Verschiedene Programmieraufgaben rund um Primzahlen

Aufgaben zu LE2 (+ LE3)

Diese Aufgaben sollen helfen, etwas praktische Erfahrung mit dem imperativen Programmieren zu sammeln – konkret mit dem Umgang mit Variablen, Zahlenwerten, Schleifen und Listenstrukturen. Je nach Gliederung der Lösung in Funktionen bzw. Prozeduren lassen sich damit auch Themen von LE3 explorieren.

Teil des Programmierens ist nicht nur das Aufschreiben von Python-Anweisungen, sondern es gehört auch dazu, eine Fragestellung zu verstehen, einen Plan für die nötigen Berechnungen zu machen und das fertige Programm daraufhin zu untersuchen, ob es richtig funktioniert.

Der Sinn der folgenden Aufgaben liegt daher darin, selbst ein Programm zu planen, aufzuschreiben und auszuprobieren. Sicher lassen sich auch fertige Programme mit einer Internet-Suche finden, die man einfach abschreiben kann, aber dabei lernt man eben nichts.

Die Aufgaben sind nicht als «Übungen» gedacht, von Dingen, die Ihr eigentlich können solltet, sondern als «Lernaufgaben», die Euch anregen sollen, Dinge auszuprobieren und Fragen aufzuwerfen, woraus Ihr etwas lernen könnt. Also verzweifelt nicht, wenn nicht alles sofort klar ist.

Wenn Ideen fehlen, Lösungen nicht funktionieren, oder nicht klar ist, ob sich etwas verbessern liesse: Nicht einfach irgendeine fertige Lösung anschauen, sondern mit dem eigenen Lösungsansatz bzw. der individuell spezifischen Frage gezielt das Verständnis erweitern. Fragen stellen kann man:

- im Space von gpr auf DS-Spaces
- mir, als Fachexperten im Büro oder per E-Mail oder via Webex oder in der Sprechstunde etc.
- Mitstudierenden
- allen Menschen, von denen man denkt, dass sie vielleicht helfen könnten (Onkel, Tante, Grosseltern, Kindern, Nachbarn, …)

Aufgabe 1 – Einfacher Primzahltest

Eine Primzahl ist eine *natürliche Zahl* grösser als 1, die durch keine andere natürliche Zahl teilbar ist, ausser 1 und sich selbst.

Um zu testen, ob eine ganze Zahl $n \geq 2$ eine Primzahl ist, kann man für alle grundsätzlich geeigneten ganzen Zahlen t prüfen, ob t die Zahl n ganzzahlig teilt – d.h. es gibt eine ganze Zahl s so dass $s \cdot t = n$ – und aus dem Fund oder Nicht-Fund geeignete Schlüsse ziehen.

Schreibe eine Python-Funktion *is_prime(n)* auf, die einen entsprechenden Wahrheitswert zurückgibt:

```
>>> is_prime(1)
False
>>> is_prime(2)
True
>>> is_prime(3)
True
>>> is_prime(4)
False
>>> is_prime(47)
True
```

Aufgabe 2 – Liste der Primzahlen im Bereich 2 bis n

Die Lösung von Aufgabe 1 kann man benutzen (d.h. die Funktion *is_prime* aufrufen), um eine Liste aller Primzahlen im Bereich ab 2 bis zu einer Obergrenze *n* zu berechnen.

Programmiere eine Funktion *all_primes(n)*, die das tut:

```
>>> all_primes(100)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83,
89, 97]
```

Zusätzliche Herausforderung: Finde heraus, was in Python mit «list comprehension»[1] gemeint ist und versuche damit den Code innerhalb von *all_primes(n)* in einer einzigen Zeile aufzuschreiben.

