

# Semesterarbeit Teil 1a

AWD FS 2018

Simon Egli [simon.egli@students.ffhs.ch]

# 1 Inhaltsverzeichnis

---

2	Abbildungsverzeichnis .....	3
3	Vorbereitung .....	4
4	<b>n</b> -te Fibonacci-Zahl bestimmen.....	5
5	Funktionsaufrufe berechnen .....	6
6	Vergleich Funktionsaufrufe und Fibonacci-Zahl.....	7
7	Messung mittels <code>time()</code> .....	8
8	Effiziente Berechnung der <b>n</b> -ten Fibonacci-Zahl.....	10
8.1	Definition des Bildungsgesetzes.....	10
8.1.1	Prüfen der Lösungen.....	10
8.1.2	Weiterentwicklung der Lösung.....	11
8.2	Implementation .....	12
8.2.1	Floating Point Arithmetic - Problem und Lösung.....	13

## 2 Abbildungsverzeichnis

---

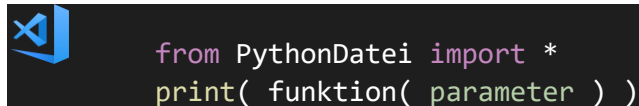
Abbildung 1: Schematische Darstellung der gesuchten Rekursion, für $n = 5$ .....	5
Abbildung 2: Laufzeitanalyse der rekursiven Implementation <code>fib(n)</code> .....	9
Abbildung 3: Laufzeitanalyse der Implementation <code>fib_formula(n)</code> .....	12
Abbildung 4: Laufzeitanalyse der Implementation <code>fib_formula(n)</code> vs. <code>fib_iterative(n)</code> .....	13

### 3 Vorbereitung

---

Dieses Dokument beinhaltet die Lösung der Semesterarbeit Teil 1a. Das gestellte Problem wird nach dem Ansatz «Think first, then act» gelöst.

Erst nach der theoretischen Konzeption sollen Python Funktionen implementiert werden. Die so geschaffenen Quellcode-Referenzen werden wie folgt dargestellt:



```
from PythonDatei import *  
print( funktion( parameter ) )
```

Unter Berücksichtigung des Dateipfads, können diese kopiert und eigenständig ausgeführt werden.

Als Arbeitsumgebung (IDE) wird Visual Studio Code verwendet:

<https://code.visualstudio.com/>

Um Python entwickeln zu können, wird die von Microsoft zur Verfügung gestellte Extension verwendet:

<https://marketplace.visualstudio.com/items?itemName=ms-python.python>

Zudem wird der Funktionsumfang der IDE mit der Extension Code Runner erweitert:

<https://marketplace.visualstudio.com/items?itemName=formulahendry.code-runner>

Wichtige Erkenntnisse und Hinweise werden dargestellt als:



Dies ist ein wichtiger Hinweis.

Der mit dieser Arbeit abgegebene Quellcode entstand aus dem Anspruch heraus erste Erfahrungen in der Python-Programmierung zu sammeln und mathematische Fertigkeiten zu schulen.

Der Quellcode entspricht nicht den in der Softwareentwicklung üblichen Qualitätsansprüchen.

Das heisst, es werden keine automatischen Tests abgegeben, die die Korrektheit der Implementation verifizieren. Auch wurde auf eine ausgiebige Fehlerbehandlung verzichtet.

Das heisst: Der hier abgegebene Quellcode wurde vom Autor manuell unter Verwendung «sinnvoller» Eingabeparameter getestet. Sinnvoll heisst am Beispiel der geforderten Funktion **fib(n)**, dass für  $n$  gilt:  $n \in \mathbb{N} \mid n \geq 0 \wedge n < 40$ .



## 5 Funktionsaufrufe berechnen

---

Bei der Betrachtung der schematischen Darstellung in Abbildung 1 fällt auf, dass die Anzahl der Endpunkte (Knoten, von denen kein Pfeil wegführt), stets der «nächsten» Fibonacci-Zahl  $f_{n+1}$  zu entsprechen scheint.

