Übung 4 — Abgabe 5.6.2020

Aufgabe 0 – Debuggen von Übung 3

Studieren Sie die Musterlösung zu Übung 3 auf Moodle und vergleichen Sie mit Ihrer eigenen Lösung. Fügen Sie Kommentare in Ihre ursprüngliche Abgabe ein, welche Fehler Sie gemacht haben und wie die Lösung korrigiert werden muss. Kommentieren Sie auch Dinge, die bei Ihnen zwar korrekt, aber umständlich oder ineffizient gelöst sind. Je besser Ihre Kommentare sind, desto weniger Punkte werden abgezogen. Insbesondere werden keine Punkte für Folgefehler abgezogen, wenn Sie die Grundursache eines Problems identifiziert und korrigiert haben. In Quellcode-Dateien kennzeichnen Sie Ihre Kommentare mit

```
# comment: <Ihr Kommentar>
```

In PDF-Dateien verwenden Sie eine geeignete Farbe. Wichtig ist, dass die Tutoren die Kommentare leicht von der ursprünglichen Abgabe unterscheiden können.

Benennen Sie die Dateien mit kommentierten Lösungen in "`kommentiert_<original-filename>`" um und geben Sie sie im zip-File für Übung 4 ab.

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 ∞/∞ 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
```

```
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 streichen
        j += k
    k += 1
return [k for k in primes if k!=0]
```

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

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 $\sum_{k \leq N} \frac{1}{k} \in O(\ln N)$ 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 $\sum_{p \leq N:\, p \text{ is prime}} \frac{1}{p} \in O(\ln(\ln N))$, 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 – Speichermanagement eines Containers 19 Punkte

Wir haben in der Vorlesung die abstrakten Containertypen *Array, Stack* und *Queue* kennengelernt. In dieser Aufgabe sollen Sie eine Python-Klasse `array_deque` implementieren und testen, die die Fähigkeiten aller drei Container vereinigt. Die folgende Funktionalität soll unterstützt werden:

