# Übung 4 — Abgabe 22.5.2019

## Aufgabe 1 – Asymptotische Komplexität — 12 Punkte

a) Beweisen Sie: $(x+20)\log_{10}(x+20) \in O(x \log_2 x)$ *3 Punkte*

b) Ordnen Sie die folgenden Funktionen nach aufsteigender asymptotischer Komplexität: *9 Punkte*

$$f_A(x) = 3^x$$
$$f_B(x) = \sqrt{x} + \log_2 x$$
$$f_C(x) = x^x$$
$$f_D(x) = x \log_2 x$$
$$f_E(x) = 2^{\sqrt{\log_2 x}}$$
$$f_F(x) = x^3 + 12x^2 + 200x + 999$$

und begründen Sie Ihre Entscheidung. Das heißt, bringen Sie die Funktionen in eine Reihenfolge $f_1 \prec f_2 \prec \cdots \prec f_5 \prec f_6$ und beweisen Sie für alle $i < 6$, dass $f_i \in O(f_{i+1})$ gilt. Sie können dafür entweder zeigen, dass der Grenzwert des Quotienten Null ist:

$$\lim_{x\to\infty} \frac{f_i(x)}{f_{i+1}(x)} = 0$$

(falls der Grenzwert unbestimmt ist, d.h. falls $\infty/\infty$ herauskommt, benutzen Sie die Regel von l'Hospital, oder Sie können vollständige Induktion verwenden: Mit dem Induktionsanfang "$f_i(x_0) \le c\, f_{i+1}(x_0)$ gilt für eine bestimmte Wahl von $c$ und $x_0$" und dem Induktionsschritt "aus $f_i(x) \le c\, f_{i+1}(x)$ folgt $f_i(x+1) \le c\, f_{i+1}(x+1)$" ist die Behauptung ebenfalls bewiesen.

## Aufgabe 2 – Komplexität des Siebs von Eratosthenes — 9 Punkte

Wir betrachten zwei Varianten vom Sieb des Eratosthenes (vgl. Übung 1):

```
def sieve1(N):
    primes = list(range(N+1))
    primes[1] = 0   # Zahl 1 streichen
    stop = N
    k = 2
    while k <= stop:

        j = 2*k
        while j <= N:
            primes[j] = 0 # j streichen
            j += k
        k += 1
    return [k for k in primes if k!=0]
```

```
def sieve2(N):
    primes = list(range(N+1))
    primes[1] = 0
    stop = N
    k = 2
    while k <= stop:
        if primes[k] != 0:
            j = 2*k
            while j <= N:
                primes[j] = 0
                j += k
        k += 1
    return [k for k in primes if k!=0]
```

Für die Syntax von [k for k …] siehe List Comprehensions.

a) Begründen Sie, warum beide Varianten die gleichen Ergebnisse liefern. *2 Punkte*

b) Die Laufzeit $f(N)$ wird zweckmäßig dadurch gemessen, dass man zählt, wie oft Zahlen gestrichen werden, d.h. wie oft '`primes[j] = 0`' aufgerufen wird. Zeigen Sie, dass bei Variante 1 gilt: $f_1(N) \in O(N \ln N)$. Die Formel $\sum_{k \le N} \frac{1}{k} \in O(\ln N)$ ist dabei hilfreich. *3 Punkte*

c) Beweisen Sie analog, dass bei Variante 2 gilt: $f_2(N) \in O(N \ln(\ln N))$. Hier hilft die Formel $\sum_{p \le N:\, p \text{ is prime}} \frac{1}{p} \in O(\ln(\ln N))$, wobei sich die Summe nur über die Primzahlen *p* bis maximal *N* erstreckt. *2 Punkte*

d) Warum wäre es ausreichend, die äußere Schleife bereits bei '`stop = sqrt(N)`' zu beenden (statt bei '`stop = N`' wie oben)? Hat diese Änderung Auswirkungen auf die Komplexität? *2 Punkte*

## Aufgabe 3 – Codeoptimierung 5 Punkte

Eine naive Implementation der Multiplikation von zwei `size*size` Matrizen lautet[1]

```
for i in range(size):
    for j in range(size):
        for k in range(size):
            C[i + j*size] += A[i + k*size] * B[k + j*size]
```

Diese Implementation ist relativ langsam, weil die innere Schleife redundante Berechnungen enthält. Optimieren Sie die Implementation, z.B. durch Umstellen der Schleifenreihenfolge und durch Verschieben von invarianten Teilausdrücken aus der inneren Schleife, und begründen Sie, warum Ihre Optimierungen sinnvoll sind. Verwenden Sie das `timeit`-Modul (Kurzanleitung siehe unten), um die Laufzeiten der beiden Versionen für `size`=100 zu vergleichen. Die optimierte Implementation sollte ungefähr doppelt so schnell sein wie die naive. Ändert sich durch die Optimierung die Komplexität des Algorithmus? Geben Sie Ihre Lösung im File `matrix.py` ab.

## Aufgabe 4 – Speichermanagement eines Containers 14 Punkte

Wir hatten im Übungsblatt 2 den `UniversalContainer` implementiert. Hier wollen wir drei Varianten dieser Datenstruktur hinsichtlich ihrer Laufzeit vergleichen. Die drei Varianten unterscheiden sich dadurch, wie effizient die Funktionen `push()` und `popFirst()` arbeiten:

`UniversalContainer1`: Wenn in `push()` der interne Speicher voll ist (`size_ == capacity_`), wird die Kapazität um ein Element vergrößert. Hier sind `push()` und `popFirst()` beide ineffizient.

