1 Algorithmen-Grundlagen und Algorithmen-Implementierung

Wir skizzieren in diesem Abschnitt die Grundlagen der Laufzeitanalyse von Algorithmen und gehen insbesondere der Frage nach, warum man den Formalismus der Groß-Oh-Notation benötigt, um die Laufzeit eines Algorithmus sinnvoll angeben zu können. Wir erklären, was man unter praktisch lösbaren Problemen versteht und skizzieren die Eigenschaft der NP-Vollständigkeit und einige wichtige NP-vollständige Probleme.

Dieses Buch legt einen besonderen Augenmerk auf die Implementierung der Algorithmen. Es gibt meistens mehrere Möglichkeiten einen Algorithmus zu implementieren, bzw. eine Datenstruktur zu repräsentieren. Wir besprechen in diesem Abschnitt die folgenden Implementierungsdimensionen:

- Iterative vs. rekursive Implementierung eines Algorithmus.
- Destruktive vs. nicht-destruktive Implementierung eines Algorithmus.
- Verwendung einer Klasse vs. Verwendung einer Liste, eines Tupel oder einer Hashtabelle zur Repräsentation einer Datenstruktur.

1.1 Laufzeitanalyse von Algorithmen

In der Informatik hat es sich seit Mitte der 60er Jahre eingebürgert, die sog. Landau-Symbole zur Beschreibung der Laufzeit von Algorithmen zu verwenden.

1.1.1 Landau-Symbole

Die nützlichste Methode, die Laufzeit von Algorithmen zu beschreiben, verwendet die sog. *Landau-Symbole*, insbesondere die sog. „Groß-Oh-Notation“, geschrieben $O(\ldots)$. Nehmen wir an, ein Algorithmus wird auf einen Datensatz einer bestimmten Größe n angewendet, so sind wir zwar an der prinzipiellen Laufzeit dieses Algorithmus interessiert; wir wollen jedoch bei der grundsätzlichen Analyse von Algorithmen die Laufzeit auch so abstrakt angeben, dass sie ...

... unabhängig von dem konkreten Computer ist, auf dem der Algorithmus abläuft.

... unabhängig von dem konkreten Compiler ist, der den im Allgemeinen in Hochsprache programmierten Algorithmus in vom Computer ausführbare Maschinensprache übersetzt.

Nur wenn wir in der Lage sind, diese technologischen Details auszuklammern, können wir von einer eigenständigen Disziplin „Algorithmik“ überhaupt erst sprechen und können Algorithmen technologieunabhängig analysieren.

Die Laufzeit eines Algorithmus geben wir immer in Abhängigkeit von der „Größe“ (was auch immer Größe im konkreten Fall bedeutet) der Eingabedaten an – oft auch als *Problemgröße* bezeichnet. Beim Sortieren einer aus n Einträgen bestehenden Liste ist beispielsweise die Problemgröße gleich n. Mit Hilfe der sog. „Groß-Oh-Notation“ kann man technologieunabhängig die Laufzeit eines Algorithmus in Abhängigkeit von der Problemgröße angeben. Behaupten wir beispielsweise unter Verwendung der Groß-Oh-Notation, ein bestimmter Sortieralgorithmus habe eine Laufzeit von $O(n^2)$, so bedeutet das eine Laufzeit, die (höchstens) quadratisch mit der Größe der Eingabe – in diesem Fall die Länge der zu sortierenden Liste – zunimmt. Ausgeklammert wird dabei die Frage, ob die Laufzeit bei einer Eingabegröße n etwa $2 \cdot n^2$ oder $4 \cdot n^2$ ist; aber ein solches „Detail“ (wie ein konstanter Multiplikationsfaktor) hängt ja in der Tat von der Leistungsfähigkeit des ausführenden Rechners ab, interessiert uns also – zumindest wenn wir uns im Fachgebiet „Algorithmik“ bewegen – weniger.

Die formale Definition zeigt, dass die Groß-Oh-Notation eigentlich eine (mathematische) Menge von Funktionen beschreibt. Es gilt:

$$O(g(n)) := \{\ f(n) \mid \text{es gibt } C \geq 0 \text{ und } n_0 \in \mathbb{N} \text{ so dass für alle } n \geq n_0 \text{ gilt: } |f(n)| \leq C \cdot |g(n)|\ \}$$

$$\Omega(g(n) := \{\ f(n) \mid \text{es gibt } C \geq 0 \text{ und } n_0 \in \mathbb{N} \text{ so dass für alle } n \geq n_0 \text{ gilt: } |f(n)| \geq C \cdot |g(n)|\ \}$$

$$\Theta(g(n)) := O(g(n)) \ \cap \ \Omega(g(n)) \qquad .$$

Mit der Konstanten C bringt man mathematisch zum Ausdruck, dass Konstanten keine Rolle spielen; bei der Frage, ob sich eine Funktion in $O(g(n))$ befindet ist nur das ungefähre Wachstum entscheidend. Ist etwa $g(n) = n^3$, so ist die Intention der Groß-Oh-Notation, dass *jede* kubische Funktion in $O(n^3)$ ist, etwa auch $f(n) = 9n^3$; in diesem Falle müsste man $C \geq 9$ wählen. Indem man verlangt, dass die gewünschte Eigenschaft nur von Funktionswerten ab einer bestimmten Größe ($n \geq n_0$) erfüllt wird, bringt man zum Ausdruck, dass man nur an dem asymptotischen Wachstumsverhalten interessiert ist, d. h. dem Wachstumsverhalten für „große“ Funktionswerte – durch Wahl von n_0 kann man selbst bestimmten, was „groß“ ist. Für die konstante Funktion $f(x) = 5$ wäre beispielsweise $f(x) \in O(\ln(x))$, was sich durch Wahl von $n_0 = \lceil e^5 \rceil = 149$ leicht bestätigen lässt.

