Rekursion und dynamische Programmierung

1 Didaktische Einbettung

Ziel: Diese Einheit ist darauf ausgelegt an einem ersten Beispiel die Stärke der dynamischen Programmierung erlebbar zu machen. Anhand der Fibonacci Folge sollen die Schülerinnen und Schüler mit Zeitmessungen sehen wie viel effizienter die dynamische Programmierung nur schon bei der Berechnung einer so einfachen Funktion sein kann.

Einsatzmöglichkeiten: Dieses Kapitel eignet sich nach der Behandlung der Rekursion als Vertiefung des Themas anhand einer klassischen Aufgabe mit mehreren rekursiven Aufrufen.

Man kann die Aufgabe auch als Übergang zu dynamischer Programmierung nutzen, falls man diese mit einer starken Klasse ansieht.

Da alle Lösungen angegeben sind, eignet sich der Text auch zur selbständigen Bearbeitung für schnelle Schülerinnen und Schüler, der ihnen eine Möglichkeit gibt einen Blick auf eine weitere Technik zu werfen, ohne dabei den Rest der Klasse zu verlieren.

Überlegungen zum Aufbau: Die Schülerinnen und Schüler finden anhand einer Sachaufgabe die Fibonacci Folge. Die Einführungsaufgabe zu den Fibonacci Folgen ist bewusst sehr feingliedrig, so dass möglichst alle Schülerinnen und Schüler diese berühmte Folge anhand eines Sachbeispieles selbständig finden können. Ebenfalls wird bewusst, darauf verzichtet das Kaninchenproblem direkt zu nehmen, da es immer wieder Lernende gibt, welche dieses Problem bereits kennen.
Wenn schon genügend Beispiele mit rekursiver Programmierung gemacht wurden, so können sie hier selbst ein Programm dazu schreiben, es kann aber auch auf das angegebene Programm zurückgegriffen und anhand des gegebenen Codes experimentiert werden.

Vorwissen: Im Text wird die Programmierumgebung TigerJyton genutzt. Die Kenntnis von rekursiven Funktionen wird empfohlen, da in dieser Unterrichtseinheit direkt doppelt rekursiv ersetzt wird. Der rekursive Aufruf an sich steht dabei nicht im Mittelpunkt, sondern die Verbesserungsmöglichkeit der Rechenzeit über dynamische Programmierung. Es wird für den wesentlichen Teil kein spezifisches mathematisches Wissen benötigt. Die Aufgabe am Schluss bietet die Möglichkeit einer Limesbetrachtung.

Zeitaufwand: ca. 1-2 Lektionen

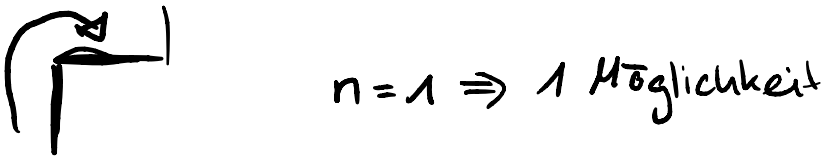
2 Lernaufgabe mit Lösungen

Treppen steigen ...

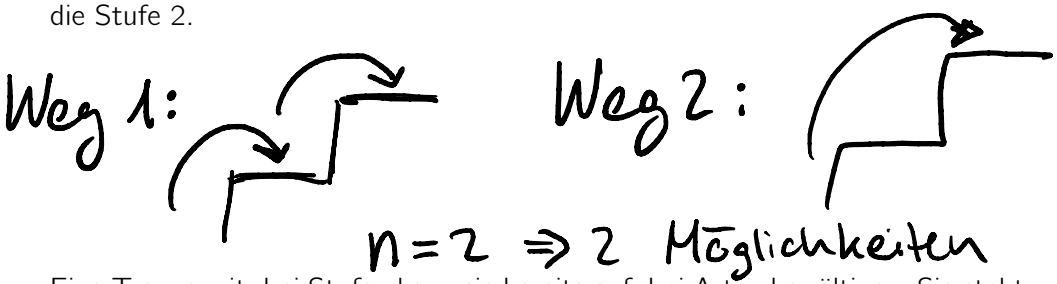
Die Treppe im Schulhaus besteht von Stock zu Stock aus 24 Stufen. Simone fragt sich nun, auf wie viele Arten sie die Treppe erklimmen kann, wenn sie jeweils eine oder zwei Stufen in einem Schritt bewältigen kann.
Bemerkung: Dieses Problem soll nicht mit Kombinatorik gelöst werden, sondern aufbauend auf einfachen Fällen entwickelt werden.

