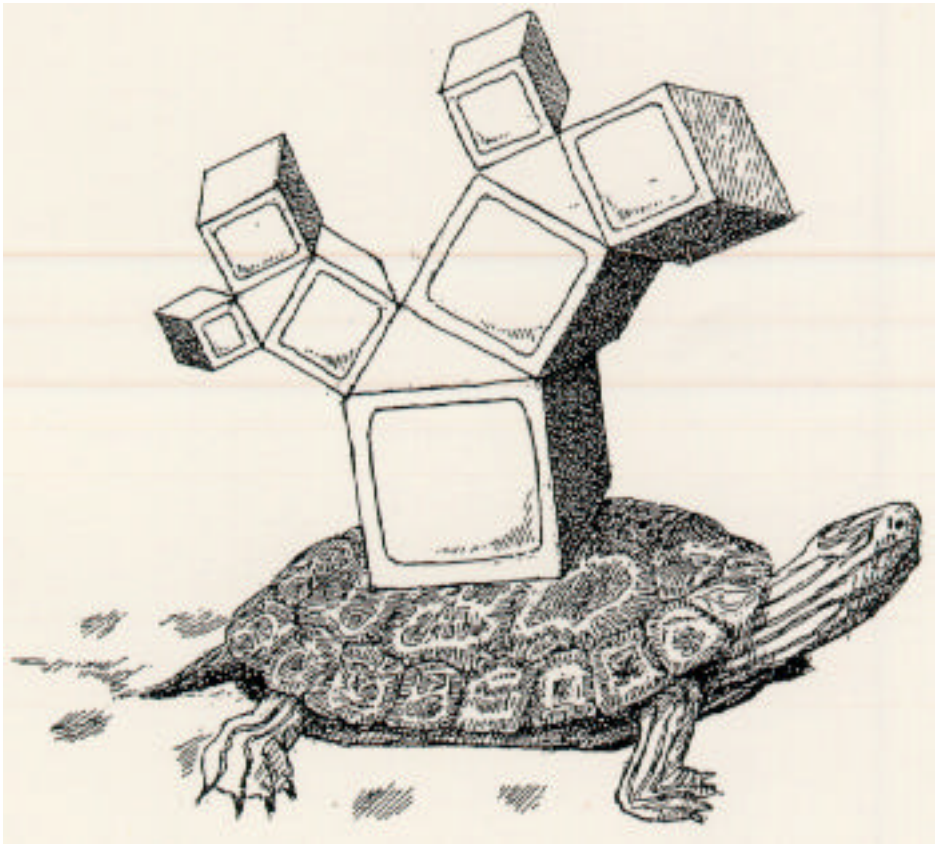
Rekursives



Programmieren

Ein Leitprogramm in Informatik

Verfasst von
Andrea Alder, Marco Bettinaglio
Werner Hartmann, Patrick Leoni
Markus Meier, Florian Schlotke
Herausgegeben durch U. Kirchgraber und W. Hartmann

Leitprogramm „Rekursives Programmieren“

Version Mai 1995

Stufe, Schulbereich
Gymnasium, Fachhochschule

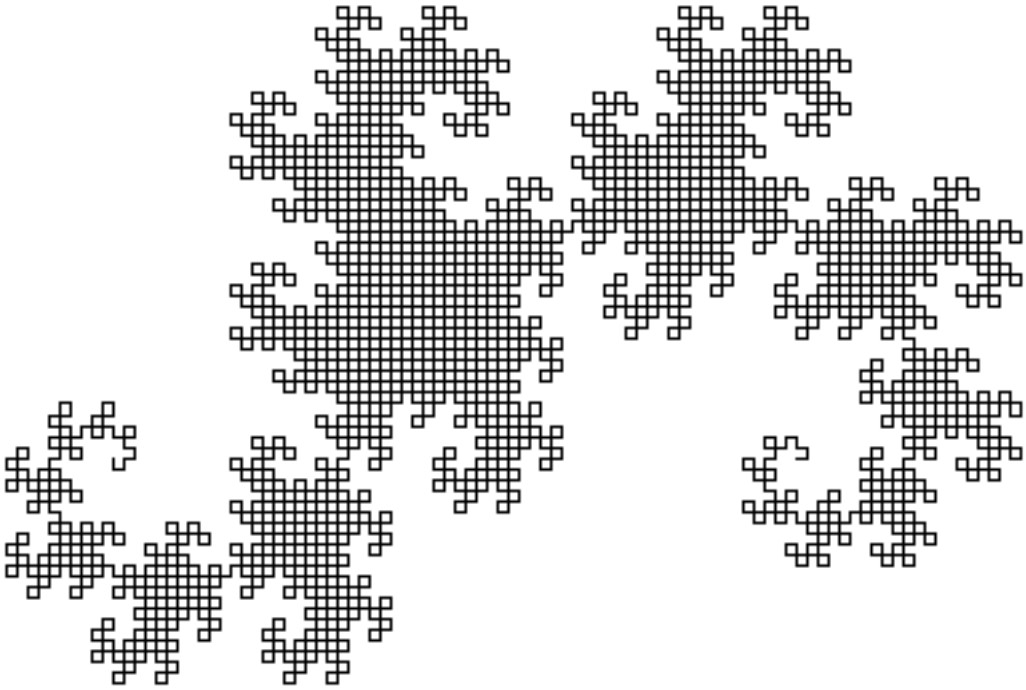
Fachliche Vorkenntnisse
Begriff der Rekursion und Iteration aus der Mathematik
Programmierkenntnisse in einer strukturierten Sprache

Bearbeitungsdauer
15 - 20 Lektionen
(abhängig von der Anzahl der implementierten Programme)

Die *Leitprogramme* waren ein Gemeinschaftsprojekt von Karl Frey und Angela Frey-Eiling (Initiatoren), Walter Caprez (Chemie), Hanspeter Dreyer (Physik), Werner Hartmann (Informatik), Urs Kirchgraber (Mathematik), Hansmartin Ryser (Biologie), Jörg Roth (Geographie), zusammen mit den Autorinnen und Autoren.

Inhaltsverzeichnis

Einführung



Worum geht es?

In der Mathematik, Informatik, Grafik, Biologie, aber auch in unserem Alltag, treffen wir oft auf die Situation, dass sich ein Problem lösen lässt, indem eine bestimmte Lösungsmethode in gleicher Form mehrmals angewendet wird. Bei der *Iteration* wird diese Methode schrittweise nacheinander angewendet, bei der *Rekursion* dagegen ineinander geschachtelt.

Rekursion ist ein gedanklich anspruchsvolles Prinzip und wird daher oft als „zu schwierig“ gemieden. Rekursion ist aber so fundamental, dass man ihr gar nicht aus dem Weg gehen kann: Selbstreproduktion, welche die Grundlage jeglichen Lebens bildet, ist ein rekursiver Vorgang.

Welche Bedeutung hat die rekursive Programmierung?

Mit Hilfe der Rekursion kann man viele Probleme elegant lösen. Das gilt speziell, wenn man die Berechnung dem Computer überlassen will. Dieses Leitprogramm stellt dir in den ersten drei Kapiteln die Rekursion als Technik und Programmierkonzept vor.

Daneben eignet sich die Rekursion hervorragend für grafische Spielereien. Schon mit wenigen Kenntnissen lassen sich hübsche, rekursive *Computergrafiken* verwirklichen. Das nötige „Handwerk“ dazu wird dir in den letzten beiden Kapiteln vermittelt.

Was kannst du nach der Bearbeitung dieses Leitprogramms?

Schon in wenigen Stunden wirst du die oben abgebildete Grafik programmieren. Das ist aber noch lange nicht alles…

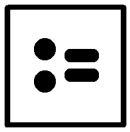
Arbeitsanleitung

Mit Hilfe dieses Leitprogramms wirst du das Thema „Rekursives Programmieren" *selbständig* erarbeiten. Das Programm besteht aus 5 Kapiteln. Diese sind immer gleich strukturiert:

Übersicht Worum geht es in diesem Kapitel? Was ist zu tun?

Lernziele Was kann ich nach dem Bearbeiten des Kapitels?

Im eigentlichen Lernstoff-Abschnitt begegnest du folgenden Piktogrammen:



Definition

Information

Wissenssicherung

Aufgabe

Programmieraufgabe

Lernkontrolle Habe ich alles verstanden?

Lösungen zu allen gestellten Problemen.

Nach jedem Kapitel wird dein Wissen in einem *Kapiteltest* geprüft.
Die Testfragen holst du bei deiner Lehrerin oder deinem Lehrer.

Wenn du einmal nicht weiter weisst ...

Solltest du einen Abschnitt auch nach der dritten Durchsicht noch nicht verstehen, fragst du eine Mitschülerin um Rat. Wenn ihr auch gemeinsam nicht weiterkommt, dürft ihr die Lehrerin um Hilfe bitten.

Muss das ganze Leitprogramm durchgearbeitet werden?

Nein! - Obligatorisch zu behandeln sind die Kapitel 1 bis 4. Falls du danach noch genügend Zeit hast, oder wenn dich das Thema fasziniert, kannst du dich an das fakultative Kapitel 5 wagen. Dieses vermittelt dir einen Ausblick auf weiterführende Themen der rekursiven Computergrafik.

1 Rekursiv definierte Folgen in der Mathematik

Alle Kreter sind Lügner.
Epimenides, Kreter.

Übersicht

Was lernst du hier?

Dieses Kapitel führt euch in eine wichtige Technik der Mathematik und Informatik ein. Mit ihr können wir viele Probleme sehr einfach beschreiben und lösen. Wir werden diese Technik zuerst an einem Beispiel zeigen. Dabei wirst du – vielleicht erstaunt – feststellen, dass sie dir gar nicht neu ist: Aus der Mathematik kennen wir den Begriff der *rekursiv definierten Folgen.*

Was tust du?

Lies die Abschnitte der Reihe nach durch und versuche die Aufgaben zu lösen. Lege dir am besten gleich jetzt einen Stapel Sudelpapier zurecht. Für manche Aufgaben wirst du einen Taschenrechner brauchen. Den Computer lassen wir fürs erste ruhen. Schau dir unten einmal die Lernziele an, dann weisst du, worum es geht.

Lernziele

Nachdem du dieses erste Kapitel durchgearbeitet hast,

- kannst du den Begriff „rekursiv definierte Folgen“ erklären
- weisst du, wie man rekursiv definierte Folgen iterativ berechnet
- weisst du, warum A4-Blätter 21.0 x 29.7 cm messen.

Alles klar? Schön, jetzt geht’s los …

1.1 Erinnern wir uns an die Folgen

Wahrscheinlich hast du in sogenannten Intelligenztests auch schon Fragen angetroffen wie:

„Finde die nächsten Zahlen in der Folge 2, 4, 8, …"

Um die Antwort zu finden, muss man die Gesetzmässigkeit dieser Folge erkennen. In diesem Fall ist jede Zahl das Doppelte der vorangegangenen Zahl. Im Mathematikunterricht hast du gelernt, wie man dieses Bildungsgesetz korrekt aufschreibt. Die einzelnen Glieder der Folge werden durchnumeriert. In der Informatik beginnt man traditionellerweise eine Numerierung mit 0. Mit a_n bezeichnen wir die n-te Zahl der Folge. Das Bildungsgesetz für die Folge 2, 4, 8, … lautet dann:

$a_0 = 2$

$a_n = 2 \cdot a_{n-1}$ für n = 1, 2, 3, …

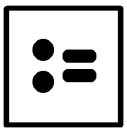
Nicht alle Gesetzmässigkeiten erkennt man so schnell. Welches ist z.B. das nächste Glied der Folge 0.5, 2, 5, 11, 23, … ?

Die Antwort ist 47, denn das Bildungsgesetz lautet hier:

$a_0 = 0.5$

$a_n = 2 \cdot a_{n-1} + 1$ für n = 1, 2, 3, …

Die Gleichungen $a_n = 2 \cdot a_{n-1}$ bzw. $a_n = 2 \cdot a_{n-1} + 1$ berechnen also ein neues Glied a_n mit Hilfe seines Vorgängers a_{n-1}.



Definition

- Bei einer rekursiv definierten Folge kann ein bestimmtes Glied a_n der Folge aufgrund der Kenntnis vorangegangener Folgeglieder berechnet werden.
- Gleichungen der Form $a_n = f(a_{n-1})$, $a_n = f(a_{n-1}, a_{n-2})$, ... heissen *Rekursionsvorschriften* . f kann dabei irgendeine Funktion sein.
- Die Gleichungen, die das erste Glied einer Folge festlegen, heissen *Anfangsbedingungen* oder *Rekursionsbasis.*
- Eine rekursiv definierte Folge wird also durch eine *Rekursionsvorschrift* und eine *Rekursionsbasis* beschrieben.

Achtung: Lass dich von den vielen Definitionen, Formeln und Indizes nicht abschrecken! Das Ganze ist eigentlich keine grosse Hexerei. Die Begriffe in der Definition solltest du dir aber gut merken, sie werden dir im Leitprogramm immer wieder begegnen.

In der folgenden Wissenssicherung 1 und in Aufgabe 2 kannst du überprüfen, wie gut du die Welt der rekursiv definierten Folgen verstanden hast …

Wissenssicherung 1

Berechne die ersten 6 Glieder der Folge:

$a_0 = 1$

$a_n = 2 \cdot a_{n-1} + 1$ für n = 1, 2, 3, …

Aufgabe 2

Berechne a_{100}. Wie stellst du es am geschicktesten an?

Es gibt mehrere Möglichkeiten a_{100} zu berechnen. Es lohnt sich auf alle Fälle zuerst genau zu überlegen. Erst dann solltest du den Taschenrechner oder Computer bemühen. Eine ausführliche Lösung findest du am Ende des Kapitels.

Aufgabe 3: „Mit dem Papier ist es so eine Sache“

Hast du dich auch schon gefragt, warum wir eigentlich meist auf A4-Blättern schreiben? Papier könnte ja irgend ein beliebiges Format haben. Es könnte quadratisch sein, es könnte 20 x 25 cm gross oder 10 x 30 cm gross sein. Aber nein, unser Papier (auch das, welches du gerade liest), ist eben meist DIN A4, also 21.0 x 29.7 cm gross.

Früher war das nicht so. Jede Person, die Papier herstellte, schnitt es irgendwie zu: gerade so, wie es ihr passte. Diese Tatsache bereitete vielen Leuten Mühe, etwa den Sekretärinnen, deren Briefpapiere nie in irgendwelchen Couverts Platz fanden.

Einige praktische Köpfe nahmen sich deshalb vor, ein für allemal aufzuräumen mit diesem Papierdurcheinander. Man erfand die DIN-Norm 472 (DIN = Deutsche Industrie Norm). Darin wurde festgelegt, in welchen Grössen die Papierbögen herzustellen und zu gebrauchen sind. Zuerst wurde das grösste Papierformat festgelegt. Man nannte es „Weltformat“ und definierte folgendes:

Seine Fläche beträgt 1 m^2, sein Seitenverhältnis beträgt 1 : $\sqrt{2}$.

Finde heraus, wie gross so ein Bogen im Weltformat ist, also seine Länge und Breite.

Aufgabe 4: „Mit dem Papier ist es so eine Sache“ (Fortsetzung)

Das Weltformat heisst auch A0.. Die weiteren Formate wurden wie folgt festgelegt:

- Das Papierformat A1 erhält man, indem man ein A0 Blatt so faltet, dass seine Breitseiten aufeinander zu liegen kommen.
- Auf dieselbe Weise – durch wiederholtes Falten – erhält man die weiteren Papierformate der DIN-A-Reihe.

Schreibe die Grösse der Papierformate A0 bis A4 auf.

Bei genauerer Betrachtung fällt natürlich sofort die Ähnlichkeit zur Rekursion ins Auge. Auch hier wird eine Rekursionsbasis und eine Rekursionsvorschrift gegeben. Die Papierformate sind rekursiv definiert !

Wie lautet die Rekursionsvorschrift?
Welches ist die Rekursionsbasis?

Aufgabe 5

Du bist jetzt drei Stichworten begegnet: *iterativ*, *explizit* und *rekursiv*. Es ist wichtig, dass wir uns klar machen, was genau mit diesen Begriffen gemeint ist.

Suche in deinem Mathematikbuch oder in einem Lexikon eine Umschreibung der Begriffe *rekursiv definierte Folge*, *explizite Darstellung eines Folgegliedes* und *iterative Berechnung einer rekursiv definierten Folge*.

Gib diese Definitionen dann mit eigenen Worten wieder.

Eine verflixte Sache …

Bis jetzt sieht alles ganz einfach aus! Es gibt aber auch rekursiv definierte Folgen, die nicht mehr so einfach durchschaubar sind.

Das nächste Beispiel sieht auf den ersten Blick recht harmlos aus. Eine nähere Betrachtung wird deinen Denkapparat aber ganz schön auf Touren bringen. Das Beispiel in Aufgabe 6 ist aber nicht einfach eine mathematische Spielerei. Es taucht in der Praxis häufig bei Wachstumsmodellen auf:

Aufgabe 6:

Betrachte die Folge

a_0 = Startwert zwischen 0 und 1

$a_n = k \cdot a_{n-1} \cdot (1 - a_{n-1})$ für n = 1, 2, 3, …

wobei k eine Konstante ist.
Nimm als Startwert 0.2. Mit Hilfe deines Taschenrechners kannst du jetzt untersuchen, wie sich die Folge für die Werte k = 2, 3.2, 3.5, 3.56 und 4 entwickelt. Für k = 2 siehst du schon nach dem 5. Glied, was passiert. Für höhere k-Werte brauchst du vielleicht etwas mehr Glieder. Was stellst du fest? Erkennst du ein explizites Bildungsgesetz für die Folge, also eine Formel, mit der du problemlos a_{100} berechnen kannst?