Es hat sich eingebürgert, statt $f(n) \in O(g(n))$ einfach $f(n) = O(g(n))$ zu schreiben. Man sollte jedoch nicht vergessen, dass das hier verwendete Symbol „=“ eigentlich ein „$\in$“ darstellt und daher auch nicht kommutativ ist.

Aufgabe 1.1

Geben Sie konkrete Werte der Konstanten C und n_0 an, die zeigen, dass gilt:

(a) $3n^2 + 10 \in O(n^2)$
(b) $3n^2 + n + 1 \in O(n^2)$

Aufgabe 1.2

Entscheiden Sie die Gültigkeit der folgenden Aussagen (nicht notwendigerweise formal; sie dürfen auch intuitiv argumentieren):

(a) $n^{100} = O(1.01^n)$
(b) $10^{\log n} = O(2^n)$
(c) $10^{\sqrt{n}} = O(2^n)$
(d) $10^n = O(2^n)$

Während Konstanten tatsächlich oft technologische Besonderheiten widerspiegeln (moderne Rechner sind etwa 10 bis 100 mal schneller als die Rechner vor 10 Jahren), so spiegeln die durch die Groß-Oh-Notation ausgedrückten Laufzeiten eher prinzipielle Eigenschaften der zugrunde liegenden Algorithmen wider. Beispielsweise würde der modernste und schnellste Rechner mit einem schlecht implementierten Sortieralgorithmus (Laufzeit $O(n^2)$) um Größenordnungen langsamer sortieren als ein sehr alter langsamer Rechner, der einen schnellen Sortieralgorithmus (Laufzeit $O(n \log(n))$) verwendet – wenn die Länge der zu sortierenden Liste nur lang genug ist.

Aufgabe 1.3

Wir lassen einen schnellen Rechner A (100 Millionen Instruktionen pro Sekunde) mit einem langsamen Sortieralgorithmus (Laufzeit $O(n^2)$) gegen einen sehr langsamen Rechner B (100000 Instruktionen pro Sekunde) mit einem schnellen Sortieralgorithmus (Laufzeit $O(n \log(n))$ gegeneinander antreten.

Füllen Sie die folgende Tabelle mit den ungefähren Laufzeiten.

	Länge der Liste				
	100000	1 Mio	10 Mio	100 Mio	1 Mrd
Rechner A					
Rechner B					

1.1.2 Worst-Case, Average-Case und amortisierte Laufzeit

In der Laufzeitanalyse von Algorithmen unterscheidet man häufig zwischen ...

- *Worst-Case-Laufzeit*: Dies ist die Laufzeit, die der Algorithmus im schlechtest denkbaren Fall brauchen würde. Auch dann, wenn dieser „schlechteste“ Fall sehr unwahrscheinlich ist bzw. sehr selten auftritt, mag eine sehr ungünstige Worst-Case-Laufzeit – wenn man Wert auf konstantes, vorhersagbares Verhalten legt – kritisch sein.

- *Average-Case-Laufzeit*: Dies ist die Laufzeit, die der Algorithmus im Mittel benötigt, mathematisch modelliert durch den Erwartungswert der Laufzeit. Bei der Berechnung dieses Erwartungswerts wird die Laufzeit aller Situationen nach der Wahrscheinlichkeit gewichtet, mit der die entsprechende Situation eintritt; die Laufzeit unwahrscheinlicher Konstallationen fällt entsprechend weniger ins Gewicht als die Laufzeit wahrscheinlicher Konstellationen.

  Häufig interessiert man sich für die Average-Case-Laufzeit eines Algorithmus.

- *Amortisierte Laufzeit*: Bei dieser Art der Laufzeitanalyse betrachtet man Folgen von Operationen auf einer Datenstruktur; die Laufzeit einer Rechenzeit-aufwändigen Operation kann hierbei durch die Laufzeit von weniger aufwändigen Funktionen ausgeglichen werden. Es gibt mehrere Methoden eine amortisierte Laufzeitanalyse durchzuführen; für die Laufzeitanalyse von Fibonacci-Heaps (siehe Abschnitt 4.3 verwenden wir etwa die sog. Potentialmethode.

1.1.3 Praktisch lösbar vs. exponentielle Laufzeit

Wir wollen Probleme, für die es einen Algorithmus mit Laufzeit $O(n^p)$ mit $p \in \mathbb{N}$ gibt, als *praktisch lösbar* bezeichnen; manchmal werden sie lax auch als *polynomiell* bezeichnet, da ihre Laufzeit begrenzt ist durch ein Polynom in der Eingabegröße. Genaugenommen wäre jedoch ein Algorithmus mit Laufzeit von beispielsweise $O(n^{100})$ für große Werte von n eigentlich nicht wirklich „praktisch“, denn schon für eine Eingabegröße $n = 10$ wäre die Laufzeit für die Lösung eines solchen Problems astronomisch groß. Zwar kann man sich theoretisch für jedes gegebene $p \in \mathbb{N}$ ein Problem konstruieren, für dessen Lösung ein Algorithmus mit Laufzeit $O(n^p)$ nötig ist, für alle praktisch relevanten Probleme ist, sofern sie polynomiell sind, jedoch $p \leq 4$; insofern macht es tatsächlich Sinn polynomielle Probleme als „praktisch lösbar“ zu bezeichnen.

