SoSe 2019

Prof. Dr. Margarita Esponda

# Objektorientierte Programmierung

## 4. Übungsblatt

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

### 1. Aufgabe (8 Punkte)

a) (2 P.) Programmieren Sie eine rekursive Variante des Mergesort-Algorithmus aus der Vorlesung, indem Teilarrays, die kleiner gleich 9 sind, nicht mehr mit Mergesort, sondern mit Insertsort sortiert werden.

b) (6 P.) Programmieren Sie eine nicht rekursive Version des Mergesort-Algorithmus, indem ein Hilfsarray verwendet wird, das genau so groß wie die zu sortierenden Zahlenmengen ist und die als Zwischenlagerung von Teilmengen des Arrays innerhalb der **merge**-Funktion verwendet wird.

### 2. Aufgabe (10 Punkte)

a) Nehmen Sie an, wir haben eine Reihe von Aufgaben, die mit Prioritäten zwischen 1 und 100 abgearbeitet werden müssen.

$$[(a_1, \text{priority}_1), (a_2, \text{priority}_2),...,(a_n, \text{priority}_n)]$$

Simulieren Sie unter Verwendung einer binären *Max-Heap*-Struktur (siehe Vorlesungsfolien) eine Warteschlange, womit die Aufgaben nach Priorität abgearbeitet werden (*Priority Queue*).

Eine Prioritätswarteschlange speichert Objekte nach einer Prioritätseigenschaft und entfernt immer als erstes das Objekt mit der höchsten Priorität.

Schreiben Sie dafür folgende vier Hilfsfunktionen:

```
def buildPriorityQueue( taskList )
    """ Die taskList wird in eine Heap-Struktur transformiert """

def insert( priorityQueue, newTask )
    """ Ein neuer Task (Aufgabe) wird in dem priorityQueue eingefügt """

def isEmpty( priorityQueue )
    """ gibt einen Wahrheitswert als Rückgabewert, je nachdem, ob die
        Prioritätswarteschlange leer ist oder nicht """

def removeTask( priorityQueue )
    """ Die Aufgabe mit der höchsten Priorität wird aus der priorityQueue entfernt
        und als Ergebnis der Funktion zurückgegeben """
```

b) Analysieren Sie die Komplexität der Funktionen.

c) Schreiben Sie eine Test-Funktion, die zufällig Aufgaben einfügt oder entfernt. Die Prioritäten der Aufgaben sollen auch zufällig ausgewählt werden. Wenn die Prioritätswarteschlange leer ist, soll die **removeTask**-Funktion den Wert **None** zurückgeben.

**3. Aufgabe** (6 Punkte)

a) Definieren Sie eine Variante des *Countingsort*-Algorithmus aus der Vorlesung, bei dem Sie die Elemente **in-Place** sortieren. Ihre *Countingsort* Variante muss nicht stabil sein. Sie dürfen als einzigen zusätzlichen Speicher das Hilfsrarray C verwenden (siehe Vorlesungsfolien). Das Hilfsarray **B** darf nicht mehr verwendet werden.

b) Analysieren Sie die Komplexität ihres Algorithmus.

**4. Aufgabe** (6 Punkte)

Beantworten Sie die Frage, die Google an Obama im Jahr 2008 stellte.
"*What ist the most efficient way to sort a million 32-bit integers?*". Begründen Sie Ihre Antwort.
Implementieren Sie Ihre Lösung und testen Sie diese mit einer Million zufällig erzeugter 32-Bit Zahlen. Erläutern Sie Vor- und Nachteile Ihrer Lösung.

**5. Aufgabe** (8 Bonuspunkte)

Fortsetzung der **7. Aufgabe** des **3.Übungsblattes**.
In dieser Aufgabe möchten wir die restliche Funktionen für das *Löcher*-Spiel definieren.

a) Schreiben Sie eine Funktion **start_game**, die bei Eingabe einer Wahrscheinlichkeit **p** und einer Spielfeldgröße (**n**, **m**) eine Matrix mit entsprechender Lösung intern produziert und diese zuerst mit gedeckten Feldern ausgibt.

```
X X X X X X X X X X
X X X X X X X X X X
X X X X X X X X X X
X X X X X X X X X X
```

b) Schreiben Sie zum Schluss eine Funktion **play**, die das Spiel mit folgenden Regeln startet:

1) Der Spieler wird jedes mal aufgefordert eine **x**, **y** Position einzugeben.

2) Wenn in der eingegebenen Position ein Loch existiert, fällt der Spieler in das Loch und das Spiel ist beendet.

3) Wenn an der gewählten **x**, **y** Position kein Loch ist, aber mindestens ein Loch in der Nachbarschaft existiert, wird nur diese Position mit entsprechender Zahl aufgedeckt.

4) Wenn die gewählte Position keinen Nachbarn mit einem Loch hat (Position mit '.' - Zeichen), wird diese aufgedeckt und alle seine Nachbarn (so lange noch welche vorhanden sind) werden auch automatisch aufgedeckt. Die neu aufgedeckten Nachbarn, die selber keinen Nachbarn mit Loch haben (Nachbarn ohne Zahlen), verursachen wiederum die Aufdeckung ihrer Nachbarn. Diese Kettenreaktion wird so lange kein Loch, kein Nachbar mit Zahlen und kein Rand vorkommt, wiederholt.

5) Wenn alle Positionen aufgedeckt sind und der Spieler noch am Leben ist, hat er gewonnen!

**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.