Semesterarbeit Teil 1a

AWD FS 2018

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

Inhaltsverzeichnis

[2 Abbildungsverzeichnis 3](#_Toc507418481)

[3 Tabellenverzeichnis 4](#_Toc507418482)

[4 Vorbereitung 5](#_Toc507418483)

[5 -te Fibonacci-Zahl bestimmen 6](#_Toc507418484)

[6 Funktionsaufrufe berechnen 7](#_Toc507418485)

[7 Vergleich Funktionsaufrufe und Fibonacci-Zahl 8](#_Toc507418486)

[8 Messung mittels time() 9](#_Toc507418487)

[9 Effiziente Berechnung der -ten Fibonacci-Zahl 11](#_Toc507418488)

[9.1 Definition des Bildungsgesetzes 11](#_Toc507418489)

[9.1.1 Prüfen der Lösungen 11](#_Toc507418490)

[9.1.2 Weiterentwicklung der Lösung 12](#_Toc507418491)

[9.2 Implementation 13](#_Toc507418492)

[9.2.1 Floating Point Arithmetic Problem und Lösung 14](#_Toc507418493)

# Abbildungsverzeichnis

[Abbildung 1: Schematische Darstellung der gesuchten Rekursion, für 6](#_Toc507352141)

[Abbildung 2: Laufzeitanalyse der rekursiven Implementation fib(n) 10](#_Toc507352142)

[Abbildung 3: Laufzeitanalyse der rekursiven Implementation fib\_formula(n) 13](#_Toc507352143)

# Tabellenverzeichnis

**Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.**

# 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 gilt: .

# -te Fibonacci-Zahl bestimmen

Mit Fibonacci-Zahlen sind die Zahlen gemeint.

In diesem Dokument:

Diese Zahlen sind festgelegt durch das Bildungsgesetz: «Jede Zahl wird gebildet durch die Summe der beiden vorhergegangenen Zahlen». Also:

für mit den Anfangswerten und .

Da die Aufgabenstellung in einem weiteren Schritt eine «Analyse der Funktionsaufrufe» fordert, sollte zu Beginn eine Funktion implementiert werden, die die -te Fibonacci-Zahl rekursiv bestimmt.

Die Funktion muss sich also solange selbst Aufrufen, bis der Funktionsparameter die Wertigkeit 1 oder 0 erreicht hat:

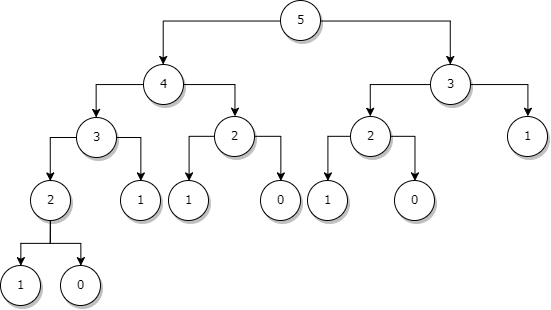


Abbildung : Schematische Darstellung der gesuchten Rekursion, für

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

* from SEgli\_01 import \*  
   print( fib( 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 zu entsprechen scheint.

Dies soll mittels Induktion gezeigt werden:

**Hypothese:** Die Anzahl der Endknoten entspricht der «nächsten» Fibonacci-Zahl , so dass gilt:

**Induktionsanfang :**

**Induktionsschritt, schliessen von auf :**

Induktionsannahme:

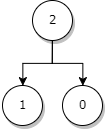
Zu zeigen:

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

Sei die Anzahl Endpunkte und die inneren Knoten (keine Endpunkte), dann sei die zu beweisende Hypothese:

**Induktionsanfang**:

****- Ein Baum mit null inneren Knoten () hat einen Endpunkt () und die Höhe:

- Ein Baum mit einem inneren Knoten () hat zwei Endpunkte ():

**Induktionsschritt:**

Zeige, dass ein Fibonacci-Rekursionsbaum mit inneren Knoten, Endpunkte hat.

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

Die gesuchte Python Funktion fib\_rec\_count(n) kann wie folgt aufgerufen werden.