In der Komplexitätstheorie wird die Menge aller Probleme, zu deren Lösung ein polynomieller Algorithmus existiert, als P bezeichnet. P ist ein Beispiel für eine *Komplexitätsklasse*. Probleme, für die nur Algorithmen bekannt sind, deren Laufzeit exponentiell mit der Größe der Eingabe steigt, mögen zwar theoretisch nicht jedoch praktisch lösbar sein.

Aufgabe 1.4

Angenommen ein bestimmtes Problem, z. B. die Primfaktorzerlegung einer n-stelligen Zahl, benötigt $O(2^n)$ viele Schritte; die Laufzeit ist also exponentiell in der Größe der Eingabe.

(a) Angenommen, uns steht ein äußerst leistungsfähiger Rechner zur Verfügung, der für eine elementare Operation $50ps$ benötigt. Füllen Sie nun folgende Tabelle mit den ungefähren Laufzeiten aus:

	Länge der zu zerlegenden Zahl					
	10	20	50	100	200	1000
Laufzeit						

(b) Wir nehmen Kontakt zu einer außerirdischen Zivilisation auf, die der unseren technologisch sehr überlegen ist. Sie können Rechner bauen, die 1 Mio mal schneller sind als die unsrigen; nehmen wir weiter an, jeder Außerirdische auf dem mit 20 Mrd Individuen hoffnungslos überbevölkerten Planeten besitzt einen solchen schnellen Rechner. Zudem sind sie in der Lage alle 20 Mrd Rechner zu einem Cluster zusammen zu schließen, das dann tatsächlich etwa 20 Mrd mal schneller ein bestimmtes Problem lösen kann als ein einzelner Rechner. Füllen Sie nun die folgende Tabelle mit den ungefähren Laufzeiten aus, die dieses Alien-Cluster zur Primfaktorzerlegung benötigt.

	Länge der zu zerlegenden Zahl				
	50	100	200	1000	5000
Laufzeit					

Eine weitere wichtige Komplexitätsklasse ist die Klasse *NP*, die alle Probleme beinhaltet, die durch eine nicht-deterministische Rechenmaschine in polynomieller Zeit „berechnet“ werden können. Einer nicht-deterministischen Rechenmaschine (mathematisch modelliert durch eine nicht-deterministische Turingmaschine) kann man mehrere alternative Rechenwege zur Verfügung stellen; die „Ausführung“ eines Programms auf einer solchen Maschine besteht darin, dass sie sich (durch „Magie“) immer die richtige zum Ziel führende Alternative auswählt. Es gilt $P \subseteq NP$, da jeder polynomielle Algorithmus auch genauso gut auf einer nicht-deterministischen Maschine (ohne jedoch dieses Nicht-Determinismus-„Feature“ zu nutzen) in polynomieller Zeit ausgeführt werden kann. Interessanterweise konnte bisher noch nicht gezeigt werden, dass $P \neq NP$, auch wenn die meisten Spezialisten dies stark vermuten.

Es gibt eine Klasse von Problemen, die sog. *NP*-vollständigen Probleme, die (intuitiv gesprochen) „schwersten“ Probleme in *NP*; zudem kann man (wiederum intuitiv gesprochen) sagen, dass alle *NP*-vollständigen Probleme in gewissem Sinne gleich schwer sind. Wenn man für eines dieser *NP*-vollständigen Probleme einen polynomiellen Algorithmus finden würde, so wäre man in der Lage, dieses polynomielle Verfahren auf alle

anderen *NP*-vollständigen Probleme zu übertragen und – da diese gewissermaßen die schwersten Probleme in *NP* sind – somit auf alle Probleme in *NP* zu übertragen. Dann hätte man gezeigt, dass $P = NP$. Bisher hat jedoch noch niemand einen polynomiellen Algorithmus für ein solches *NP*-vollständiges Problem gefunden und somit bleibt weiterhin unbewiesenermaßen zu vermuten, dass $P \neq NP$ ist.

Rabin Karp „entdeckte“ diese Ähnlichkeit der NP-vollständigen Probleme; in seinem ursprünglichen Artikel [11] beschrieb er insgesamt 21 solche Probleme. Wir geben hier eine kleine Auswahl davon an:

- 3SAT: Das Erfüllbarkeitsproblem (Satisfiability) für 3-KNF-Formeln, d. h. für boolesche Formeln in Konjunktiver Normalform (also Konjunktionen von Disjunktionen), wobei jede Klausel genau drei Variablen enthält, besteht darin, nach einer Belegung der Variablen zu suchen, so dass die Formel erfüllt ist (d. h. den Wahrheitswert „True“ liefert).
- Rucksack-Problem: Das Problem besteht darin, aus einer Menge von Objekten, die jeweils einen Wert und ein Gewicht haben, eine Teilmenge so auszuwählen, dass deren Gesamtgewicht eine vorgegebene Schwelle nicht überschreitet und der Wert der Objekte maximal ist.
- Clique: Gegeben sei ein Graph. Das Problem besteht darin, einen vollständigen Teilgraphen mit k Knoten zu finden. (Ein Graph heißt vollständig, wenn jeder Knoten mit jedem anderen verbunden ist).
- Travelling-Salesman-Problem (Kurz: TSP). Das Problem besteht darin, eine Reihenfolge für den Besuch einer gegebenen Anzahl von Orten so auszuwählen, dass die zurückgelegte Wegstrecke minimal ist.