Des Weiteren scheint die Anzahl der Knoten, von denen mindestens ein Pfeil wegführt, stets um eins kleiner zu sein als die Anzahl der Endpunkte.

Somit kann die Anzahl der Funktionsaufrufe mit einer Funktion  $c$  (für count) ermittelt werden, welche definiert ist als:

$$c_n = f_{n+1} + f_{n+1} - 1 = 2f_{n+1} - 1$$

Im Folgenden soll versucht werden, dies mittels Induktion zu beweisen.

Induktionsanfang:

Für  $n = 0$  gilt  $c_0 = 2f_{0+1} - 1 = 1$

Für  $n = 1$  gilt  $c_1 = 2f_{1+1} - 1 = 1$

Induktionsschritt:

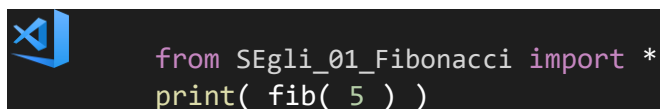
Schliessen von  $n = k$  auf  $n = k + 1$

Induktionsannahme:  $c_k = 2f_{k+1} - 1$

Zu zeigen:  $c_{k+1} = 2f_{k+1+1} - 1$

$$\begin{aligned} c_{k+1} &= c_k + c_{k-1} + 1 \\ &= f_{k+1} + f_{k+1} + c_{k-1} \\ &= f_{k+1} + f_{k+1} + f_k + f_k - 1 \\ &= f_k + f_{k+1} + f_k + f_{k+1} - 1 \\ &= f_{k+2} + f_{k+2} - 1 \\ &= 2f_{k+2} - 1 \end{aligned}$$

Die gesuchte Python Funktion `fib_rec_count(n)` kann wie folgt aufgerufen werden.



```
from SEgli_01_Fibonacci import *
print( fib( 5 ) )
```

## 6 Vergleich Funktionsaufrufe und Fibonacci-Zahl

---

Die Funktionsaufrufe schematisch dargestellt ergeben einen binären Baum<sup>1</sup>. Als Spezialfall der AVL-Bäume<sup>2</sup>, weist der Fibonacci-Baum zu gegebener Höhe die kleinstmögliche Anzahl Knoten aus.

Genau gleich wie die Fibonacci-Folge, ist der Fibonacci-Baum rekursiv definiert, was mittels Induktion bewiesen wurde.



Nach der Erarbeitung des Kapitels 8 wird klar, dass das Bildungsgesetz des Fibonacci-Baums ebenfalls die Zahl  $\frac{(1+\sqrt{5})}{2}$  als Basis beinhaltet. Somit ist der Fibonacci Baum ebenfalls eine Fibonacci-Folge, wenn auch diese Aussage mathematisch nicht haltbar ist.

---

<sup>1</sup> Definition Binärbaum: Ein Knoten darf höchstens zwei direkte Nachkommen haben.

<sup>2</sup> Definition AVL-Baum: Die Höhe des linken und rechten Teilbaums differieren maximal um 1.

## 7 Messung mittels `time()`

---

Im Sinne der Aufgabenstellung, wird eine Laufzeitmessung der rekursiven Implementation mittels `time()` durchgeführt.

An dieser Stelle muss darauf hingewiesen werden, dass das Modul `timeit` für diese Aufgabe aus drei Gründen besser geeignet wäre:

- Die Testdurchläufe werden wiederholt, um die Einflüsse anderer Tasks die auf dem Computer laufen zu minimieren (z.B Disk flushing oder OS scheduling)
- Der Garbage Collector wird ausgeschaltet um zu verhindern, dass nicht mehr verwendete Objekte zu einem ungünstigen Zeitpunkt abgeräumt werden
- Das Modul wählt den am besten geeigneten Timer des Betriebssystems

Um trotzdem einigermaßen glaubhafte Ergebnisse zu erhalten, wiederholt auch die hier vorgestellte Implementation jede Messung zehn Mal und gibt anschliessend den besten Wert zurück. Dies aus der Annahme heraus, dass der Beste Wert am wenigsten durch «äussere» Einflüsse gestört wurde.