* from SEgli\_01 import \*  
   print( fib( 5 ) )

Vergleich Funktionsaufrufe und Fibonacci-Zahl

D

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 der am besten geeignete 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 import \*  
   print( get\_best\_recursion\_runtime(5) )

Um die Benutzerfreundlichkeit zu erhöhen und trotzdem innerhalt des Rahmens der Aufgabenstellung zu bleiben, wurde in einer eigenen Datei eine weitere Funktion get\_best\_recursion\_runtime(n) implementiert, die ausgehend von inkrementell bis eine Laufzeitanalyse durchführt. Für gilt :

* 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:

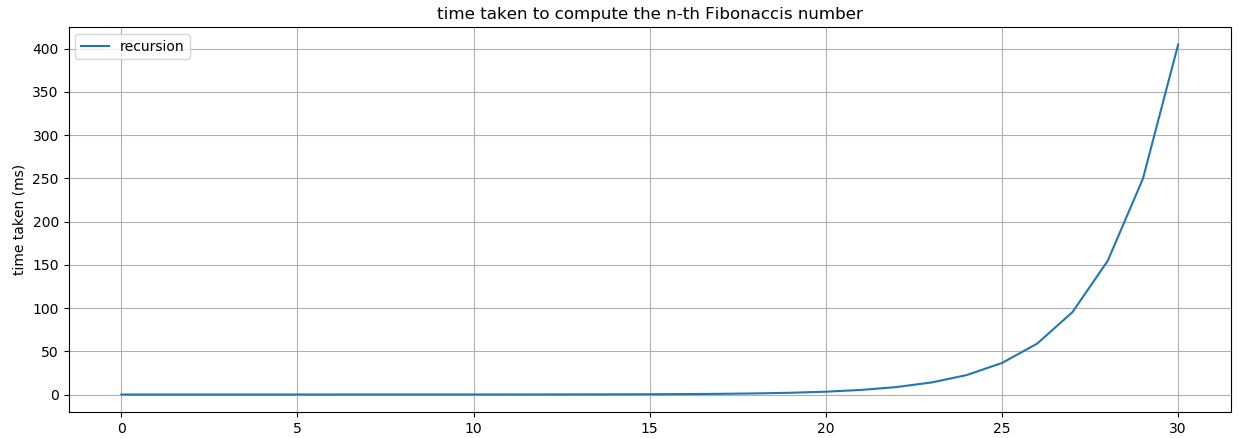


Abbildung : Laufzeitanalyse der rekursiven Implementation fib(n)

Allein die Betrachtung der Laufzeitanalyse für zeigt, wie drastisch sich die in Kapitel 6 errechneten Funktionsaufrufe auf die Laufzeit auswirken.

Effiziente Berechnung der -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 den durch die Monotonie ergibt sich:

1. und folglich
2. und folglich

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 statt definiert werden.

## Definition des Bildungsgesetzes

Idee:

Prüfen, dass gilt: also :

faktorisieren

:

ausrechnen

Potenz vereinfachen

In Normalform bringen

Koeffizienten (a=-1, b=1, c=1) einsetzen in Mitternachtsformel

=

### Prüfen der Lösungen

Sei und

Dann muss gelten:

= + Diese Aussage ist wahr!

= + Diese Aussage ist wahr!

### Weiterentwicklung der Lösung

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

Sei erneut und

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | … |
| gegeben: | 1 |  |  | … |
| gegeben: | 1 |  |  | … |
| gesucht: | 0 | 1 | 1 | … |

Die erste Datenspalte ( = 0), lässt sich einfach durch Subtraktion bilden ().

Also . In diesem Fall ist .

Für die zweite Datenspalte ( = 1): , was nicht gewünscht ist.

Allerdings kann die Formel durch geteilt werden, ohne dabei zu verlieren:

Nun kann in eingesetzt werden und das Bildungsgesetz vereinfacht werden:

Vereinfachen

Vereinfachen

## Implementation

Bei näherer Betrachtung der Formel fällt auf, dass der zweite Summand vom Betrag her kleiner ist als 1. Je grösser , desto näher bei null wird das Resultat liegen ().

