Die Fibonacci-Folge

Leonardo Fibonacci (er vermittelte dem Abendland die arabischen Rechentechniken) stellte 1202 in seinem *liber abaci* folgende Aufgabe: Gegeben sei ein Kaninchenpaar. Jedes Kaninchenweibchen bringt jeden Monat ein neues Paar zur Welt. Junge Kaninchen haben nach einem Monat ihre ersten Jungen. Kaninchen sterben nicht. Wie viele Kaninchenpaare gibt es nach 12 Monaten?

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,...

Diese Zahlenfolge F(n) ist folgendermaßen definiert: $$F(n) = \begin{cases} 1 & n \leqslant 2 \\ F(n-1)+F(n-2) & sonst \end{cases}$$

Wie verstehen wir diese mathematische Formel:

- **ITERATIV**: Wir starten mit zwei Einsern. Nun ergeben sich die weiteren Elemente, indem man jeweils die Summe der beiden letzten anfügt.
- **REKURSIV**: Jedes Element ist laut Formel die Summe der beiden vorherigen Elemente. Wir verschieben also die Aufgabe der Berechnung von F(n) auf die Berechnung von F(n-1) und F(n-2), von zwei gleichartigen aber einfacheren Aufgaben, ab. Erst F(2) und F(1) sind bekannt, weil vorgegeben.

Im Folgenden möchte ich einige einfache Algorithmen zur Berechnung dieser Zahlenfolge besprechen.

Algorithmus 1:

Algorithmus 1 : Genau so, wie es in der Formel steht

```
define fibo(n):
   if n<=2:
      return 1
   else:
      return fibo(n-1)+fibo(n-2)
```

Den 'else' Zweig des Algorithmus kann man einsparen, da der Programmlauf für n<=2 ohnehin mit dem 'return 1' endet.

Algorithmus 1a : kompakt

```
define fibo(n):
   if n<=2:
      return 1
   return fibo(n-1)+fibo(n-2)
```

Vorteil:Dies ist die exakte Übertragung der mathematischen Definition.
Nachteil: Wie oft ruft sich die Funktion rekursiv auf? Um F(20) zu berechnen ruft sie F(19) und F(18) auf. Da diese Werte unbekannt sind, müssen F(18) und F(17) für F(19) sowie F(17) und F(16) für F(18) berechnet werden. Es werden also Werte mehrfach berechnet, was eigentlich nicht sehr sinnvoll ist.
(**Übung**: ergänze Deine Version von Algorithmus 1 um eine Liste und eine Zeile, und zähle mit, welche Rekursionstiefe, bzw. welche kleinere Fibonaccizahl, wie oft besucht wird! Wärest Du überrascht, wenn Du als Lösung wiederum die Fibonacci-Folge antriffst?)
Um nach diesem Algorithmus F(20) zu erhalten, wird etwa F(3) 2584 mal errechnet, für F(30) sogar 317811 mal! Der Zeitbedarf dieses Algorithmus wächst dadurch mit wachsendem n sehr, sehr rasch, nämlich exponentiell.

Algorithmus 2:

Wir merken uns bereits berechnete Werte in einer Liste namens 'fiboliste'. Bereits berechnete Werte stehen dann sofort zur Verfügung, es tritt keine Mehrfachberechnung mehr auf.

Doch wird diese Liste verwaltet? Da im Vorhinein unbekannt ist, die wievielte Fibonacci-Zahl wir wünschen, ist die

nötige Größe der Liste unbekannt. Entweder erlaubt die Programmiersprache eine DYNAMISCHE Liste, oder man muss mühsam mit eigenen Programmabschnitten laufend für passenden freien Speicherplatz sorgen, oder man dimensioniert im Voraus die Liste in der Hoffnung, dass sie für alle Fälle groß genug ist (dann ist sie aber für kleine n sicher viel zu groß und eine Speicherplatzverschwendung). Zwar unterstützen etwa Java, Forth,... die Erzeugung von Arrays zur Laufzeit, doch stehen diese nach der Berechnung nicht mehr für die nächsten Abfragen zur Verfügung, falls eine größere Fibonacci-Zahl zu berechnen ist. Diese Technik interessiert uns hier also nicht.