1.2 Implementierung von Algorithmen

Insbesondere dann, wenn man Algorithmen in ausführbaren Programmiersprachen beschreiben möchte, muss man sich Gedanken um die Implementierung machen. Es gibt immer mehrere Möglichkeiten einen Algorithmus zu implementieren. Man muss sich entscheiden, ob man einen Algorithmus durch rekursive Funktionen oder durch Iteration implementiert. Man muss sich entscheiden, ob ein Algorithmus eine Datenstruktur verändern soll, oder ob er die „alte“ Struktur belässt und als Rückgabewert eine „neue“ Datenstruktur zurückliefert. Und man muss sich entscheiden, ob man eine Datenstruktur durch eine Klasse oder etwa durch eine Liste oder gar durch eine Hash-Tabelle implementiert.

1.2.1 Rekursive vs. iterative Implementierung

Ein Funktion heißt genau dann rekursiv, wenn der Funktionskörper mindestens einen Aufruf der Funktion selbst enthält, die Funktion also die folgende Form hat:

def *rekFunc*(*x*):
  ...
  ... *rekFunc*(*i*) ...
  ...

Intuitiv vermutet man hier eine Endlos„schleife“ (die Funktion ruft sich endlos selbst auf) – wir werden jedoch gleich sehen, dass dies nicht notwendigerweise der Fall sein muss.

Beispiel: Implementierung der Fakultätsfunktion. Betrachten wir als erstes Beispiel die Implementierung einer Funktion, die die Fakultät einer Zahl n berechnet. Eine iterative Implementierung könnte folgendermaßen aussehen:

```
1 def facIter(n):
2   erg = 1
3   for i in range(1,n+1)
4     erg = erg*i
5   return erg
```

Aufgabe 1.5

Verwenden Sie die Python-Funktion *reduce*, um eine Funktion *prod*(*lst*) zu definieren, die als Ergebnis die Aufmultiplikation der Zahlen in *lst* zurückliefert. Mathematisch ausgedrückt, sollte für *prod* gelten:

$$\texttt{prod(lst)} \stackrel{!}{=} \prod_{x \in \texttt{lst}} x$$

Implementieren Sie nun *facIter* mit Hilfe von *prod*.

Man kann die Fakultätsfunktion auch rekursiv definieren, wie in Listing 1.1 gezeigt. Man beachte, dass diese Funktionsdefinition im Gegensatz zur iterativen Definition keine Schleife benötigt.

```
1 def fac(n):
2   if n==0:
3     return 1
4   else:
5     return n*fac(n-1)
```

Listing 1.1: *Rekursive Implementierung der Fakultätsfunktion*

Um zu verstehen, wie *fac* einen Wert berechnet, zeigt Abbildung 1.1 im Detail an einem Beispiel, wie etwa ein Aufruf von *fac*(4) abläuft. Für den Programmierer ist es interessant zu wissen, dass der rekursive Abstieg immer mit einer zunehmenden

Belegung des Stacks[1] einhergeht; ein zu langer rekursiver Abstieg kann hierbei evtl. in einem „Stack Overflow“, d. h. einem Überlauf des Stackspeichers, enden.

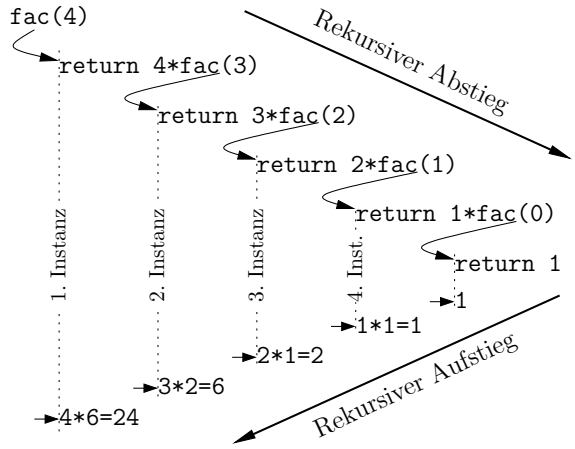


Abb. 1.1: *Bei einem Aufruf von fac(4) wird (da 4==0 nicht zutrifft) sofort die Anweisung* ***return*** *4*fac(3) (Zeile 5, Listing 1.1) ausgeführt, was zu dem Aufruf fac(3), also einem rekursiven Aufruf, führt. Ab diesem Zeitpunkt sind zwei* Instanzen *der Funktion fac zugleich aktiv: Die erste Instanz wartet auf die Ergebnisse, die die zweite Instanz liefert und die Befehle der zweiten Instanz werden aktuell ausgeführt. Alle Anweisungen dieser zweiten Instanz sind in der Abbildung eingerückt dargestellt. Bei diesem Aufruf von fac(3) wird (da 3==0 nicht zutrifft) sofort die Anweisung* ***return*** *3*fac(2) ausgeführt, was zu dem Aufruf fac(2), also einem weiteren rekursiven Aufruf führt, usw. Dieser Prozess des wiederholten rekursiven Aufrufs einer Funktion (in Richtung auf den Rekursionsabbruch) nennt man auch den rekursiven Abstieg. In der 5. Instanz schließlich ist mit dem Aufruf fac(0) der Rekursionsabbruch erreicht: nach Beenden der 5. Instanz kann der Wert der* ***return****-Anweisung der 4. Instanz bestimmt werden und anschließend die 4. Instanz beendet werden, usw. Diese sukzessive Beenden der durch rekursive Aufrufe entstandenen Instanzen nennt man auch den rekursiven Aufstieg.*

Damit eine rekursive Funktion sich nicht endlos immer wieder selbst aufruft, sollte sie die beiden folgenden Eigenschaften haben:

1. Rekursionsabbruch: Es muss eine Abfrage vorhanden sein, ob das Argument des Aufrufs „klein“ genug ist – „klein“ muss in diesem Zusammenhang nicht notwendigerweise „numerisch klein“ bedeuten, sondern kann je nach involviertem Datentyp auch strukturell klein bedeuten. In diesem Fall soll die Rekursion beendet werden; es sollen also keine weiteren rekursiven Aufrufe stattfinden. In diesem Fall sollte der Rückgabewert einfach direkt berechnet werden. In Listing 1.1 besteht der Rekursionsabbruch in Zeile 2 und 3 darin zu testen, ob die übergebene Zahl eine Null ist – in diesem Fall ist die Fakultät definitionsgemäß 1.

