LU07g - Leistung verbessern

Sie haben gemessen (LU07f) und wissen, wo die Zeit hingeht. Jetzt geht es um die Frage, *was* man dagegen tut. Die wirksamsten Massnahmen sind fast nie «cleverer Code», sondern eine andere **Datenstruktur** oder ein anderer **Algorithmus**.

Komplexität in einer Minute

Die Landau-Notation beschreibt, wie die Laufzeit wächst, wenn die Datenmenge wächst:

Notation	Heisst	Beispiel	10-mal mehr Daten
O(1)	konstant	`x in set`, `dict[key]`	gleich schnell
O(log n)	logarithmisch	Suche in sortierter Liste (`bisect`)	kaum langsamer
O(n)	linear	`x in list`, eine Schleife	10-mal langsamer
O(n log n)		`sorted()`	gut 10-mal langsamer
O(n²)	quadratisch	Schleife in Schleife	**100-mal langsamer**

Der Sprung von O(n²) auf O(n) ist der einzige, der bei wachsenden Daten wirklich rettet. Mikrooptimierungen innerhalb einer O(n²)-Schleife verschieben das Problem nur.

Massnahme 1: Die richtige Datenstruktur

Operation	list	set / dict
`x in sammlung`	O(n)	O(1)
Element anfügen	O(1)	O(1)
`sammlung.insert(0, x)`	O(n)	-
Zugriff über Index	O(1)	-
Reihenfolge bleibt erhalten	ja	dict ja, set nein
Duplikate möglich	ja	nein

Der Klassiker ist die Zugehörigkeitsprüfung. Gemessen mit 2000 Anfragen gegen 10'000 gesperrte Konten:

```python
# vorher
gesperrt = [f"user{i}" for i in range(10000)]
treffer = sum(1 for a in anfragen if a in gesperrt)

# nachher
gesperrt = {f"user{i}" for i in range(10000)}     # set statt list
treffer = sum(1 for a in anfragen if a in gesperrt)
```

```
list  0.107    s
set   0.000068 s     -> rund 1500-mal schneller, identisches Ergebnis
```

Eine einzige geänderte Klammer. Genau solche Fälle sind für **D1A** gemeint, wenn dort von «geeignete Algorithmen oder Datenstrukturen auswählen» die Rede ist.

Massnahme 2: Verschachtelte Schleifen durch einen Index ersetzen

Zwei Datenquellen zusammenführen ist im ersten Wurf fast immer $O(n^2)$:

```
# vorher - für jede Bestellung alle Kunden durchsuchen
def zuordnen(bestellungen, kunden):
    out = []
    for b in bestellungen:
        for k in kunden:
            if k["id"] == b["kunde_id"]:
                out.append((k["name"], b["betrag"]))
                break
    return out
```

```
# nachher - einmal einen Index bauen, dann direkt zugreifen
def zuordnen(bestellungen, kunden):
    index = {k["id"]: k for k in kunden}
    return [(index[b["kunde_id"]]["name"], b["betrag"]) for b in
bestellungen]
```

```
verschachtelt  0.0585  s
mit Index      0.00028 s     -> rund 200-mal schneller  (2000 x 2000
Datensätze)
```

Und die Nachher-Version ist zusätzlich kürzer und deklarativ - Refactoring und Optimierung fallen hier zusammen.

Massnahme 3: Zwischenergebnisse merken (Memoization)

Wenn dieselbe Berechnung mit denselben Argumenten mehrfach vorkommt, speichert `functools.cache` das Ergebnis. Das funktioniert **nur bei pure functions** - eine Funktion mit Seiteneffekten würde beim zweiten Aufruf ihren Seiteneffekt überspringen.

```
from functools import cache

@cache
def fibonacci(n):
    return n if n < 2 else fibonacci(n - 1) + fibonacci(n - 2)
```

```
fibonacci(30) ohne cache   0.0899    s
fibonacci(30) mit  cache   0.0000003 s
```

Der Grund steht in LU03b: Ohne Cache berechnet der Aufrufbaum dieselben Teilprobleme tausendfach neu. Die Animation dort zeigt es für `fibonacci(4)`.

Grenzen von `@cache`

- Nur für **pure functions** - sonst verschwinden Seiteneffekte ab dem zweiten Aufruf.
- Argumente müssen **hashbar** sein: `int`, `str`, `tuple`, `frozenset`. Eine Liste als Argument wirft `TypeError`.
- Der Cache wächst unbegrenzt. Bei vielen verschiedenen Argumenten besser `@lru_cache(maxsize=1000)`.
- **Veraltete Daten:** Wer eine Datenbankabfrage cacht, sieht Änderungen nicht mehr. Das ist der häufigste Fehler in Web-Projekten.

Massnahme 4: Generatoren statt Listen

Eine Comprehension in eckigen Klammern baut die ganze Liste im Speicher auf. Runde Klammern liefern einen Generator, der die Werte einzeln erzeugt (LU04i).

```
total = sum([zeilenumsatz(e) for e in eintraege])   # baut erst die Liste
total = sum(zeilenumsatz(e) for e in eintraege)     # erzeugt Werte einzeln
```

Bei 10 Einträgen egal, bei einer Million Zeilen aus einer CSV entscheidend. Preis: Der Generator ist nur **einmal** durchlaufbar (siehe LU07e).

Dazu passt der Kurzschluss von `any` und `all`:

```
# vorher - prüft alle Einträge, auch wenn der erste schon passt
gefunden = len([e for e in eintraege if e["id"] == gesucht]) > 0

# nachher - hört beim ersten Treffer auf
gefunden = any(e["id"] == gesucht for e in eintraege)
```

Massnahme 5: Arbeit aus der Schleife herausziehen

```
# vorher - sortiert bei jedem Durchlauf neu
for kunde in kunden:
    top = sorted(produkte, key=lambda p: p["preis"])[:3]
    ...

# nachher - einmal sortieren, dann verwenden
top = sorted(produkte, key=lambda p: p["preis"])[:3]
for kunde in kunden:
    ...
```

Dasselbe gilt für Datenbankabfragen in Schleifen - in Flask der mit Abstand häufigste Performance-Fehler: eine Abfrage pro Zeile statt einer Abfrage für alle Zeilen.

Reihenfolge der Massnahmen

1. **Messen**, wo die Zeit hingeht.
2. Steckt der Hotspot in einer verschachtelten Schleife oder in `in` auf einer Liste? Dann **Datenstruktur oder Algorithmus** ändern - das bringt Faktoren.
3. Wird dasselbe mehrfach berechnet? **Cache**, sofern die Funktion pure ist.
4. Wird zu viel im Speicher gehalten? **Generator**.
5. Nachmessen und das Ergebnis mit `assert` auf Gleichheit prüfen.

Was Sie im Portfolio für D1A zeigen: eine gemessene Zahl vorher, eine nachher, dieselben Testdaten - und die Begründung, *warum* Sie diese Datenstruktur gewählt haben und was der Nachteil ist. Ein `set` ist nicht gratis: Es verliert die Reihenfolge, entfernt Duplikate und braucht hashbare Elemente. Wer das mitschreibt, argumentiert auf Niveau A.

M323-LU07, M323-D1A