SoSe 2019

Prof. Dr. Margarita Esponda

# Objektorientierte Programmierung

## 3. Übungsblatt

**Ziel**: Auseinandersetzung mit Komplexitätsanalyse und Sortieralgorithmen.

### 1. Aufgabe (4 Punkte)

a) Wie groß muss eine sortierte Liste sein, um bei einer rekursiven linearen Suche innerhalb der Liste eventuell einen Stapelüberlauf der Python-Virtuelle-Maschine zu produzieren?

b) Wie groß kann eine sortierte Liste maximal sein, bevor bei einer rekursiven Binärsuche ein Stapelüberlauf verursacht wird?

c) Erläutern Sie beide Antworten.

### 2. Aufgabe (3 Punkte)

Definieren Sie eine **isSorted** Funktion, die bei Eingabe eines Vergleichsoperators und einer Liste überprüft, ob die Liste nach dem eingegebenen Vergleichsoperator (Vergleichsfunktion) sortiert ist.

Anwendungsbeispiele:

```
>>> isSorted (operator.lt, [2, 2, 4, 5, 8, 9])
False
>>> isSorted (operator.gt,  [2, 2, 4, 5, 8, 9])
False
>>> isSorted (operator.le,  [2, 2, 4, 5, 8, 9])
True
```

### 3. Aufgabe (5 Punkte)

a) Ersetzen Sie die **for**-Schleife des Insertsort-Algorithmus der Vorlesung mit einer **while**-Schleife.

b) Definieren Sie eine Testfunktion, die mit Hilfe von Zufallszahlen den Algorithmus ausführlich testet. Verwenden Sie dafür Ihre **isSorted** Funktion aus Aufgabe 2.

c) Wann ist es sinnvoll, den Insertsort- oder den Shellsort-Algorithmus zu verwenden? Begründen Sie Ihre Antwort.

### 4. Aufgabe (4 Punkte)

Schreiben Sie eine möglichst effiziente Python-Funktion, die bei Eingabe einer Zahlenmenge beliebigen Datentyps, die zwei Zahlen findet, deren Werte am nächsten sind. Analysieren Sie die Komplexität Ihres Algorithmus.

Anwendungsbeispiele:

```
>>> min_diff ([3, 10, 3, 9, 5, 1, 2, 7, 6, 8])
(3, 3)
```

**5. Aufgabe** (6 Punkte)

Schreiben Sie eine möglichst effiziente Python Funktion, die bei Eingabe eines Arrays aus ganzen Zahlen alle Zahlenpaaren findet, die zusammen addiert und geteilt durch zwei gleich dem mittleren Wert (Durchschnitt) aller Zahlen ist. Die Liste aller Zahlenpaaren soll als Ergebnis der Funktion zurückgeliefert werden. Analysieren Sie die Komplexität Ihres Algorithmus.

Verwenden Sie für die Berechnungen nur ganzzahlige Division.

Anwendungsbeispiele:

```
>>> same_average ([6, 25, 29, 10, 6, 5, 8, 0, 20, 19])
[(6, 19), (25, 0), (6, 19), (5, 20), (5, 19)]
```

**6. Aufgabe** (8 Punkte)

a) Erläutern Sie mit einem konkreten Zahlenbeispiel, warum der Quicksort-Algorithmus aus der Vorlesung nicht stabil ist.

b) Können Sie den Algorithmus aus der Vorlesung stabil machen? Erläutern Sie Ihre Antwort, indem Sie die entsprechende Variante in Python programmieren. Oder erläutern Sie warum es nicht möglich ist.

c) Wie könnten Sie, falls eine Lösung existiert, mit entsprechenden Testdaten automatisch kontrollieren, dass dieser Algorithmus funktioniert?

d) Wie oft kann während der Ausführung des Quicksort-Algorithmus das größte Element maximal bewegt werden? Begründen Sie Ihre Antwort.

**7. Aufgabe** (8 Bonuspunkte)

In dieser Aufgabe möchten wir einige Funktionen definieren, die später für ein Spiel verwendet werden können. Das Spiel soll analog zum *Minesweeper*-Spiel funktionieren. Wir haben aber statt Bomben Löcher. Das Spielfeld soll mit Hilfe einer Matrix modelliert werden.

a) Schreiben Sie eine Funktion, die nach Eingabe der Argumente n, m eine nxm Matrix mit Punkte initialisiert.

b) Definieren Sie eine showGameField Funktion, die das Feld ausgibt.

c) Schreiben Sie eine Funktion newGame, die nach Eingabe von p und einer nxm Matrix das Spielfeld (Matrix) initialisiert, in dem mit Wahrscheinlichkeit p in jeder beliebigen (x, y)-Position ein Loch vorkommen kann. Ein '.'-Zeichen bedeutet kein Loch und mit einem 'O'-Zeichen werden die Positionen mit Löchern markiert.

d) Schreiben Sie eine Funktion genSolution, mit der in jeder Position des Feldes, an der kein Loch existiert, die Anzahl der benachbarten Löcher ausgegeben wird. Wenn keine Nachbarn vorhanden sind, soll das Punkt-Zeichen hinterlassen werden.

Das SpielFeld (Matrix) soll am Ende beispielsweise folgendermaßen aussehen:

```
. . 1 1 1 2 2 1 . .
. 1 O 1 2 O O 1 . .
. 1 1 1 2 O 3 1 . .
. . 1 1 1 . . . . .
```

**Wichtige Hinweise:**

1) Verwenden Sie geeignete Namen für Ihre Variablen und Funktionen, die den semantischen Inhalt der Variablen oder die Funktionalität der Funktionen darstellen.
2) Verwenden Sie vorgegebene Funktionsnamen, falls diese angegeben werden.
3) Kommentieren Sie Ihre Programme.
4) Verwenden Sie geeignete Hilfsvariablen und Hilfsfunktionen in Ihren Programmen.
5) Löschen Sie alle Programmzeilen und Variablen, die nicht verwendet werden.
6) Schreiben Sie getrennte Test-Funktionen für alle Aufgaben.
7) Die Lösungen sollen in Papierform und elektronisch (KVV-Upload) abgegeben werden.