Algorithmus 2 : Algorithmus 1 mit Gedächtnis

```
fiboliste = leer

define fibo(n):
   if fiboliste[n] bekannt :
      return fiboliste[n]
   if n<=2:
      return 1
   ergebnis = fiboliste[n] = fibo(n-1)+fibo(n-2)
   return ergebnis
```

Algorithmus 2a : kompakter

```
fiboliste = {1:1,2:1}

define fibo(n):
   if fiboliste[n] bekannt :
      return fiboliste[n]
   ergebnis = fiboliste[n] = fibo(n-1)+fibo(n-2)
   return ergebnis
```

Vorteil: immer noch genau die Formel. Schnelles Ergebnis, da bereits errechnete Werte sofort zur Verfügung stehen.

Algorithmus 3:

Und wie macht man es mit Zettel und Bleistift oder überhaupt im Kopf?
Man schreibt zwei Einser an und addiert dann so lange die jeweils letzten zwei Zahlen, bis man beim geforderten n anlangt. Diese Vorgehensweise ist ITERATIV: 1 1 2 3 ..... a b neu=a+b .....

Algorithmus 3 : Die Kopfrechenmethode - iterativ

```
define fibo(n):
   if n<=2:
      return 1
   a = 1
   b = 1
   for zaehler = 3..n:
      neu = a+b
      a = b
      b = neu
   return neu
```

(Wer mag, kann statt a 'vorletztes' schreiben, und statt b 'letztes'.)

Vorteil: maximales Tempo, keine Hilfsliste nötig.

Bemerkung: Wenn die Programmiersprache Mehrfachzuweisungen erlaubt, kann man die Variable 'neu' einsparen und a,b = b,a+b programmieren.
Trick: Erlaubt man eine zusätzliche Rechenoperation, kann man auch mit Einfachzuweisung die Variable 'neu' einsparen:

```
b = b+a
a = b-a
```

sieht als Programmtext sogar elegant aus und wird klarer wenn man

$b_{neu} = a + b$

$a = b_{neu} - a = a+b - a = b$

schreibt.

Es geht noch eleganter! Vielleicht können wir sogar das 'if n <=2' einsparen!
Trick 1: beginnen wir die Schleife bei

Algorithmus 3	wir sparen den Fall n=2 ein, indem wir immer den alten wert ausgeben und einmal öfter die Schleife durchlaufen. Wir rechnen 'auf Vorrat'	und wenn wir ein 'nulltes' Folgenelement erfinden, das nie ausgegeben wird, wandert auch n=1 in die Schleife. Was muss vor 1,1,2,3,.. stehen? Natürlich Null!

define fibo(n):
  if n<=2:
  return 1
  a = 1
  b = 1
  for zaehler = 3..n:
  neu = a+b
  a = b
  b = neu
  return neu
	define fibo(n):
	  if n<=1:
	  return 1
	  a = 1
	  b = 1
	  for zaehler = 2..n:
	  neu = a+b
	  a = b
	  b = neu
	  return a
		define fibo(n):
		
		  a,b = 0,1
		
		  for zaehler = 1..n:
		  neu = a+b
		  a = b
		  b = neu
		  return a

Algorithmus 3a : iterativ elegant	

	define fibo(n):
	  a,b = 0,1
	  for zaehler = 1..n:
	  a,b = b,a+b
	  return a

Diese Version ist speziell für Folgengeneratoren praktisch (siehe unten)

Überlegung:

Warum erkläre ich die rekursiven Algorithmen 1 und 2, wenn die iterative Technik hier so überlegen ist?
1.) Weil es hübsch ist, wenn das Programm exakt die mathematische Notation wiederspiegelt
2.) Weil es viele Aufgaben gibt, die zwar elegant und einfach rekursiv, aber nicht gut iterativ formuliert werden können (speziell in der Graphen- und Spieletheorie).
3.) Es gibt Programmiersprachen, die für Rekursionen optimiert sind. Dort braucht man sich selten iterativ zu plagen.

