Informatik 5. Klasse

Examensvorbereitung

Teil 2

Lösungen

OOP: Aufgabe 1

(a) Die Grundidee von OOP ist es, Programme so zu gestalten, dass sie die zu verarbeitende Realität möglichst gut abbilden.

(b) Konzepte:

- *Klasse:* ein Bauplan, aus dem zur Laufzeit Objekte erzeugt werden.
- *Vererbung:* Aus bestehenden Klassen (Eltern- oder Superklasse) werden neue Klassen (Kind- oder Subklasse) abgeleitet. Damit kann die Kindklasse auf den Code der Elternklasse zugreifen. Die Kindklasse kann bestehende Eigenschaften und Methoden durch neue ersetzen oder zusätzliche Eigenschaften oder Methoden hinzufügen.
- *Polymorphie:* Die Klassenzugehörigkeit bestimmt, welche Variable und und welche Methode aufgerufen wird. Daher können dieselben Variablen und Funktionsnamen in verschiedenen Klassen verwendet werden.

(c) Eine *Klasse* ist ein Bauplan für Objekte.

(d) Eine *Instanz* ist ein Ding (eben ein *Objekt*), das Eigenschaften (Variablen) und Methoden (Funktionen) besitzt und zur Laufzeit aus einer Klassendefinition erzeugt wird.

(e) *Objektvariablen* sind Variablen, die den Zustand (Eigenschaften) eines Objekts darstellen. Jedes Objekt kann dafür eigene Werte haben. *Klassenvariablen*, sind Variablen, die den Zustand der ganzen Klasse darstellen. Sie haben jeweils nur einen Wert, der für die ganze Klasse gilt

(f) *Objektmethoden* sind Funktionen, die von den Objekten einer Klasse ausgeführt werden können. Sie sind im Allgemeinen vom inneren Zustand der Objekte abhängig.

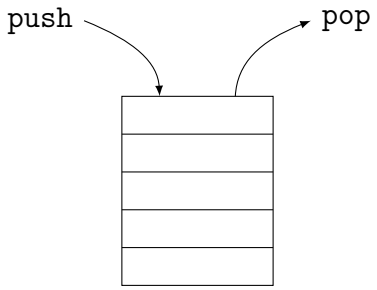
Klassenmethoden sind Funktionen die von der Klasse selbst ausgeführt werden können und sind vom Zustand der Objekte unabhängig.

OOP: Aufgabe 2

```
1  class Example:
2
3      a = 5  # Klassenvariable
4
5      def __init__(self, x): # Konstruktor
6          self.x = x         # x -> Objektvariable self.x
7
8      def __str__(self):  # überschreibt str()-Funktion
9          return('{0}'.format(self.x)) # gibt self.x als String zurück
10
11     def bar(self, y):  # Objektmethode mit Argument y
12         self.x += y    # Objektvariable wird um y vergrössert
13
14     def foo(b):  # Klassenmethode mit einem Parameter b
15         Example.a = b   # Weist der Klassenvariable den Wert b zu
16
17 e = Example(1) # Aufruf von __init__(...) => e.x=1
18 e.bar(3)       # Verändert die Objektvariable $e.x ist neu 1+3=4
19 print(e)       # => '4' (str-Darstellung von e)
20 Example.foo(7) # Aufruf Klassenmethode mit b=7 => Example.a ist neu 7
21 print(Example.a)  # 7 (siehe Zeile 20)
```

OOP: Aufgabe 3

Die Klasse implementiert eine Datenstruktur, die *Stack* (deutsch *Kellerspeicher*) genannt wird. Es handelt sich um eine Sammlung von Objekten, bei der mit der Operation `push(item)` ein Element `item` auf den Stapel gelegt wird und mit der Operation `pop()` das zuletzt hinzugefügte Element wieder vom Stapel entfernt wird. Dieses Prinzip wird *Last In-First Out* oder kurz *LIFO* genannt.