Bevor du dich auf das Problem mit den vielen Stufen stürzt, betrachte zunächst ein einfacheres Problem: Treppen mit wenigen Stufen. Es ergibt sich dann daraus einen Zusammenhang, der recht einfach auf grössere Treppen schliessen lässt.

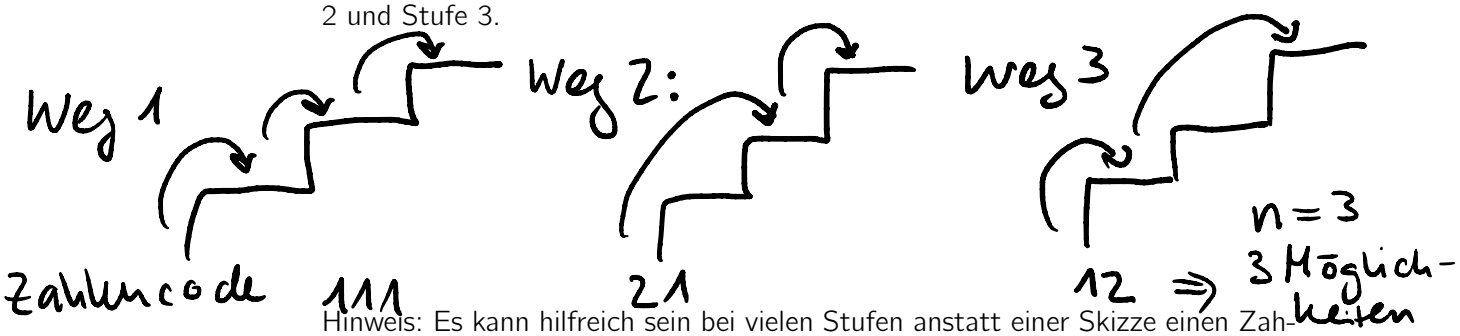
1. Vergewissere dich zum Beispiel mit Hilfe von Skizzen oder eines nützlichen Nummerncodes, dass folgende Aussagen stimmen:
   - Eine Treppe mit einer Stufe kann Simone nur auf eine Variante erklimmen, indem sie auf die eine Stufe tritt.



   - Eine Treppe mit zwei Stufen kann sie auf zwei Arten hochsteigen: Sie steht auf die Stufe 1 und die Stufe 2 oder sie überspringt die Stufe 1 und steht direkt auf die Stufe 2.



   - Eine Treppe mit drei Stufen kann sie bereits auf drei Arten bewältigen: Sie steht auf alle Stufen, sie steht auf Stufe 1 und dann Stufe 3 oder sie steht auf Stufe 2 und Stufe 3.



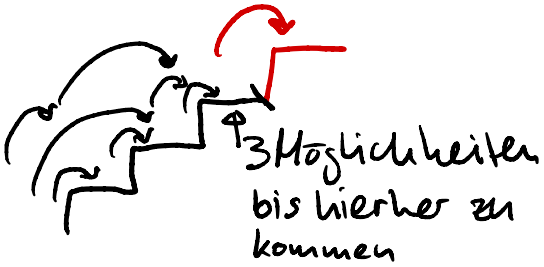
   Hinweis: Es kann hilfreich sein bei vielen Stufen anstatt einer Skizze einen Zahlencode einzuführen. Zum Beispiel 111 bedeutet es wurde 3 mal eine Stufe genommen, 12 bedeutet zuerst wurde eine Stufe genommen, danach 2.

2. Wie viele Möglichkeiten gibt es für Treppen mit 4, 5, 6 und 7 Stufen? Überlege mit Hilfe von Skizzen oder Nummerncodes und trage die Anzahl in die Tabelle ein.