Erweiterung:

Beginnt man die Fibonacci-Folge nicht mit zwei Einsern, sondern zwei beliebigen Zahlen a und b, so erhält man andere Zahlenfolgen, die aber ganz ähnliche Eigenschaften besitzen. Man nennt sie auch LUCAS-Folgen L(n,a,b) oder 'erweiterte Fibonacci-Folgen'.

$$L(n,a,b) = \begin{cases} a & n=1 \\ b & n=2 \\ L(n-1)+L(n-2) & sonst \end{cases}$$

mit 6 und 2 lauten die Zahlen L(n,6,2): 6 2 8 10 18 28 46 ....
mit 1 und -1 erhält man L(n,1,-1): 1 -1 0 -1 -1 -2 -3 -5 .....
mit -1 und 1 erhält man L(n,-1,1): -1 1 0 1 1 2 3 5 ....

mit 3 und 5 ergibt sich die Folge L(n,5,8): 5 8 13 21 34 55 89 ...
Genaues Hinsehen zeigt: das ist doch genau die Fibonacci-Folge, aber erst ein späterer Abschnitt!
Die Zahl 55 ist entweder als F(10) oder als L(6) mit a=5,b=8 schreibbar. Also: F(n) =L(n,1,1)= L(n-4,5,8).

Denken wir an Algorithmus 1, so könnten wir ihn vielleicht so verbessern, dass wir statt der gewünschten Fibonacci-Folge einfach eine Lucas-Folge mit gut gewählten Startzahlen a und b berechnen. Aber welche? Auch diese Entscheidung können wir getrost unserem Programm überlassen!

So können wir zu einem trickreichen neuen Algorithmus gelangen. Ich formuliere ihn gleich für Lucas-Folgen:

Statt die Folge a b a+b ..... bis zum n-ten Element zu berechnen, also L(n,a,b)
berechne die Folge b a+b ..... ein Element weniger weit, also L(n-1,b,a+b)
und das berechnen wir auch nicht direkt, sondern reduzieren gleich wieder die Ordnung n-1 mit dem selben Trick auf n-2 , und tricksen weiter, bis das Ergebnis klar ist!

Algorithmus 4 : Der rekursive Fibonacci-Trick	

	`define fibo(n,a,b):`
	`  if n=1: return a`
	`  if n=2: return b`
	`  return fibo(n-1,b,a+b)`

Vorteil: schnell, Anzahl der Rekursionsschritte ist linear in n
Nachteil: keiner. Der Trick hat aber ganz direkt mit der Struktur dieser Zahlenfolge zu tun und kann leider nicht so einfach auf andere rekursive Aufgaben übertragen werden...

Rückblick:

Wir begannen mit den Fibonacci-Zahlen, verallgemeinerten zum 'schwierigeren' Problem der Luas-Folgen, erkannten an ihnen einen interessanten Zusammenhang und konnten damit das ursprüngliche Problem auf eine ganz neue und geschickte Art lösen.

Schlechtes Gewissen:

Sollten wir diesen Algorithmus 4 überhaupt 'fibo' nennen, wo er doch Lucas-Zahlen ermittelt?

Alternative 1: Wir nennen die Funktion 'lucas' und schreiben eine eigene 'fibo', die den richtigen Aufruf besorgt:	**Alternative 2:** Einige Programmiersprachen kennen eine Vorbelegung von Aufrufvariablen mit Defaultwerten. Gebe ich diese Parameter beim Aufruf nicht an, treten automatisch die Vorgabewerte in Kraft. Ich schreibe sie einfach mit einem = Zeichen in den Funktionskopf.