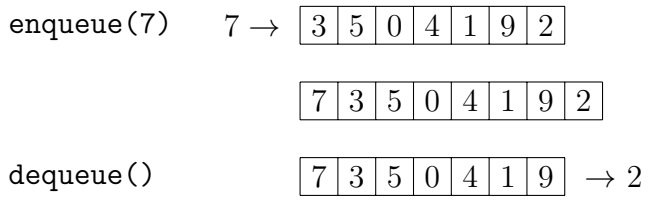


Der Stack wird hier als Liste implementiert, wobei das Hinzufügen und Entfernen der Elemente aus Effizienzgründen am rechten Listende erfolgt. Klassenintern wird `push(item)` mit der `append(...)`-Methode und `pop()` mit der gleichnamingen Python-Methode für Listen realisiert.

```
1  class Stack: # Zeilen 1-16 definieren die Klasse "Stack"
2
3      def __init__(self):  # Konstruktur
4          self.items = []  # Objektvariable: self.items ist leere Liste
5
6      def __str__(self): # überschreibt str()-Funktion
7          return('{0} <top>'.format(self.items)) # Ausgabe der Liste
8                                                 # top zeigt "oben" an
9      def __len__(self):  # überschreibt die Python-Funktion len() und gibt
10         return len(self.items) # die Anzahl Elemente von items zurück
11
12     def push(self, x): # definiert die push(...)-Methode des Stacks
13         self.items.append(x) # legt x auf den Stack ab (-> ans Listenende)
14
15     def pop(self):  # definiert die push(...)-Methode des Stacks
16         return self.items.pop() # entfernt oberstes Element und gibt es zurück
17
18 s = Stack()  # Ruft Konstruktor auf (keine Argumente) => s.items = []
19 s.push(5)    # Legt 5 auf dem Stack ab => s.items = [5]
20 s.push(3)    # Legt 3 auf dem Stack ab => s.items = [5, 3]
21 s.push(4)    # Legt 4 auf dem Stack ab => s.items = [5, 3, 4]
22 print(len(s)) # => 3 (da aktuell 3 Elemente auf dem Stack liegen)
23 x = s.pop()   # => x <- 4  und s.items = [5,3]
24 y = s.pop()   # => x <- 3  und s.items = [5]
25 s.push(x+y)   # legt x+y = 7 auf dem Stack ab  => s.items = [5, 7]
26 print(s)      # => [5, 7] <top> [top zeigt Seite an, wo Elmt. ein-/ausgehen]
```

OOP: Aufgabe 4

Die folgende Klasse definiert eine „Queue“ (Warteschlange). Dies ist eine Datenstruktur mit einer Einfügeoperation `enqueue(<item>)`, welche Objekte auf einer bestimmte Seite einfügt und einer Entnahmeoperation `dequeue(<item>)`, welche Objekte auf der anderen Seite entntimmt (FIFO-Prinzip: First In-First-Out).



Die Implementierung mittels einer Python-List ist insofern „naiv“, da die Einfügeoperation durch die `insert()`-Methode realisiert wird, welche eine schlechte Effizienz ($O(n)$) hat. Die Entnahmeoperation mit der `pop()`-Methode für Python-Listen ist hingegen effizient $O(1)$. Man könnte die Seiten für das Einfügen und das Entnehmen der Elemente tauschen aber das Effizienzproblem würde sich dann bei der Entnahme der Elemente stellen.

```
1  class Queue: # Zeilen 1-16: Definition der Klasse Queue (FIFO-Datenstruktur)
2
3      def __init__(self):  # Konstruktor
4          self.items = []  # Leere liste [] -> self.items (Objektvariable)
5
6      def __str__(self):  # Spezialmethode überschreibt str()-Funktion
7          return 'in {0} out'.format(self.items) # gibt Queue als Liste aus
8
9      def __len__(self):  # Spezialmethode überschreibt len()-Funktion
10         return len(self.items) # gibt die Anzahl Elemente in der Queue zurück
11
12     def enqueue(self, item): # Methode definiert Einfügeoperation
13         self.items.insert(0, item)  # item wird links eingeführt
14
15     def dequeue(self): # Methode definiert Entnahmeperation
16         return self.items.pop()  # Elemt. wird rechts entfernt und
               zurückgegeben
17
18 q = Queue()      # erzeugt ein neues Queue-Objekt => q.items = []
19 q.enqueue(4)    # fügt 4 in Queue ein (-> Liste) => q.items = [4]
20 q.enqueue(7)    # fügt 7 in Queue ein (-> Liste) => q.items = [7, 4]
21 x = q.dequeue() # entnimmt 4->x aus Queue (Liste ->) => q.items = [7]
22 print(x)        # => 4
23 q.enqueue(1)    # fügt 1 in Queue ein (-> Liste) => q.items = [1, 7]
24 q.enqueue(2)    # fügt 2 in Queue ein (-> Liste) => q.items = [2, 1, 7]
25 print(q)        # in [2, 1, 7] out
26 print(len(q))   # => 3
```

