

## Informatik I: Einführung in die Programmierung

Prof. Dr. Peter Thiemann  
Hannes Saffrich, Marc Fuchs  
Wintersemester 2020

Universität Freiburg  
Institut für Informatik

### Übungsblatt 12

**Abgabe: Montag, 08.02.2021, 9:00 Uhr morgens, über git<sup>1</sup>**

#### Hinweis

Aufgabenteile werden mit **0 Punkten** bewertet wenn:

- Dateien und Definitionen nicht so benannt sind, wie im Aufgabentext gefordert;
- Dateien falsche Formate haben, z.B. PDF statt plaintext; oder
- Pythonskripte wegen eines Syntaxfehlers nicht ausführbar sind.

Gruppenaufgaben müssen von allen Mitgliedern abgegeben werden und in der ersten Zeile müssen die Mitglieder in einem Kommentar vermerkt werden, z.B:

```
# Gruppe: xy123, yz56, zx934
```

#### Hinweis

In diesem Übungsblatt müssen Sie **keine** Typannotationen schreiben.

In den meisten Aufgaben müssen Sie Funktionen definieren, deren Rumpf aus einem einzigen Ausdruck besteht - wie üblich in der Funktionalen Programmierung. Beispiel:

```
def inc(x):  
    return x + 1
```

Diese Funktionen können Sie auch als Variable mit Funktionswert definieren:

```
inc = lambda x: x + 1
```

Diese Schreibweise ist insbesondere bei bestimmten verschachtelten Funktionen angenehmer zu lesen:

```
def mul_1(x):  
    def mul_with_x(y):  
        return x * y  
    return mul_with_x
```

```
mul_2 = lambda x: lambda y: x * y
```

```
assert mul_1(2)(3) == 6  
assert mul_2(2)(3) == 6
```

---

<sup>1</sup><https://inpro.informatik.uni-freiburg.de/>

**Aufgabe 12.1** (Funktionskomposition; Datei: `compose.py`; 2 Punkte)

In dieser Aufgabe sollen Ihre Funktionsdefinitionen außer einer `return`-Anweisung keine weiteren Zeilen enthalten, oder wie im Hinweis als Variable mit Funktionswert definiert werden.

Schreiben Sie eine Funktion `compose`, die zwei einstellige Funktionen `f` und `g` als Argument nimmt und die Funktionskomposition  $f \circ g$  zurückgibt.

Beispiel:

```
inc = lambda x: x + 1
mul2 = lambda x: x * 2
assert compose(inc, mul2)(4) == 9
```

**Aufgabe 12.2** (Vektoren; Datei: `vectors.py`; 4 Punkte)

In dieser Aufgabe sollen Ihre Funktionsdefinitionen außer einer `return`-Anweisung keine weiteren Zeilen enthalten, oder wie im Hinweis als Variable mit Funktionswert definiert werden.

Die folgende Datenklasse beschreibt 3-dimensionale Vektoren:

```
@dataclass
class V3:
    x: int
    y: int
    z: int
```

Die komponentenweise Addition zweier Vektoren ist definiert durch:

```
cadd = lambda v, w: V3(v.x + w.x, v.y + w.y, v.z + w.z)
```

Die komponentenweise Multiplikation zweier Vektoren ist definiert durch:

```
cmul = lambda v, w: V3(v.x * w.x, v.y * w.y, v.z * w.z)
```

Schreiben Sie eine Funktion `map2`, die eine zweistellige Funktion `f` auf ganzen Zahlen als Argument nimmt und eine zweistellige Funktion auf Vektoren zurückgibt, welche `f` komponentenweise auf den Vektoren anwendet.