Rekursiv definierte Folgen

In der Mathematik spielen *rekursiv definierte Folgen* eine wichtige Rolle. Typischerweise werden die Glieder einer rekursiv definierten Folge *iterativ berechnet.* Ausgangspunkt für die Iteration ist dabei die Rekursionsbasis. In manchen Fällen lässt sich auch eine explizite Darstellung für die Folgeglieder finden.

Nach diesen geistigen Höhenflügen wenden wir uns wieder einfacheren Dingen zu: In der Lernkontrolle lernst du eine der wohl bekanntesten rekursiv definierten Folgen kennen!

Lernkontrolle

Aufgabe 7

Die Folge 1, 1, 2, 3, 5, 8, 13, 21, ... der *Fibonacci - Zahlen* ist rekursiv definiert durch

$$fib(0) = fib(1) = 1$$
$$fib(n) = fib(n-1) + fib(n-2) \qquad \text{für n = 2, 3, 4, ...}$$

Schreibe auf deinem Taschenrechner oder einem Computer ein kleines Programm, das die ersten 50 Fibonacci - Zahlen iterativ berechnet.

Wenn dir alle obigen Aufgaben klar sind, kannst du bei deiner Lehrerin oder deinem Lehrer den Kapiteltest holen.

Lösungen Kapitel 1

Wissenssicherung 1

Die Folge $a_0 = 1$

$a_n = 2 \cdot a_{n-1} + 1$ für n = 1, 2, 3, …

beginnt mit 1, 3, 7, 15, 31, 63, …

Aufgabe 2

a_{100} kannst du auf mindestens 3 Arten berechnen:

1. Iterative Berechnung

Du setzt die Folge Schritt für Schritt - eben iterativ - fort, bis du a_{100} berechnet hast. Dazu kannst du dir ein kleines Programm schreiben, zum Beispiel auf deinem programmierbaren Taschenrechner. Aber aufgepasst: Das genaue Resultat wirst du wohl kaum erhalten: es hat sehr viele Stellen …

2. Explizite Berechnung

Vielleicht hast du es gemerkt: Für a_n scheint es eine explizite Formel zu geben:

$$a_n = 2^{n+1} - 1$$

Also ist

$$a_{100} = 2^{101} - 1 = 2535301200456458802993406410751$$

Jetzt weisst du warum dein Taschenrechner Probleme hatte. Es ist eben nicht jedermanns Sache, mit 31-stelligen Zahlen zu jonglieren.
Wenn du die vollständige Induktion als Beweismethode kennst, wird du auch ohne Probleme die Gültigkeit der obigen expliziten Formel für a_n nachweisen.

3. Rekursive Berechnung

So hast du es wohl kaum gemacht - niemand käme auf diese Idee. Doch der Vollständigkeit halber:

Zuerst führen wir a_{100} auf a_{99} zurück:

$$a_{100} = 2 \cdot a_{99} + 1$$

Analog können wir a_{99} auf a_{98} zurückführen:

$$a_{99} = 2 \cdot a_{98} + 1$$

So erhalten wir für a_{100}:

$$a_{100} = 2 \cdot (2 \cdot a_{98} + 1) + 1 = 4 \cdot a_{98} + 3$$

Wenn wir noch einen Schritt weiter gehen erhalten wir:

$$a_{100} = 8 \cdot a_{97} + 7$$

Wir können so weiterfahren, bis wir a_{100} auf a_0 zurückgeführt haben. Da wir wissen, dass $a_0 = 1$ ist, sind wir am Ende unserer Rechnung angelangt. Die Rekursionsbasis bewahrt uns also vor endloser Rechnerei.

Aufgabe 3

Länge(Weltformat) = 1189 mm
Breite(Weltformat) = 841 mm

Aufgabe 4

	A0	A1	A2	A3	A4
Länge [mm]	1189	841	595	420	297
Breite [mm]	841	595	420	297	210

Beachte: Die Breite eines A(i)-Blattes entspricht der Länge eines A(i-1)-Blattes.

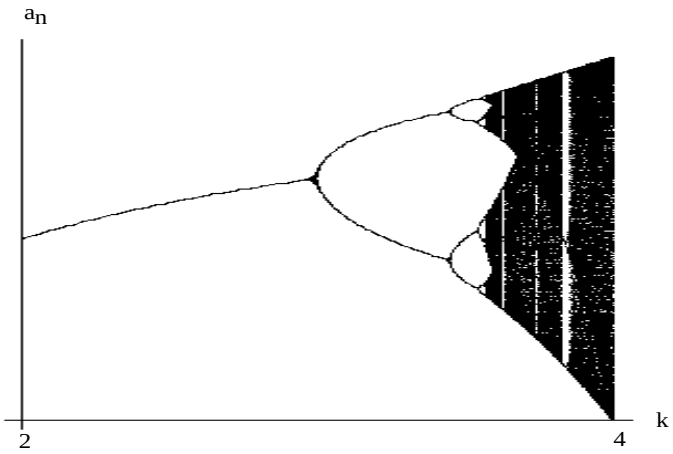
Rekursionsbasis	Ein A0-Bogen misst 1189 x 841 mm
Rekursionsvorschrift	A(i) entsteht aus A(i-1) durch Falten, so dass die Breitseiten aufeinander liegen. Die Rekursionsvorschrift lautet also: Länge(Ai) = Breite(Ai-1) und Breite(Ai) = $\frac{1}{2}$ Länge(Ai-1)

Aufgabe 5

Du liegst sicher richtig, wenn deine Definition derjenigen im Leitprogramm entspricht!

Aufgabe 6

Wir nehmen es gleich vorweg: Eine explizite Formel für die Folge wirst du nicht gefunden haben.
Wenn du den Anfang der Folge für k = 4 aufschreibst, kannst du wohl keine Gesetzmässigkeit erkennen. Uns geht es auch so! Kein Wunder: mit diesem Beispiel sind wir mitten im Bereich *Chaos und Fraktale.* Vielleicht hast du schon einmal das nachstehende Bild - das sogenannte *Feigenbaum-Diagramm* angetroffen.



Diese Abbildung hat direkt mit unserer Folge zu tun. Mehr wollen wir hier nicht verraten, es würde den Rahmen unseres Leitprogramms sprengen.

Vielleicht kannst du aber deine Lehrerin oder deinen Lehrer überzeugen, später auch logistisches Wachstum, Feigenbaum-Diagramme oder Apfelmännchen, kurz Chaos und Fraktale zu behandeln. Sicher findest du in der Schulbibliothek auch Literatur zu diesem faszinierenden Thema. Sehr empfehlenswert sind die Bücher von H. O. Peitgen, die auch einfach zu lesen sind.

Zurück zu unserer Frage: Obwohl du keine explizite Formel für die Folge gefunden hast, konntest du sicher a_{100} iterativ berechnen.

In Pseudocode sieht dein Programm zum Beispiel so aus:

```
Eingabe k
a = 0.2
FOR n = 1 TO 100
    Ersetze a durch k · a · (1-a)
    Ausgabe a
END
```

Das Programm berechnet sukzessive alle Glieder der Folge bis a_{100}. Ausgehend von der Rekursionsbasis sind wir wie auf einer Treppe von einer Zahl zur nächsten hochgestiegen.

Aufgabe 7 (Lernkontrolle)

Die Programmierung der iterativen Berechnung der Fibonacci-Zahlen liegt auf der Hand. Dein Programm sollte in Pseudocode ähnlich wie das Programm in Aufgabe 6 aussehen. Der einzige Unterschied ist, dass die Rekursionsbasis zwei Glieder der Folge umfasst und dass die Rekursionsvorschrift immer die beiden vorangehenden Glieder der Folge benötigt. Einfach ist es, wenn du alle bereits berechneten Fibonacci-Zahlen in einem Array *fib()* abspeicherst. Dann sieht dein Programm in Pseudocode zum Beispiel so aus:

```
fib(0) = 1
fib(1) = 1
FOR  n = 2  TO  50
      fib(n) = fib(n-1) + fib(n-2)
      Ausgabe fib(n)
END
```

Übrigens: Für die Fibonacci-Zahlen gibt es auch eine explizite Formel

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

Mit der Methode der vollständigen Induktion kann man die Gültigkeit dieser Formel ohne Mühe beweisen. Auf die Formel selber zu stossen ist schon ein bisschen schwieriger....

2 Rekursion in der Informatik was ist das ?

Divide et impera! (*teile und herrsche!)*
Louis XI.

Übersicht

Ein kleines Experiment!

Nimm einen Papierstreifen und versuche ihn so zu falten, dass sieben genau gleich grosse Teile entstehen. Dabei darfst du keinen Massstab oder sonst ein Hilfsmittel verwenden. Gar nicht so einfach diese Aufgabe!

Wenn du statt sieben jedoch acht Teile machen willst, wird es plötzlich einfach: Einmal in der Mitte falten, dann nochmals falten .....

Genau das ist das Prinzip der Rekursion in der Informatik: Ein Problem wird auf ein „kleineres“ Problem zurückgeführt, das wiederum nach demselben Verfahren bearbeitet wird. Rekursion ist eine ganz wichtige algorithmische Technik in der Informatik.

Am obigen Beispiel hast du auch gesehen, dass die Lösung einer Aufgabe, wenn sie mit Rekursion möglich ist, sehr einfach wird.

Was lernst du hier?

In diesem Kapitel erstellst du rekursive Programme. Du erfährst, wie auf dem Computer rekursiv programmiert werden kann. Keine Angst, das Grundsätzliche der Rekursion ändert sich dabei nicht. Vielmehr geht es um die Art und Weise, wie sie in den Computer eingegeben wird.

Zentral ist dabei die Möglichkeit, dass ein Unterprogramm sich selbst aufrufen kann. Diese Eigenschaft von Unterprogrammen erleichtert uns die Arbeit: Teilaufgaben, die bei einer Berechnung immer wieder vorkommen, müssen nur einmal definiert werden. Dieses einfache und sehr nützliche Mittel bildet den Kern der rekursiven Programmierung. Hier lernst du diesen *„Selbstaufruf“* kennen und anwenden.

Was tust du?

Zuerst studierst du die Lernziele. Nimm dir Zeit! Es ist wichtig, dass du weisst, worauf es bei diesem Kapitel ankommt.

Dann bearbeitest du das Beispiel des Telefonalarms und den Streckenalgorithmus. Mit Hilfe von Pseudocode wirst du schon in kurzer Zeit die ersten Programme rekursiv aufschreiben können.

Lernziele

Wichtigstes	Du lernst die Rekursion als ein in vielen Fällen sehr elegantes Problemlöseverfahren der Informatik kennen.
Zweitwichtigstes	Du verstehst, was ein Selbstaufruf eines Unterprogrammes bewirkt und worauf du bei der Definition achten musst.
Auch wichtig	Du hast dich an den Pseudocode in diesem Leitprogramm gewöhnt und weisst, welches die entsprechenden Befehle in deiner Programmiersprache sind.

Denke noch zwei Minuten über die Aussage nach, dass eine Arbeit sich immer wieder selbst aufrufen kann: Was geschieht bei zwei Spiegeln, die eine Kerze gegenseitig widerspiegeln? Beachte dabei, dass jede Kopie etwas kleiner wird. Was ist hier die gleichbleibende „Arbeit“, und mit welcher Startvorgabe beginnt sie?

2.1 Der Telefonalarm

Das Prinzip des Telefonalarms ist uns allen bekannt, auch Leuten, die nie etwas über Rekursion gehört haben:

Die 15 Mitglieder einer Volleyball-Mannschaft haben die gleiche Liste aller Spielerinnen mit ihren Telefonnummern. Die erste Spielerin auf der Liste erhält die Meldung, das nächste Spiel sei verschoben worden. Sie greift zum Hörer und ruft die zweite Spielerin auf der Liste an. Diese wiederum ruft die dritte Spielerin an und so geht es weiter, bis die ganze Mannschaft informiert ist.

Aufgabe 1

Angenommen jede Spielerin benötigt pro Telefon eine Viertelstunde. („Wollen wir anstatt des Matches ins Kino gehen …") Dann geht es insgesamt mehr als drei Stunden, bis alle Spielerinnen informiert sind. Überlege dir deshalb einen schnelleren Telefonalarm! Das Ziel ist es, mit deinem Telefonalarm in weniger als zwei Stunden die ganze Mannschaft zu informieren. Dabei soll jede Spielerin höchstens zwei Telefone machen müssen. Und natürlich sollte der Telefonalarm auch funktionieren, wenn eine Spielerin krank ist!

Wissenssicherung 2

Entwirf einen Telefonalarm für eine Mannschaft mit 24 Spielerinnen. Verwende dabei das Verfahren, das in der Lösung von Aufgabe 1 beschrieben wurde, und zeichne ein neues, entsprechendes Baumdiagramm.

Du hast es sicher schon bemerkt: Hinter dem Telefonalarm steckt das Prinzip „Teile und herrsche", also das Prinzip der Rekursion.

Aufgabe 3

Analysiere das rekursive Verhalten des Telefonalarms!

Rekursionsvorschrift

Rekursionsbasis

Wir haben gesehen: Alle Spielerinnen machen im Prinzip immer dasselbe, die gleiche Arbeit. Es genügt deshalb, eine für alle Spielerinnen gleichlautende Telefonalarm-Anleitung zu schreiben:

Anleitung Telefonalarm

Beginn

Falls ich die Meldung nicht weitergeben muss, mache ich nichts

Sonst halbiere ich meine Liste in zwei Hälften;

Dann rufe ich die erste Person auf der ersten Listenhälfte an;
Ich teile dieser Person die Meldung mit und beauftrage sie, die Meldung mittels Telefonalarm an die Personen auf der ersten Listenhälfte weiterzugeben;

Dann rufe ich die erste Person auf der zweiten Listenhälfte an;
Ich teile dieser Person die Meldung mit und beauftrage sie, die Meldung mittels Telefonalarm an die Personen auf der zweiten Listenhälfte weiterzugeben;

Fertig

Ende

Diese Anleitung ist nichts anderes als der Pseudocode einer *rekursiven Prozedur* . Also eines Unterprogramms, das sich selbst wieder aufruft.

Wenn wir die Anleitung noch ein bisschen computergerechter aufschreiben, wird das klarer:

Programm Telefonalarm (Liste)

BEGIN

IF Liste enthält nur noch eine Person **THEN** fertig

ELSE

Teile Liste in 2 Hälften;

Aufruf des Unterprogrammes **Telefonalarm** (1. Listenhälfte)

Aufruf des Unterprogrammes **Telefonalarm** (2. Listenhälfte)

END

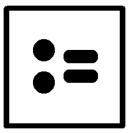
END

Wir übergeben der Prozedur „Telefonalarm“ den Parameter „Liste”. Falls die Liste noch Telefonnummern enthält, wird sie in zwei kleinere Listen zerlegt, und diese werden zur Bearbeitung wiederum an die Prozedur „Telefonalarm“ übergeben. Mit anderen Worten: Die Prozedur „Telefonalarm“ ruft sich selbst auf!

Bei der rekursiven Programmierung wird es immer ähnlich aussehen:

Das Ausgangsproblem wird in kleinere, einfachere Teilprobleme zerlegt. Die Teilprobleme sind dem Ausgangsproblem ähnlich und können deshalb analog zum Ausgangsproblem bearbeitet werden. Diese Unterteilung wird solange fortgeführt, bis die Teilprobleme so einfach sind, dass sie direkt gelöst werden können.

Im Prinzip besitzt also jedes rekursive Programm die gleiche Grundstruktur. Das folgende Schema zeigt diese Struktur in Pseudocode:



Grundstruktur rekursiver Programme

```
Programm Löse (Problem)
BEGIN
    IF  Problem einfach  THEN
        Löse es direkt
    ELSE
        Unterteile das Problem in ähnliche Teilprobleme;
        Löse (Teilproblem)
    END
END
```

Das Kernstück dieses Programmschemas ist das **IF … THEN … ELSE**. Es enthält die *Rekursionsbasis* (im IF) und die *Rekursionsvorschrift* (im ELSE).

2.2 Vom Pseudocode zum Programm

Vielleicht hast du jetzt vor lauter Telefonalarm und Rekursion ein Durcheinander; vielleicht möchtest du jetzt endlich ein rekursives Programm auf deinem Computer laufen sehen? Geduld, wir sind gleich so weit!

Wir wissen nicht, in welcher Programmiersprache du deine Programme implementierst. Darum schreiben wir unsere Programme hier im Leitprogramm in Pseudocode. Pseudocode bedeutet, dass wir keine eigentliche Programmiersprache verwenden. Die Algorithmen im Leitprogramm werden in einer Art Umgangssprache beschrieben. Wir brauchen dabei aber Konstrukte und Ausdrücke, die in jeder richtigen Programmiersprache in der einen oder anderen Form vorkommen.

PROCEDURE	Damit meinen wir ein Unterprogramm.
Parameter	Falls ein Unterprogramm zur Ausführung irgendwelche Angaben benötigt, werden diese in Form von Parametern mit dem Aufruf übergeben.
IF … THEN … ELSE	Bedingte Entscheidung
FOR … TO	Zählschleife
Variablen	Speicherplätze zur Aufbewahrung von Daten.