OOP: Aufgabe 5

Hinweis: Im Quellcode der gedruckten Version dieser Aufgabe müsste es auf Zeile 20 korrekt `R = P.add(Q)` statt `R = add(P, Q)` heissen. Der Fehler wurde hier korrigiert.

```
1  class Point:  # Definition der Klasse in den Zeilen 1-16
2
3      def __init__(self, x, y): # Konstruktur mit 1+2 Argumenten
4          self.x = x  # x -> x-Eigenschaft des Objekts (self)
5          self.y = y  # x -> y-Eigenschaft des Objekts (self)
6
7      def __str__(self): # überschreiben der str()-Funktion
8          return('({0}, {1})'.format(self.x, self.y)) # => (self.x, self.y)
9
10     def add(self, other): # Objektmethode, die eine Addition suggeriert
11         x = self.x + other.x # Additon der x-Eigenschaften von self und other
12         y = self.y + other.y # Additon der y-Eigenschaften von self und other
13         return Point(x, y) # x, y => neues Point-Objekt; gibt es zurück
14
15     def swap(self): # Objektmethode; vertauscht die Werte der ...
16         self.x, self.y = self.y, self.x # x- und y-Eigeschaften des Objekts
17
18 P = Point(3, 4) # Ruft den Konstruktor von Point => P.x=3 und P.y=4
19 Q = Point(1, 2) # Ruft den Konstruktor von Point => Q.x=1 und Q.y=2
20 R = P.add(Q)    # Erzeugt einen neuen Punkt mit R.x=4 und R.y=6
21 Q.swap()       # Ändert Zustand von Q: Q.x=2 und Q.x=1
22
23 print(P)        # => (3, 4)
24 print(R)        # => (4, 6)
25 print(Q)        # => (2, 1)
```

OOP: Aufgabe 6

Hinweis: In der Aufgabenstellung der gedruckten Version hätte noch stehen sollen, dass der Code der Methode `umfang()` in Zeile 15 zu vervollständigen ist.

```
1  class Rechteck: # Definition der Klasse "Rechteck" (Zeilen 1-15)
2
3      def __init__(self, a, b): # Konstruktor mit Parametern a, b
4          self.a = a   # Objekteigenschaft self.a ~ Länge
5          self.b = b   # Objekteigenschaft self.b ~ Breite
6
7      def __str__(self): # Methode überschreibt str()-Funktion
8          txt = 'Länge: {0}, Breite {1}'.format(self.a, self.b)
9          return txt # gibt den Text zurück - z.B. bei print(<obj>)
10
11     def inhalt(self): # Objektmethode: berechnet den Flächeninhalt
12         return self.a * self.b  # und gibt ihn als Wert zurück
13
14     def umfang(self): # Objektmethode: berechnet den Umfang des Objekts
15         return 2*(self.a + self.b) # und gibt ihn als Wert zurück
16
17 class Quadrat(Rechteck):  # Klasse "Quadrat" erbt von Klasse Rechteck
18
19     def __init__(self, a): # Konstruktor von Quadrat ruft den Konstruktor
20         super().__init__(a, a)f # der Superklasse mit Länge = Breite = a auf
21
22 r = Rechteck(3,4) # Ruft Konstruktor von Rechteck auf => r.a=3, r.b=4
23 print(r.inhalt()) # Ruft Methode inhalt() für Objekt r auf => Ausgabe: 12
24 q = Quadrat(5)    # Ruft Konstruktor von Quadrat->Rechteck auf => q.a=5, q.b=5
25 print(q.inhalt()) # Ruft inhalt() für Quadrat->Rechteck auf => Ausgabe: 25
```

Heiratsproblem: Aufgabe 1

(a)
- Die Anzahl der Kandidaten muss im Voraus bekannt sein.
- Die Kandidaten erscheinen in zufälliger Reihenfolge.
- Allen Kandidaten lässt sich eindeutig ein Rang zuordnen.