`define fibo(n):`
`  return lucas(n,1,1)`
 	
`define lucas(n,a,b):`
`  if n=1: return a`
`  if n=2: return b`
`  return lucas(n-1,b,a+b)`
	`define fibo(n,a=1,b=1):`
	`  if n=1: return a`
	`  if n=2: return b`
	`  return fibo(n-1,b,a+b)`

	Dann ist fibo(5) die fünfte Fibonaccizahl, fibo(5,10,-10) ergibt aber für den neugierigen Spezialisten eine Lucas-Zahl. Und das alles mit dem gleichen Programmtext!

Vorsicht: Bei Experimenten mit größeren Werten für n (ab etwa 40) kann es sinnlose Ergebnisse (negative Zahlen) oder Fehlermeldungen geben! Die meisten Programmiersprachen haben für den ganzzahligen Datentyp eine untere und obere Schranke (meist 2 hoch 16, 2 hoch 32 oder 2 hoch 64 Werte möglich). Wird dieser Bereich überschritten, muss man vorsichtig sein (bei solchen Programmiersprachen...)!
Glücklich sind die Python-Nutzer: hier gibt es keine Obergrenze (einzige Einschränkung: die Zahl muss im Speicher des Computers Platz haben...)

Das Fibo Gesellschaftsspiel:

Was braucht man? Wenn man etwa F(6) als Gesellschaftsspiel berechnen möchte, braucht man 6 Leute, 2 bis 6 Zettel Papier (je nach Algorithmus), Schreibzeug.

So gehts:
Die Spieler S1,S2,...,S6 stellen sich in einem Kreisbogen auf, damit jeder jeden sehen kann. Jeder Spieler hat als Aufgabe die Beantwortung der Frage nach 'seiner' Fibonacci-Zahl. S1 ist für F(1) zuständig, S2 für F(2),..., S6 für S(6). Jetzt wählt man einen der Algorithmen.

Algorithmus 1 (der mit dem größten Spaßfaktor):
S1 und S2 erhalten einen Zettel, auf den sie '1' schreiben. Sie können eine Anfrage sofort beantworten.
Jetzt wird S6 nach F(6) gefragt. Der hat keine Ahnung und weiß nur, daß er für seine Antwort die Antworten von S5 und S4 zusammenzählen muß. Also bittet er S5, ihm zu antworten. Achtung: nicht gleichzeitig S4 fragen, erst auf die Rückmeldung von S5 warten!. Nach Algorithmus 1 ist jetzt S5 an der Reihe, denkt nach und erkennt, dass er S4 und S3 fragen muss. Also fragte er ersteinmal S4 und wartet auf eine Antwort . Jetzt gehts runter (immer merken, wem man antworten muss!) bis bei S2 und S1 die Antworten klar sind. Und jetzt gehts wieder rauf nach oben. Keiner der Spieler S3 bis S6 darf sich aber Notizen über seine Antwort machen! Immer weiterfragen. Und nach einiger Zeit hat S6 die Information von S5. Jetzt braucht er noch die Antwort von S4, um die Addition der beiden Werte zu erledigen, und der Spaß geht weiter.
Hier merkt man übrigens, warum es *nicht* genügt, im Algorithmus nur F(1) vorzugeben!

Algorithmus 2:
Jeder Spieler S3 bis S6, der einmal geantwortet hat, darf sich seine einmal wie oben gewonnene Antwort auf seinen Zettel notieren und bei einer neuerlichen Anfrage sofort diese Antwort herunterlesen.

Algorithmus 3:
S1 und S2 haben ihre Zettel. Nun addiert S3 seine beiden Vorgänger, schreibt das Ergebnis auf und hält den Zettel hoch. S1 kann seinen Zettel ablegen. Jeder weiter Spieler schaut auf seine zwei Vorgänger, addiert auf, zeigt seinen Zettel und der drittletzte Spieler legt seinen Zettel ab. Es sind immer 2 Zettel in der Höhe ('a' und 'b' des Algorithmus), gelegentlich kurz ein dritter ('neu' des Algorithmus).