Zudem wird `time.process_time()` verwendet, da diese Funktion einen vom Prozessor abgeleiteten Wert liefert, welcher nur aktualisiert wird, wenn der Python-Prozess auf dem Prozessor ausgeführt wird:



```
from SEgli_01_Fibonacci import *  
print( get_best_recursion_runtime(5) )
```

Um die Benutzerfreundlichkeit zu erhöhen und trotzdem innerhalb des Rahmens der Aufgabenstellung zu bleiben, wurde in einer eigenen Datei eine weitere Funktion `get_best_recursion_runtime(n)` implementiert, die ausgehend von  $n = 0$  inkrementell bis  $n = nMax$  eine Laufzeitanalyse durchführt. Für  $n$  gilt  $n \in \mathbb{N} \mid n \geq 0 \wedge n < nMax$ :



```
from SEgli_01_Measurement import *  
show_measurement_recursion(31)
```



Die gesammelten Daten werden auf der Konsole, sowie als Grafik unter Verwendung des Moduls `matplotlib` ausgegeben:

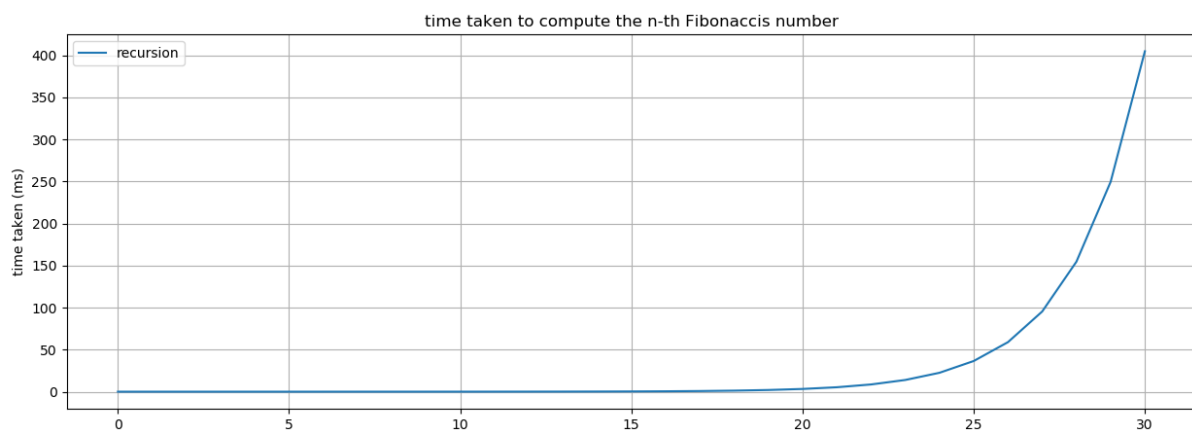


Abbildung 2: Laufzeitanalyse der rekursiven Implementation  $\text{fib}(n)$

Allein die Betrachtung der Laufzeitanalyse für  $n \leq 30$  zeigt, wie drastisch sich die in Kapitel 5 errechneten Funktionsaufrufe auf die Laufzeit auswirken.

## 8 Effiziente Berechnung der $n$ -ten Fibonacci-Zahl

Wird die Fibonacci-Folge betrachtet, so fällt auf, dass diese sehr schnell anwächst. Genauer gesagt fällt auf, dass diese monoton wächst. Dass die Folge gar exponentiell wächst, ist einfach zu erkennen, denn durch die Monotonie ergibt sich:

1.  $f_n = f_{n-1} + f_{n-2} \leq 2f_{n-1}$  und folglich  $f_n \leq 2^n$
2.  $f_n = f_{n-1} + f_{n-2} \geq 2f_{n-2}$  und folglich  $f_n \geq 2^{n-1}$

Dies lässt vermuten, dass eine reelle Zahl existieren muss, welche als Basis zur effizienten Berechnung der Folge herangezogen werden kann.

Das Problem soll vereinfacht werden, in dem die Anfangswerte der Folge ignoriert werden. Aus diesem Grund soll das Bildungsgesetz der neuen Folge in der Funktion  $g_n$  statt  $f_n$  definiert werden.