(b) Begutachte die ersten r Kandidaten, lehne sie ab und wähle danach den ersten, der besser ist, als die ersten r. Falls es keinen besseren gibt, wähle den letzten.

(c) Die in (b) beschriebene Stategie führt dazu, dass mit maximaler Wahrscheinlichkeit der Kandidat mit dem höchsten Rang ausgewählt wird.

Heiratsproblem: Aufgabe 2

(a) [8, 4, 10, 2, 5, 1, 3, 9, 7, 6], $r = 4 \Rightarrow$ wähle 6

(b) [2, 1, 6, 7, 4, 5, 10, 9, 3, 8], $r = 5 \Rightarrow$ wähle 10

(c) [2, 1, 4, 6, 5, 10, 9, 7, 8, 3], $r = 2 \Rightarrow$ wähle 4

(d) [7, 4, 1, 6, 2, 3, 10, 8, 9, 5], $r = 0 \Rightarrow$ wähle 7

Heiratsproblem: Aufgabe 3

- Situation 1: Man lernt in der Testphase nur Kandidaten mit einem schlechten Rang kennen und gerät dann gleich danach an an jemanden, der nur ein bisschen besser ist. Diese(r) muss dann gewählt werden, wenn man den Algorithmus befolgt.
- Situation 2: Man lernt bereits in der Testphase den optimalen Partner kennen, muss ihn aber wieder verlassen, wenn man den Algorithmus befolgt. In diese Fall kann man keinen besseren finden und landet beim letzten in der Liste.

Heiratsproblem: Aufgabe 4

(a)

Permutation	$r = 0$	$r = 1$	$r = 2$
1, 2, 3	1	2	3
1, 3, 2	1	3	2
2, 1, 3	2	3	3
2, 3, 1	2	3	1
3, 1, 2	3	2	2
3, 2, 1	3	1	1
Summe der Dreien	2	3	2

(c) für $r = 1$

(d) für $r = n/e \approx 0.37 \cdot n$

Heiratsproblem: Aufgabe 5

Bei 200 Kandidaten müsste man etwa $200 \cdot \frac{1}{e} = 200 \cdot 0.367879 \approx 74$ ablehnen.

Heiratsproblem: Aufgabe 6

7 verschiedene Objekte kann man auf $7! = 5040$ verschiedene Arten in eine Reihenfolge bringen.

Heiratsproblem: Aufgabe 7

```
L = [3, 4, 2, 1, 5, 7, 8]
m = max(L[:5]) # => max([3, 4, 2, 1, 5]) => 5
print(m) # => 5
```

Sortieren: Aufgabe 1

```
1 def sort(L):
2     n = len(L)
3     i = 0
4     while i < n:
5         if i == 0 or L[i-1] < L[i]:
6             i = i + 1
7         else:
8             L[i-1], L[i] = L[i], L[i-1]
9             i = i - 1
```

(a) Gnomesort

Der „Gartenzwerg“ geht jeweils eine Position nach rechts, wenn er sich an der Position 0 befindet oder die Elemente an der aktuellen und der links davor liegenden Position in der richtigen Reihenfolge sind (Zeilen 5–6). Andernfalls vertauscht der Gartenzwerg die Elemente an der aktuellen und der links davon befindlichen Position und geht eine Position nach links zurück (Zeilen 7–9). Der Zwerg führt diese Schritte so lange aus bis er rechts vom letzten Element steht (Zeile 4).

(b) Die unterstrichene Zahl kennzeichnet die Position des Gartenzwergs.

				# Vergleiche
$\underline{3}$	1	4	2	0
3	$\underline{1}$	4	2	1
$\underline{1}$	3	4	2	0
1	$\underline{3}$	4	2	1
1	3	$\underline{4}$	2	1
1	3	4	$\underline{2}$	1
1	3	$\underline{2}$	4	1
1	$\underline{2}$	3	4	1
1	2	$\underline{3}$	4	1
1	2	3	$\underline{4}$	1
1	2	3	4	

(c) *Best Case:* Liste ist aufsteigend sortiert $\Rightarrow O(n)$

An $n-1$ Positionen wird ein Vergleich durchgeführt. Da alle Elemente in der richten Reihenfolge sind, gibt es keine Vertauschungen und damit auch keine Schritte zurück. Somit kommen auch keine zusätzlichen Vergleiche hinzu.