Generatoren

Einige Programmiersprachen ermöglichen die Konstruktion von Generatoren. Darunter verstehen wir Funktionen, die sozusagen einen Wert der Folge nach dem anderen produzieren bzw. zur Verfügung stellen (englisch: yield). Jeder Aufruf des Generators ergibt ein weiteres Folgenelement

Algorithmus 5 : Der Fibonacci-Generator

```
define fibo_generator():
   setze Startwerte
   loop
      berechne das nächste Folgenelement
      yield Folgenelement
   end loop
```

Verwendung	`fibo = fibo_generator()`
	
	`for n = 1 to 20:`
	`  print fibo.next()`

Noch mehr Mathematik (Weitere Möglichkeiten zur Berechnung der Fibonacci-Zahlen)

1.) Durch Induktion beweist der Zusammenhang

$$F_{n+m}=F_{n+1}F_m+F_nF_{m-1}$$

Diesen kann man als die Verdopplungsformeln F_{2n} und F_{2n+1} schreiben. Damit erhält man die Möglichkeit, etwa F_{20} sofort als Kombination aus F_{10} und F_9 schreiben zu können. Das kann man einfach rekursiv weitertreiben. Es entsteht ein logarithmischer Arbeitsaufwand!

2.) Der Zusammenhang der Fibonacci-Folge kann auch so formuliert werden (Beweis induktiv):

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

Hier wird die Berechnung der Folge auf das Potenzieren einer 2x2 Matrix abgeschoben. Dies kann man ebenfalls schnell mit logarithmischem Aufwand erledigen.

3.) Die Binet'sche Formel liefert sofort (Aufwand von Ordnung 1) die n-te Fibonaccizahl, allerdings muss man mit reellen Zahlen arbeiten. Durch die begrenzte Stellenzahl des float-Datentyps ist sie nur für kleine n brauchbar, wenn man die Zahl exakt haben möchte. (Beweis ebenfalls induktiv)

.

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right) = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n)$$

(Bemerkung: da der Beitrag des zweiten Summanden kleiner als ½ ist und potenziert wird, genügt es, nur den ersten zu berechnen und dann auf die nächstgelegene ganze Zahl zu runden. Dann erhält man die n-te Fibonaccizahl aus dem goldenen Schnitt hoch n durch die Wurzel aus 5!)

4.) Die erzeugende Funktion G(z), deren Reihenentwicklung die Fibonacci-Zahlen als Koeffizienten hat, ist

$$G(z)=\frac{z}{1-z-z^2}=\sum_{n=0}^{\infty} F_n z^n$$

Beweis: durch Partialbruchzerlegung mit $1-z-z^2=-(1-\phi)(1-\hat{\phi})$

Umsetzung von Algorithmen:

Um praktische Beispiele zu erhalten, folgen nun einige Umsetzungen dieser Algorithmen in unterschiedlichen Programmiersprachen. Man kann dabei sehr schön die verschiedenen 'Stile' sehen, in denen man dabei die Programme verfasst. Ich habe dabei nur jene Algorithmen berücksichtigt, die sich in der betreffenden Sprache 'natürlich' und ohne zusätzlichen Aufwand formulieren lassen (etwa Algorithmus 2 nur dort, wo die Sprache dynamische Daten direkt unterstützt) . Weiters habe ich auf spezielle Sprachkonstrukte verzichtet, die ein 'Parallel-Lesen' der Prozeduren erschweren und in unserem Fall nicht nötig sind (wie etwa switch oder CASE bei mehrfachen if-Abfragen).

Sollte ich in dem einen oder anderen Fall den 'Geist' und Stil einer Programmiersprache nicht ganz getroffen haben, bin

ich für Verbesserungsvorschläge immer dankbar!

Wolfgang.Urban@schule.at