Aufgabe 4

Weisst du noch, wie die obigen Begriffe in deiner Programmiersprache heissen und welche Bedeutung sie haben?

Wenn du nicht sicher bist, solltest du jetzt in einem Handbuch nachschauen oder mit deinem Nachbarn darüber diskutieren.

Es ist gut, wenn du gleich ein kleines Programm in deiner Programmiersprache schreibst. In der nächsten Aufgabe schreibst du aber auch gerade dein erstes rekursives Programm.

Aufgabe 5

Die Fibonacci-Zahlen sind rekursiv definiert und man kann sie deshalb auch rekursiv berechnen. In Pseudocode sieht das Programm zum Beispiel so aus:

PROCEDURE fib (n)
BEGIN
    IF n = 0 oder n = 1 **THEN**
        fib(n) = 1
    ELSE
        Rufe Programm fib für die beiden Parameterwerte
        n-1 und n-2 auf, d.h. berechne fib(n-1) und fib(n-2):
        fib(n) = fib(n-1) + fib(n-2)
    END

END

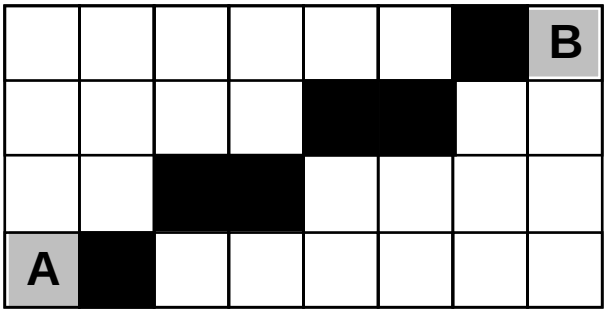
Hauptprogramm

BEGIN
Eingabe von n
Rufe Prozedur **fib**(n) auf
END

Schreibe das obige Programm in deiner Programmiersprache und teste es auf deinem Computer.

2.3 Wie zeichnet der Computer Strecken?

Hast du dir schon einmal überlegt, wie der Computer auf dem Bildschirm eine Strecke zeichnet? Das ist nämlich gar keine einfache Aufgabe. Der Bildschirm besteht aus einem Gitter von einzelnen Bildpunkten, den sogenannten Pixeln. Jedes einzelne Pixel kann vom Computer gefärbt werden. Während wir uns gewohnt sind, zwei Punkte auf dem Papier durch eine simple Linie in einem Strich zu verbinden, muss für den Bildschirm abgeklärt werden, welche Pixel „angezündet" werden sollen.



So könnte zum Beispiel - stark vergrössert - die Verbindungsstrecke von A nach B auf dem Bildschirm aussehen.

Aufgabe 6

Überlege dir ein *rekursives* Verfahren, das die Pixel bestimmt, die zur Verbindungsstrecke zweier Punkte gehören.

Suche dir eine Partnerin, die im Leitprogramm gleich weit ist, und erläutere ihr dein Verfahren.

Wenn du Probleme hast, hier ein Tip: Genauso wie beim Telefon, muss man hier die Pixel informieren, damit sie leuchten …

Jetzt wird es ernst!

Programmieraufgabe 7

Implementiere deinen rekursiven Streckenalgorithmus in deiner Programmiersprache. Teste dein Programm für verschiedene Strecken!

Achtung: Wenn dein Programm nicht auf Anhieb läuft, darfst du nicht verzweifeln. Rekursive Programme haben so ihre Tücken …

Information

Unser rekursiver Streckenalgorithmus ist nicht besonders effizient. Für Zeichnungen mit Tausenden von Strecken ist er viel zu langsam. Unser Ziel war es nur, dir ein rekursives Programm zu zeigen, bei dem du die Rekursion förmlich mitverfolgen kannst. Wenn du wirklich wissen möchtest, wie der Computer Strecken zeichnet, dann schaue in einigen gängigen Büchern über Algorithmen und Datenstrukturen unter dem Stichwort „Bresenham-Algorithmus“ nach.

Lernkontrolle

Aufgabe 8

Eine Möglichkeit, den *grössten gemeinsamen Teiler (ggT)* zweier natürlicher Zahlen a und b zu berechnen, ist die folgende:

- Falls a > b ist, subtrahiere b von a und berechne ggT (a–b, b) anstelle von ggT(a, b).
- Falls b > a ist, subtrahiere a von b und berechne ggT (a, b–a) anstelle von ggT(a, b).
- Falls a = b ist, gilt ggT (a, b) = a = b und man ist fertig.

1. Überlege dir, warum dieser Algorithmus korrekt ist.
2. Du hast sicher bemerkt, dass auch dieser Algorithmus rekursiv ist. Welches ist die Rekursionsbasis, welches die Rekursionsvorschrift?
3. Schreibe ein rekursives Programm in Pseudocode und anschliessend in deiner Programmiersprache. Teste dein Programm an einigen (auch speziellen) Beispielen.

Wenn dir alle obigen Aufgaben klar sind, kannst du bei deiner Lehrerin oder deinem Lehrer den Kapiteltest holen.

Viel Glück !

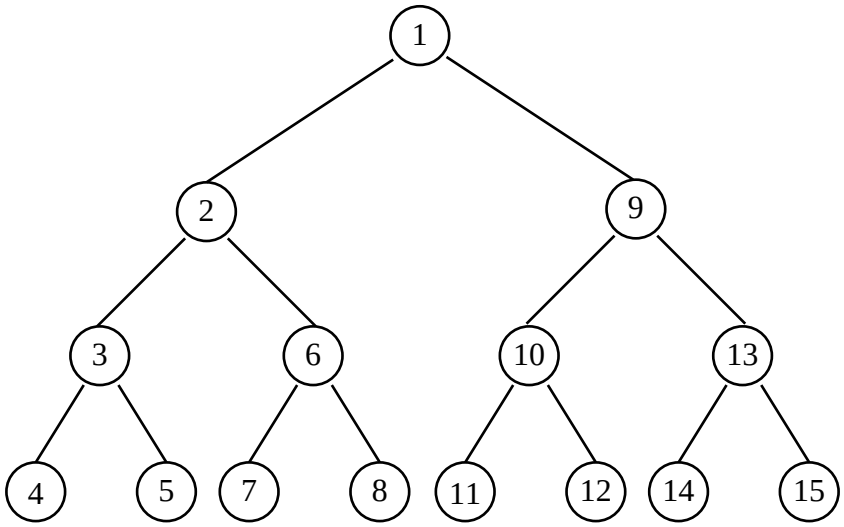
Lösungen Kapitel 2

Aufgabe 1

Die Spielerin zuoberst auf der Liste muss 14 Spielerinnen informieren. Getreu dem Motto „Divide et impera!" teilt sie die 14 Spielerinnen in zwei Hälften auf. Aus jeder Hälfte wählt sie eine Spielerin aus und beauftragt diese, ihre Hälfte weiter zu informieren. Zum Beispiel erhält die 2. Spielerin den Auftrag, die Spielerinnen 3 bis 8 zu informieren. Analog soll die 9. Spielerin die Spielerinnen 10 bis 15 informieren.

Die 2. und die 9. Spielerin verfahren nun genau gleich. Auch Sie teilen ihren Auftrag wieder in zwei Teile. Die 2. Spielerin wird also die 3. Spielerin beauftragen, Nummer 4 und 5 zu informieren. Spielerin Nummer 6 soll sich um Nummer 7 und 8 kümmern.

Schematisch sieht unser Telefonalarm also so aus.



Es gibt keine überflüssigen Anrufe. Keine Spielerin macht mehr als zwei Telefongespräche, und der Telefonalarm ist nach anderthalb Stunden erledigt.

Wissenssicherung 2

Wir geben hier bewusst keine Lösung an. Du wirst selber auf die Schwierigkeit gestossen sein: Mit den Hälften geht es nicht immer so schön auf. Je nachdem, welche „Hälfte" du ein bisschen grösser gewählt hast, sieht dein Baumdiagramm ein wenig anders aus.

Aufgabe 3

Rekursionsbasis	Es muss keine weitere Spielerin mehr informiert werden.
Rekursionsvorschrift	Jede Spielerin, die noch weitere Spielerinnen informieren muss, streicht sich selber auf der Liste und teilt ihre Liste in zwei Hälften, ruft je die Spielerin zuoberst auf jeder Listenhälfte an und beauftragt diese, die Meldung weiterzugeben.

Aufgabe 4

Die Lösung hängt von deiner Programmiersprache ab. Wenn du nicht alle Elemente herausgefunden hast, dann frage einen Kollegen oder deine Lehrerin.

Aufgabe 5

Die Lösung hängt von deiner Programmiersprache ab. Wenn du Schwierigkeiten hast, dann frage eine Kollegin oder deinen Lehrer.

Aufgabe 6

Wir schlagen dir folgendes Verfahren vor: Sind die Punkte bzw. die Pixel A und B benachbart, so markieren wir die beiden Punkte. Andernfalls halbieren wir die Strecke und zeichnen die beiden kürzeren Strecken nach demselben Verfahren. Das Programm sieht in unserem Rekursionsschema folgendermassen aus:

```
PROCEDURE  Strecke (A, B)
BEGIN
    IF  A und B benachbart  THEN  Markiere A und B
    ELSE  Bestimme die Mitte der Strecke von A und B
        Strecke (A, M)
        Strecke (M, B)
    END
END
```

Programmieraufgabe 7

Um die obige Lösung wirklich in einer Programmiersprache zu implementieren, müssen wir das Programm noch verfeinern. Dazu führen wir die folgenden Variablen ein:

Ax, Ay	**:**	x- und y- Koordinate von Punkt A
Bx, By	**:**	x- und y- Koordinate von Punkt B
Mx, My	**:**	x- und y- Koordinate der Mitte von A und B

Numerieren wir die Pixel auf dem Bildschirm in x- und y- Richtung, so hat jedes Pixel auf dem Bildschirm ganzzahlige Koordinaten. Die Variablen Ax, …, My können also nur ganzzahlige Werte annehmen.

Mit diesen Bezeichnungen sieht unser verfeinertes Programm wie folgt aus:

```
PROCEDURE Strecke (Ax, Ay,  Bx, By)
BEGIN
    IF |Ax - Bx| ≤ 1 AND |Ay - By| ≤ 1 THEN
        Markiere (Ax, Ay)
        Markiere (Bx, By)
    ELSE
        Mx := (Ax + Bx) DIV 2
        My := (Ay + By) DIV 2
        Strecke (Ax, Ay,  Mx, My)
        Strecke (Mx, My,  Bx, By)
    END
END
```

DIV bezeichnet dabei die Division mit Rest. Schaue im Handbuch deiner Programmiersprache nach, wie die entsprechende Operation heisst!

Deine endgültige Lösung hängt stark von der verwendeten Programmiersprache ab. Deshalb findest du in diesem Leitprogramm keine konkrete Lösung. Du kannst dir aber bei deiner Lehrerin eine Lösung holen.

Möchtest du die Rekursion besser sichtbar machen? Dann empfehlen wir dir, jeden neu berechneten Mittelpunkt M auf dem Bildschirm einzuzeichnen. Wenn du das Programm zusätzlich künstlich verlangsamst, kannst du sehr schön mitverfolgen, wie die Rekursion abgearbeitet wird.

Hinweis: Ganz besonders schön kann man die verschiedenen Rekursionsstufen sichtbar machen, wenn man die Mittelpunkte M je nach Rekursionstiefe mit einer anderen Farbe färbt. Zu diesem Zweck empfiehlt es sich, die Rekursionstiefe als Parameter der Prozedur Strecke zu übergeben und bei jedem Prozedur-Aufruf um 1 zu erhöhen.

Aufgabe 8 (Lernkontrolle)

1. Der grösste gemeinsame Teiler von a und b ist sicher ein Teiler von a und b und damit auch ein Teiler von a – b bzw. b – a. Das sieht man am leichtesten ein, wenn der ggT als Faktor von a und von b ausgeklammert wird.

2. Die Vorschriften, wie die Fälle a > b bzw. a < b abzuarbeiten sind, sind die Rekursionsvorschrift. Die Vorschrift für den Fall a = b ist die Rekursionsbasis.

3.
```
PROCEDURE ggT (a, b)
BEGIN
    IF a = b THEN ggT := a
    ELSE
        IF a > b THEN
            ggT (a - b, b)
        ELSE
            ggT (a, b - a)
        END
    END
END
```

Information

Wie schon unser rekursiver Streckenalgorithmus ist auch die Berechnung des grössten gemeinsamen Teilers nicht sehr effizient. Das Verfahren enthält aber die dem *Euklid'schen Algorithmus* zugrundeliegenden Ideen.

Den Euklid'schen Algorithmus - einer der schönsten und effizientesten Algorithmen überhaupt - findest du zum Beispiel im Schülerduden Informatik. Wenn du Zeit hast, solltest du dir diese Perle unter den Algorithmen anschauen und programmieren!

3 Probleme und Grenzen

Aus kleinem Reis wird ein grosser Baum.
Deutsches Sprichwort

Übersicht

Was lernst du hier?

Ein Programm ruft sich selber auf..... irgendwie hat das im letzten Kapitel immer geklappt. Wie arbeitet der Computer aber intern eine rekursiv gestellte Aufgabe ab? Wie schafft er es, nicht den Überblick zu verlieren?

Dazu benötigt er für jeden (Selbst-)Aufruf eines Unterprogrammes ein Stückchen Speicherplatz. Darin werden alle notwendigen Informationen abgelegt. So sind sie später wieder abrufbar. Dieser Platz bleibt so lange erhalten, bis das ganze Unterprogramm durchlaufen ist. Wie das im Detail funktioniert, erfährst du in diesem Kapitel. Darüber Bescheid zu wissen ist nötig. Erst dann kannst du entscheiden, wann eine rekursive Lösung geeignet und effizient ist.

Was tust du?

Wie dieses Wertespeichern konkret vor sich geht, erfährst du in den nächsten 30 Minuten wie folgt:

- Nachdem du die Lernziele angeschaut hast, beginnst du mit dem kleinen Theorieteil über die Computerbuchhaltung.
- Das behandelte Beispiel des Streckenalgorithmus benutzt du, um einmal die Buchhaltung des Computers selbst zu übernehmen.
- Für interessierte Leserinnen gibt es einen fakultativen Teil über diese Verwaltungsaufgaben.

Lernziele

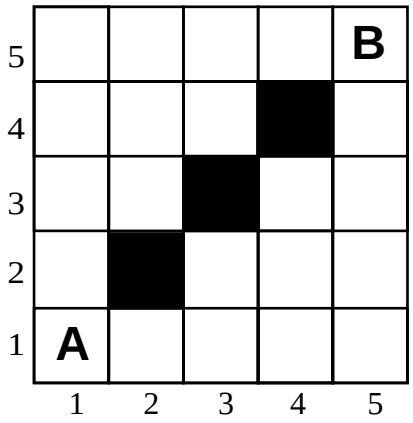
- Du verstehst, wie der Computer aufgerufene und noch nicht beendete Unterprogramme bearbeitet. Es geht nicht um kleinste Details, sondern um den Mechanismus.

- Mit diesem Wissen fällt es dir leicht, bei einem gegebenen Problem zu entscheiden, ob eher eine rekursive oder eine iterative Lösung angezeigt ist.

3.1 Ein Blick in die Buchhaltung des Computers

Bis zu diesem Punkt interessierte uns nur die Strategie der rekursiven Programmierung. Nun fragen wir uns, wie der Computer rekursive Programme abarbeitet.

Wenden wir uns nochmals dem Streckenalgorithmus zu:




```
PROCEDURE Strecke (A, B)
BEGIN
    IF  A und B benachbart  THEN
        Markiere A und B
    ELSE
        Bestimme die Mitte von A und B;
        Strecke (A, M);
        Strecke (M, B)
    END
END
```

In der abgebildeten Situation sind A und B nicht benachbart. Also wird die Mitte bestimmt. Dann wird die Prozedur Strecke für die Endpunkte A und M aufgerufen. Das ursprüngliche Programm Strecke (A, B) wird unterbrochen. Erst wenn das Programm Strecke (A, M) abgearbeitet ist, fährt man an dieser Stelle weiter und arbeitet das Programm Strecke(M,B) ab.