[1] Der Zustand der aufrufenden Funktion – dazu gehören unter Anderem Werte von lokalen Variablen und die Werte der Aufrufparameter – wird immer auf dem Stack des Rechners gespeichert. Jede Instanz einer Funktion, die sich noch in Abarbeitung befindet, belegt hierbei einen Teil des Stacks.

2. Rekursive Aufrufe sollten als Argument (strukturell oder numerisch) „kleinere“ Werte übergeben bekommen. Handelt es sich bei den Argumenten etwa um natürliche Zahlen, so sollten die rekursiven Aufrufe kleinere Zahlen übergeben bekommen. Handelt es sich bei den Argumenten etwa um Listen, so sollten die rekursiven Aufrufe kürzere Listen übergeben bekommen; handelt es sich bei den Argumenten etwa um Bäume, so sollten die rekursiven Aufrufe Bäume geringerer Höhe (oder Bäume mit weniger Einträgen) übergeben bekommen, usw.
   Die in Listing 1.1 gezeigte rekursive Implementierung der Fakultätsfunktion erfüllt diese Voraussetzung: Der rekursive Aufruf in Zeile 5 erfolgt mit einem Argument, das um eins kleiner ist als das Argument der aufrufenden Funktion.

Rekursive Aufrufe mit kleineren Argumenten stellen einen rekursiven Abstieg sicher; der Rekursionsabbruch beendet den rekursiven Abstieg und leitet den rekursiven Aufstieg ein.

Offensichtlich erfüllt also die in Listing 1.1 gezeigte rekursive Implementierung der Fakultätsfunktion diese Eigenschaften und ist somit wohldefiniert.

Aufgabe 1.6

Angenommen, eine rekursive Funktion erhält als Argument eine reelle Zahl. Warum ist es für eine korrekt funktionierende rekursive Funktion nicht ausreichend zu fordern, dass die rekursiven Aufrufe als Argumente kleinere reelle Zahlen erhalten als die aufrufende Funktion?

Aufgabe 1.7

(a) Definieren Sie die Funktion $sum(n)$, die die Summe der Zahlen von 1 bis n berechnen soll, rekursiv.

(b) Definieren Sie die Funktion $len(\,lst\,)$, die die Länge der Liste lst berechnen soll, rekursiv.

Beispiel: Beschriftung eines Meterstabs. Wir haben gesehen, dass das vorige Beispiel einer rekursiv definierten Funktion auch ebenso einfach iterativ programmiert werden konnte. Das gilt für die folgende Aufgabe nicht: Sie ist sehr einfach durch eine rekursive Funktion umzusetzen; die Umsetzung durch eine iterative Funktion ist in diesem Fall jedoch deutlich schwerer[2]. Wir wollen ein Programm schreiben, das Striche auf ein Lineal folgendermaßen zeichnet: In der Mitte des Lineals soll sich ein Strich der Höhe h befinden. Die linke Hälfte und die rechte Hälfte des Lineals sollen wiederum vollständig beschriftete Lineale sein, in deren Mitten sich jeweils Striche der Höhe $h-1$ befinden, usw. Abbildung 1.2 zeigt solch ein Lineal (das mit dem Pythonskript aus Listing 1.2 gezeichnet wurde).

[2]Dies gilt allgemein auch für alle nach dem sog. Divide-And-Conquer Schema aufgebauten Algorithmen.

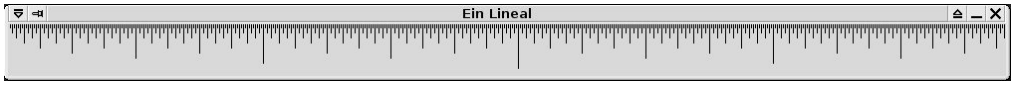


Abb. 1.2: *Das durch Aufruf von lineal(0,1024,45) gezeichnete Lineal in dem durch GraphWin("Ein Lineal",1024,50) (Zeile 3, Listing 1.2) erzeugten Fenster.*

```
1  from graphics import *
2
3  linealCanv = GraphWin('Ein Lineal',1000,50)
4
5  def strich(x,h):
6      l = Line(Point(x,0),Point(x,h))
7      l.draw(linealCanv)
8
9  def lineal(l,r,h):
10     step = 5
11     if (h<1): return
12     m = (l+r)/2
13     strich(m,h)
14     lineal(l,m,h-step)
15     lineal(m,r,h-step)
```

Listing 1.2: *Die rekursiv definierte Funktion lineal zeichnet das in Abbildung 1.2 dargestellte Lineal.*

Der Rekursionsabbruch der rekursiv definierten Funktion *lineal* befindet sich in Zeile 11; die rekursiven Aufrufe (mit kleinerem dritten Parameter) befinden sich in Zeile 14 und Zeile 15. Das verwendete *graphics*-Modul ist eine kleine, sehr einfach gehaltene Graphik-Bibliothek, geschrieben von John Zelle, der es in seinem Python-Buch [19] verwendet. Der Konstruktor *GraphWin* in Zeile 3 erzeugt ein Fenster der Größe 1000×50 Pixel; die Funktion *strich*(x,h) zeichnet an Position x des zuvor erzeugten Fensters eine vertikale Linie der Länge h.