### 8.1 Definition des Bildungsgesetzes

Idee:  $g_n = a^n \mid a > 0 \wedge a \in \mathbb{R}$

Prüfen, dass gilt:  $g_n = g_{n-2} + g_{n-1}$  also  $a^n = a^{n-2} + a^{n-1}$ :

$a^n = a^{n-2} + a^{n-1}$	faktorisieren
$a^n = (a + 1)a^{n-2}$	: $a^{n-2}$
$\frac{a^n}{a^{n-2}} = a + 1$	ausrechnen
$a^{n-n-2} = a + 1$	Potenz vereinfachen
$a^2 = a + 1$	In Normalform bringen
$0 = -a^2 + a + 1$	Koeffizienten (a=-1, b=1, c=1) einsetzen in Mitternachtsformel

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \Rightarrow \frac{-(1) \pm \sqrt{1^2 - 4 \cdot (-1) \cdot 1}}{2 \cdot (-1)} = \frac{-(1) \pm \sqrt{1 - -4}}{-2} = \frac{-(1) \pm \sqrt{5}}{-2} = \frac{1 \pm \sqrt{5}}{2}$$

$$\mathbb{L} = \left\{ \frac{1 - \sqrt{5}}{2}, \frac{1 + \sqrt{5}}{2} \right\}$$

#### 8.1.1 Prüfen der Lösungen

Sei  $a = \frac{1 - \sqrt{5}}{2}$  und  $b = \frac{1 + \sqrt{5}}{2}$

Dann muss gelten:

$$a^n = a^{n-2} + a^{n-1} \Rightarrow \left(\frac{1 - \sqrt{5}}{2}\right)^n = \left(\frac{1 - \sqrt{5}}{2}\right)^{n-2} + \left(\frac{1 - \sqrt{5}}{2}\right)^{n-1} \quad \text{Diese Aussage ist wahr!}$$

$$b^n = b^{n-2} + b^{n-1} \Rightarrow \left(\frac{1 + \sqrt{5}}{2}\right)^n = \left(\frac{1 + \sqrt{5}}{2}\right)^{n-2} + \left(\frac{1 + \sqrt{5}}{2}\right)^{n-1} \quad \text{Diese Aussage ist wahr!}$$

## 8.1.2 Weiterentwicklung der Lösung

Im Anschluss muss versucht werden, die Anfangswerte der Folge zu bilden:

Sei erneut  $a = \frac{1-\sqrt{5}}{2}$  und  $b = \frac{1+\sqrt{5}}{2}$

$n =$	0	1	2	...
gegeben: $a^n$	1	$a$	$a^2$	...
gegeben: $b^n$	1	$b$	$b^2$	...
gesucht: $f_n$	0	1	1	...

Die erste Datenspalte ( $n = 0$ ), lässt sich einfach durch Subtraktion bilden ( $1 - 1 = 0$ ).

Also  $g_n = a_n - b_n$ . In diesem Fall ist  $g_0 = 0$ .

Für die zweite Datenspalte ( $n = 1$ ):  $g_1 = a^1 - b^1 = \frac{1-\sqrt{5}}{2} - \frac{1+\sqrt{5}}{2} = -\sqrt{5}$ , was nicht gewünscht ist.

Allerdings kann die Formel durch  $-\sqrt{5}$  geteilt werden, ohne dabei  $g_0$  zu verlieren:

$$g_0 = a^0 - b^0 = \frac{\left(\frac{1}{2}(1-\sqrt{5})\right)^0}{-\sqrt{5}} - \frac{\left(\frac{1}{2}(1+\sqrt{5})\right)^0}{-\sqrt{5}} = 0$$

$$g_1 = a^1 - b^1 = \frac{\left(\frac{1}{2}(1-\sqrt{5})\right)^1}{-\sqrt{5}} - \frac{\left(\frac{1}{2}(1+\sqrt{5})\right)^1}{-\sqrt{5}} = 1$$