[1] Zwei Quellen zu «list comprehension»: docs.python.org/3/tutorial/datastructures.html#list-comprehensions und youtu.be/AhSvKGTh28Q

Aufgabe 3 – Sieb des Eratosthenes

Ein anderer Lösungsansatz für die oben beschriebene Aufgabe alle Primzahlen ab 2 bis zu einer festen Grenze *n* zu ermitteln, ist *das Sieb des Eratosthenes*.[2]

Programmiere eine Funktion *eratosthenes(n)*, die dieses Verfahren implementiert:

```
>>> eratosthenes(100)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83,
89, 97]
```

Es gibt verschiedene Möglichkeiten, wie man die Idee von Eratosthenes in ein Computerprogramm (in Python) umsetzen kann. Hier ist bewusst keine davon vorgegeben, denn es gehört zu den Herausforderungen beim Programmieren, verschiedene Möglichkeiten zu überlegen und sich für eine gute zu entscheiden.

Aufgabe 4 – Abstände zwischen Primzahlen

Mathematiker treibt seit langem die Frage nach Regelmässigkeiten bei Primzahlen um. (Vermutlich gibt es eben keine. Aber das zu Beweisen ist eben auch schwierig.) In diesem Zusammenhang ist die Frage interessant, in welchen Abständen Primzahlen vorkommen.

Programmiert eine Methode *distances(primes)*, die eine Liste der Abstände zwischen je zwei aufeinanderfolgenden Primzahlen berechnet, wenn man eine Liste mit aufeinanderfolgenden Primzahlen als Parameter gibt. Diese Liste ist natürlich um einen Wert kürzer als die Liste der Primzahlen im gleichen Bereich:

```
>>> distances(eratosthenes(100))
[1, 2, 2, 4, 2, 4, 2, 4, 6, 2, 6, 4, 2, 4, 6, 6, 2, 6, 4, 2, 6, 4, 6, 8]
```

Aufgabe 5 – Heuristik der Abstände zwischen Primzahlen

Was man einer Liste, wie sie von Aufgabe 4 produziert wird, nicht so gut ansieht, ist wie häufig die einzelnen Abstände zwischen Primzahlen auftreten – vor allem wenn die Liste länger wird. Deswegen soll als letzte Aufgabe eine Heuristik der in der Abstandsliste auftretenden Zahlen programmiert werden.

Eine solche Heuristik besteht aus Paaren von Zahlen: Je ein Distanzwert und dessen Häufigkeit in der analysierten Liste.

```
>>> heuristic(distances(eratosthenes(100)))
[(1, 1), (2, 8), (4, 7), (6, 7), (8, 1)]
```

Aufgabe 6 – Laufzeitmessungen

Die Ausführung der Programme aus den Aufgaben 1 bis 5 braucht jeweils eine gewisse Rechenzeit. Die hängt von verschiedenen Faktoren ab. Am deutlichsten wohl von der Grösse der betrachteten Primzahlen bzw. der Grösse des betrachteten Zahlenbereichs. Manche Faktoren sind in der Laufzeitumgebung begründet, wie der Hardware und des Programmiersystems bzw. der Programmiersprache. Weitere Faktoren – und die haben oft viel grössere Auswirkungen – haben mit dem programmierten Verfahren zu tun.

Um Einsichten in solche Zusammenhänge zu erhalten, ist es interessant, Laufzeit messen zu können.

Strategie 1

Ein naheliegender Ansatz ist das zu analysierende Programm einmal laufen zu lassen und vorher sowie nachher «auf die Uhr zu schauen». Bei Programmen, die sehr lange laufen, genügt das. Viele Programme sind aber so schnell, dass man die Ablesung der Uhr automatisieren und in das Programm integrieren muss, um sinnvolle Ergebnisse zu erhalten.

Bei sehr schnellen Programmen liegt die Laufzeit aber immer noch unter oder gerade in der Grössenordnung der Auflösung der Zeitmessung: Wenn auf Hundertstelsekunden genau gemessen wird, ist es schwierig, Erkenntnisse über ein Programm zu gewinnen, das nur 2 ms läuft.