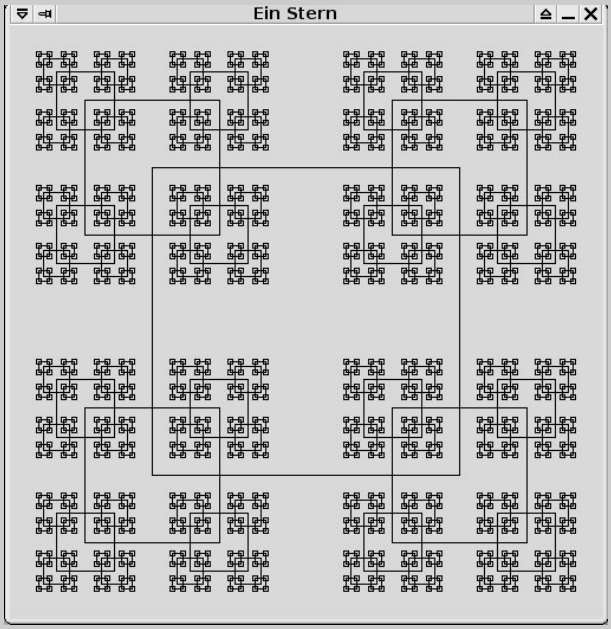
Versucht man dieselbe *lineal*-Funktion dagegen iterativ zu programmieren, muss man sich erheblich mehr Gedanken machen: Entweder muss man die rekursive Aufrufhierarchie unter Verwendung eines Stacks „simulieren“ (in Abschnitt 2.3.5 ab Seite 30 zeigen wir im Detail am Beispiel des Quicksort-Algorithmus wie man hierbei vorgehen kann) oder man muss entschlüsseln, welche Höhe ein Strich an Position x haben muss.

Aufgabe 1.8

Verwenden Sie Iteration um eine *lineal*-Funktion zu programmieren, die äquivalent zur *lineal*-Funktion aus Listing 1.2 ist.

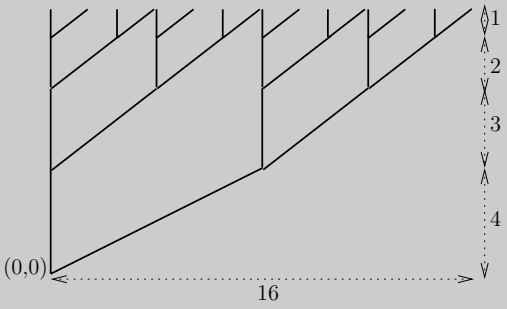
Aufgabe 1.9

Zeichnen Sie durch eine rekursiv definierte Python-Funktion und unter Verwendung der *graphics*-Bibliothek folgenden Stern:



Aufgabe 1.10

Schreiben Sie eine rekursive Prozedur *baum*(*x*,*y*,*b*,*h*) zum Zeichnen eines (binären) Baumes derart, dass die Wurzel sich bei (x, y) befindet, der Baum b breit und h hoch ist. Definieren Sie hierzu eine Python-Prozedur *line*(*x1*,*y2*,*x2*,*y2*), die eine Linie vom Punkt (*x1*,*y2*) zum Punkt (*x2*,*y2*) zeichnet. Folgende Abbildung zeigt ein Beispiel für die Ausgabe die der Aufruf *baum*(0,0,16,4) erzeugt.



Aufgabe 1.11

Das sog. Sierpinski-Dreieck kann folgendermaßen konstruiert werden. **1.** Man wähle zunächst eine (eigentlich beliebige) Form – wir starten hier mit einem gleichschenkeligen Dreieck, also einem beliebig großen Dreieck mit drei gleichlangen Seiten. **2.** Nun verkleinern wir diese Form um genau die Hälfte ihrer ursprünglichen Größe und positionieren zwei dieser Formen direkt nebeneinander und eine dritte mittig direkt darüber. **3.** Man wiederhole nun mit der so erhaltenen Form den Schritt 2. rekursiv. Das folgende Bild zeigt die ersten 5 Schritte beim Zeichnen eines Sierpinski-Dreiecks.

Schreiben Sie eine rekursive Prozedur *sierpinski*(x, y, n), die ein Sierpinski-Dreieck der Seitenlänge n und Mittelpunkt (x, y) zeichnet.

1.2.2 Warum Rekursion (statt Iteration)?

Rekursive Implementierungen mögen für den Informatik-„Anfänger“ schwieriger zu verstehen sein und für manche Compiler/Interpreter problematischer zu übersetzen sein, sie haben jedoch einen entscheidenden Vorteil: Man braucht sich nicht der Lösung des kompletten Problems zu widmen, sondern es genügt, sich über den „Rekursionsschritt“ Gedanken zu machen. Man muss sich dabei „nur“ überlegen, wie man sich aus einer (bzw. mehrerer) „kleineren“[3] Lösung(en) des Problems eine „größere“ Lösung konstruieren kann. Dies ist meist viel weniger komplex als sich zu überlegen, wie die Lösung von Grund auf zu konstruieren ist.

