1. (3 Punkte) Sortiere die Zahlenfolge mit SelectionSort. Schreibe für die ersten drei Durchgänge je eine Zeile.

   `88 14 3 16 20 42 9`

Lösung:

```
3 14 88 16 20 42 9
3 9 88 16 20 42 14
3 9 14 16 20 42 88
```

2. (4 Punkte) Notiere den Code, der für den SelectionSort-Algorithmus fehlt. Gib auch die Stufe der Einrückung an.

```
def selection_sort(a):                  #E0
    for i in range(len(a)-1):           #E1
        pos = i                         #E2
                                      ???(1)
        for j in range(i+1,len(a)):     #E2
                                      ???(2)
                pos = j                 #E4
                min = a[j]              #E4
        a[pos],a[i] = a[i],a[pos]       #E2
```

Lösung:

```
(1)   min = a[i]              #E2
(2)    if a[j] < min:         #E3
```

3. (3 Punkte) Sortiere die Zahlenfolge mit BubbleSort. Schreibe für die ersten drei Durchgänge je eine Zeile.

   `13 7 1 25 42 18 9`

Lösung:

```
7 1 13 25 18 9 42
1 7 13 18 9 25 42
1 7 13 9 18 25 42
```

4. (4 Punkte) Notiere den Code, der für den BubbleSort-Algorithmus fehlt. Gib auch die Stufe der Einrückung an.

```
def bubble_sort(a):                          #E0
    getauscht = True                         #E1
    while getauscht:                         #E1
                                             ???(1)
        for i in range(len(a)-1):            #E2
            if a[i] > a[i+1]:                #E3
                                             ???(2)
                getauscht = True             #E4
```

Lösung:

```
(1) getauscht = False          #E2
(2) a[i],a[i+1]=a[i+1],a[i]    #E4
```

5. (5 Punkte) Die Liste `a = [8,3,1,22,6]` wird mit dem rekursiven mergeSort-Algorithmus aus dem Unterricht sortiert.
   a. Wieviel mal wird `merge` aufgerufen?
   b. Wieviel mal wird `mergeSort` aufgerufen? (der erste Aufruf zählt mit).

Lösung: a. 4 b. 9

Bei jedem Aufruf von `merge` wird eine Liste zurückgegeben. Notiere die Listen in der Reihenfolge, in der sie zurückgegeben werden.

Lösung:

```
3 8
6 22
1 6 22
1 3 6 8 22
```

6. (2 Punkte) Der nächste Quicksort-Durchgang bearbeitet die Liste von Index 5 bis Index 8.

   `3 5 8 9 11 42 19 16 22`

   Schreibe die Protokollzeilen vor und nach dem Durchgang.

Lösung:

```
3 5 8 9 11 42 19 16 22 unten=5 oben=8 pivot=19
3 5 8 9 11 16 19 42 22 unten=7 oben=8 pivot=42
```

7. (2 Punkte) Quicksort erhält die Liste zur Sortierung. Schreibe die Protokollzeilen vor und nach dem ersten Durchgang.

   `0 1 2 3 7 6`

Lösung:

```
0 1 2 3 7 6 unten=0 oben=5 pivot=2
0 1 2 3 7 6 unten=0 oben=1 pivot=0
```

8. (3 Punkte) Mache aus der Liste einen Heap nach dem Verfahren aus dem Unterricht:

   `33 12 5 9 39 42 18 7 69 29`

Lösung: 5 7 18 9 29 42 33 12 69 39

9. (2 Punkte) Eine Liste wird mit HeapSort sortiert. Die Zeile zeigt die in einen Heap umgewandelte Liste. Füge für die beiden folgenden Reorganisationen je eine Zeile hinzu.

   `7 12 11 13 47 42 82 36`

Lösung:

```
11 12 36 13 47 42 82 7
12 13 36 82 47 42 11 7
```

10. (2 Punkte) Welche Komplexität hat die Laufzeit von MergeSort im best, worst und average-case? In welcher Komplexitätsklasse ist der zusätzliche Platzbedarf?