Für den Computer ist es nun wichtig, dass er sich merkt, an welcher Stelle er nach Abarbeitung des Programms Strecke (A, M) weiterfahren soll. Dazu friert er alle benötigten Werte (Informationen) ein. Dieses Einfrieren stellen wir uns zunächst wie folgt vor:

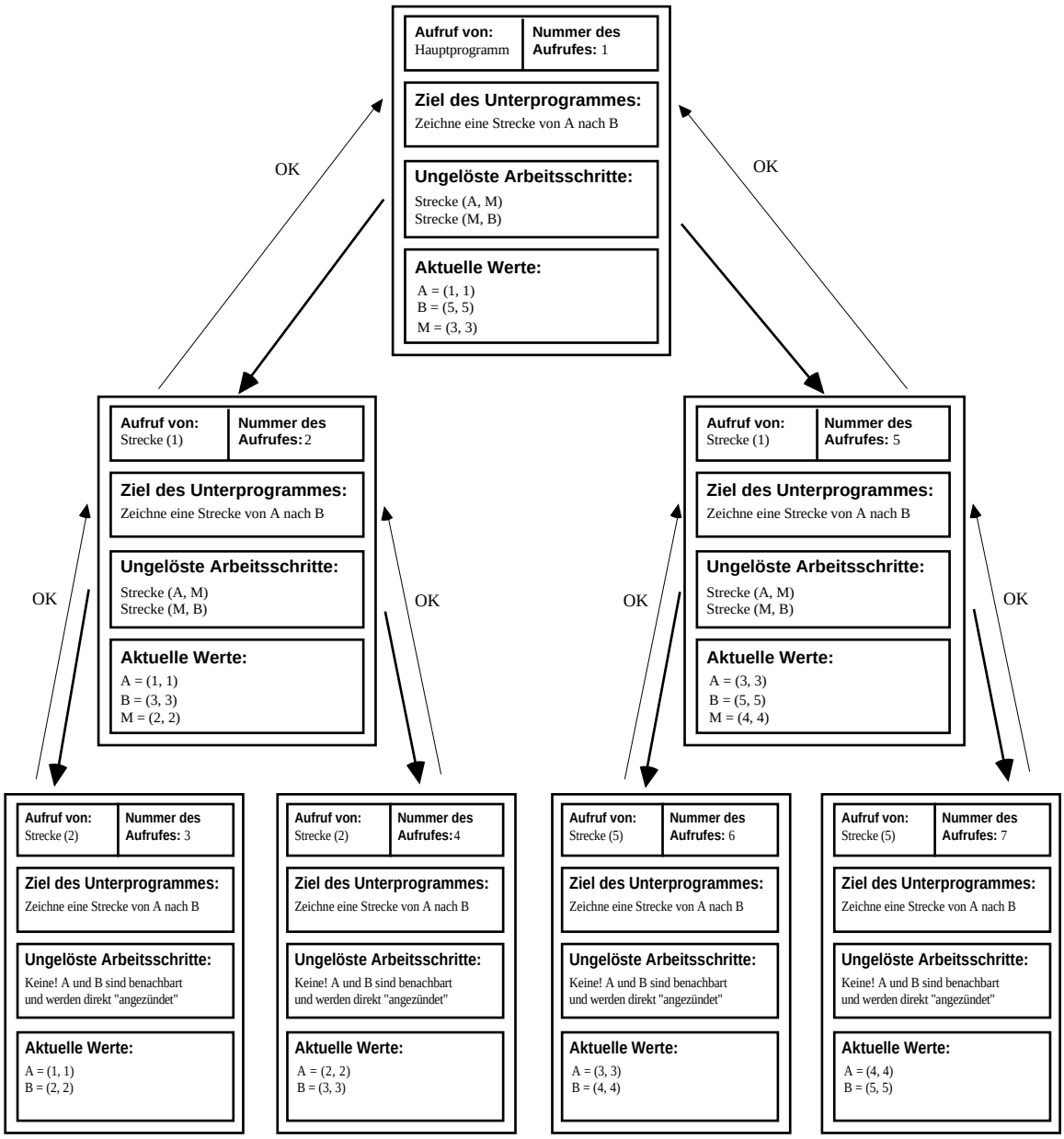
- Jedes Unterprogramm bekommt eine Karteikarte, auf welche alle wichtigen Werte geschrieben werden.
- Bei jedem neu aufgerufenen Unterprogramm entsteht automatisch eine neue Karte.
- Die Nummer des Aufrufes entspricht gerade derjenigen der Kartennummer.
- Jede neue Karte kommt *auf* die vorangegangenen zu liegen. Dadurch werden die früheren verdeckt. Es ist immer nur die oberste Karte „aktiv“.
- Wenn ein Unterprogramm zu Ende ist, wird die entsprechende Karte vom Stapel genommen und vernichtet. Damit gelangt aber die daruntergelegene nach oben und wird „aktiv“.

Schauen wir uns so eine Karte genauer an:

Deine Lehrerin oder dein Lehrer hält leere Karten für dich bereit. Damit kannst du die Aufgaben in diesem Kapitel durchspielen.

Auf der nächsten Seite findest du das Beispiel des Streckenalgorithmus - dargestellt mit Hilfe des Karteikartensystems.

Der Streckenalgorithmus:



Die Aufrufe 3, 4, 6 und 7 werden vollständig durchgeführt, da sie jeweils die Rekursionsbasis erreicht haben. Bekanntlich ist dann das Problem ohne weitere Vereinfachung lösbar: Die Strecke zweier benachbarter Punkte wird gezeichnet. Nachdem die Teilstrecke gezeichnet wurde, kommt das *recurrere* - das Zurücklaufen. Betrachten wir das Zurücklaufen am Beispiel der Karte 3:

Die Strecke ((1, 1) (2, 2)) wurde gezeichnet. Bevor wir die Karte 3 löschen können, müssen wir noch schnell die Karte 2 benachrichtigen, dass alles zur vollsten Zufriedenheit erledigt wurde. Die Karte 2 ist mit ihrem Auftrag jetzt halb fertig: Sie muss nur noch die Karte 4 veranlassen die Strecke ((2, 2) (3, 3)) zu zeichnen. Nachdem das geschehen ist, kann sie stolz und zufrieden ein ”OK” an die oberste Karte geben. So wird der ganze Auftrag durch ständiges Aufteilen, Delegieren und wieder Zusammensetzen abgearbeitet. Beachte die Reihenfolge der Karten: Es wird immer zuerst links in die Tiefe gestiegen!

Aufgabe 1

Suche dir eine Partnerin, die im Leitprogramm gleich weit ist. Versucht gemeinsam die Computerbuchhaltung für den Telefonalarm aus Kapitel 2 einmal selbst in die Hand zu nehmen.

Für den Streckenalgorithmus oder den Telefonalarm haben wir noch nicht allzu viele Karteikarten benötigt. Der Buchhaltungsaufwand hielt sich in Grenzen. Sobald wir aber eine längere Strecke zeichnen oder eine grössere Telefonliste abarbeiten müssen, geraten wir arg ins Schwitzen. Hier zeigt sich die Stärke des Computers:

Der grosse Buchhaltungsaufwand kümmert den Computer im Gegensatz zu uns wenig!

Genauer genommen: Der Buchhaltungsaufwand für den Computer ist ebenso gross wie für uns, nur arbeitet er viel schneller.

Aber sogar für den Computer kann der Buchhaltungsaufwand ein verheerendes Ausmass annehmen. Im Kapitel 1 hast du die *Fibonacci-Folge* untersucht:

$$\text{fib}(0) = \text{fib}(1) = 1$$

$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \qquad \text{für } n = 2, 3, \ldots$$

Wissenssicherung 2

Berechne einige grosse Fibonacci-Zahlen mit deinem rekursiven Programm aus Kapitel 1. Was fällt dir auf?

Wenn du Zeit (und Lust) hast, kannst du den Buchhaltungsaufwand bei der rekursiven Programmierung der Fibonacci-Zahlen genauer untersuchen. Dann wird dir der Unterschied zwischen rekursiver und iterativer Berechnung schnell klar.

Programmieraufgabe 3 (freiwillig)

Berechne die ersten 20 Fibonacci-Zahlen und den Quotienten

$$q(n) = fib(n) / fib(n-1)$$

zweier aufeinanderfolgenden Fibonacci-Zahlen.

Miss gleichzeitig die Zeit t(n), welche dein rekursives Programm zur Berechnung von fib(n) braucht, und berechne die Quotienten

$$p(n) = t(n) / t(n-1).$$

Vergleiche die Quotienten q(n) und p(n). Was fällt dir auf?

Wir haben gesehen, dass nicht jedes rekursive Programm effizient ist. Deshalb ist es wichtig, dass du dir vor der Verwendung der Rekursion Gedanken machst:

- Wie stark wächst die Anzahl der Programmaufrufe bei zunehmender Unterteilung des Problems in Teilprobleme?
- Gibt es nicht eine einfache, iterative Lösung?

Faustregel

Rekursive Algorithmen sollten dann verwendet werden, wenn Rekursion eine naheliegende Problemlösung ist und ein und dieselbe Sache nicht in offensichtlicher Weise mehrfach berechnet wird!

Bist du im Leitprogramm nicht zu sehr im Rückstand und daran interessiert zu wissen, wie der Computer das Karteikarten-System handhabt, so kannst du jetzt das fakultative Kapitel 3.2 in Angriff nehmen. Du lernst dann auch, dass der Computer jedes rekursive Programm de facto iterativ abarbeitet, Schritt für Schritt. Er entlastet uns einfach vom grossen Buchhaltungsaufwand über die auszuführenden Schritte.

Für den Kapiteltest ist das Kapitel 3.2 nicht notwendig.

3.2 Die Verwaltung mit dem Stapelspeicher

Auf dem Computer wird die Verwaltung von unterbrochenen Unterprogrammen natürlich nicht mit Karteikarten ausgeführt.

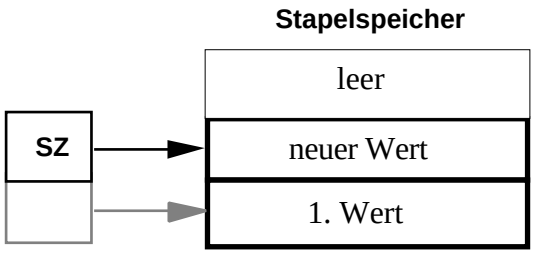
Das Prinzip hingegen bleibt das gleiche: Bei jedem Aufruf wird ein Teil eines speziellen, internen Speichers reserviert, um die nötigen „Karteidaten“ abzulegen. Man nennt diesen Speicher auch *Stapelspeicher* oder auf Englisch *stack*. Der Name deutet seine Eigenschaften schon an:

- Er stapelt Daten in der Reihenfolge wie sie hereinkommen und gibt sie in der umgekehrten Reihenfolge wieder heraus. Man nennt diese Art des Abspeicherns auch *last in - first out.*
- Jeder Aufruf eines Unterprogrammes erhält ein genau zugewiesenes Stück (Ort und Grösse) des Stapelspeichers. Dieses Stück ist quasi die Karteikarte des Computers.

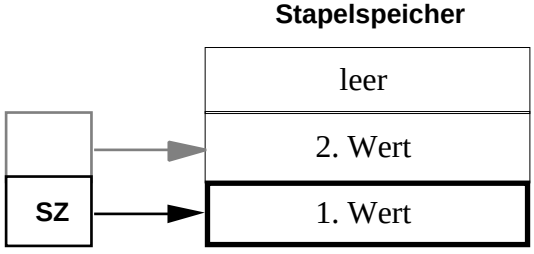
Den Aufbau eines Stapelspeichers kann man sich als lange Kette von aneinandergereihten Speicherzellen vorstellen, mit einem *Stapelzeiger*, der immer auf den zuletzt eingegebenen Wert zeigt.

Der Stapelspeicher kann mit zwei Operationen manipuliert werden:

PUSH: Speichert einen Wert in eine leere Speicherzelle. Der Stapelzeiger (SZ) ist danach auf diese Zelle gerichtet. Wird ein neuer Wert gespeichert, so wächst der Stapel von unten nach oben.



POP: Gibt den zuletzt eingegebenen Wert zurück und setzt den Stapelzeiger auf den vorletzten Wert.

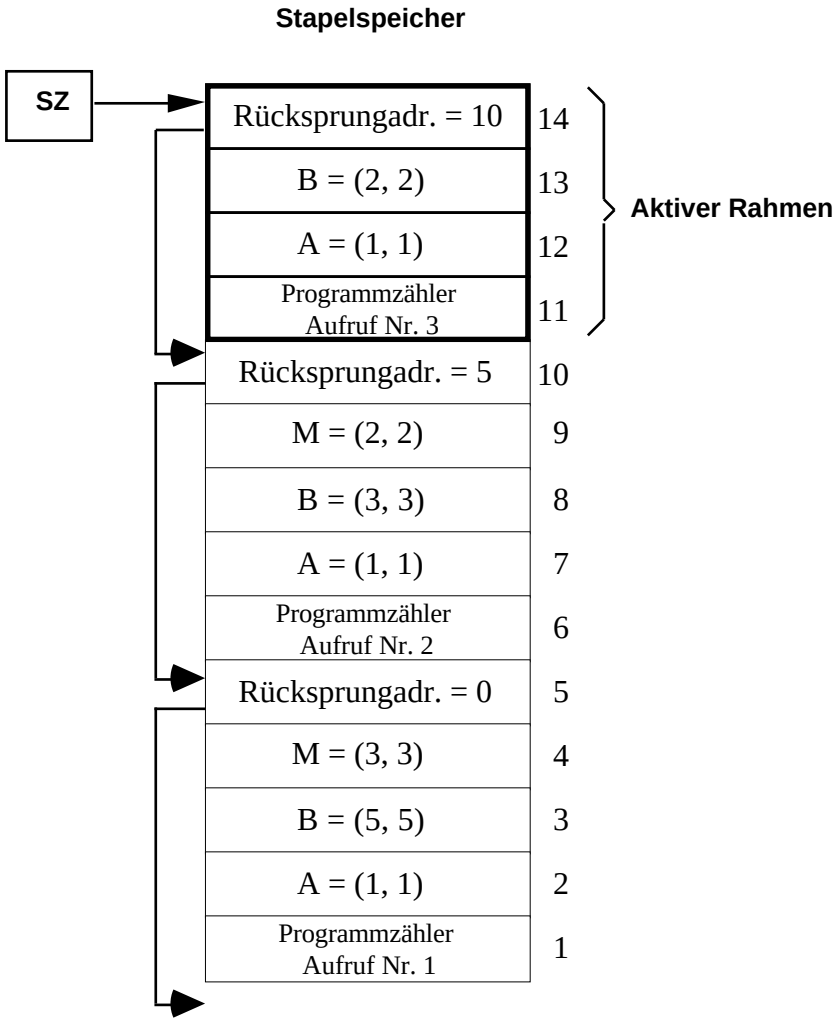


Du hast sicher bemerkt, dass der 2. Wert dabei nicht gelöscht wird. Man hat aber mit keiner Stapelanweisung mehr Zugriff auf ihn. Die Zelle wird als leer betrachtet und ein neuer Wert wird einfach darübergeschrieben.

Ein Aufruf eines ganzen Unterprogrammes benötigt aber nicht nur eine Speicherzelle. Deshalb werden für alle laufenden Parameter Speicherzellen reserviert und in einer Art „Rahmen“ im Stapelspeicher abgelegt. Dieser Rahmen umschliesst alle Daten, die zu einem bestimmten Aufruf eines Unterprogrammes gehören. Neben den Prozedurparametern werden noch die „Rücksprungadresse“ und der Programmzähler auf dem Stapel abgelegt. Die Rücksprungadresse entspricht einer Klammer um den Rahmen und sagt dem Computer, wo der vorhergehende Rahmen aufhört. Der Programmzähler enthält die Stelle, an der die entsprechende Prozedur unterbrochen wurde, um die nächste Unterprozedur aufzurufen.

Bei einem Rücksprung aus einer Unterprozedur wird der aktive Rahmen gemäss Rücksprungadresse nach unten verschoben. Die Werte der ursprünglichen Prozedur sind wieder aktiv und sie kann weiterarbeiten.

Für den ersten Teil des Streckenalgorithmus sieht der Stapelspeicher folgendermassen aus:



Während des ganzen Streckenalgorithmus wächst und schrumpft der Stapelspeicher ständig. Je nachdem wie tief er sich gerade in der Rekursion befindet.

Du siehst, dass doch einiges im Hintergrund abläuft, um die Rekursion so bequem programmieren zu können.