Anzahl Stufen	1	2	3	4	5	6	7
Anzahl Möglichkeiten	1	2	3				

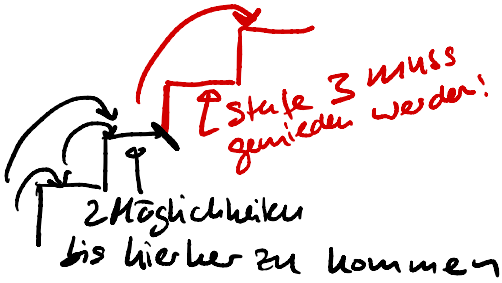
3. Sobald es mehr Stufen werden, ist die Überlegung gar nicht mehr so einfach. Falls du bei der obigen Aufgabe Mühe hattest, so schau dir den folgenden Hinweis an und versuche es nochmal. Wir betrachten nochmal eine Treppe mit 4 Stufen und zwar können wir diese so betrachten als ob es eine

Treppe mit 3 plus 1 Stufen



Wie wir auf Stufe 3 kommen wissen wir, denn es gibt dazu 3 Möglichkeiten. Von Stufe 3 auf 4 kommen wir mit einem zusätzlichen Schritt von einer Stufe. Wir haben hier insgesamt also 3 Möglichkeiten.

oder eine Treppe mit 2 plus 2 Stufen ist.



Wie wir auf Stufe 2 kommen wissen wir, denn es gibt dazu 2 Möglichkeiten. Von Stufe 2 auf 4 kommen wir mit einem zusätzlichen Schritt von 2 Stufen. Wir müssen dabei Stufe 3 meiden, denn diese Möglichkeiten haben wir bereits links gezählt. Wir haben also nun zwei weitere Möglichkeiten.

Total haben wir nun $3+2=5$ Möglichkeiten eine Treppe mit 4 Stufen zu erklimmen.

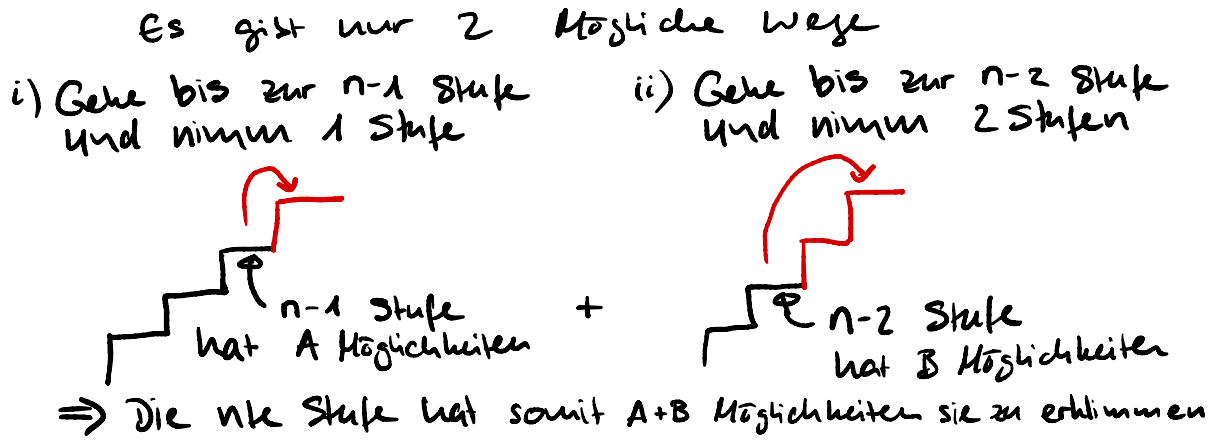
Versuche analog zum Beispiel dir nun anhand deines Wissens zu Treppen mit 4 Stufen zu überlegen wie du eine Treppe mit 5 Stufen erklimmst, danach eine mit 6 und so weiter.

4. Du hast du hoffentlich folgende Tabelle gefunden:

Anzahl Stufen	1	2	3	4	5	6	7	8	9	
Anzahl Möglichkeiten	1	2	3	5	8	13	21			

   Betrachte nun alle Möglichkeiten und suchen einen Zusammenhang zwischen zwischen einer beliebigen Anzahl Treppenstufen 8, 9, 10, beliebig viele...