Nun kann  $g_n$  in  $f_n$  eingesetzt werden und das Bildungsgesetz vereinfacht werden:

$$f_n = f_{n-1} + f_{n-2} = \frac{\left(\frac{1}{2}(1-\sqrt{5})\right)^{n-1}}{-\sqrt{5}} - \frac{\left(\frac{1}{2}(1+\sqrt{5})\right)^{n-1}}{-\sqrt{5}} + \frac{\left(\frac{1}{2}(1-\sqrt{5})\right)^{n-2}}{-\sqrt{5}} - \frac{\left(\frac{1}{2}(1+\sqrt{5})\right)^{n-2}}{-\sqrt{5}}$$

$$f_n = \frac{\left(\frac{1}{2}(1-\sqrt{5})\right)^n}{-\sqrt{5}} - \frac{\left(\frac{1}{2}(1+\sqrt{5})\right)^n}{-\sqrt{5}} \quad \left| \quad \text{Vereinfachen} \right.$$

$$f_n = \frac{\left(\frac{1}{2}(1+\sqrt{5})\right)^n}{\sqrt{5}} - \frac{\left(\frac{1}{2}(1-\sqrt{5})\right)^n}{\sqrt{5}} \quad \left| \quad \text{Vereinfachen} \right.$$

$$f_n = \frac{1}{\sqrt{5}} \left[ \left( \frac{1+\sqrt{5}}{2} \right)^n - \left( \frac{1-\sqrt{5}}{2} \right)^n \right]$$


## 8.2 Implementation

Bei näherer Betrachtung der Formel fällt auf, dass der zweite Summand  $\frac{1-\sqrt{5}}{2}$  vom Betrag her kleiner ist als 1. Je grösser  $n$ , desto näher bei null wird das Resultat liegen ( $\lim_{n \rightarrow \infty} \left(\frac{1-\sqrt{5}}{2}\right)^n = 0$ ).

Da immer  $\left| \frac{1}{\sqrt{5}} \cdot \left(\frac{1-\sqrt{5}}{2}\right)^n \right| < \frac{1}{2}$  (da  $\sqrt{5} > 2$ ), folgt, dass  $f_n$  nur mit  $\frac{1}{\sqrt{5}} \cdot \left(\frac{1+\sqrt{5}}{2}\right)^n$  berechnet werden kann, wenn anschliessend das Resultat gerundet wird:

$$f_n = \left\lfloor \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor$$

Diese Vereinfachung wurde bei der Implementation der Formel in Python berücksichtigt und kann wie folgt getestet werden:

```
 from SEgli_01_Fibonacci import *  
print( fib_formula(5) )
```

Der Erfolg ist messbar. Auch für grössere  $n$ , bleibt die Laufzeit nahezu konstant:

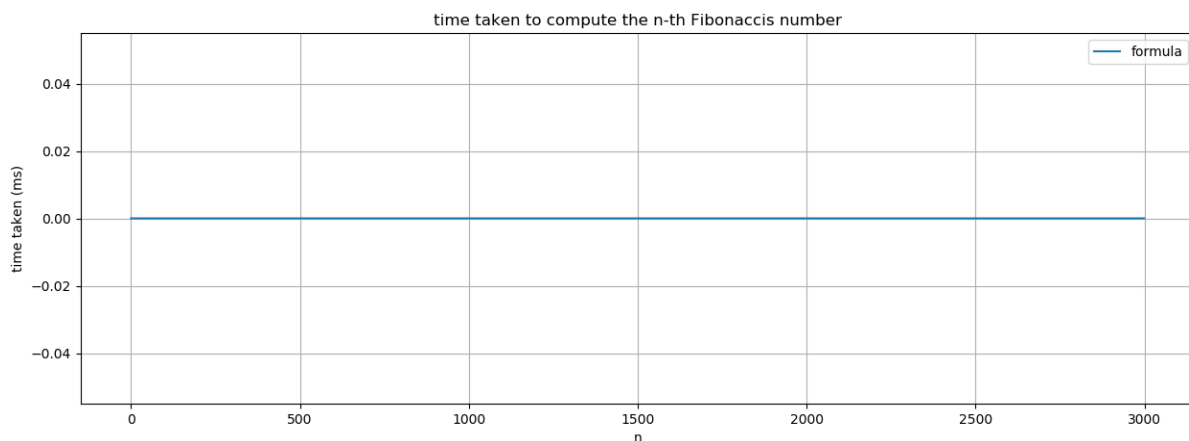


Abbildung 3: Laufzeitanalyse der Implementation `fib_formula(n)`

## 8.2.1 Floating Point Arithmetic - Problem und Lösung

Eine Hürde bei der Implementation stellt die numerische Präzision dar. Genauer gesagt, die Verwendung von  $\sqrt{5}$ . Wird der Standard Fließkommatyp Float verwendet, wird  $\sqrt{5}$  gerundet zu 2.23606797749979. Diese Rundung führt zu Fehlern in der Berechnung der Fibonacci-Folge ab etwa dem 70. Glied.

Eine mögliche und hier verwendete Lösung bietet das Modul **decimal**. Dieses Modul lässt den Entwickler bestimmen, mit wie vielen Nachkommastellen gerechnet werden soll.

Manuelle Tests haben ergeben, dass  $\frac{1}{4}n$  für  $n > 40$  ausreichend ist.

Diese (notwendige) Anpassung der Präzision führt dazu, dass sich mit steigenden  $n$  die Laufzeit der Funktion stärker verschlechtert.

Aus diesem Grund und um die Korrektheit der Berechnung der einzelnen Glieder zu verifizieren, wurde eine möglichst effiziente iterative Funktion **fib\_iterative(n)** implementiert:

```
from SEgli_01_Fibonacci import *  
print( fib_iterative(5) )
```

Diese Funktion wird herangezogen um einen Vergleich der Laufzeit zwischen iterativer und formelbasierter Berechnung durchzuführen:

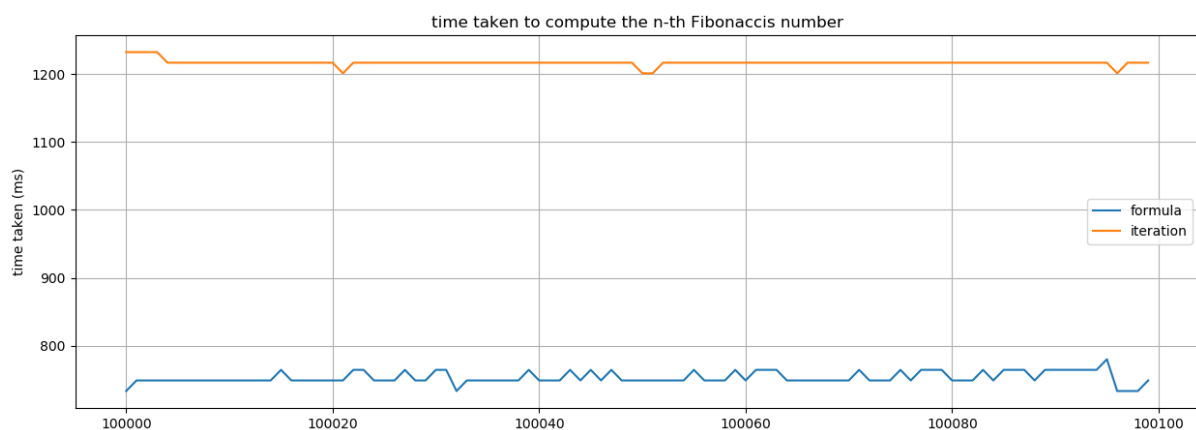


Abbildung 4: Laufzeitanalyse der Implementation *fib\_formula(n)* vs. *fib\_iterative(n)*

Je grösser  $n$ , umso drastischer der Unterschied. Ein «smoke test» unter Verwendung der IDE Visual Studio Code ergab für die iterative Berechnung mit  $n = 1'000'000$  eine Laufzeit von 126.8 Sekunden. Die Formel basierte Lösung lieferte das exakt gleiche Resultat in lediglich 19.6 Sekunden. Wobei auch Messwerte für „mittlere  $n$ “ existieren, die gegenteilige Resultate zeigen.