Lernkontrolle

Aufgabe 4

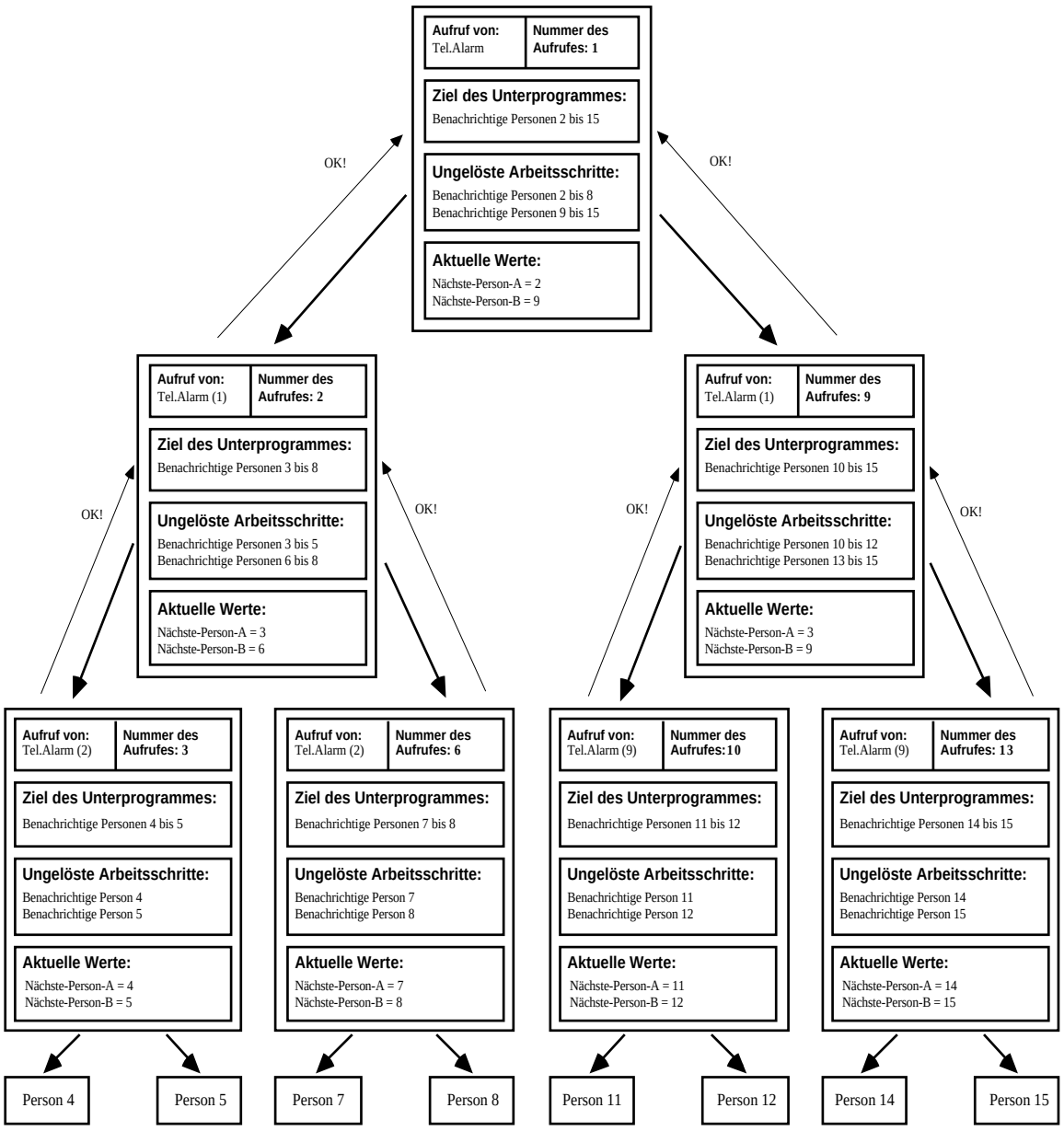
Für die Fibonacci-Zahlen kannst du nun die Buchhaltung mit den Karteikarten ausführen. Spiele selber den Computer und berechne fib(5).

Es ist wichtig, dass du das Beispiel sehr sorgfältig nachvollziehst. Schritt für Schritt. Überspringe nicht sogenannte „offensichtliche“ Dinge (Mundart: „Esch jo logisch...“). Der Computer kann es schliesslich auch nicht!

Zeit für den Kapiteltest, er liegt wie gewohnt bei der Lehrerin oder dem Lehrer für dich bereit.

Lösungen Kapitel 3

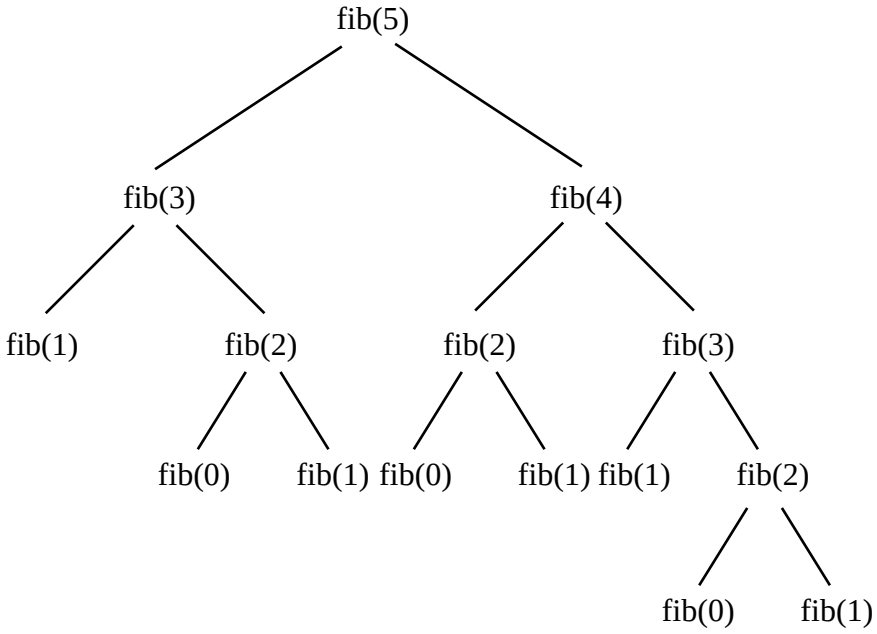
Aufgabe 1



Interessant ist die Reihenfolge der Programmaufrufe, welche durch die Aufruf-Nr. festgelegt ist. Man sieht dabei, wie der Baum Ast für Ast von links nach rechts und von oben nach unten durchlaufen wird.

Wissenssicherung 2

Sicher ist dir aufgefallen, dass du schon bei der Berechnung kleiner Fibonacci-Zahlen recht lange auf das Ergebnis warten musst. Im Baumdiagramm sieht man, dass die Anzahl der Programmaufrufe bei der Berechnung der Fibonacci-Zahlen förmlich explodiert:



Dass dieses Verfahren zur Berechnung der Fibonacci-Zahlen nicht effizient sein kann, leuchtet ein. Zum Beispiel wird fib(2) dreimal berechnet, eine völlig unnötige Arbeit. Zudem werden unnötig viele Karteikarten benötigt, also viel Speicherplatz verschwendet.

Dass unser rekursives Programm nicht speziell effizient, ja eigentlich unbrauchbar ist, braucht dich nicht weiter zu stören. Die Fibonacci-Zahlen kann man viel einfacher iterativ berechnen. Wenn du dich nicht mehr daran erinnerst, wie man rekursiv definierte Folgen effizient berechnet, so schaue im Kapitel 1 nach!

Programmieraufgabe 3

Die Quotienten q(1), q(2), q(3), … nähern sich immer mehr der Zahl 1.618…, das heisst, langfristig verhalten sich die Fibonacci-Zahlen wie eine geometrische Folge mit der Zahl 1.618… als Quotienten zweier aufeinanderfolgender Folgenglieder. Insbesondere kann man daraus folgern, dass die Fibonacci-Zahlen exponentiell wachsen.

1.618… ? Kommt dir diese Zahl bekannt vor? Im Kapitel 1 hast du sie schon angetroffen:

$$\frac{1+\sqrt{5}}{2} = 1.618\ldots$$

Kein Zufall! Diese Zahl hat auch etwas mit dem *goldenen Schnitt* zu tun. Der goldene Schnitt wird in der Architektur verwendet. Aber auch in der Natur kommt er vor: bei Sonnenblumen, Tannenzapfen … Möchtest du mehr wissen? Es gibt unzählige Bücher darüber und vielleicht schaltet deine Lehrerin einmal ein ganzes Kapitel zu diesen Themen ein!

1.618…? Ja, auch die Quotienten p(1), p(2), p(3), … der einzelnen Rechenzeiten nähern sich langfristig dieser Zahl. Der Rechenaufwand bei der rekursiven Berechnung der Fibonacci-Zahlen steigt also auch exponentiell. Du wirst es nie schaffen, fib(100) rekursiv zu berechnen! Wenn du dich genauer für den Rechenaufwand interessierst: Die Anzahl A(n) der rekursiven Aufrufe in unserem Programm wird selber rekursiv beschrieben durch

$$A(n) = A(n-1) + A(n-2) + 2$$

Betrachte das Baumdiagramm, und die Rekursionsvorschrift wird dir sofort klar! Wegen dem Summanden „+ 2“ wächst die Anzahl rekursiver Aufrufe also noch schneller als die Fibonacci-Zahlen.

Aufgabe 4 (Lernkontrolle)

Dein Vorgehen ist sicher richtig, wenn du die Karteikarten gemäss der Baumstruktur in der Lösung zur Wissenssicherung 2 abgearbeitet hast.

4 Rekursiv definierte Kurven

Es braucht immer länger, als man erwartet, sogar wenn man das Hofstadtersche Gesetz berücksichtigt.
Hofstadtersches Gesetz

Übersicht

Was lernst du hier?

Rekursion ist nicht nur eine bewährte Rechenmethode. Sie hat auch in die Grafik als elegantes Verfahren Einzug gehalten. Du hast mittlerweile genügend Kenntnisse über die Rekursion, dass du auch diese grafischen Anwendungen kennenlernen kannst. Nach einer theoretischen Einführung sammelst du praktische Erfahrungen im Bereich der Computergrafik.

Was tust du?

Zunächst faltest du eine interessante Figur. Sie veranschaulicht dir, was rekursive Grafik überhaupt ist. Am Computer wendest du dein Wissen spielerisch an.

Lernziele

Am Ende dieses Kapitels ...

... kennst du einige rekursive Kurven.

... kannst du mit Grafikhilfsmitteln am Computer umgehen.

... stellst du rekursive Figuren auf dem Bildschirm dar.

4.1 Drachenkurven (Teil 1)

Aufgabe 1

Nimm ein A4-Blatt und schneide einen etwa 2 cm breiten Streifen aus.

0. Falte den Papierstreifen einmal in der Mitte. Entfalte ihn wieder, bis die zwei Hälften einen rechten Winkel bilden. Stelle ihn dazu hochkant auf den Tisch. Das ist noch kein sehr interessantes Gebilde. Also:
1. Falte den Streifen wieder ganz zusammen zu einem neuen Ausgangsstreifen.
2. Falte den Streifen zusätzlich in der Mitte. (Er ist jetzt doppelt so dick wie vorher.)
3. Entfalte den Streifen wieder. Und zwar so, dass alle benachbarten Teilstücke einen rechten Winkel zueinander bilden. Am besten stellst du das Gebilde wieder hochkant auf den Tisch.
4. Skizziere auf einfache Art den Umriss des entstandenen Gebildes.

Führe die Punkte 1 bis 4 vier- oder fünfmal aus.

Deine aufgezeichneten Figuren nennt man *Drachenkurven*. Vergleiche deine Zeichnungen mit denjenigen einer Mitschülerin! Vielleicht sind eure Drachenkurven verschieden. Das hängt davon ab, in welche Richtung jeweils gefaltet wurde und ist für unsere Betrachtung unwichtig. Die Gemeinsamkeit aller Kurven ist die *Symmetrie* bezüglich des Mittelpunktes des Papierstreifens. Du stellst schnell fest:

a) Die halbe Kurve wiederholt sich - um 90° gedreht - nach der Papierstreifenmitte.

b) Die halbe Kurve hat eine ähnliche Form wie die ganze Drachenkurve. Sie ist demnach auch eine Drachenkurve!

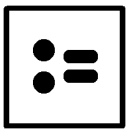
Erkenntnis: Eine Drachenkurve besteht aus Drachenkurven!

Die Feststellungen a) und b) können wir auch so zusammenfassen:

Rekursionsbasis
Der ungefaltete Papierstreifen ist eine Drachenkurve

Rekursionsvorschrift
Wenn zwei Kopien einer Drachenkurve senkrecht aneinandergefügt werden, entsteht wieder eine Drachenkurve.

Dieses *Rekursionsschema,* welches du seit Kapitel 1 kennst, kommt auch bei der Grafik zur Anwendung. Etwas allgemeiner können wir sagen:



Definition

Eine *rekursive Figur* enthält sich selbst als Teil.

Wir möchten nun rekursive Figuren wie die Drachenkurve auf dem Computerbildschirm darstellen. Der folgende Abschnitt verrät uns die nötigen Techniken der Computergrafik.

4.2 Turtle-Grafik

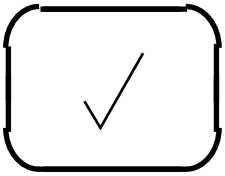
Was jetzt folgt, ist ein wichtiges Hilfsmittel zur Erzeugung von Grafiken auf dem Computer: die *Turtle-Grafik.* „Turtle“ ist das englische Wort für „Schildkröte“ – ein Tier, das Kindern gefällt. Diese „Schildkröten“-Grafik wurde für die ersten Gehversuche von Primarschülern auf Computern entwickelt. Es hat sich aber bald gezeigt, dass sie ein gutes Hilfsmittel zur Erstellung von Zeichnungen auf dem Bildschirm darstellt. Darum hat sich die Idee der Turtle-Grafik auch ausserhalb der Primarschule durchgesetzt.

Die Idee ist folgende: Der Bildschirm entspricht einem Blatt Papier. Darauf steht eine kleine Schildkröte mit einem Stift. Dieser Schildkröte können wir nun den Befehl geben, herumzugehen und zu zeichnen. Ein Beispiel:

Wir geben der Schildkröte die 4 Befehle:

Turn (-60)
Forward (20)
Turn (120)
Forward (60)

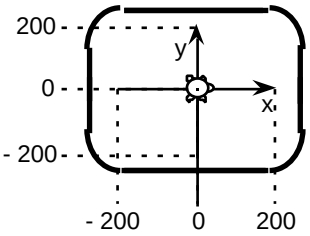
und sie zeichnet einen Haken auf den Schirm:



Alles klar? Wenn nicht, dann schaue dir doch die folgende Zusammenstellung aller Befehle an, welche die Schildkröte versteht:

GraphicInit Dieser Befehl dient dazu, den Bildschirm auf die Schildkröte einzurichten. Er stellt quasi ein neues, leeres Zeichenfeld zur Verfügung. Diesen Befehl werden wir in Zukunft an den Anfang aller unserer Programme stellen, damit wir dann der (unsichtbaren) Schildkröte Befehle geben können.

Als Folge dieses Befehls wird auch ein (ebenfalls unsichtbares) Koordinatensystem eingerichtet. Dieses hat feste Bereiche, in denen sich die Schildkröte bewegen darf. Diese Bereiche hängen von der Programmierung der Turtle-Grafik auf deinem Schulcomputer ab. In den Beispielen gehen wir einmal von den folgenden Begrenzungen aus:



- Der Bildschirm (oder, je nach Computer, das Fenster) hat den Nullpunkt in der Mitte.
- Die x- und die y-Koordinaten liegen beide zwischen -200 und 200. Wir haben also ein quadratisches Bildfeld mit 400 Bildpunkten Seitenlänge.

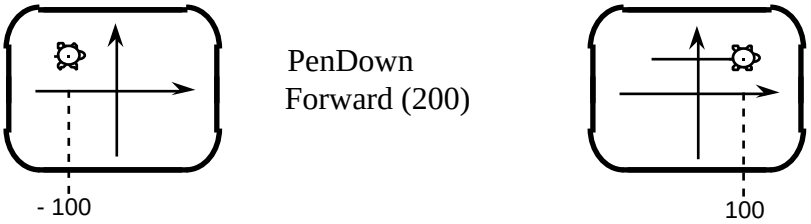
Zusätzlich gelten folgende zwei Anfangszustände:

- Die Schildkröte befindet sich in der Mitte des Bildschirms, im Punkt (0,0)
- Sie schaut nach rechts (= 0 Grad).

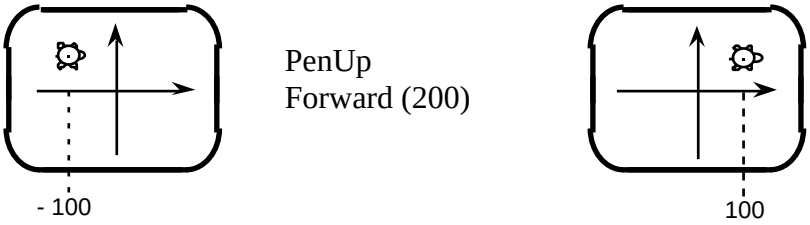
Lass dich nicht verwirren, wenn die Bereiche und Anfangsbedingungen bei dir anders gewählt sind. Das Prinzip ist das gleiche.

PenDown Wenn die Schildkröte diesen Befehl erhält, setzt sie ihren Stift aufs Papier bzw. auf den Schirm. Beim Herumgehen hinterlässt sie eine Linie.

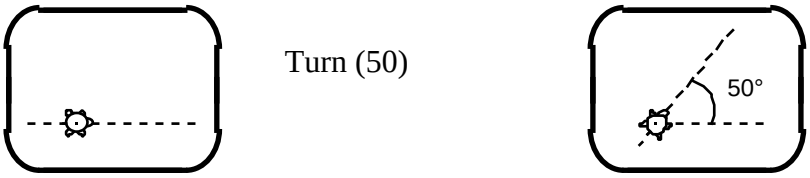
Forward (*Dist*) Die Schildkröte bewegt sich um *Dist* vorwärts. Wenn der Stift „down“ ist, zeichnet sie dabei eine Linie.



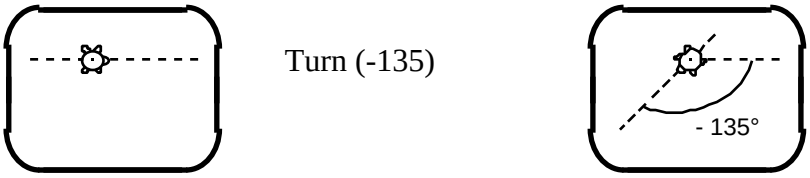
PenUp Hier nimmt, wer hätte es gedacht, die Schildkröte ihren Stift vom Papier. Wenn sie danach einen Forward-Befehl erhält, bewegt sie sich, ohne zu zeichnen.