1.2.3 „Kochrezept“ für das Entwickeln eines rekursiven Algorithmus

(a) Zunächst kann man sich den Rekursionsabbruch überlegen, also:

- Was ist der „triviale“, einfache Fall? Üblicherweise ist der einfache Fall für eine Eingabe der Größe $n = 1$, $n = 0$ oder einem anderen kleinen Wert für n gegeben.
- Was muss der Algorithmus noch tun, wenn er solch einen einfachen Fall vorliegen hat? Üblicherweise sind nur noch (wenn überhaupt) einfache Manipulationen der Eingabe vorzunehmen.

(b) Dann muss man sich eines Gedanken„tricks“ bedienen. Man nehme an, dass die Aufgabenstellung schon für ein oder mehrere „kleinere“ Probleme gelöst sei und überlegt sich (unter dieser Annahme), wie man aus den Lösungen der kleineren Aufgaben, die Lösung der Gesamtaufgabe konstruieren kann. Die Implementierung dieses Schritts wird auch als der „Rekursionsschritt“ bezeichnet.

[3] Was auch immer „kleiner“ im Einzelfall heißen mag; falls die Eingaben Listen wären, würde man darunter eine kürzere Liste verstehen.

(c) Das Ausprogrammieren der rekursiven Lösung erfolgt dann prinzipiell wie in folgendem Pseudo-Python-Code-Listing gezeigt:

```
1  def rekAlg(x):
2    if groesse(x) is kleingenug:
3      return loesung(x)
4    else:
5      (x1,x2, ... ) = aufteilen(x) # len(x1) < x, len(x2) < x,  ...
6      return rekSchritt(rekAlg(x1),rekAlg(x2),  ... )
```

Die rekursive Funktion startet mit dem Test, ob die Rekursion abgebrochen werden kann, was dann der Fall ist, wenn die Größe der Eingabe klein genug ist und so die Lösung einfach berechnet werden kann. Andernfalls wird der Algorithmus rekursiv evtl. mehrmals aufgerufen um so Teillösungen zu produzieren; die Entscheidung, wie die Eingabe aufgeteilt werden soll, überlassen wir der Funktion *aufteilen*, die für jeden rekursiven Algorithmus individuell ausprogrammiert werden muss. Diese Teillösungen werden dann wieder zusammengefügt – hier dargestellt durch Ausführung der Funktion *rekSchritt*. In der Ausprogrammierung dieses Rekursionsschritts besteht im Allgemeinen die eigentliche algorithmische Herausforderung bei der Lösung eines gegebenen Problems.

1.3 Nicht-destruktive vs. In-place Implementierung

Viele in imperativen Programmiersprachen wie C, C++ oder Python implementierte Algorithmen operieren auf ihrer Eingabe „destruktiv“, d. h. sie zerstören bzw. überschreiben ihre ursprüngliche Form; sie „bauen“ die Struktur des übergebenen Parameters so um, dass das gewünschte Ergebnis entsteht. Dies geschieht etwa, wenn man mit Hilfe der in Python eingebauten Sortierfunktion *sort*() eine Liste sortiert. Eine Liste wird dem Sortieralgorithmus übergeben, der diese in destruktiver Weise sortiert („>>>“ ist die Eingabeaufforderung der Python-Shell):

```
>>> lst=[17, 46, 45, 47, 43, 25, 35, 60, 80, 62, 60, 41, 43, 14]
>>> lst.sort()
>>> print lst
[14, 17, 25, 35, 41, 43, 43, 45, 46, 47, 60, 60, 62, 80]
```

Nach Aufruf von *lst.sort*() werden die Werte, die ursprünglich in *lst* standen überschrieben, und zwar so, dass eine sortierte Liste entsteht. Wir können nun nicht mehr auf den ursprünglichen Wert von *lst* zugreifen. Der große Vorteil einer solchen „destruktiven“ Implementierung ist jedoch, dass sie i. A. „in-place“ – also „an Ort und Stelle“ – erfolgen kann, d. h. dass der Algorithmus (so gut wie) keinen weiteren Speicherbereich belegen muss, sondern für die Sortierung ausschließlich den Speicherbereich benötigt, der durch *lst* bereits belegt ist.

Viele in funktionalen Sprachen, wie Haskell, ML oder Lisp, implementierte Algorithmen dagegen verarbeiten die Eingabe „nicht destruktiv“, d. h. sie zerstören die Eingabe nicht. Stattdessen erzeugen sie sich als Ergebnis (d. h. als Rückgabewert; in Python durch das

return-Kommando übergeben) eine neue Struktur, die sich teilweise oder ganz in einem neuen Speicherbereich befindet.

Pythons eingebaute Funktion *sorted*(*xs*) verarbeitet ihre Eingabe nicht-destruktiv:

```
>>> lst=[17, 46, 45, 47, 43, 25, 35, 60, 80, 62, 60, 41, 43, 14]
>>> lst2=sorted(lst1)
>>> print lst2
[14, 17, 25, 35, 41, 43, 43, 45, 46, 47, 60, 60, 62, 80]
```

Der Nachteil nicht-destruktiver Implementierungen ist offensichtlich: sie brauchen mehr Speicherplatz, als entsprechende In-place-Implementierungen.