Verwenden Sie `map2` um die komponentenweise Addition, Multiplikation, Subtraktion, ganzzahlige Division, Exponentiation durch jeweils einen einzigen `map2`-Aufruf zu definieren:

```
cadd = map2(...)
cmul = map2(...)
csub = map2(...)
cdiv = map2(...)
cpow = map2(...)
```

**Aufgabe 12.3** (Comprehensions; Datei: `comprehensions.py`; Punkte: 3+3 )

In dieser Aufgabe sollen Ihre Funktionsdefinitionen außer einer `return`-Anweisung keine weiteren Zeilen enthalten, oder wie im Hinweis als Variable mit Funktionswert definiert werden.

- (a) Eine Liste `xs` ist sortiert, wenn für alle Listen-Indexe `i` und `j` gilt, dass wenn `i <= j` dann `xs[i] <= xs[j]`.

Schreiben Sie eine Funktion `is_sorted`, welche nach diesem Verfahren prüft ob eine Liste sortiert ist. Verwenden Sie dafür eine Kombination aus der `all`-Funktion und einer Generator-Comprehension.

Hinweis: Die `all`-Funktion verknüpft alle Elemente eines `bool`-Iterable mit der `and` Funktion, z.B. `all([x, y, z]) == (x and y and z)`.

- (b) Schreiben Sie eine Funktion `bits_to_int` die eine Liste von Bits als Argument nimmt und die zugehörige positive ganze Zahl zurückgibt.

Die Bits sind im Big-Endian Format, d.h. das erste Bit hat wie gewohnt den größten Exponenten. Beispiel:

```
assert bits_to_int([1,0,1,0]) == 10
```

Verwenden Sie dafür eine Kombination aus der `sum`-Funktion und einer Generator-Comprehension.

**Aufgabe 12.4** (Magische Dekoratoren; Datei: `decorators.py`; Punkte: 6 )

Die Fibonacci-Folge kann rekursiv über folgende Funktion beschrieben werden:

```
def fib(n: int) -> int:
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Auch wenn diese Funktion auf den ersten Blick eher unschuldig aussieht, hat sie es doch ganz schön in sich: um `fib(n)` zu berechnen, muss ungefähr  $2^n$  mal die `fib`-Funktion ausgewertet werden. Die Ausführungszeit von `fib` wächst also exponentiell mit der Größe von `n`. Auf modernen Computern ist die Ausführung für `n < 20` blitzschnell, für `n = 30` dauert es bereits Sekunden und für `n > 40` eine Ewigkeit.

Die Fibonacci-Funktion kann jedoch auch sehr viel effizienter implementiert werden. Der Trick basiert dabei auf der Beobachtung, dass viele der rekursiven Aufrufe doppelt ausgeführt werden. In Abbildung 1 sieht man die rekursiven Funktionsaufrufe die für `fib(5)` nötig sind als Baum visualisiert. Die Knoten von doppelten Funktionsaufrufe sind in grauem Text hervorgehoben. Würden wir uns z.B. das Ergebnis von `fib(3)` im linken Teilbaum merken, dann könnten wir es auf der rechten Seite

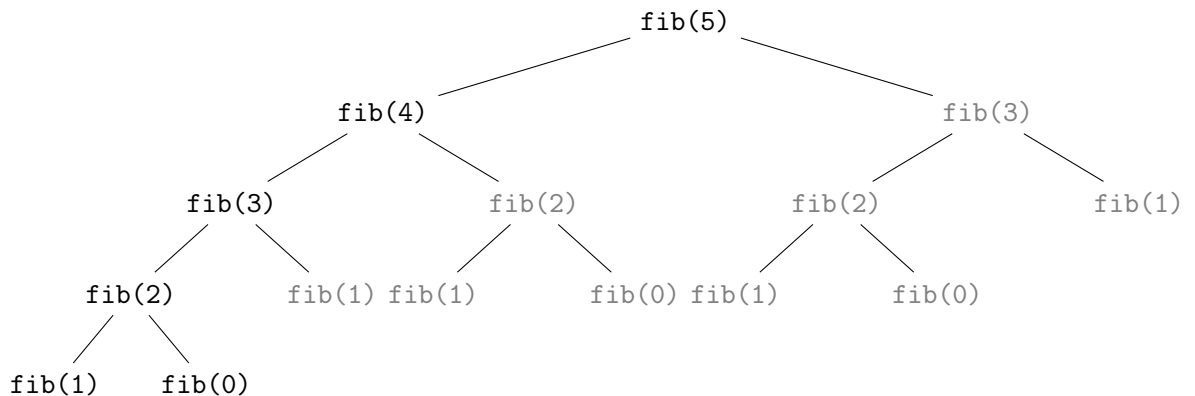


Abbildung 1: Rekursive Funktionsaufrufe für `fib(5)`

einfach nachschlagen, und der gesamte rechte Teilbaum von `fib(3)` würde wegfallen. Macht man dies für alle `n`, so fallen alle grauen Knoten weg, und es bleibt (fast) eine Linie von `n` Knoten übrig - wir können `fib(n)` also mit nur `n` rekursiven Aufrufen berechnen.

Der folgende Code implementiert solch eine optimierte Fibonacci-Funktion. Hierbei wird ein Dictionary `cache` verwendet, in welchem wir uns die Argumente und Rückgabewerte der bisherigen Funktionsaufrufe merken.