5. Bestimmt hast du es gemerkt! Die neue Anzahl Möglichkeiten setzt sich immer aus der Summe der beiden vorhergehenden errechnen. Wenn dir das noch nicht vollkommen klar ist, so lies folgende Begründung, ansonsten springe zur nächsten Aufgabe.



6. Diese Zahlenfolge ist nach dem Italienischen Mathematiker Fibonacci benannt. Leonardo von Pisa wurde als Fibonacci „Sohn des Bonaccus – Filius Bonacci“ bekannt. Er lebte im 13 Jahrhundert und schrieb das Buch Liber Abaci - Die Kunst des Rechnens. Auch wenn nun klar ist, wie man die Anzahl Möglichkeiten zu berechnen hat, so sind wir doch noch nicht am Ende. Wie viele Möglichkeiten gibt es nun die 24 Treppenstufen zwischen zwei Stockwerken hochzugehen? Wenn du Zeit zum Rechnen hast, dass holst du ein Blatt Papier und fängst an, oder... erinnerst dich an die rekursive Programmierung und überlässt die Rechenarbeit einem Computer.

   Definiere eine Funktion fibonacci(n): Erinnere dich daran, dass die Funktion einen Basisfall enthalten muss, hier für $n = 0$ und $n = 1$, da auf zwei zurückliegende Werte zugegriffen wird. Der Rekursionsfall beinhaltet auch 2 Werte über welche die Rekursion läuft. Lass sich davon nicht irritieren.

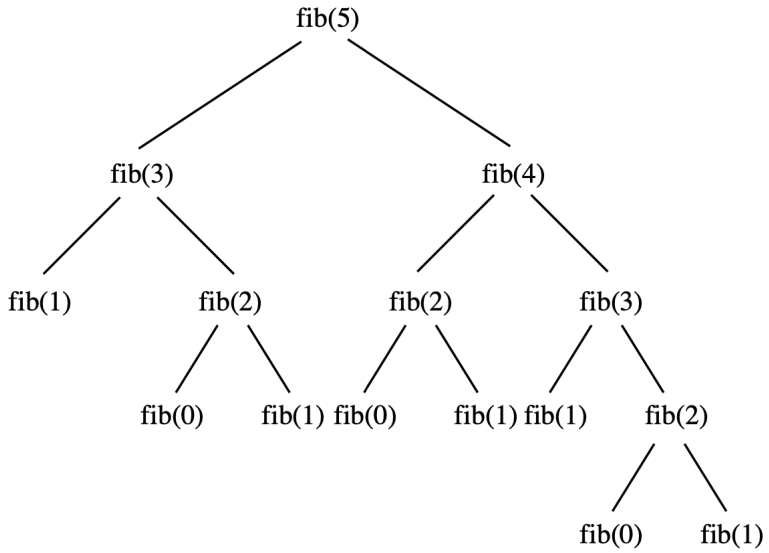
7. Deine Lösung wird vermutlich ähnlich wie die unterhalb aussehen und sollte für 24 Stufen den Wert 46 368 ausgeben.
   Wenn du verschiedne Stufenzahlen ausprobierst merkst du, dass du zum Teil ziemlich lange warten musst oder den Computer sogar zum Absturz bringen kannst. Konsultiere auch den debugger, er hilft dir zu verstehen, was im Hintergrund des Programmes abläuft. Finde einen Wert, so dass du etwa 5 Sekunden lange auf das Resultat warten musst.

```
1 def fibonacci(n):
2     if n <= 1:
3         return n
4     else:
5         return fibonacci(n-1) + fibonacci(n-2)
6 n = input("Enter number:")
7 print fibonacci(n)
```

   Kannst du dir erklären wieso der Computer so lange braucht, um das Resultat zu berechnen?

Teilschritte zwischenspeichern – dynamische Programmierung

Durch die doppelte Rekursion rechnet der Computer viele der Werte mehrfach nach, so dass sogar für die schnellen Rechner bald ein fühlbarer Rechenaufwand entsteht. Im Baum sind die Rechenschritte für fibonacci(5) hier abgekürzt fib(5) aufgezeichnet. Wir erkennen, dass fib(1) 5 mal aufgerufen wird, was der 4ten Fibonacci Zahl entspricht.