Worst Case: Liste ist absteigend sortiert $\Rightarrow O(n^2)$

Ist der Zwerg an der Position 2, ist 1 Vergleich nötig.

Ist der Zwerg an der Position 3, sind 2 Vergleiche nötig.

...

Ist der Zwerg an der Position n, sind $n-1$ Vergleiche nötig.

$1 + 2 + \cdots + (n-2) + (n-1) = n(n-1)/2 = \frac{1}{2}n^2 + \ldots$

Wir müssten noch die Vergleiche zählen, die der Zwerg braucht, um von Position 1 zur gegebenen Position k kommt aber diese „Kosten“ verdoppeln höchstens die Anzahl der Vergleiche des letzten Schritts und können bei der Bestimmung der Laufzeitkomplexität vernachlässigt werden.

Sortieren: Aufgabe 2

```
1  def sort(L):
2      n = len(L)
3      for i in range(0, n-1):
4          k = i
5          for j in range(i+1, n):
6              if L[j] < L[k]:
7                  k = j
8          L[i], L[k] = L[k], L[i]
```

(a) Selectionsort

Es handelt sich um eine doppelte for-Schleife, wobei die erste (äussere) for-Schleife mit `i` vom ersten bis zum zweitletzten Element geht und die zweite (innere) for-Schleife vom i-ten bis zum letzten Elementen das kleinste und seine Position `k` bestimmt und dann dieses Element mit dem an der Position `i` vertauscht. Auf diese Weise gelangt das Element im i-ten Rang an die i-te Position.

(b) Das erste unterstrichene Element ist das i-te Element in dem unsortierten Teil der Liste. Das zweite unterstrichene Element ist das kleinste Element in der Teilliste vom i-ten bis zum letzten Element. Wenn das i-te Element bereits an seiner richtigen Position ist, dann ist nur es unterstrichen und wird mit sich selber vertauscht, was kein Problem ist. Die bereits fertig sortierte Teil der Liste ist fettgedruckt.

					# Vergleiche
3	1	5	2	4	4
1	3	5	2	4	3
1	**2**	5	3	4	2
1	**2**	**3**	5	4	1
1	**2**	**3**	**4**	5	

(c) *Best Case* und *Worst Case:* immer $O(n^2)$

Liste der Länge n: $(n-1)+(n-2)+\cdots+2+1 = \dfrac{n(n-1)}{2} = \frac{1}{2}n^2+\ldots \quad \Rightarrow \quad O(n^2)$

Da insgesamt nur $n-1$ Vertauschungen durchgeführt werden, können sie gegenüber der mit n quadratisch wachsenden Anzahl Vergleiche vernachlässigt werden.

Sortieren: Aufgabe 3

```
1 def sort(L):
2     n = len(L)
3     for i in range(1, n):
4         x = L[i]
5         j = i-1
6         while j >= 0 and L[j] > x:
7             L[j+1] = L[j]
8             j = j-1
9         L[j+1] = x
```

(a) Insertionsort

In der äusseren for-Schleife läuft `i` von der zweiten bis zur letzten Position der Liste. Das Element an der Position `i` wird in `x` zwischengespeichert (Zeile 4). In der inneren while-Schleife werden die Elemente `L[j]` an den Positionen `j = (i-1), (i-2), ..., 0` um eine Position nach rechts verschoben, wenn sie grösser als `x` sind. Sind sie nicht grösser als `x` oder ist der Listenanfang erreicht (`j = -1`), endet die innere Schleife und `x` wird an die „offene“ Postion `j+1` gesetzt.

(b) Der in sich korrekt sortierte Teil der Liste ist fett hervorgehoben. Das nächste, in diese Liste einzusortierende Element ist unterstrichen.

					# Vergleiche
4	2	6	5	1	1
2	**4**	6	5	1	1
2	**4**	**6**	5	1	2
2	**4**	**5**	**6**	1	4
1	**2**	**4**	**5**	**6**	

(c) *Best Case:* Liste ist aufsteigend sortiert: $O(n)$

Position 2: 1 Vergleich

Position 3: 1 Vergleich

...

Position n: 1 Vergleich

Summe: $1 + 1 + \cdots + 1 = (n-1) \in O(n)$

Worst Case: Liste ist absteigend sortiert $O(n^2)$

Position 2: 1 Vergleich

Position 3: 2 Vergleiche

...