Turn (*Grad*) Dieser Befehl veranlasst die Schildkröte, sich um den Winkel *Grad* zu drehen. Der Parameter *Grad* liegt zwischen -360 und 360°. Positive Winkel bewirken eine Drehung nach links, negative nach rechts. Zwei Beispiele:



also eine Drehung nach rechts, sowie



Damit das Programmieren noch etwas komfortabler wird, gibt es dazu noch die folgenden zwei Befehle:

SetPosition (*x*,*y*) Mit SetPosition setzt man die Schildkröte an einen beliebigen Ort auf dem Bildschirm. Dabei zeichnet sie nicht. Die Parameter *x* und *y* liegen im oben beschriebenen Bereich.

SetDirection (*Grad*) Dieser Befehl ähnelt dem Befehl **Turn**. Mit SetDirection wird die Blickrichtung der Schildkröte auf den angegebenen Wert eingestellt. *Grad* liegt, wie gehabt, zwischen -360 und 360.

Diese sieben Befehle reichen aus für ziemlich komplizierte Zeichnungen; wir werden auch in den folgenden Beispielen mit diesen Befehlen auskommen. Was oben über die Standards steht, gilt auch hier. Vielleicht hast du auf deinem Computer eine Turtle-Grafik, wo die Befehle anders heissen. Es gibt beispielsweise Turtle-Grafik-Pakete, in denen der Befehl **Turn** durch die zwei Befehle **Left** und **Right** ersetzt ist, also durch Befehle, welche die Schildkröte nach links oder rechts drehen. Darum die folgende

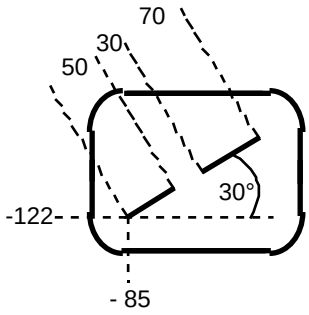
Aufgabe 2

Kläre ab, wie die Turtle-Befehle auf deinem Computer genau heissen. Schreibe diese in eine Tabelle, zusammen mit den entsprechenden Befehlen dieses Leitprogramms.

Nun bist du gerüstet für eine Anwendung der Befehle:

Programmieraufgabe 3

Zeichne mit Turtle-Grafik die unterbrochene dicke Linie der folgenden Zeichnung auf den Schirm:



Es ist die Linie mit dem Anfangspunkt (-85,-122), der Richtung 30° und den in der Zeichnung angegebenen Längen.

Hier noch eine Aufgabe:

Programmieraufgabe 4

Zeichne mit Turtle-Grafik ein regelmässiges Fünfeck auf den Bildschirm.
Den Startpunkt und die Seitenlänge kannst du frei wählen. Du kannst aber auch darauf achten, dass der Mittelpunkt des Fünfecks bei (0,0) liegt.

4.3 Drachenkurven (Teil 2)

Du hast eben die „Turtle-Bibliothek“ ausprobiert und bist jetzt schon ein kleiner Schildkrötenspezialist. Du hast nun genug Wissen, um dich der Programmierung von Drachenkurven zu widmen.

Rufen wir uns die Definition der Drachenkurve in Erinnerung:

- Eine Strecke (der ungefaltete Papierstreifen) ist eine Drachenkurve. Diese Strecke ist die *Basis*.

  Wir sagen auch: Eine Strecke ist eine Drachenkurve mit der *Rekursionstiefe* 0.

- Bezeichnen wir mit D eine Drachenkurve. Fügen wir an einem Ende eine Kopie an, und zwar im rechten Winkel, dann entsteht wieder eine Drachenkurve. Für diese zusammengesetzte Kurve führen wir ein Zeichen ein:

  D ◁ D (◁ : rechter Winkel)

  Zu dieser *Rekursionsvorschrift* kommt noch hinzu: Nach jeder Anwendung der Rekursionsvorschrift vergrössert sich die Rekursionstiefe um 1.

Rekursionstiefe: 0 1 2 … …

Betrachte noch einmal das Bild auf Seite 4. Es ist eine Drachenkurve der Tiefe …? Die Tiefe verraten wir nicht! Mit ein wenig Phantasie kannst du übrigens den „Seedrachen“ erkennen, nach dem diese Figur benannt worden ist.

Frage: Wie sieht die Drachenkurve mit der Rekursionstiefe 3 aus?

Die Rekursionsvorschrift sagt uns:

Drachen (3) = Drachen (2) ◁ Drachen (2).

Drachen (2) = Drachen (1) ◁ Drachen (1).

Drachen (1) = Drachen (0) ◁ Drachen (0)

Aha! - Drachen (0) kennen wir doch. Wir sind bei der Basis angelangt!

Drachen (0) ist die Strecke ———.

Setzen wir in die obige „Rekursionstreppe“ rückwärts ein! Wir erhalten:

Drachen (0):

Drachen (1): 

Drachen (2): 

Drachen (3): 

Für das Zeichnen der Drachenkurve entwickeln wir schrittweise ein Computerprogramm. Wir verwenden dabei die *Turtle-Grafikbefehle*, die du vom letzten Abschnitt her kennst. Im folgenden bezeichnen wir die Rekursionstiefe mit *t*. Die Frage ist: Wie sieht der allgemeine Drachen (*t*) aus?

- Falls $t = 0$, dann ist Drachen (*t*): *Basis*

  Der Computer soll also in diesem Fall bloss eine Linie zeichnen. Wie lang diese ist, ist vorläufig egal. Bezeichnen wir ihre Länge einfach einmal mit *s*. Der entsprechende Turtle-Befehl lautet hier:

  Forward (*s*) **.**

- Sonst ($t > 0$) ist Drachen (*t*) = Drachen (*t* -1)  Drachen (*t* -1) *Vorschrift*

  Der Computer muss eine Drachenkurve mit Rekursionstiefe *t* -1 zeichnen, danach den Zeichenstift um 90° drehen und wieder eine Drachenkurve mit Rekursionstiefe *t* -1 zeichnen. Die Befehlsfolge dazu lautet:

  Drachen (*t* -1) zeichnen
  Turn (90)
  Drachen (*t* -1) zeichnen

Es gibt jetzt aber noch ein Problem: Wir können die Drachen(*t* -1) nicht beide Male genau gleich zeichnen lassen! Wieso?

Beim ersten Drachen (*t* -1) zeichnen wir *von aussen her* gegen die Mitte des späteren Drachen (*t*). Dann drehen wir um 90°. Den zweiten Drachen (*t* -1) zeichnen wir hingegen von der Mitte des Drachen (*t*) *nach aussen hin!*

Was ist zu tun? - Damit beide Drachen (*t* -1) deckungsgleich herauskommen, muss der zweite Drachen spiegelverkehrt gezeichnet werden! Das heisst, dass alle seine Winkel

(a) *gespiegelt* liegen und

(b) in *umgekehrter Reihenfolge* gezeichnet werden müssen.

Wissenssicherung 5

Überprüfe die eben gemachte Aussage anhand deiner Drachenskizzen! Wenn die Aussage mit deinen Skizzen übereinstimmt, ist alles in Ordnung. Wenn nicht, dann hast du wohl einen Fehler beim Zeichnen gemacht.

Wie erreichen wir nun das Gewünschte? Der Trick ist einfach: Wir führen ein *Vorzeichen vz* ein! (*vz* = 1 heisst „+“, *vz* = -1 bedeutet „-“). Zuerst lassen wir einen „gewöhnlichen“ Drachen (t -1) mit positivem Vorzeichen (*vz* = 1) entstehen. Dann drehen wir um 90° und zeichnen einen Drachen (*t* -1) mit negativem Vorzeichen (*vz* = -1). Bei diesem kehren wir die Winkel um, was sich auf den Turn-Befehl auswirkt. Also:

Drachen (*t* -1) (+) zeichnen
Turn (vz • 90)
Drachen (*t* -1) (-) zeichnen

Das Pseudocode-Programm sieht nun folgendermassen aus:

```
PROCEDURE  ZeichneDrachen (t, vz)
BEGIN
    IF  t = 0  THEN  Forward (s)
    ELSE
        ZeichneDrachen (t -1, 1)
        Turn (vz * 90)
        ZeichneDrachen (t -1, -1)
    END
END
```

Wenn du zum Beispiel die Drachenkurve mit Rekursionstiefe 3 kennen möchtest, rufst du das Unterprogramm aus dem Hauptprogramm auf mit

ZeichneDrachen (3, 1) oder **ZeichneDrachen (3, -1)**.

Der Aufruf mit negativem Vorzeichen ergibt einen spiegelverkehrten Drachen. Diese Vorzeichengeschichte ist nicht ganz einfach einzusehen. Wenn du damit Probleme hast, darfst du ruhig deine Lehrerin zu Rate ziehen.

Noch ein Wort zur Länge s der Strecke, die jeweils mit Forward(s) gezeichnet wird:

Damit unser Drachen nicht immer grösser wird, sollten wir bei jedem rekursiven Aufruf der Prozedur **ZeichneDrachen** die Streckenlänge halbieren. Es lohnt sich deshalb, die Streckenlänge s als Parameter der Prozedur zu übergeben.

```
PROCEDURE  ZeichneDrachen (t, vz, s)
BEGIN
    IF  t = 0  THEN  Forward (s)
    ELSE
        ZeichneDrachen (t -1, 1, s/2)
        Turn (vz * 90)
        ZeichneDrachen (t -1, -1, s/2)
    END
END
```

Der Aufruf aus dem Hauptprogramm sieht zum Beispiel folgendermassen aus:

ZeichneDrachen (5, 1, 10).

Programmieraufgabe 6

Schreibe ein Programm in deiner vertrauten Computersprache, welches die Drachenkurve auf den Bildschirm zeichnet. Gib das Programm in den Computer ein und experimentiere etwas damit.

Zeichnet dein Programm die Drachenkurve etwas langsam auf den Bildschirm? Rekursive Algorithmen sind häufig nicht sehr effizient. Wenn du Lust und Zeit hast, kannst du dir auch eine iterative Lösung zur Drachenkurve überlegen. Du kannst nämlich die Drachenkurve durch eine Folge von Befehlen der Art „links, links, rechts....“ beschreiben. Überlege dir zuerst, nach welcher Gesetzmässigkeit diese Befehlsfolge bei der Drachenkurve mit Rekursionstiefe k aufgebaut werden kann. Anschliessend kannst du die Befehlsfolge beispielsweise als Folge von „l“ und „r“ in einer Zeichenkette abspeichern. Nach Bedarf kannst du diese Zeichenkette dann ganz einfach auf dem Bildschirm graphisch darstellen.

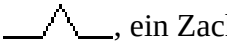
4.4 KOCH-Kurven

Eine häufige Aufgabenstellung in der Computergrafik besteht darin, eine Strecke durch eine vorgegebene Form zu ersetzen. Diese heisst *Grundfigur*. Man erhält so eine neue Kurve. In dieser ersetzt man ebenfalls alle Strecken durch die (verkleinerte) Grundfigur. Man erhält eine neue Kurve. In dieser ersetzt man wieder alle Strecken durch die (verkleinerte) Grundfigur. Man erhält eine neue Kurve. In dieser ersetzt man ...

Figuren, welche nach diesem Muster gebildet werden, nennt man KOCH-Kurven. Sie sind nach dem schwedischen Mathematiker Helge von Koch benannt. Er beschäftigte sich um die Jahrhundertwende mit solchen Figuren.

Beispiel: Ersetze durch  .

- Die feste Strecke ist hier die *Basis.*
  Die Computergrafiker nennen die Basis auch *Initiator* .

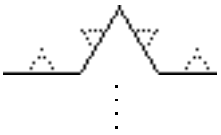
- Die Grundfigur  , ein Zacken, wirkt als *Rekursionsvorschrift* .
  Diese wird auch *Generator* genannt.

In diesem Beispiel ersetzen wir also zuerst die gegebene Strecke durch einen Zacken. Im nächsten Schritt ersetzen wir jede Strecke der neuen Figur durch proportional verkleinerte Zacken usw.

Die folgende Zusammenstellung zeigt dir, wie sich die KOCH-Kurve schrittweise entwickelt.

Initiator:

Der *Generator* bewirkt:  Kurve nach dem 1. Schritt: 

Der *Generator* bewirkt:  Kurve nach dem 2. Schritt: 

Bei jedem Schritt vergrössert sich die *Rekursionstiefe*. Der Initiator hat die Rekursionstiefe 0. Nach dem ersten Generator-Schritt hat die KOCH-Kurve die Rekursionstiefe 1, nach dem zweiten Schritt die Tiefe 2 usw.

Tiefe: 0 1 2 3 ...

Wissenssicherung 7

Nun bist du an der Reihe: Zeichne selber eine KOCH-Kurve!
Gegeben ist wieder die Strecke ——— als Initiator.

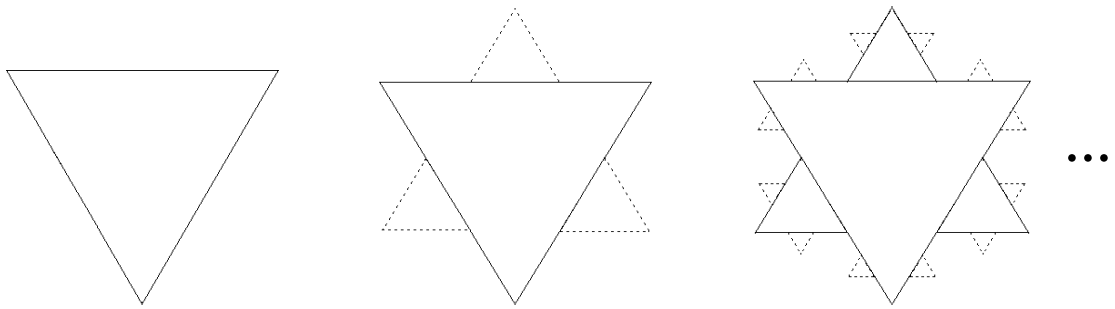
Als Generator verwendest du nun die Grundfigur: ⎍ .
Skizziere, wie sich die KOCH-Kurve bis und mit Rekursionstiefe 2 entwickelt.

Programmieraufgabe 8

Schreibe ein Programm auf deinem Computer, das eine Kochkurve auf den Bildschirm zeichnet. Du kannst den Generator frei wählen, es kann irgendeine Figur sein – etwa ein Zacken oder eine Kurve wie in der letzten Aufgabe. Überlege dir zuerst genau die folgenden Punkte:

- Wohin auf den Bildschirm soll die Kurve zu liegen kommen?
- Wie geht die Rekursion genau? Schreibe das Programm unbedingt zuerst in Pseudocode auf, bevor du es in den Computer eingibst. Vergleiche dein Programm mit dem einer Nachbarin. Wenn ihr grosse Probleme habt, könnt ihr die Lehrerin zu Hilfe rufen.

Die bisherigen Kochkurven waren zwar ganz schön, das Nonplusultra waren sie indes noch keineswegs. Wir nähern uns jetzt nämlich dem ersten Höhepunkt dieses Leitprogramms, den Schneeflockenkurven. Das sind rekursive Kurven, welche die Form einer Schneeflocke annehmen. Die (kleine aber feine) Idee dahinter ist es, mehrere Initiatoren zu einer Figur zusammenzusetzen, etwa zu einem Dreieck.



Tiefe 0 Tiefe 1 Tiefe 2 und so weiter

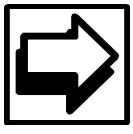
Es wurden also drei Initiatoren zu einem Dreieck zusammengefügt. Dann wurde unser bekanntes Zackenprogramm mit Rekursionstiefe 3 ausgeführt.

Nicht zu verleugnen, die Verwandtschaft von Mathematik und Natur …

Lernkontrolle

Programmieraufgabe 9

Erweitere das Programm aus der letzten Aufgabe so, dass es Schneeflockenkurven zeichnet. Schreibe die neuen Teile deines Programms auch hier zuerst in Pseudocode auf. Wenn du Lust und Zeit hast, kannst du Initiator und Generator variieren!



Es ist soweit, der Kapiteltest ruft.

Lösungen Kapitel 4

Aufgabe 1

Wenn du Mühe mit dieser Aufgabe hattest, so besprich dein Problem mit einer Kollegin. Mit vereinten Kräften sollte diese Aufgabe dann zu schaffen sein.

Aufgabe 2

Wenn es sie überhaupt braucht, könnte diese Tabelle etwa so aussehen:
Die Turtle-Befehle:

Im Leitprogramm	*Auf meinem Computer (Beispiel)*
• **GraphicInit**	• OpenWindow
• **PenDown**	• PenDown
• **PenUp**	• PenUp
• **Forward** (Dist)	• Draw (Dist)
• **Turn** (Grad)	• Left / Right (Grad)
• **SetPosition** (x, y)	• SetPosDir (x, y, Grad)
• **SetDirection** (Grad)	•
	• Circle (r)
Weitere Befehle:	• Rectangle (x, y)
	• ...

Programmieraufgabe 3

In unserem Pseudocode sieht die Lösung zum Beispiel so aus:

Pseudocode	*Bemerkung*
GraphicInit	Initialisieren
SetPosition (-122,-85)	Stift zum Ausgangspunkt
SetDirection (30)	Richtung auf 30° einstellen
PenDown	Stift senken
Forward (50)	Linie der Länge 50 zeichnen
PenUp	Stift heben
Forward (30)	Stift ohne zu zeichnen bewegen
PenDown	Stift senken
Forward (70)	Linie der Länge 70 zeichnen

Programmieraufgabe 4

Das Kernstück des Programms ist eine Schleife, welche die fünf Seiten des Fünfecks mit der gewünschten Kantenlänge zeichnet.

```
FOR  i = 1  TO  5
     Forward (Kantenlänge)
     Turn (72)
END
```

Wissenssicherung 5

Wenn die Aussage mit deinen Skizzen übereinstimmt, ist alles in Butter. Sonst hast du wohl einen Fehler gemacht beim Zeichnen. Wenn man richtig faltet und abzeichnet, *muss* etwas herauskommen, das die Aussage erfüllt.