Im nächsten Schritt fib(6)wird fib(1) bereits $5 + 3 = 8$ mal aufgerufen und zwar 5 mal für fib(5) und 3 mal für fib(4). Auch hier entsteht wieder die Fibonacci Folge einfach jeweils um einen Wert kleiner. Um fib(24) zu berechnen, muss der Computer also fib(23)= 28 657 mal fin(1) berechnen. Echt häufig, oder?

Das geht besser! Rechnen wir von Hand, so rechnen wir die vorhergegangen Werte ja auch nicht jedesmal 'wie irr' nach, sondern wir merken uns den neuen Wert nach jeder Berechnung und und schreiben ihn zum Beispiel in einer Tabelle nieder. Die Tabelle wird so immer grösser, sie entwickelt sich „dynamisch".

Diese einfache Idee, gewisse Werte geschickt zu speichern und während des Aufrufs wieder darauf zurückzugreifen ist auch im Programmieren eine zentrale Idee und man nennt diesen Trick, **dynamische Programmierung**, denn die Speicher-Tabelle wird während der Berechnung dynamisch verändert.

Dynamisch Programmiert sparst du nicht nur Zeit, sondern auch der Zwischenspeicher wird geschont und du kannst viel höhere Fibonacci Werte berechnen. Probiere aus. Ein mögliches Programm ist auf der nächsten angegeben.

8. Spiele mit dem Programm und vergleiche die Zeitwerte für die nomale Rekursion im Vergleich zur dynamischen Programmierung. Beantworte dabei folgende Fragen:

   - Gebe zweimal den gleichen Wert, zu Beispiel 24 ein, was beobachtest du?
   - Gib eine kleine Zahl ein, zum Beispiel fib(5), was stellst du fest?
   - Bei welcher Zahl bekommt die normale rekursive Funktion Probleme bei der Berechnung?
   - Wo liegen die Grenzen für die dynamisch berechnete Funktion?

Lösung: Die Zeiten unterscheiden sich, das kann einerseits daran liegen, dass time.clock() nicht sehr genau misst. Doch der Computer berechnet die Werte auch nicht immer auf die gleiche Art und Weise oder arbeitet noch an etwas anderem, so dass die Zeiten sich leicht unterscheiden.

Für kleine Fibonacci Werte ist der rekursive Funktionsaufruf schneller, das mag daran liegen, dass das erstellen der Tabelle auch eine gewisse Zeit benötigt.

Die Grenzen des Computers sind individuell, doch bereits bei so kleinen Werten wie fobonacci(35) hat der rekursive Aufruf wohl mühe.

Der dynamische Aufruf ist ungleich stärker, auch fibonacci(1000) kann er bewältigen.

```
1  import time
2  #Fibonacci dynamisch Programmiert, Werte werden in results zwischengespeichert.
3  def dp_fibonacci(n):
4      if n <= 1:
5          return n
6
7      if results[n] == None:
8          results[n] = dp_fibonacci(n-1) + dp_fibonacci(n-2)
9
10     return results[n]
11
12 #Fibonacci normal Rekursiv
13 def fibonacci(n):
14     if n <= 1:
15         return n
16     else:
17         return fibonacci(n-1) + fibonacci(n-2)
18
19 n = input("Enter number:")
20 #die Tabelle muss im vornherein mit der richtigen Grösse erzeugt werden
21 results = [None] * (n+1)
22 start= time.clock() #Zeitmessung, um die beiden Funktionen zu vergleichen
23 print dp_fibonacci(n)
24 end= time.clock()
25 print "dp_zeit" , end-start
26
27 start= time.clock()
28 print fibonacci(n)
29 end= time.clock()
30 print "rekursion_zeit" , end-start
```

Aufgabe 1 **Folge von Mittelwerten** Es soll eine Folge von Mittelwerten generiert werden, das heisst, das nächste Glied ist immer der Mittelwert der vorhergehenden beiden.