Position n: $(n-1)$ Vergleiche

Summe: $1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n \in O(n^2)$

Sortieren: Aufgabe 4

```
1 def sort(L):
2     n = len(L)
3     for i in range(0, n-1):
4         for j in range(0, n-i-1):
5             if L[j] > L[j+1]:
6                 L[j], L[j+1] = L[j+1], L[j]
```

(a) Bubblesort

Man erkennt Bubblesort an den häufigen Vertauschungen in der inneren for-Schleife. Im Gegensatz zu Selectionsort, das auch zwei for-Schleifen hat, wächst bei Bubblesort die Teilliste der bereits sortieren Elemente von rechts nach links, während sie bei Selectionsort von links nach rechts wächst.

(b) Die bereits korrekt sortierten Elemente sind fett hervorgehoben. Die Elemente, welche gerade verglichen werden, sind unterstrichen.

					# Vergleiche
$\underline{4}$	$\underline{2}$	6	5	1	1
2	$\underline{4}$	$\underline{6}$	5	1	1
2	4	$\underline{6}$	$\underline{5}$	1	1
2	4	5	$\underline{6}$	$\underline{1}$	1
$\underline{2}$	$\underline{4}$	5	1	**6**	1
2	$\underline{4}$	$\underline{5}$	1	**6**	1
2	4	$\underline{5}$	$\underline{1}$	**6**	1
$\underline{2}$	$\underline{4}$	1	**5**	**6**	1
2	$\underline{4}$	$\underline{1}$	**5**	**6**	1
$\underline{2}$	$\underline{1}$	**4**	**5**	**6**	1
1	**2**	**4**	**5**	**6**	1

(c) *Best Case* und *Worst Case*:

0 Elemente am Listenende richtig: $(n-1)$ Vergleiche

1 Element am Listenende richtig: $(n-2)$ Vergleiche

...

$(n-2)$ Elemente am Listenende richtig: 1 Vergleich

Summe: $(n-1)+(n-2)+\cdots+2+1 = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n \in O(n^2)$

Sortieren: Aufgabe 5

```
def is_sorted(L):
    for i in range(0, len(L)-1):
        if L[i] > L[i+1]:
            return False
    return True
```

Sortieren: Aufgabe 6

```
1  def quicksort(A):
2      quicksort_helper(A, 0, len(A))
3
4  def quicksort_helper(A, a, b):
5      if a < b:
6          m = partition(A, a, b)
7          quicksort_helper(A, a, m)
8          quicksort_helper(A, m+1, b)
9
10 def partition(A, a, b):
11     pivot = A[b-1]
12     i = a-1
13     for j in range(a, b-1):
14         if A[j] <= pivot:
15             i += 1
16             A[i], A[j] = A[j], A[i]
17     A[i+1], A[b-1] = A[b-1], A[i+1]
18     return i+1
```

(a) Vor dem ersten vertikalen Balken stehen die Elemente, welche kleiner oder gleich dem Pivotelement (Unterstrichen) sind. Zwischen den vertikalen Balken stehen die Elemente die grösser als das Pivotelement sind und nach dem zweiten vertikalen Balken stehen die Elemente, welche noch nicht einsortiert wurden.

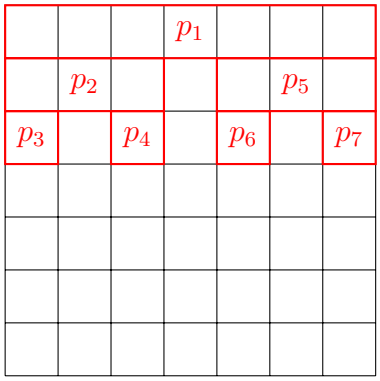
| | 8 9 2 1 5 $\underline{4}$	$8 > 4 \Rightarrow$	verschiebe Balken 2
| 8 | 9 2 1 5 $\underline{4}$	$9 > 4 \Rightarrow$	verschiebe Balken 2
| 8 9 | 2 1 5 $\underline{4}$	$2 \leq 4 \Rightarrow$	tausche 2 mit 8 nach Balken 1 und verschiebe beide Balken
2 | 9 8 | 1 5 $\underline{4}$	$1 \leq 4 \Rightarrow$	tausche 1 mit 9 nach Balken 1 und verschiebe beide Balken
2 1 | 8 9 | 5 $\underline{4}$	$5 > 4 \Rightarrow$	verschiebe Balken 2
2 1 | 8 9 5 | $\underline{4}$		tausche Pivot mit 8 nach Balken 1
2 1 $\underline{4}$ 9 5 8		(die Balken braucht es jetzt nicht mehr)