Programmieraufgabe 6

```
PROCEDURE  ZeichneDrachen (t, vz, s)
BEGIN
     IF  t = 0  THEN  Forward (s)
     ELSE
          ZeichneDrachen (t -1, 1, s/2)
          Turn (vz * 90)
          ZeichneDrachen (t -1, -1, s/2)
     END
END
```

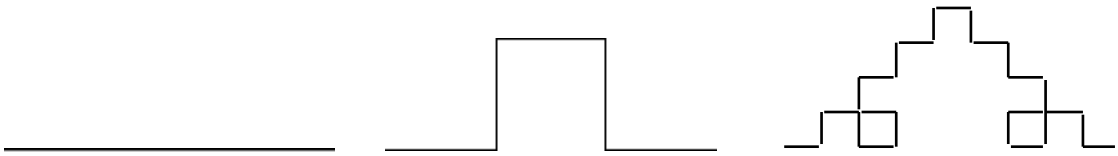
Man bettet die Prozedur **ZeichneDrachen** in ein Hauptprogramm ein, das diese dann aufruft: In PASCAL sieht das Programm zum Beispiel folgendermassen aus:

```
PROGRAM  Drachen;
{$I TURTLE.PAS}
VAR s, rektiefe, i: INTEGER;

PROCEDURE  ZeichneDrachen (t, vz, s: INTEGER);
BEGIN
     IF  t = 0  THEN  Forward (s)
     ELSE
          BEGIN
          ZeichneDrachen (t-1, 1, s/2);
```

```
            Turn (vz * 90);
            ZeichneDrachen (t-1, -1, s/2)
        END
    END;

    BEGIN
        Write („Rekursionstiefe =„); ReadLn (rektiefe);
        s:= 64;          (* Startwert von s; bestimmt die Grösse der Figur *)
        GraphicInit;
        SetPosition (-100,-100)   (* Schildkröte an die Startstelle bewegen *)
        PenDown;
        ZeichneDrachen (rektiefe, 1)          (* Rekursion starten, vz=1*)
    END.
```

Wissenssicherung 7

Die quadratische KOCH-Kurve sieht so aus:



Tiefe 0 Tiefe 1 Tiefe 2

Programmieraufgabe 8

```
    PROCEDURE  Zacken (t, s)
    BEGIN
        IF  t = 0  THEN  Forward (s)
        ELSE
            Zacken (t -1, s/3)   Turn (60)
            Zacken (t -1, s/3)   Turn (-120)
            Zacken (t -1, s/3)   Turn (60)
            Zacken (t -1, s/3)
        END
    END
```

Wieder steht s für die Länge einer Strecke eines Zackens. Das Hauptprogramm aus der Programmieraufgabe 7 bleibt gleich. Es muss lediglich die Prozedur **ZeichneDrachen** durch **Zacken** ersetzt werden.

Programmieraufgabe 9 (Lernkontrolle):

Die Schneeflocke wird aus drei Zacken zusammengesetzt. Die Prozedur **Zacken** bleibt gleich. Es muss lediglich das Hauptprogramm geändert werden.

```
BEGIN (* Hauptprogramm *)

    (* Startposition Stift vorbereiten *)
    GraphicInit
    Eingabe Rekursionstiefe t, Kantenlänge s
    SetPosition (x,y)
    SetDirection (Grad)
    PenDown

    (* Zeichnen des Anfangsdreiecks *)
    Zacken (t,s)
    Turn (120)
    Zacken (t,s)
    Turn (120)
    Zacken (t,s)

END
```

Natürlich kannst du die drei Aufrufe der Prozedur **Zacken** in einer Schleife zusammenfassen. Das lohnt sich insbesondere, wenn du anstelle des gleichseitigen Dreiecks zum Beispiel ein reguläres Sechseck als Ausgangsfigur nimmst.

5 Erweiterte Grafik

Ich bin, der ich bin.
Exodus, 3, 14

Übersicht

Was lernst du hier?

Dieses letzte Kapitel richtet sich an schnelle oder interessierte Leserinnen. Es wird hier mehr experimentiert als Wissen vermittelt. Du wirst spezielle Kurven programmieren und sie nicht nur betrachten, sondern auch ein wenig hinterfragen.

So ist es beispielsweise interessant, wie sich Kurven bei zunehmender Rekursionsstufe verhalten und welche Eigenschaften sie haben. Dazu stellen wir im zweiten Teil einige Fragen. Es ist dir aber freigestellt, ob du in die Materie der rekursiven Grafik durch zusätzliches Literaturstudium tiefer eindringen willst oder nicht.

Auf alle Fälle lohnt es sich!

Was tust du?

Je nachdem wie intensiv du die zweite Hälfte bearbeitest, dauert dieses Kapitel kürzer oder länger:

- Du programmierst die drei Kurven im ersten Teil des Kapitels.
- Die Fragen im zweiten Teil kannst du alleine oder mit Kollegen bearbeiten.
- Den endgültigen Durchblick erhältst du mit einem individuellen Fachstudium.

Lernziele

- Du lernst, schwierigere Grafiken zu programmieren.
- In diesem Kapitel kannst du dir auch selber Ziele setzen!

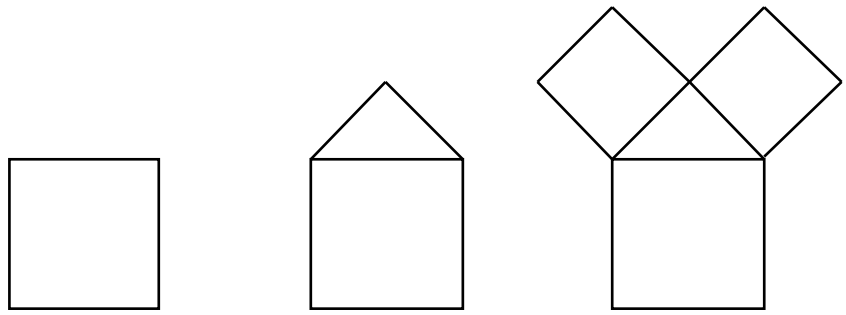
Und nun auf zu einem alten Bekannten, dem Herrn Pythagoras …

5.1 Drei ungewöhnliche Kurven

Der Pythagorasbaum

Wir kennen alle die berühmte Formel : $a^2 + b^2 = c^2$. Aus dieser Idee lässt sich aber auch eine wunderbare Grafik entwickeln, indem wir folgende Konstruktion rekursiv programmieren:

1. Zeichne ein Quadrat.
2. Füge ein rechtwinkliges, gleichschenkliges Dreieck an einer Seite an.
3. Füge an jeder freien Dreiecksseite ein Quadrat an.



1. Schritt 2. Schritt 3. Schritt

An jedem so neu entstandenen Quadrat werden diese Schritte wiederholt, und so weiter...

Aufgabe 1

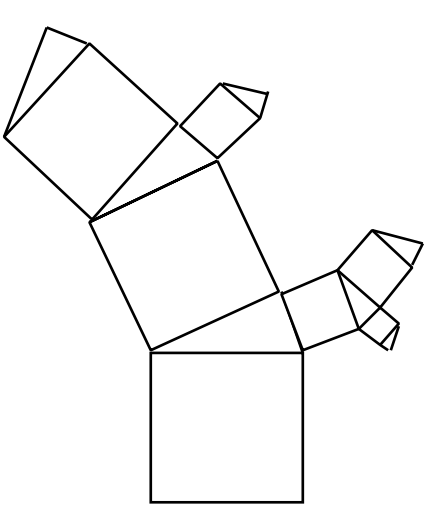
Setze die obige Konstruktion mit den „Turtle“- Befehlen um.
Die Grösse des Quadrates sollte 1/6 der Bildschirmbreite nicht überschreiten.

Versuche anschliessend, das ganze rekursive Programm in Pseudocode aufzuschreiben. Nachdem du dein Pseudocodeprogramm mit einem Mitschüler verglichen hast, kannst du es in deiner Programmiersprache implementieren.

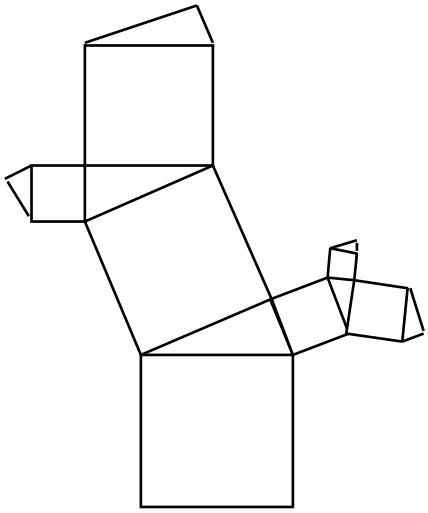
Interessante Variationen des Pythagorasbaumes lassen sich wie folgt erzielen:

a) Es wird nicht ein gleichschenkliges, sondern irgendein rechtwinkliges Dreieck an das Quadrat angefügt.

b) Ein beliebiges rechtwinkliges Dreieck wird angefügt und zwar mit immer wechselnder Orientierung zum Quadrat -> vergleiche mit folgendem Bild!

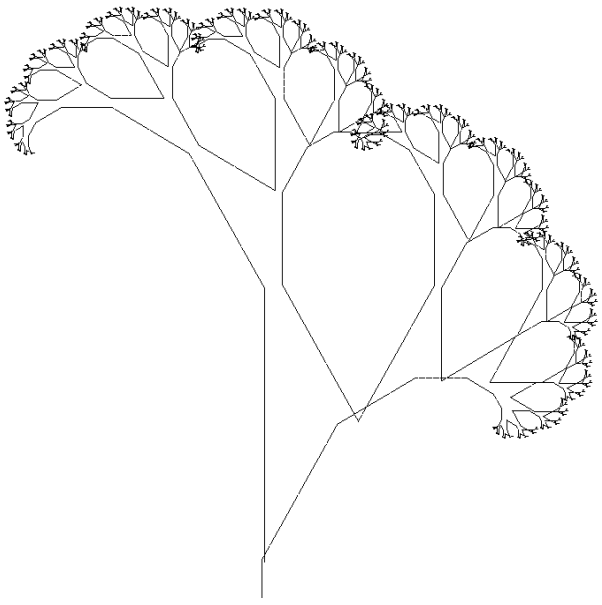


Variation a)



Variation b)

Vielleicht wagst du dich auch an das *Blumenkohlexperiment* : Untersuche die Struktur von einem Blumenkohl, und versuche, ein ebenes Bild davon zu erzeugen. Der Pythagorasbaum muss nur geschickt variiert werden …



Für Feinschmecker empfiehlt sich besonders der Broccoli!

Der Cantor-Staub

Vom deutschen Mathematiker Georg Cantor (1845 - 1918) stammt ein weiteres interessantes Gebilde mit folgendem Prinzip:

1. Man nimmt eine beliebige Strecke, sagen wir der Länge 1.
2. Man entfernt das mittlere Drittel.
3. Das Ganze wird mit den übrigen Streckenteilen wiederholt.

Vielleicht ahnst du hier schon, wie das Gebilde sich bei zunehmender Rekursionstiefe verhält – jedenfalls gibt der Name „Cantor-Staub“ einen Anhaltspunkt. Bevor du aber weitergehst, hilft dir vielleicht eine kleine Handskizze bis etwa zur Rekursionstiefe 5, um den Verlauf der Grafik zu erkennen.

Aufgabe 2

Setze diese Konstruktion mit den „Turtle“-Befehlen um. Dazu kannst du statt einer horizontalen Linie als Ursprungsfigur einen gleich langen, aber etwa einen halben Bildschirm hohen Balken nehmen und dann die Schritte eins und zwei anwenden.

Versuche anschliessend das ganze rekursive Programm in Pseudocode aufzuschreiben.

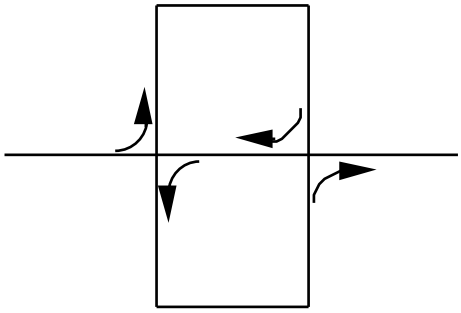
Nachdem du dein Pseudocodeprogramm mit einer Mitschülerin verglichen hast, kannst du es in deiner Programmiersprache implementieren.

Die Peano-Kurve

Du kennst das Prinzip des Initiators und Generators. Bei der Peano-Kurve sieht dies wie folgt aus:

Initiator

Generator



Die Pfeile im Generatorbild bezeichnen die Durchlaufrichtung des Zeichnungsstiftes.

Aufgabe 3

Nachdem du ein Pseudocodeprogramm für die Peano-Kurve aufgeschrieben hast, kannst du es in deiner Programmiersprache implementieren.

Nimm als Initiator eine Strecke in der Bildschirmmitte mit 2/3 Bildschirmlänge!

Schau dir bei dieser Kurve vor allem an, wie die Grafik entsteht. Versuche auch hier zu erkennen, was bei zunehmender Rekursionstiefe geschieht. Probiere einfach einmal Rekursionstiefe 6 aus.

5.2 Wieso sind diese Kurven ungewöhnlich?

Eine ganz grundsätzliche Frage stellt sich uns hier: Was sind denn eigentlich diese Grafiken? Ist es nun eine (geordnete) Ansammlung von Punkten, oder vielleicht eher von Strecken, oder ist es gar eine Fläche ?

Schauen wir uns dazu die Cantor-Staub-Kurve an. Sie wird bei jedem Rekursionsschritt um einen Drittel verkürzt. Einfacher gesagt: Am Anfang ist es eine Strecke, dann werden es mehrere, immer kleinere Strecken, bis bald nur noch Punkte erkennbar sind. Nehmen wir hier nun eine Lupe zur Hand, die alles 10 x grösser darstellt, so sehen wir wieder kleine Strecken. Diese Geschichte können wir wiederholen und immer mehr Rekursionsschritte durchführen, eine noch stärkere Lupe nehmen, und so weiter.

Wir können uns natürlich auch sagen, dass diese Kurve einfach ein „Zwischending“ zwischen einem Punkt und einer Strecke ist. Dies bedeutet aber, dass es weder der Dimension 0 (= Punkt) noch der Dimension 1 (= Strecke/Linie) zugeordnet werden kann!

Jetzt wird es aber gefährlich, denn bis jetzt wurden in der Geometrie ja nur die Dimensionen 0 bis 3 zugelassen: Punkt, Linie, Fläche, Körper.

Mit der obigen Erkenntnis müssen wir nun aber auch ein Zwischending oder eben eine gebrochene Zahl zwischen 0 und 1 berücksichtigen. Solche Kurven nennt man daher auch *Fraktale* (Gebrochene). Die Cantor-Staub-Kurve erhält zum Beispiel eine Dimension von 0.6309...

Willst du wissen, wie wir auf 0.6309... kommen, so ist es deiner Initiative überlassen, ein Fachbuch zur Hand zu nehmen, um mehr darüber zu erfahren. Fürs Studium empfehlen wir die Bücher am Ende dieses Kapitels.

Bevor du dich aber auf die Bücher stürzt, überlege folgendes:

Es gibt natürlich auch Kurven, deren Dimension irgendwo zwischen der Dimension einer Linie und einer Fläche liegt, beispielsweise die Kochkurve oder der Pythagorasbaum. Sie alle füllen eine begrenzte Fläche nie vollständig aus. Daher können es rein anschaulich auch keine Flächen sein. Da sie aber an einigen Stellen (vorwiegend an den Rändern) durch zunehmende Rekursionsschritte immer flächendeckender werden, sind es auch keine richtigen Linien mehr.

Nehmen wir aber die Peano-Kurve, so sehen wir, dass sie ganz gleichmässig eine Fläche in der Form eines Quadrates ausfüllt. Je grösser die Rekursionstiefe wird, desto besser bedeckt sie die Fläche. Anders ausgedrückt: Der Zeichenstift berührt bei unendlicher Rekursionstiefe *jeden* Punkt dieses Quadrates. Das heisst, die Peano-Kurve wird zu einer Fläche und bekommt daher die Dimension 2!

Hier bist du nun am Ende dieses Leitprogrammes angelangt. Falls du neugierig geworden bist auf das Zauberwort *Fraktale* oder *Computergrafik* ganz generell, wünschen wir dir beim Lesen oder Durchblättern der weiterführenden Literatur viel Spass!

