (mit Duplikaten) implementieren und verwenden, um verschiedene Suchoperationen umzusetzen. Arbeiten Sie hierzu in der gegebenen Klasse `BinSearch`. Implementieren Sie Ihre eigene Version der binären Suche. Nutzen Sie für Vergleiche und Überprüfung auf Gleichheit das `Comparable` Interface. Werfen Sie bei erfolglosen Suchen eine passende Exception.

a) Vervollständigen Sie die Methode `isSorted`, indem Sie das Array mit einer `for`-Schleife durchlaufen, um zu überprüfen, ob es sortiert ist. Geben Sie einen entsprechenden booleschen Wert zurück.

b) Vervollständigen Sie die Methode `findFirst`, indem Sie zunächst mit der binären Suche ein beliebiges Auftreten des gesuchten Elementes finden und anschließend ausgehend von diesem das erste Vorkommen eines gleichen Elementes (gemäß compareTo) im Array (d.h. mit kleinstem Array Index) finden und zurückgeben.

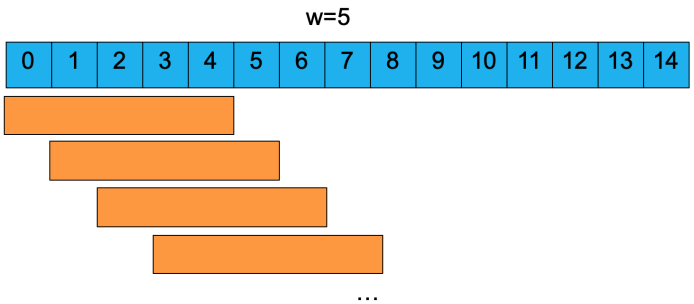
c) Vervollständigen Sie die Methode `findLast`, indem Sie zunächst mit der binären Suche ein beliebiges Auftreten des gesuchten Elementes finden und anschließend ausgehend von diesem das letzte Vorkommen eines gleichen Elementes (gemäß compareTo) im Array (d.h. mit größten Array Index) finden und zurückgeben.

d) Vervollständigen Sie die Methode `findAll`, indem Sie zunächst mit der binären Suche ein beliebiges Auftreten des gesuchten Elementes finden. Implementieren Sie nun einen Iterator, welche das gefundene Element nutzt, um sämtliche gleiche Elemente aufsteigend nach Ihrem Array Index zurückzugeben.

Aufgabe 1.4: Wiederholung: Arrays und Schleifen* (3+2) **(5 Punkte)**

Zur Wiederholung zum Umgang mit Arrays und Schleifen werden in dieser Aufgabe sogenannte `Window`-Funktionen aus dem Bereich von Datenstrom Analyse auf Arrays adaptiert. Eine `Window`-Funktion betrachtet mehrere Bereiche eines Arrays, jeweils mit fester Größe w, und führt auf diesen eine Funktion f aus. Vervollständigen Sie zur Bearbeitung dieser Aufgabe die gegebene Klasse `WindowFunctions`.

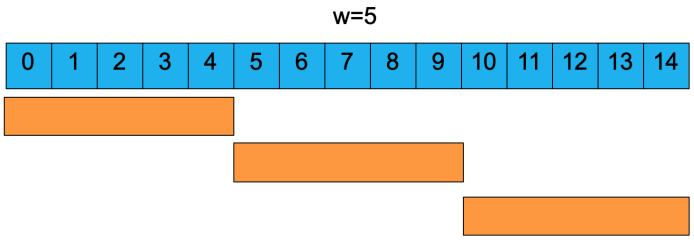
a) Ein *Sliding Window* berechnet zunächst auf den ersten w Elementen eines Arrays (ab Array Index 0) die Funktion f. Anschließend wandert das *Window* ein Element weiter und wiederholt diesen Vorgang (ab Array Index 1). Dies geschieht so lange, bis alle Elemente betrachtet wurde. Ein Beispiel finden Sie in folgender Abbildung.



Bitte beachten Sie, dass bei aufeinanderfolgenden Schritten innerhalb der Berechnung von einem `Sliding Window` $w - 1$ Elemente überlappen.

Vervollständigen Sie die Methode `slidingWindowSum`, indem Sie auf dem `input` Array ein *Sliding Window* ausführen, welches jeweils die *Summe* der im Fenster liegenden Elemente berechnet. Legen Sie für jede vollständige Ausführung der Summenfunktion auf einem Bereich ein Ergebnis im zurückzugebenen Array ab. Setzen Sie voraus, dass das `input` Array groß genug ist, um mindestens ein Fenster zu enthalten und werfen Sie bei Verletzung dieser Vorbedingung eine passende Exception.

b) Ein *Tumbling Window* berechnet zunächst auf den ersten w Elementen eines Arrays (ab Array Index 0) die Funktion f. Anschließend wandert das *Window* w Elemente weiter und wiederholt diesen Vorgang (ab Array Index w). Dies geschieht so lange, bis alle Elemente betrachtet wurde. Ein Beispiel finden Sie in folgender Abbildung.



Bitte beachten Sie, dass bei aufeinanderfolgenden Schritten innerhalb der Berechnung von einem *Tumbling Window* keine Elemente überlappen.

Vervollständigen Sie die Methode `tumblingWindowAverage`, indem Sie auf dem `input` Array ein `Tumbling Window` ausführen, welches jeweils den *Durchschnitt* der im Fenster liegenden Elemente berechnet. Legen Sie für jede vollständige Ausführung der Durchschnittsfunktion auf einem Bereich ein Ergebnis im zurückzugebenen Array ab. Setzen Sie voraus, dass das `input` Array genau eine ganzzahlige Anzahl von Fenstern der Größe w enthält und werfen Sie bei Verletzung dieser Vorbedingung eine passende Exception.

*** Aufgabe 1.4 ist für Lehramtsstudierende optional.**