1.3.1 Warum nicht-destruktive Implementierungen?

Wenn nicht-destruktive Implementierungen mehr Speicherplatz benötigen und daher meist auch etwas langsamer sind als destruktive (d. h. In-place-)Implementierungen, warum sollte man nicht-destruktive Implementierung überhaupt in Erwägung ziehen? Der Grund ist einfach: nicht-destruktive Implementierungen sind oft kompakter, leichter zu verstehen und entsprechend schneller und fehlerfreier zu implementieren. Um den Grund dafür wiederum zu erklären, müssen wir etwas weiter ausholen:

- Jedes destruktive Update einer Datenstruktur verändert den internen Zustand eines Programms.
- Je größer die Anzahl der möglichen Zustände im Laufe des Programmablaufs, desto mehr potentielle Abfragen, und desto mehr potentielle Fehler können sich einschleichen.
- Eine Funktion, die keine destruktiven Updates verwendet (die einer mathematischen Funktion also relativ ähnlich ist), führt keine Zustände ein; im optimalen Fall verändert ein gegebenes Programm den globalen Zustand überhaupt nicht, und diese zustandsfreie Situation erlaubt es dem Programmierer, leichter den Überblick zu bewahren.

Viele moderne Compiler und Interpreter sind inzwischen schon „intelligent“ genug, den durch nicht-destruktive Implementierungen verwendeten Speicher selbstständig wieder freizugeben, wenn klar ist, dass Daten nicht mehr verwendet werden. Dies ermöglicht es, tatsächlich Programme, die ausschließlich nicht-destruktive Updates beinhalten, in praktisch genauso schnellen Maschinencode zu übersetzen wie Programme, die ausschließlich In-place-Implementierungen verwenden.

1.4 Repräsentation von Datenstrukturen

Möchte man eine Datenstruktur repräsentieren, die aus mehreren Informations-Komponenten besteht, so bieten sich in Python hierzu mehrere Möglichkeiten an. Nehmen wir beispielsweise an, wir wollen einen Baum repräsentieren, der aus den Komponenten Schlüsseleintrag, Werteintrag, linker Teilbaum und rechter Teilbaum besteht.

1.4.1 Repräsentation als Klasse

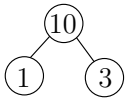
Das Paradigma der Objektorientierten Programmierung schlägt die Repräsentation als Klasse vor, wie in Listing 1.3 gezeigt.

```
1 class Baum(object):
2     __init__ ( self,key,val, ltree=None,rtree=None):
3         self.key = key ;  self.val=val
4         self. ltree  =  ltree  ;  self. rtree  =  rtree
```

Listing 1.3: *Repräsentation eines Baums durch eine Klasse*

Der rechts gezeigte einfache Baum kann dann folgendermaßen mittels des Klassenkonstruktors erzeugt werden:

Baum(10,20,*Baum*(1,2),*Baum*(3,4))



Diese Art der Repräsentation ist in vielen Fällen die sinnvollste; die Klassenrepräsentation wird in diesem Buch für viele Bäume (außer für Heaps) und für Graphen verwendet.

1.4.2 Repräsentation als Liste

Eine Klasse ist nicht die einzige Möglichkeit der Repräsentation. Man könnte auch eine Liste verwendet, um die (in diesem Fall vier) Informations-Komponenten zu einem Bündel, das dann den Baum darstellt, zusammenzufassen. Der Baum aus obiger Abbildung ließe sich dann folgendermaßen definieren:

[10,20, [1,2], [3,4]]

Mit ebenso viel Recht könnten wir uns aber auch dazu entscheiden, leere Teilbäume explizit aufzuführen und etwa durch „*None*“ zu repräsentieren. Dann hätte obiger Baum die folgende Repräsentation:

[10,20, [1,2, *None*,*None*], [3,4, *None*,*None*]]

Diese Art der Darstellung ist kompakter als die Darstellung über eine Klasse, und es kann sich in einigen Fällen durchaus als vernünftig herausstellen, diese Art der Repräsentation zu wählen. Ein „Problem“ ist jedoch oben schon angedeutet: Es gibt viele Freiheitsgrade, wie diese Liste zu gestalten ist. Zusätzlich ist eine Repräsentation über eine Klasse typsicherer: Der Wert *Baum*(10,20) ist *immer* ein Baum; der Wert [10,20] könnte dagegen ebenso eine einfache Liste sein.

Ein Repräsentation über Listen wurde in diesem Buch beispielsweise für Binomial-Heaps gewählt (siehe Abschnitt 4.2).

1.4.3 Repräsentation als Dictionary

Die Repräsentation als Dictionary stellt in gewissem Sinn einen Kompromiss zwischen der mit verhältnismäßig viel Overhead verbundenen Repräsentation als Klasse und der

sehr einfachen aber nicht typsicheren Repräsentation als Liste dar. Jede Informations-Komponente erhält hierbei eine Kennung (etwa einen String), und die Datenstruktur stellt dann nichts anderes als eine Sammlung solcher mit Kennung versehener Komponenten dar. Der oben im Bild dargestellte Baum könnte so folgendermaßen repräsentiert werden:

```
{ 'key':10 , 'val':20 ,
  'ltree': {'key':1 , 'val':2 , 'ltree':None , 'rtree':None} ,
  'rtree': {'key':3 , 'val':4 , 'ltree':None , 'rtree':None}
}
```

Tatsächlich erfolgt Python-intern der Zugriff auf die Attribute und Methoden einer Klasse nach dem gleichen Prinzip wie der Zugriff auf die Einträge eines Dictionary-Objektes: nämlich über eine Hash-Tabelle; diese Datenstruktur beschreiben wir in Abschnitt 3.4 ab Seite 72). Insofern ist zumindest technisch gesehen die Repräsentation über ein Dictionary schon recht nah an der Repräsentation über eine Klasse.

Wir verwenden diese Art der Repräsentation beispielsweise für die Implementierung von Fibonacci-Heaps (Abschnitt 4.3 auf Seite 127) und Pairing-Heaps (Abschnitt 4.4 auf Seite 142).