```
c = array_deque()     # create empty container
c.size()              # get number of elements
c.capacity()          # get number of available memory cells
c.push(v)             # append element v at the end
c.pop_first()         # remove first element (Queue functionality)
c.pop_last()          # remove last element (Stack functionality)
v = c[k]              # read element at index k (Array functionality)
c[k] = v              # replace element at k (Array functionality)
v = c.first()         # read first element (Queue functionality)
v = c.last()          # read last element (Stack functionality)
```

Einen Rahmen, den Sie nur noch vervollständigen müssen, finden Sie im File array_deque.py auf Moodle.

Die Klasse verwaltet einen internen Speicherbereich `_data` der Größe `_capacity`, der aber nur bis zur Größe `_size` gefüllt ist. Beim Aufruf `c.push(v)` wird Element v in die nächste freie Speicherzelle von `_data` geschrieben und `_size` inkrementiert. Gilt allerdings `_size == _capacity`, muss in der Funktion `push()` erst ein Speicherbereich mit größerer (z.B. verdoppelter) `_capacity` geschaffen und die vorhandenen Daten dahin umkopiert werden. Das entspricht dem Vorgehen bei dynamischen Arrays aus der Vorlesung.

Damit auch `c.pop_first()` und `c.pop_last()` effizient sind, d.h. konstante Komplexität haben, soll die Datenstruktur als Ringpuffer implementiert werden. Dabei wird der interne Speicher `_data` zyklisch adressiert, d.h. nach dem letzten Index (`_capacity-1`) folgt wieder der Index 0. Umgekehrt liegt vor Index 0 der Index (`_capacity-1`). Das erreicht man elegant, indem man Indexoperationen mit Hilfe der Modulo-Operation (bezüglich `_capacity`) implementiert.

Der Ringpuffer ermöglicht es, den gerade aktiven Bereich des internen Speichers auf dem Ring beliebig zu verschieben: Er erstreckt sich nicht zwangsläufig von Index 0 bis Index `_size-1` (wie beim normalen dynamischen Array), sondern wird durch einen beliebigen Anfangs- und korrespondierenden Endindex gekennzeichnet. Dadurch kann `pop_first()` einfach den Anfangsindex inkrementieren, und `pop_last()` den Endindex dekrementieren – es müssen keine Elemente umkopiert werden. Bei Indexzugriffen mittels c[i] muss natürlich intern der Anfangsindex addiert werden.

Jetzt kann in `push()` ein dritter Fall eintreten: Wenn der aktuelle Endindex auf `_capacity-1` steht, ist *am Ende* des Speicherbereichs `_data` kein Platz mehr frei. Im normalen dynamischen Array müsste man jetzt den Speicher verdoppeln. Im Ringpuffer gilt jedoch: ist der Anfangsindex größer als 0, gibt es *am Anfang* von _data noch freien Speicherplatz, den `push()` nutzen kann: es setzt den Endindex auf `0` zurück und kopiert das neue Element an diese Position. Dadurch wird der Endindex kleiner als der Anfangsindex, und das bedeutet einfach, dass der aktive Speicherbereich die "Nahtstelle" des Rings enthält. Eine Verdoppelung des Speichers ist erst notwendig, wenn der Endindex durch weitere `push()`-Aufrufe den Anfangsindex einholt.

Die Datenstruktur soll folgende Axiome erfüllen:

- Ein neuer Container hat die Größe 0.
- Zu jeder Zeit gilt: `c.size() <= c.capacity()`.
- Nach einem push gilt: (i) die Größe hat sich um eins erhöht, (ii) das gerade eingefügte Element ist jetzt das letzte, (iii) alle anderen Elemente haben sich nicht verändert, (iv) wenn der Container vorher leer war, ist das eingefügte Element auch das erste, andernfalls bleibt das erste Element unverändert, (v) ein nachfolgendes `pop_last()` reproduziert wieder den Container vor dem push (Sie können für diesen Vergleich mit der Funktion `deepcopy()` aus dem Modul copy eine Kopie des ursprünglichen Containers erzeugen).
- Nach `c[k] = v` gilt: (i) die Größe bleibt unverändert, (ii) Index `k` enthält das Element `v`, (iii) die übrigen Elemente haben sich nicht verändert.
- Nach `c.pop_last()` gilt: (i) die Größe hat sich um eins verringert, (ii) die Elemente vom ersten bis zum vorletzten bleiben unverändert. War des Container bereits leer, wird eine Exception ausgelöst.
- Nach `c.pop_first()` gilt: (i) die Größe hat sich um eins verringert, (ii) das zweite bis letzte Element sind einen Index nach unten gerückt. War des Container bereits leer, wird eine Exception ausgelöst.
- Wenn der Container nicht leer ist, gilt stets (i) `c.first() == c[0]` und (ii) `c.last() == c[c.size()-1]`.

Aufgaben (geben Sie das vervollständigte File `array_deque.py` bzw. `array_deque.npynb` ab):

a) Vervollständigen Sie die Implementierung von `array_deque` und fügen Sie Dokumentation hinzu. *7 Punkte*

b) Implementieren Sie in der Funktion `test_array_deque()` Tests für die obigen Axiome, die mit `pytest` ausgeführt werden können (siehe Übungsblatt 3). Beachten Sie dabei, dass alle Möglichkeiten abgedeckt werden, wie der aktive Speicherbereich des Containers liegen kann (inklusive der jeweiligen Randfälle). *7 Punkte*

   Testen Sie außerdem Codebeispiele in Ihrer Dokumentation mittels `doctest`.

c) Prüfen Sie mit `timeit`, dass die Funktion `push()` tatsächlich amortisierte konstante Komplexität hat. *2 Punkte*

d) Implementieren Sie die abgeleitete Klasse `slow_array_deque`, die die `push()`-Funktion naiv implementiert: Ist der Speicher voll, wird er nur um ein Element vergrößert (statt ihn zu verdoppeln). Prüfen Sie, dass die Tests auch für diese Datenstruktur funktionieren, und zeigen Sie mit `timeit`, dass `push()` jetzt lineare Komplexität hat. *3 Punkte*

Bitte laden Sie Ihre Lösung bis zum 5.6.2020 um 16:00 Uhr auf Moodle hoch.