Informatik I: Einführung in die Programmierung

Prof. Dr. Peter Thiemann
Hannes Saffrich, Michael Uhl
Wintersemester 2022

Universität Freiburg
Institut für Informatik

Übungsblatt 12

Abgabe: Montag, 23.01.2023, 9:00 Uhr morgens

Hinweis: Es gelten die selben Regeln wie bisher, diese können in Blatt 10 eingesehen werden.

Hinweis: Eine Generator-Funktion, die einen Iterator von ganzen Zahlen als Argument nimmt und Strings generiert, hat folgende Typsignatur:

```
from typing import Iterator

def ints_to_strs(int_gen: Iterator[int]) -> Iterator[str]:
    for i in int_gen:
        yield str(i)
```

Hinweis: In diesem Blatt sollen Sie bestimmte Generatorfunktionen implementieren. Diese sollen dabei das folgende Kriterium erfüllen: Um das nächste Element zu generieren, dürfen nur die Berechnungen durchgeführt werden, die dafür unbedingt notwendig sind. Insbesondere dürfen die Berechnungen nicht durchgeführt werden, welche die darauffolgenden Elemente erzeugen.[1] **Nur** für die Ausgabe des Ergebnisses dürfen Sie `list` auf einen Generator anwenden. Das Modul `itertools` dürfen Sie **nicht** verwenden.

[1] Der Sinn eines Generators besteht ja gerade darin, die Elemente erst bei Bedarf zu generieren ("lazy evaluation"). Dadurch kann in bestimmten Fällen Speicherbedarf und Ausführungszeit gespart werden. Andernfalls könnten wir auch gleich eine normale Funktion schreiben, die eine Liste aller zu generierenden Elemente zurückgibt (zumindest wenn der Generator endlich ist).

Aufgabe 12.1 (Generatoren; Datei: `generators.py`; Punkte: 2+1+1+1+1)

In der Mathematik interessiert man sich oftmals für Nullstellen verschiedener Funktionen. Im Folgenden möchten wir mit dem Newton-Verfahren Nullstellen approximieren.

(a) Implementieren Sie die Generatorfunktion `newton`, welche eine einstellige Funktion `f`, welche einen `float` nimmt sowie zurückgibt und einen Startwert `x` entgegennimmt. Sie soll sich wie folgt verhalten:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

```
>>> generator = newton(lambda x: 2 * x + 1, 3)
>>> for i in range(3):
...     print(next(generator))
...
-0.4999999999770708
-0.5
-0.5
```

Sie dürfen hierfür die Funktion `differentiate` aus der Musterlösung des letzten Übungsblatts verwenden.

(b) Implementieren Sie die Generatorfunktion `generate_target`. Diese nimmt als Argumente ein iterierbares Objekt `iterable`, das Zahlen liefert, eine Funktion `f`, welche wie oben sowohl einen `float` nimmt als auch zurückgibt, und einen Wert `target (float)`. Sie liefert einen Generator, der die Werte von `iterable` generiert, bis ein Wert generiert wurde, für den `abs(f(x)) < target` gilt. Wählen Sie `target` also sinnvoll klein, z.B. `1e-12`. Mathematisch entspricht `target` einer sogenannten ε-Umgebung.

Beispiel:

```
>>> f = lambda x: 2 * x + 1
>>> generator = newton(f, 3)
>>> list(generate_target(generator, f, 1e-12))
[-0.4999999999770708, -0.5]
```

(c) Implementieren Sie die Generatorfunktion `arithmetic_mean`. Diese nimmt ein iterierbares Objekt `iterable`, das Zahlen liefert, als Argument und liefert einen Generator, der die laufenden arithmetischen Mittelwerte der Werte von `iterable` generiert (also: die erste Zahl, Mittelwert erste und zweite Zahl, Mittelwert der ersten drei Zahlen usw). Insbesondere soll der Generator nur konstanten Platz verbrauchen, das heißt, Sie dürfen die bisher generierten Werte nicht in einer Liste speichern.

```
>>> list(arithmetic_mean(iter(range(0, 21, 4))))
[0.0, 2.0, 4.0, 6.0, 8.0, 10.0]
```

(d) Schreiben Sie eine Funktion `map13`, die unter Verwendung der eingebauten Funktion `map` aus einem iterierbaren Objekt die von diesem generierten `int`s jeweils modulo 13 wieder als iterierbares Objekt liefert.

Beispiel:

```
>>> input_iterator = iter(range(0, 26, 5))
>>> assert list(map13(input_iterator)) == [0, 5, 10, 2, 7, 12]
```

(e) Schreiben Sie eine Funktion `filter57`, die unter Verwendung der eingebauten Funktion `filter` aus einem iterierbaren Objekt ein iterierbares Objekt erzeugt, das nur die Werte des Eingabeobjekts liefert, welche durch 5 oder 7 teilbar sind.

Beispiel:

```
>>> input_iterator = iter(range(20))
>>> assert list(filter57(input_iterator)) == [0, 5, 7, 10, 14, 15]
```

Aufgabe 12.2 (Magische Dekoratoren; Datei: `decorators.py`; Punkte: 7)

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.