Um unter solchen Umständen genauer messen zu können, kann man das Programm in einer Schleife mehrfach direkt hintereinander ausführen lassen und die gemessene Zeit durch die Anzahl der Ausführungen teilen. Manchmal ist es auch sinnvoll, die *Problemgrösse* zu verändern, also z.B. die Primzahlen im Bereich 2 bis 10^9 zu untersuchen – wobei da evtl. etwas anderes passiert, als wenn man ein kleineres Problem mehrfach rechnet.

[2] Quelle zu youtu.be/mX6VQtbNywg?t=91

Strategie 2

Besonders für Programme, die in wenigen Sekunden oder in Sekundenbruchteilen ablaufen, bietet sich noch eine andere Messmethode an: Man lässt in einer Schleife das Programm so lange wiederholt ausführen, bis eine vorher festgelegte Zeitspanne verstrichen ist. Dann dividiert man die tatsächlich verstrichene Zeit durch die Anzahl Ausführungen, die in dieser Zeit möglich waren.

Ein Vorteil dieses zweiten Verfahrens ist, dass die Messungen selbst immer gleich viel Zeit in Anspruch nehmen. Will man z.B. sehr unterschiedlich schnell ablaufende Programme vergleichen, wie unsere Funktion *all_primes* gegenüber der Funktion *eratosthenes*, muss man sich nicht Gedanken machen, wie oft man die Programme jeweils wiederholen soll, um vernünftige Messungen möglich zu machen. Ebenso, wenn man die Messungen auf ganz verschiedenen Computern gemacht werden (z.B. von Studierenden) hilft es, wenn man die Dauer der Zeitmessung fixieren und die Anzahl der Programmläufe an die Notwendigkeiten anpassen kann.

Werkzeuge zur Zeitmessung

Mögliche Werkzeuge zur Zeitmessung sind *timeit*[3] und *datetime.datetime.now*[4]. Beide sind in eigenen Modulen in der Python-Grundausstattung enthalten (sind also zusammen mit dem Interpreter schon installiert), müssen aber *importiert* werden.

timeit bietet einen leicht zu nutzenden Vermessungsservice für rasch laufende Programme. Beispielsweise kann man die gewünschte Anzahl Wiederholungen gerade mit einem Parameter festlegen und braucht nicht selbst eine Schleife zu programmieren.

Mit dem Modul *datetime* kann auch mit Zeitangaben und -Intervallen gerechnet werden. Dabei wird dann *datetime.timedelta*[5] benutzt:

```
start = datetime.datetime.now()
# put here the code to investigate
dur = (datetime.datetime.now() - start)
time_in_seconds = (dur.microseconds + 10 ** 6 * dur.total_seconds()) / cnt / 10 ** 6
```

In jedem Fall sollte man immer eine Reihe von Messungen machen, damit sehen kann, wie stark die Messergebnisse schwanken. Das gibt einen Eindruck von der Genauigkeit und der Aussagekraft der gemessenen Zahlen.

Experimentiere mit den beschriebenen Zeitmessungs-Strategien!

Die Tabelle zeigt jeweils 3 Vergleichs-Messung mit Lösungs-Programmen verschiedener der obigen Aufgaben und verschiedenen Problemgrössen, gemessen auf Surface 7 Pro.

	all_primes	**eratosthenes**	**heuristic distances eratosthenes**
n = 10'000	0.011 0.010 0.010	0.001 0.001 0.001	0.002 0.001 0.002
n = 100'000	0.239 0.197 0.246	0.016 0.013 0.015	0.017 0.017 0.017
n = 1'000'000	6.430 5.847 6.415	0.183 0.156 0.181	0.192 0.195 0.197

[3] Dokumentation und Beispiele zu *timeit*: docs.python.org/3/library/timeit.html
[4] Dokumentation und Beispiele zu datetime:
[5] timedelta