Da immer (da ), folgt, dass nur mit berechnet werden kann, wenn anschliessend das Resultat gerundet wird:

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

* from SEgli\_01 import \*  
   print( fib\_formula(5) )

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

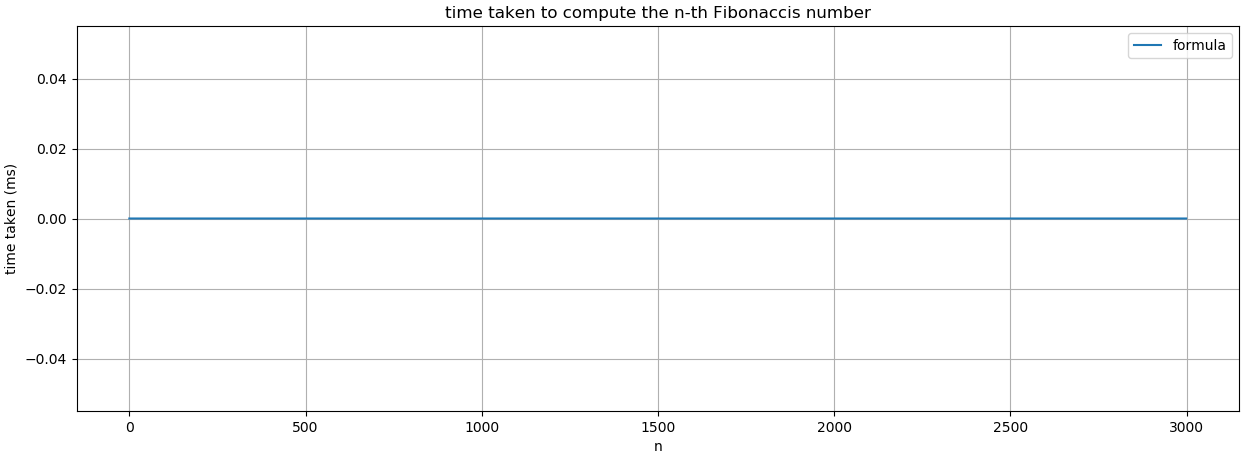


Abbildung : Laufzeitanalyse der Implementation fib\_formula(n)

### Floating Point Arithmetic Problem und Lösung

Eine Hürde bei der Implementation stellt die numerische Präzision dar. Genauer gesagt, die Verwendung von. Wird der Standard Fliesskommadatentyp Float verwendet, wird 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 für ausreichend ist.

Diese (notwendige) Anpassung der Präzision führt dazu, dass sich mit steigenden 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 import \*  
   print( fib\_iterative(5) )

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

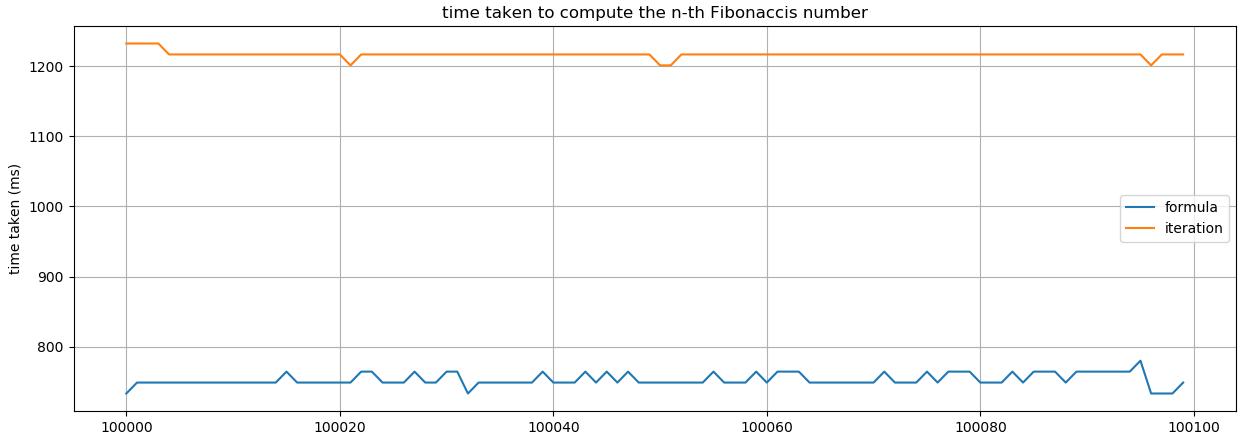


Abbildung : Laufzeitanalyse der Implementation fib\_formula(n) vs. fib\_iterative(n)

Je grösser umso drastischer der Unterschied. Ein «smoke test» unter Verwendung der IDE Visual Studio Code ergab für die iterative Berechnung mit 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 “ existieren, die gegenteilige Resultate zeigen.