```

def fib_fast(n: int) -> int:
    return fib_fast_cache(n, dict())

def fib_fast_cache(n: int, cache: dict[int,int]) -> int:
    # If we already computed fib(n), then return the previously computed result.
    if n in cache:
        return cache[n]

    # Otherwise we compute the result,
    result = None
    if n == 0:
        result = 0
    elif n == 1:
        result = 1
    else:
        result = fib_fast_cache(n-1, cache) + fib_fast_cache(n-2, cache)

    # put the result in the cache - in case we need it again later,
    cache[n] = result

    # and return the result.
    return result

```

Ihre Aufgabe ist es nun einen Decorator `cached` zu implementieren, welcher es erlaubt den Code von `fib` hinzuschreiben, aber die optimierte Implementierung von `fib_fast` zu erhalten:

```
@cached
def fib_fast_and_simple(n: int) -> int:
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_fast_and_simple(n-1) + fib_fast_and_simple(n-2)
```

`cached(f)` soll also beliebige einstellige Funktionen `f` so dekorieren, dass ihre Funktionsaufrufe in einem Dictionary zwischengespeichert und bei Bedarf wieder abgerufen werden.

Verwenden Sie wie in der Vorlesung die `time.time()`-Funktion, um die Ausführungszeiten der Funktionsaufrufe zu vergleichen. Auf meinem Rechner, benötigt der Aufruf `fib(32)` in etwa 0.83 Sekunden um den Wert 2178309 zu berechnen, wohingegen `fib_fast` und `fib_fast_and_simple` lediglich 0.00011 Sekunden benötigen. Je größer `n` ist desto drastischer wird der Unterschied.

*Hinweis:* Die `wrapper`-Funktion des Dekorators muss sich ein Dictionary merken, welches mehrere Funktionsaufrufe überlebt. Dies erreicht man durch das Einfangen einer Variable die man außerhalb des `wrappers` definiert (variable capture).

*Hinweis:* Bei einer rekursiven Funktion wirkt sich ein Decorator auch auf die rekursiven Aufrufe auf.

### **Aufgabe 12.5** (Erfahrungen; Datei: `erfahrungen.txt`; 2 Punkte)

Notieren Sie hier Ihre Erfahrungen mit diesem Übungsblatt (benötigter Zeitaufwand, Probleme, Bezug zur Vorlesung, Interessantes, etc.).

Schreiben Sie den groben Zeitaufwand, den Sie für das *gesamte* Blatt benötigt haben, bitte wie folgt in die erste Zeile:

Zeitaufwand: 3.5h

[... Freitext, wie bisher ...]

Wenn sich genug Leute daran halten, dann können wir ein Pythonskript schreiben, welches automatisiert die durchschnittliche Bearbeitungszeit berechnet ;-)