`UniversalContainer2`: Wie Variante 1, aber die Kapazität wird jeweils verdoppelt. Dadurch wird `push()` effizient (Verhalten eines dynamischen Arrays).

`UniversalContainer3`: Wie Variante 2, aber der interne Speicher wird als Ringpuffer verwaltet. Dadurch wird auch `popFirst()` effizient.

Ein Ringpuffer funktioniert folgendermaßen: Der aktive Bereich des internen Speichers erstreckt sich nicht immer von 0 bis `size_-1`, sondern wird durch einen Anfangs- und einen Endindex bezeichnet. `popLast()` dekrementiert wie bisher den Endindex, aber `popFirst()` muss nicht mehr aufwändig die Daten einen Index nach vorn kopieren, sondern inkrementiert einfach den Anfangsindex. Beim Indexzugriff mittels `c[i]` muss deshalb intern der Anfangsindex addiert werden.

Um den vorhandenen Speicher optimal auszunutzen, wird `push()` modifiziert: Solange der Endindex das Ende des internen Speichers noch nicht erreicht hat, wird er normal inkrementiert. Ist er jedoch bei

[1] Matrizen sollen durch eindimensionale Arrays implementiert werden, so dass z.B. das Matrixelement $C_{ij}$ als `C[i+j*size]` indexiert wird. Für die Zeitmessung nehmen wir an, dass alle $C_{ij}$ bereits auf 0 initialisiert wurden.

`capacity_-1` angekommen, überprüfen wir den Anfangsindex: Ist dieser größer als Null, haben wir *am Anfang* des internen Speichers noch freie Kapazität. Wir setzen den Endindex deshalb auf Null und organisieren den Speicher somit als "Ring" (das ist äquivalent dazu, dass wir den Endindex modulo `capacity_` inkrementieren). Bei darauffolgenden `push()`–Aufrufen wird der Endindex weiter inkrementiert, und erst wenn er den Anfangsindex einholt, ist die Kapazität erschöpft und muss verdoppelt werden (mit Rücksetzen des Anfangsindex auf Null und dem üblichen Umkopieren der Daten). Entsprechend müssen auch das Inkrementieren/Dekrementieren in `popFirst()`/`popLast()` und der Indexzugriff in `__getitem__()`/`__setitem__()` zyklisch modulo `capacity_` behandelt werden.

a) Implementieren Sie die drei Varianten der Datenstruktur im File `ringbuffer.py`. Bauen Sie dabei auf Ihrer Lösung zu Übungsblatt 2 auf. Erweitern Sie die Tests aus dieser Übung, so dass alle drei Varianten getestet werden. Das sorgfältige Testen des Ringpuffers ist besonders wichtig, weil bei den zyklischen Berechnungen leicht Fehler passieren. Damit das zyklische Verhalten tatsächlich auftritt (das gehört zur "Code Coverage" beim Testen), müssen Aufrufe von `push()` und `popFirst()` in geeigneter Weise abwechseln – erst nach mindestens einem `popFirst()` kann überhaupt ein Ring entstehen. *8 Punkte*

b) Implementieren Sie mit dem `timeit`-Modul eine Funktion, die den Zeitaufwand für N aufeinanderfolgende `push()`-Aufrufe misst. Bestimmen Sie den Aufwand der drei Varianten für verschiedene Werte von N und zeigen Sie dadurch, dass die obigen Aussagen zur Effizienz korrekt sind. Schätzen Sie aus Ihren Messungen die amortisierte Komplexität. Beziehen Sie in den Vergleich auch die Python-Klasse `list` (also deren `append()`-Funktion, die unserem `push()` entspricht) ein. *3 Punkte*

c) Führen Sie das analoge Experiment für N aufeinanderfolgende Aufrufe von `popFirst()` durch. Die Zeit für das vorherige Einfügen der Elemente in die Datenstruktur soll in die Messung nicht mit eingehen. Bei der Klasse `list` verwenden Sie `pop(0)` anstelle von `popFirst()`. *3 Punkte*

Bitte laden Sie Ihre Lösung bis zum 22.5.2019 um 12:00 Uhr auf Moodle hoch.

## Verwendung des timeit-Moduls

Das `timeit`-Modul (siehe docs.python.org/3/library/timeit.html) dient zur Messung der Laufzeit in Python. Der Code, dessen Laufzeit gemessen werden soll, wird dem `Timeit`-Konstruktor als (mehrzeiliger) String übergeben. In einem zweiten String kann man Initialisierungscode angeben, der zuvor ausgeführt werden muss, dessen Zeit aber nicht in die Messung eingehen soll. Für das Beispiel der Klasse `UniversalContainer1` sieht dies so aus:

```
import timeit

# your class
class UniversalContainer1:
    ...

initialisation = '''
c = UniversalContainer1()
'''

code_to_be_measured = '''
for i in range(N):
    c.push(i)
```

```
'''

repeats = 10   # repeat the test so many times
N = 400        # number of pushs to execute per test
t = timeit.Timer(code_to_be_measured, initialisation, globals=globals())
```

Der Parameter `globals=globals()` ist notwendig, damit der Testcode auf die Variable `N` und die Klassendefinitionen `UniversalContainer1` etc. zugreifen kann. Die eigentliche Zeitmessung erfolgt dann mit der Methode `t.repeat(repeats, 1)`, wobei `repeats` angibt, wie oft das Programm wiederholt werden soll, damit man eine verlässliche Laufzeit bekommt. Als Laufzeit verwendet man das Minimum der Wiederholungen, weil man davon ausgeht, dass dabei die wenigsten Unterbrechungen durch andere Programme und das Betriebssystem aufgetreten sind:

```
time = min(t.repeat(repeats, 1))
print("execution time:", (time*1000), "ms")
```