Beispiel: 0, 1, 0.5, 0.75, 0.625, 0.6875, ...

a) Finde eine rekursive Funktion mittelwert(n), welche dir die gewünschte Folge der Länge n erzeugt.

b) Schreibe ein Programm, analog zu dem der Ficonacci-Folge, welches das n-te Glied der Folge ausgibt. Erstelle ebenfalls je eine Funktion, welche dynamische Programmierung benutzt und eine, welche einfache Rekursion implementiert.

c) Verändere die Funktion so, dass du beliebige Anfangswerte einsetzten kannst.

d) Baue wiederum eine Zeitmessung ein.

e) Vergleiche nun auch hier, die unterschiedlichen Zeiten. Wann bekommt die normale Rekursion Probleme?

f) Probiere unterschiedliche Anfangswerte aus. Kannst du ohne den Computer ermitteln welcher Wert rauskommt, wenn du genügend viele Rekursionen machst? Überprüfe deine Lösung mit dem Rechner. Den gesuchten Wert nennt man Limes oder Grenzwert der Folge.

Lösung:

a) mittelwert(n) $= 0.5 \cdot$ (mittelwert($n-1$)+mittelwert($n-2$)) Start: 0 für $n = 0$ und 1 für $n = 1$

b)

```python
#Mittelwert dynamisch Programmiert, Werte werden in results zwischengespeichert.
def dp_mittelwert(n):
    if n<=1:
        return n

    if results[n] == None:
        results[n] = 0.5*(dp_mittelwert(n-1) + dp_mittelwert(n-2) )

    return results[n]

#mittelwert normal Rekursiv
def mittelwert(n):
    if n<=1:
        return n
    else:
        return 0.5*(mittelwert(n-1) + mittelwert(n-2))
```

c) siehe nächste Aufgabe

d)

```python
import time
#Mittelwert dynamisch Programmiert, Werte werden in results zwischengespeichert.
def dp_mittelwert(n,a,b):
    if n==0: #Erster Anfangswert
        return a
    if n == 1:#zweiter Anfangswert
        return b

    if results[n] == None:
        results[n] = 0.5*(dp_mittelwert(n-1,a,b) + dp_mittelwert(n-2,a,b) )

    return results[n]

#mittelwert normal Rekursiv
def mittelwert(n,a,b):
    if n==0:
        return a
    if n == 1:
        return b
    else:
        return 0.5*(mittelwert(n-1,a,b) + mittelwert(n-2,a,b))

a= input("Gib das erste Glied ein:")
b= input("Gib das zweite Glied ein:")
n = input("Wie viele Iterationen sollen gemacht werden, gib eine Zahl ein:")
#die Tabelle muss im vornherein mit der richtigen Grösse erzeugt werden
results = [None] * (n+1)
start= time.clock() #Zeitmessung, um die beiden Funktionen zu vergleichen
print dp_mittelwert(n,a,b)
end= time.clock()
print "dp_zeit" , end-start

start= time.clock()
print mittelwert(n,a,b)
end= time.clock()
print "rekursion_zeit" , end-start
```

e) Individuell, aber auch ca. bei gleich vielen Rekursionen wie Fibonacci, was nicht erstaunt, da ja beide Funktionen jeweils auf 2 Werte zurückgreifen.

f) Grenzwert = Anfangswert $+\frac{2}{3}\cdot$ Differenz der Anfangswerte.
Bei unserem Anfangsbeispiel ist der Grenzwert $= 0 + \frac{2}{3} \cdot (1 - 0) = \frac{2}{3}$

Aufgabe 2 **Kaninchenvermehrung** Die Fibonacci Folge wurde ursprünglich nicht über Treppen steigen gefunden, sondern anhand der berühmten Kaninchenaufgabe. Nimm an,
dass Kaninchen beliebig lang leben,
dass ein paar Kaninchen jeden Monat ein neues Paar wirft,
dass ein junges Paar selbst das erste Mal nach zwei Monaten wirft.
Wie viele Paare sind nach 1, 2, 3, 4, 5, 6, 7, 8, 9 bzw. 10 Monaten am Leben, wenn man zum Zeitpunkt 0 Monate mit einem neugeborenen Paar beginnt.

Lösung: Natürlich bekommst du hier die Fibonacci Folge als Lösung.