Lösungen Kapitel 5

Hast du Probleme mit dem Pythagorasbaum? Will die Cantor-Strecke nicht richtig zu Staub zerfallen? Gibt es bei der Peano-Kurve ungewollte Verwicklungen? Dann musst du dich jetzt leider selbst nochmals auf Fehlersuche begeben. Nicht nur Peano-Kurven können sich verwickeln; auch unsere Hirnwindungen neigen gerade beim rekursiven Programmieren zur Knotenbildung. Es werden deshalb in diesem Kapitel bewusst keine Lösungen angegeben. Nicht alle Programme führen immer sofort zum gewünschten Resultat …

Ein kleiner Trost:

Ganz harmlos sind diese Aufgaben nicht mehr. Der Pythagorasbaum war eine der Aufgaben beim Bundeswettbewerb Informatik 1986 in Deutschland. Deine Lehrerin hat die Aufgabenstellung samt Musterlösung.

Weiterführende Literatur:

Beck U.:

Computer-Grafik, Bilder und Programme zu Fraktalen,
Chaos und Selbstähnlichkeit

Birkhäuser Verlag, Basel, Boston , 1988

„Ein verständlich geschriebenes Buch, das speziell beim Programmieren eine grosse Hilfe ist."

Peitgen H. O., Jürgens H., Saupe D.:

Fraktale: Ein Arbeitsbuch. Springer / Klett Verlag, 1992
Chaos: Ein Arbeitsbuch. Springer / Klett Verlag, 1992

„Populärwissenschaftlich geschrieben, mit vielen Beispielen und wie alle Bücher von H. O. Peitgen spannend zu lesen."

Etwas theoretischer, aber dennoch allgemein verständlich sind die beiden Bücher:

Peitgen H. O., Jürgens H., Saupe D.:

Bausteine des Chaos: Fraktale. Klett / Cotta Verlag, 1992
Chaos: Bausteine der Ordnung. Klett / Cotta Verlag, 1994

Anhang A: Kapiteltests

Kapiteltest 1

Die *Hofstadter - Folge* ist rekursiv definiert durch

$$hof(1) = hof(2) = 1$$
$$hof(n) = hof(n - hof(n-1)) + hof(n - hof(n-2)) \qquad \text{für n = 3, 4, 5, ...}$$

Berechne einige Glieder dieser Folge von Hand.

Schreibe ein iteratives Programm in deiner Programmiersprache, das ein Anfangsstück (beispielsweise die ersten 1000 Glieder) der Hofstadter Folge berechnet und zudem die Hofstadter-Zahlen *hof(n)* in Abhängigkeit von n graphisch darstellt.

Kapiteltest 2

Du kennst sicher die Zahlen im Pascal-Dreieck:

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
.....................................
```

und ihr Bildungsgesetz!

Mit P(i,j) bzeichnen wir die Zahl, die in der i-ten Zeile an der j-ten Stelle im Pascaldreieck steht. Beide Numerierungen beginnen dabei mit 0. P(0,0) ist also gleich 1, der Spitze des Pascaldreiecks. P(4,2) ist gleich 6.

Das Bildungsgesetz des Pascaldreiecks besagt $P(i,j) = P(i-1,j-1) + P(i-1,j)$. Die Zahlen im Pascaldreieck sind also rekursiv definiert.

Schreibe ein rekursives Programm in deiner Programmiersprache, das zu vorgegebenem i und j die Zahl P(i,j) berechnet.

Die Zahlen im Pascaldreieck sind rekursiv definiert. Wäre es da nicht naheliegend, diese Zahlen iterativ zu berechnen, so wie du es im 1. Kapitel gelernt hast? Zu diesen Fragen solltest du dir einige Gedanken machen. Dein Lehrer wird mit dir darüber diskutieren.

Kapiteltest 3

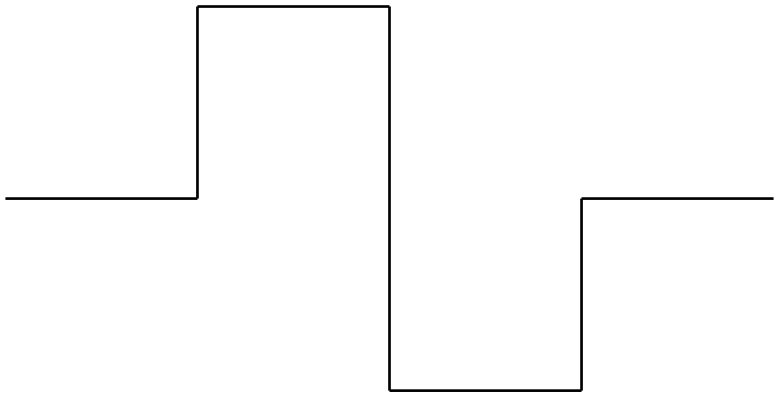
To recurse or not to recurse ist bei vielen Aufgaben die Frage.

Oft können Aufgaben sowohl iterativ als auch rekursiv gelöst werden. Typische Beispiele sind Sortieraufgaben, bei denen es darum geht, Zahlen, Spielkarten, Adressen, etc. gemäss bestimmter Regeln zu ordnen.

Überlege dir ein iteratives und ein rekursives Verfahren um 16 beliebig vorgegebene ganze Zahlen a_1, ... , a_{16} der Grösse nach zu sortieren. Du musst dazu kein Programm schreiben. Es genügt, wenn du deine beiden Algorithmen in wenigen Worten erläutern kannst.
Welcher Deiner beiden Algorithmen ist effizienter, wenn es darum geht, beispielsweise 100‘000 Zahlen zu sortieren?

Kapiteltest 4

Als Kapiteltest haben wir hier einfach eine abgeänderte Kochkurve vorgegeben. Als Generator wählen wir



Schreibe ein Programm in deiner Programmiersprache, das eine Kochkurve mit diesem Generator zeichnet. Übrigens: Wenn du die Programmieraufgabe 8 gut gelöst hast, wird dieser Kapiteltest für dich kein Problem sein!

Hinweise zu den Lösungen der Kapiteltests

Kapiteltest 1: Die Programmierung der iterativen Lösung liegt auf der Hand.
A priori ist bei der Definition der Hofstadter - Folge nicht klar, dass *hof(n-1)* ≤ *n-1* und *hof(n-2)* ≤ *n-1* sind, eine notwendige Voraussetzung für die Wohldefiniertheit der Folge.

Weitere Angaben zur Hofstadter - Folge findet man in:
MU, Der Mathematikunterricht, Friedrich VerlagVetter, Jahrgang 35, Heft 5, September 1989.

Kapiteltest 2: Die Zahlen am Rande des Pascal-Dreiecks bilden die Rekursionsbasis:

$$P(i,0) = P(i,i) = 1$$

Eine iterative Berechnung der Zahlen ist nicht mehr offensichtlich. Die rekursive Berechnung ist nicht besonders effizient, da rückwärts viele Zahlen mehrfach berechnet werden. Eine effiziente Berechnung von P(i,j) ist ausgehend von P(i,0) möglich, wenn die Beziehung zwischen den Zahlen im Pascaldreieck und den Binomialkoeffizienten bekannt ist.

Kapiteltest 3: Es gibt eine Fülle von Sortieralgorithmen. Wichtig ist nicht, dass die Schüler ein effizientes Verfahren entwickeln, sondern die klare Unterscheidung zwischen iterativem und rekursivem Vorgehen.

Kapiteltest 4: Die Aufgabe ist natürlich gelöst, wenn eine hübsche Kochkurve auf dem Bildschirm erscheint!

Anhang B: Material

Nächste Seite:

- Kopiervorlagen für die Karteikarten der Stackverwaltung

Weitere Materialien, die zur Verfügung stehen müssen:

- Mathematiklexikon, Mathematikbücher
- Schülerduden Informatik
- Papier zum Falten der Drachenkurven, Schere
- Turtle-Graphik Bibliothek zur verwendeten Programmiersprache
- Weiterführende Literatur zu Fraktalen

Vorbereitungen der Lehrperson

- Programmbeispiele in der verwendeten Sprache erstellen (Aufgabe 5, Seite 21 und Aufgabe 7, Seite 22)
- Turtle-Graphik-Bibliothek in der verwendeten Programmiersprache herstellen, wenn Sie nicht vorhanden ist. (Kapitel 4) Siehe dazu auch den Anhang D.

Aufruf von: | **Nummer des Aufrufes:**

Ziel des Unterprogrammes:

Ungelöste Arbeitsschritte:

Aktuelle Werte:

Aufruf von: | **Nummer des Aufrufes:**

Ziel des Unterprogrammes:

Ungelöste Arbeitsschritte:

Aktuelle Werte:

Aufruf von: | **Nummer des Aufrufes:**

Ziel des Unterprogrammes:

Ungelöste Arbeitsschritte:

Aktuelle Werte:

Aufruf von: | **Nummer des Aufrufes:**

Ziel des Unterprogrammes:

Ungelöste Arbeitsschritte:

Aktuelle Werte:

Anhang C: Musteraufgabe Bundeswettbewerb Informatik

(Abbdruck mit freundlicher Genehmigung des Bundeswettbewerbes Informatik)

Musteraufgabe: Baum des Pythagoras

Der „Baum des Pythagoras" setzt sich aus lauter Quadraten zusammen, die so um rechtwinklige Dreiecke angeordnet sind, daß sie den Satz des Pythagoras illustrieren. Ihre Seitenverhältnisse sind 3 : 4 : 5. Die ersten Verästelungen des Baumes zeigt Abb. A.

Schreibe ein Programm, das den Baum so weit zeichnet, wie die Auflösung des Bildschirms es zuläßt.

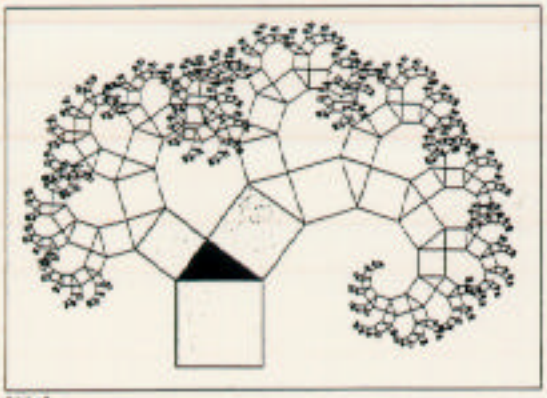


Abb. A

- Musterlösung: Nr. 1 vom Aufgabenblatt 1986

Lösungsidee

Da der Baum sich nur aus Quadraten zusammensetzt, die in konstanten Winkeln mit den Ecken aneinanderstoßen, bietet sich eine Programmlösung an, die auf der Turtlegrafik aufbaut.

Die zentrale Prozedur „baum_zeichnen" zeichnet ein Hypotenusenquadrat und ruft sich selbst zweimal mit den Vorbedingungen je eines der beiden Kathetenquadrate auf (doppelte Rekursion).

Programm-Dokumentation

Das Zeichnen beginnt mit dem größten Quadrat, dem Stamm des Baumes, dessen Seitenlänge der Benutzer eingeben kann.

Die Rekursionen brechen jeweils ab, wenn sie Quadrate zeichnen sollen, die nur noch aus einem Bildschirmpunkt bestehen (daher „seitenlaenge_kleinstes_quadrat = 2"). Dies tritt stets beim linken Kathetenquadrat zuerst auf und braucht deshalb nur dort getestet werden (Programmzeile 33).

Es ist logisch, daß die Rekursionen nach links mit weniger Schritten abbrechen müssen als nach rechts. Die Quadratseiten werden links schneller kleiner (Faktor = 3/5) als rechts (Faktor = 4/5).

Abb. B beschreibt die Bewegungen der Turtle für einen Durchlauf von „baum_zeichnen". Der Baum wächst von innen nach rechts außen und dann nach links. Damit er sich richtig entfaltet, muß die Turtle vor und nach den beiden Rekursionen (7, 11) die Schritte (1, 2, 3, 4, 5, 10, 12, 14) und die Drehungen (6, 8, 9, 13) vornehmen. Die Schritte 12, 14 und die Drehung 13 sind notwendig, damit die Turtle am Ende jedes Durchlaufs von „baum_zeichnen" wieder die gleiche Position und Richtung einnimmt wie zu Beginn des Durchlaufs.

Musterprogramm: „Baum des Pythagoras"

```
1  PROGRAM Baum_des_Pythagoras;
2  (* 3. Bundeswettbewerb Informatik, 1. Runde *)
3
4  {$I GRAPH.P}
5
6  CONST  startpunkt_x                    = 0;
7         startpunkt_y                    = -99;
8         Startrichtung                   = nach_
9         seitenlaenge_kleinstes_quadrat  = 2;
10
11 VAR    seitenlaenge_startquadrat : integer;
12
13 PROCEDURE   baum_zeichnen
14             (seitenlaenge_hypotenusenquadrat : integer);
15
16        VAR  quadratzaehlvariable,
17             seitenlaenge_rechtes_kathetenquadrat,
18             seitenlaenge_linkes_kathetenquadrat : integer;
19
20        BEGIN
21             FOR quadratzaehlvariable := 1 TO 4 DO
22
23             BEGIN
24                  forwd (seitenlaenge_hypotenusenquadrat);
25                  turnleft (90);
26             END;
27
28             seitenlaenge_rechtes_kathetenquadrat :=
29             round (4/5 * seitenlaenge_hypotenusenquadrat);
30             seitenlaenge_linkes_kathetenquadrat :=
31             round (3/5 * seitenlaenge_hypotenusenquadrat);
32
33             IF   seitenlaenge_linkes_kathetenquadrat >=
34                  seitenlaenge_kleinstes_quadrat
35             THEN
36             BEGIN forwd (seitenlaenge_hypotenusenquadrat);
37                   turnright (36.87);
38
39                   baum_zeichnen( seitenlaenge_rechtes_kathetenquadrat);
40
41                   turnleft (36.87);
42                   turnleft (53.13);
43                   forwd (seitenlaenge_rechtes_kathetenquadrat);
44
45                   baum_zeichnen( seitenlaenge_linkes_kathetenquadrat);
46
47                   back ( seitenlaenge_rechtes_kathetenquadrat);
48                   turnright (53.13);
49                   back (seitenlaenge_hypotenusenquadrat)
50             END
51        END;
52
53 BEGIN  writeln;    write ('Welche Seitenlaenge soll das Startquadrat haben? ');
54                    read (seitenlaenge_startquadrat);
55
56        graphmode;  setposition (startpunkt_x, startpunkt_y);
57                    setheading (Startrichtung);
58
59        baum_zeichnen( seitenlaenge_startquadrat)
60 END.
```

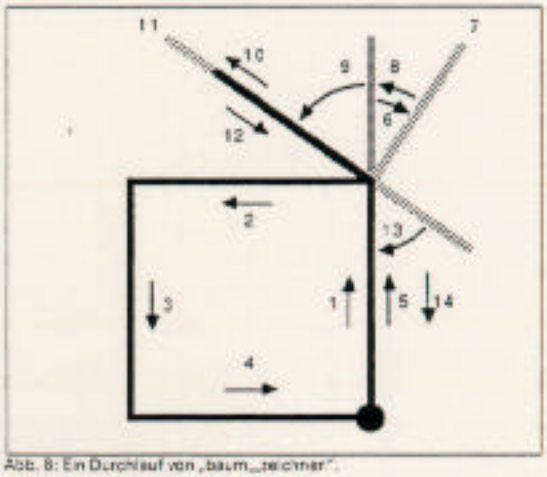


Abb. B: Ein Durchlauf von „baum_zeichnen".

Programmablauf-Protokoll

Sinnvolle Programmablauf-Protokolle sind bei dieser Aufgabe die gezeichneten Bäume selbst, mit Angabe des eingegebenen Wertes für „seitenlaenge_startquadrat" und der benötigten Zeichenzeit.

Programm-Text

Das Programm ist in Turbo-Pascal für einen Mikrocomputer mit Betriebssystem MS-DOS formuliert. In Programmzeile 4 werden die standardmäßig vorhandenen Funktionen der Turtlegrafik zugeladen.

Anhang D: Implementation einer Turtle-Graphik

Für den erfolgreichen Einsatz dieses Leitprogramms ist eine Turtle-Graphik nötig. In praktisch allen Programmiersprachen steht eine solche Bibliothek zur Verfügung. Auch einer Eigenprogrammierung steht wenig im Wege. Hier ein paar Tips:

Grundlagen:	Die Turtle-Graphik Befehle sollen auf den bestehenden Graphikroutinen zum Zeichnen von Linien aufgebaut sein. Die aktuelle Position *(x,y)* der Schildkröte, ihre Blickrichtung *Grad* und ihr Zeichenzustand *Pen* müssen in *globalen Variablen* abgespeichert werden. Für die Schüler ist es auch einfacher, wenn die realen Bildschirmkoordinaten automatisch so transformiert werden, dass man direkt in einem benutzerdefinierten Koordinatensystem arbeiten kann.
GraphicInit:	Dieser Befehl ist stark von der Programmiersprache abhängig. Oft muss nur ein Graphikfenster geöffnet werden. Manchmal muss man aber auch Farben, Strichdicken und Ähnliches einstellen. Hier sollten auch die globalen Variablen initialisiert werden.
PenDown/PenUp:	Der aktuelle Zeichenzustand der Schildkröte muss in einer (boolschen) Variablen gespeichert werden.
Forward(Dist):	Zeichnet bei abgesenktem Stift eine Linie von *(x,y)* nach *(x + Dist*cos(Grad), y + Dist*sin(Grad))*
Turn(Grad):	Zur Variablen Grad wird der übergebene Wert addiert und die Variable wieder auf das zulässige Intervall $0 \leq Grad \leq 360^\circ$ gebracht.
SetPosition(x,y) SetDirection(Grad):	Initialisieren die Position und Blickrichtung der Schildkröte.