Lösung:

best: $O(n \cdot \log n)$, average: $O(n \cdot \log n)$, worst: $O(n \cdot \log n)$, Platz: $O(n)$

11. (2 Punkte) Gegeben sei die Tupelliste `a = [(3,5,8),(7,3,4),(1,2,7),(10,1,11)]`. Erzeuge aus a zwei Listen b1 und b2. b1 ist aufsteigend sortiert nach der Summe aller Komponenten, b2 ist absteigend sortiert nach der letzten Komponente. Nutze in beiden Fällen den lambda-Operator.

Lösung:

```
b1 = sorted(a,key = lambda x: x[0]+x[1]+x[2])
b2 = sorted(a,key = lambda x: x[2], reverse=True)
```

12. (2 Punkte) Gegeben sei eine Liste `a` mit Strings:

    Sortiere die Liste aufsteigend nach der Länge des Strings. Bei gleicher Länge sortiere alphabetisch.

Lösung:

```
a.sort(key = lambda x: (len(x),x))
```

13. (3 Punkte) Die Klasse Bundesland hat die Attribute name und hauptstadt. Implementiere die Klasse Bundesland. Folgender Dialog soll möglich sein:

```
z = Bundesland("Hessen","Wiesbaden")
print(z)

Die Hauptstadt von Hessen ist Wiesbaden.
```

Lösung:

```
class Bundesland:
    def __init__(self,name,hauptstadt):
        self.name = name
        self.hauptstadt = hauptstadt

    def __str__(self):
        return "Die Hauptstadt von {} ist {}.".format(self.name,self.hauptstadt)
```

14. (5 Punkte) a. Implementiere eine Klasse `Komponist` mit den Attributen `name` und `anzahlSinfonien`. Beim Erzeugen eines Komponist-Objekts werden jeweils Werte für beide Attribute mitgegeben.
b. Erzeuge zwei Komponist-Objekte `k1` und `k2`: Beethoven hat 9 Sinfonien geschrieben, Brahms 4.
c. Erstelle eine Liste `komponisten` die `k1` und `k2` enthält.
d. Erstelle ein dictionary `kmap` mit dem Namen des Komponisten als key und dem Komponist-Objekt als value. `kmap` soll zu Beginn Einträge für Beethoven und Brahms enthalten.
Füge einen weiteren Eintrag in das dictionary `kmap` ein: Schubert hat 9 Sinfonien geschrieben.
e. `kmap` sei um weitere Einträge erweitert worden. Gehe durch das dictionary und gib die Namen aller Komponisten aus, die 9 Sinfonien geschrieben haben.

Lösung:

```
class Komponist:
    def __init__(self,name,anzahlSinfonien):
        self.name = name
        self.anzahlSinfonien = anzahlSinfonien

k1 = Komponist("Beethoven",9)
k2 = Komponist("Brahms",4)

komponisten = [k1,k2]
kmap = {"Beethoven":k1, "Brahms":k2}
kmap["Schubert"] = Komponist("Schubert",9)
for k in kmap:
    if kmap[k].anzahlSinfonien == 9:
        print(k)
```

15. (5 Punkte) a. Gib für die beiden Klassen ein passendes Klassendiagramm an.

b. Was erscheint auf der Konsole?

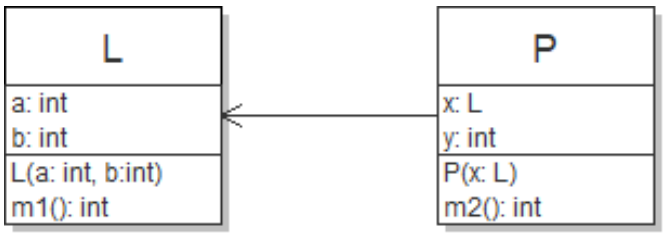
```
class L:
    def __init__(self,a,b):
        self.a = a
        self.b = b

    def m1(self):
        return self.a + self.b

class P:
    def __init__(self,x):
        self.x = x
        self.y = 3
    def m2(self):
        return self.x.m1() + self.y

k = L(4,5)
p = P(k)
print(p.m2())
```

Lösung:



b. 12

Aufgabe:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Summe:
Punkte:	3	4	3	4	5	2	2	3	2	2	2	2	3	5	5	47