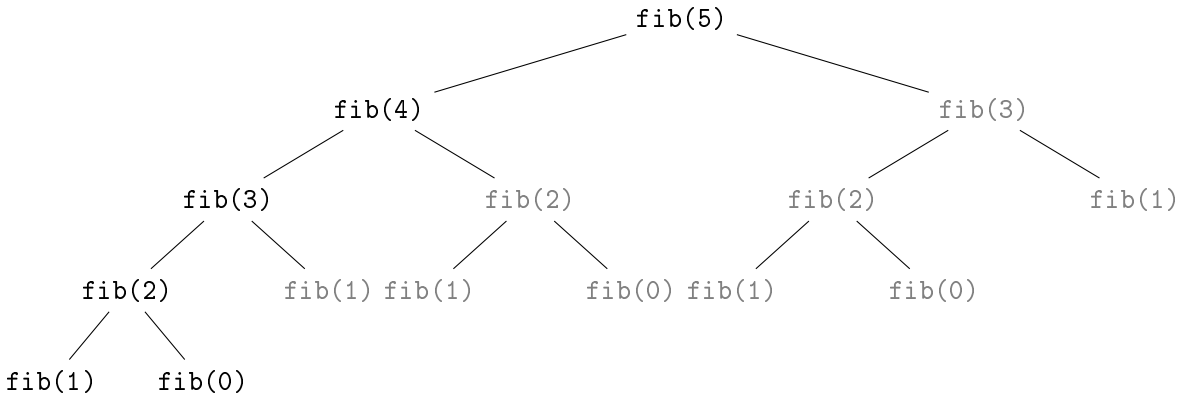

Abbildung 1: Rekursive Funktionsaufrufe für `fib(5)`

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 Funktionsaufrufen 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 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 `wrapper`s definiert (variable capture).

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

Aufgabe 12.3 (Tic Tac Toe; Datei `tictactoe.py` **(schwer)**; 1 + 3 + 1 Punkte)

In dieser Aufgabe sollen Sie mittels Backtracking und Minimax-Algorithmus einen optimalen Zug für ein gegebenes Tic-Tac-Toe-Spielfeld[2] berechnen. Es gibt ein Spielfeld `Board` (`dict[Pos, str]`) und darauf eine Position `Pos` (`tuple[int, int]`), welche von `(0, 0)` bis `(2, 2)` geht. Ein Dateigerüst wird Ihnen bereitgestellt.

Wichtig: Verändern Sie zu keinem Zeitpunkt die Inhalte übergebener Objekte. Benutzen Sie beispielsweise die `dict`-Methode `copy()`, um eine interne Arbeitskopie anzulegen.

Gehen Sie nun wie folgt vor:

(a) Erstellen Sie eine Funktion `calculate_score`, welche ein `Board` und eine aktuelle Rekursionstiefe `depth` entgegennimmt und einen ganzzahligen Punktewert zurückgibt. Ein Spiel gilt als gewonnen, wenn eine Zeile, Spalte oder Diagonale nur `X` enthält. Als verloren gilt ein Spiel, falls eine Zeile, Spalte oder Diagonale nur `O` enthält. Ein gewonnenes Spiel liefert `10 - depth` Punkte, ein verlorenes `-10 + depth` Punkte. Falls es ein Unentschieden gibt oder weder Sieg noch Niederlage festgestellt werden können, beträgt der Punktestand 0.

(b) Nun erstellen Sie eine rekursive Funktion `minimax`, welche ein Board `board`, eine aktuelle Rekursionstiefe `depth` und einen Boolean `max_mode` entgegennimmt. `max_mode` soll den aktuellen Modus, ob nun maximiert (Spieler `X`) oder minimiert (Spieler `O`) werden soll, bestimmen. Die Funktion gibt einen ganzzahligen Wert zurück und zwar:

- bei Sieg oder Niederlage `calculate_score`,
- bei keiner weiteren Zugmöglichkeit 0.

Tritt keiner der beiden Fälle ein, so soll für jede verbleibende Zugmöglichkeit rekursiv die Punktzahl herausgefunden und zurückgegeben werden, welche bei optimalem Spiel vom aktuellen Spielzustand aus erreicht werden kann. Für Spieler `X` ist dies das Maximum der Punkte aus den verbleibenden Zügen; für Spieler `O` jedoch das Minimum. Beim rekursiven Aufruf wird jeweils ein möglicher Zug ausgeführt, die Rekursionstiefe erhöht und es wechselt jeweils der Spieler.

(c) Erstellen Sie zuletzt eine Funktion `return_best_move`, welche ein Board `board` entgegennimmt und eine Spielposition `Pos` zurückgibt, welche dem optimalen Spielzug entspricht. Verwenden Sie hierzu die bereits erstellte `minimax`-Funktion.

Beispiel für das gegebene Spiel aus `game.txt`:

```
>>> board = read_board_from_file("game.txt")
>>> return_best_move(board)
(2, 1)
```

[2] https://de.wikipedia.org/wiki/Tic-Tac-Toe

Aufgabe 12.4 (Erfahrungen; 2 Punkte; Datei: `NOTES.md`)

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

Editieren Sie hierzu die Datei `NOTES.md` im Abgabeordner dieses Übungsblattes auf unserer Webplatform. Halten Sie sich an das dort vorgegebene Format, da wir den Zeitbedarf mit einem Python-Skript automatisch statistisch auswerten. Die Zeitangabe `7.5 h` steht dabei für 7 Stunden 30 Minuten.