Übungen zu Kapitel 17

17.1 Zeitkomplexität der linearen Suche:

Berechnen Sie die exakte Zeitkomplexität der linearen Suche im Erfolgsfall. Es gibt ***n*** Datenelemente; nach jedem der ***n*** Werte wird im statistischen Mittel ***gleich oft*** gesucht. Dafür sind bei ***n*** Suchvorgängen 1,2,3,...,n-1 bzw. n Vergleichsoperationen nötig. Für alle ***n*** Suchvorgänge also insgesamt 1+2+3+...+(n-1)+n Vergleichsoperationen (arithmetische Reihe). Berechnen Sie ***S***. Pro Suchvorgang sind im Mittel ***S/n*** Vergleichsvorgänge erforderlich. Berechnen Sie ***S/n*** und bilden Sie den asymptodischen Grenzwert für große Werte $n \to \infty$.

17.2 Experimente mit Suchalgorithmen:

Ergänzen Sie die Klasse `Algorithmen` um die Methode `binSearch()`. Verwenden Sie folgende ausführbare Klasse mit Stoppuhr-Funktion, um die Effektivität der ***linearen*** und ***binären Suche*** zu testen. Es werden Vielfache von 3 eingetragen, um ein sortiertes Array zu erhalten:

```
class SuchTests {
   public static void main( String[] args ) {
      int n = IO.promptAndReadInt( "Anzahl: " );
      Algorithmen serach = new Algorithmen( n );
      for( int i=0; i<n; i++) { sort.insert( 3*i ); }
      int m = IO.promptAndReadInt( "Gesucht: " );
      long start, end;   int pos;
      start = System.currentTimeMillis();
      pos = search.binSearch( m );
      end = System.currentTimeMillis();
      IO.writeln( "Binäre Suche [ms] = " + (end-start) );
      start = System.currentTimeMillis();
      pos = search.linSearch( m );
      end = System.currentTimeMillis();
      IO.writeln("Lineare Suche [ms] = " + (end-start) );
      IO.writeln("Position: " + pos);
   }
}
```

Experimentieren Sie mit deutlich unterschiedlichen Datenmengen (***n*** = 100, 500, 1000, 10000, 1000000 etc.). Notieren Sie die Ausführungszeiten, und verifizieren Sie die ***lineare*** bzw. ***logarithmische*** Zeitkomplexität, indem Sie die Zeit gegen die Datenmenge ***n*** auftragen.

17.3 Experimente mit Sortieralgorithmen:

Ergänzen Sie die Klasse `Algorithmen` um die Methoden `bubbleSort()`, `quickSort()` und `distributionSort()`. Verwenden Sie folgende ausführbare Klasse mit Stoppuhr-Funktion, um verschiedene Sortierverfahren zu testen. Die Klasse `java.util.Random` mit festem Seed-Wert liefert stets die gleiche Folge von Zufallszahlen. Somit müssen alle Sortieralgorithmen unter gleichen Bedingungen antreten.

```
import java.util.Random;
class SortierTests {
   public static void main( String[] args ) {
      int n = IO.promptAndReadInt( "Anzahl: " );
      Algorithmen sort = new Algorithmen( n );
      Random generator = new Random( 20 );
      for( int i=0; i<n; i++) {
```

```
            int z = generator.nextInt( 1000 );
            Algorithmen.insert( z );
        }
        long start, end;
        start = System.currentTimeMillis();
        sort.bubbleSort( sort.s );   // gewünschte Methode
        // sort.quickSort( sort.s );
        // sort.distributionSort( sort.s , 1000 );
        end = System.currentTimeMillis();
        IO.writeln( "Sortieren [ms] = " + (end-start) );
    }
}
```

Experimentieren Sie mit deutlich unterschiedlichen Datenmengen ***n***. Notieren Sie die Ausführungszeiten, und verifizieren Sie die verschiedenen Zeitkomplexitäten, indem Sie die Zeit gegen die Datenmenge ***n*** auftragen.

Erweitern Sie die Klassen `Algorithmen` und `SortierTests`: Der unsortierte und sortierte Datenbestand soll ausgegeben werden können, um sich (für überschaubare Datenmengen ***n***) von der korrekten Sortierung zu überzeugen.

17.4 Weitere Sortieralgorithmen:

Auf der Webseite des Buches befindet sich eine Foliensammlung zu Algorithmen und Datenstrukturen. Diese enthält auch eine Darstellung der Sortieralgorithmen SelectionSort und MergeSort.