(b) *Best Case:* Pivotelement halbiert jeweils

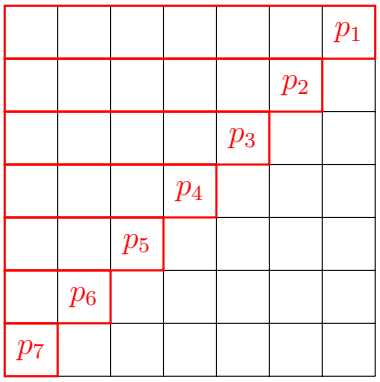
die noch zu sortienden Teilliste (ist *Median*)

(c) *Worst Case:* Pivotelement ist jeweils das Maximum (aufsteigend sortierte Liste)

Best Case



Worst Case



Sortieren: Aufgabe 7

```
1  def quicksort(A):
2      quicksort_helper(A, 0, len(A))
3
4  def quicksort_helper(A, a, b):
5      if a < b:
6          m = partition(A, a, b)
7          quicksort_helper(A, a, m)
8          quicksort_helper(A, m+1, b)
9
10 def partition(A, a, b):
11     pivot = A[b-1]
12     i = a-1
13     for j in range(a, b-1):
14         if A[j] <= pivot:
15             i += 1
16             A[i], A[j] = A[j], A[i]
17     A[i+1], A[b-1] = A[b-1], A[i+1]
18     return i+1
```

(a) $\underline{p}$: Pivot; vor Balken 1: $x \leq p$; nach Balken 1: $x > p$; nach Balken 2: x unverarbeitet

| | 3 6 1 7 2 9 $\underline{4}$	$3 < 4 \Rightarrow$	tausche 3 mit „ “ nach Balken 1 und verschiebe beide Balken
3 | | 6 1 7 2 9 $\underline{4}$	$6 > 4 \Rightarrow$	verschiebe Balken 2
3 | 6 | 1 7 2 9 $\underline{4}$	$1 \leq 4 \Rightarrow$	tausche 1 mit 6 nach Balken 1 und verschiebe beide Balken
3 1 | 6 | 7 2 9 $\underline{4}$	$7 > 4 \Rightarrow$	verschiebe Balken 2
3 1 | 6 7 | 2 9 $\underline{4}$	$2 \leq 4 \Rightarrow$	tausche 2 mit 6 nach Balken 1 und verschiebe beide Balken
3 1 2 | 7 6 | 9 $\underline{4}$	$9 > 4 \Rightarrow$	verschiebe Balken 2
3 1 2 | 7 6 9 | $\underline{4}$		tausche Pivot mit dem Element nach Balken 1
3 1 2 $\underline{4}$ 6 9 7		(die Balken braucht es jetzt nicht mehr)

(b) `quicksort_helper(...)` ist eine *rekursive Funktion*, da sie sich so lange selber aufruft, bis der *Base Case* erreicht wird.

(c) Der *Best Case* tritt dann auf, wenn an der Pivotposition der noch zu sortierenden Teilliste jeweils der Median der Elemente dieser Liste steht. Die asymptotische Laufzeit beträgt dann $O(n \log(n))$.

Der *Worst Case* tritt dann auf, wenn an der Pivotposition der noch zu sortierenden Teilliste jeweils das grösste (oder das kleinste) Element dieser Liste steht. Die asymptotische Laufzeit beträgt dann $O(n^2)$

(d) Modifikationen am Quicksort-Algorithmus um den Worst Case zu verhindern:

- *Randomized Quicksort:* Wähle zufällig ein Element aus dem zu partitionierenden Teil der Liste `L[a,b]` und tausche es mit dem letzten Element an der Pivotposition.
- *Median of Three:* Bestimme den Median der Elemente an der ersten, mittleren und letzten Position und vertausche ihn mit dem Element an der Pivotposition.