Rekursive Algorithmen programmiert in Python

- Version 1.0 -

Thomas Peters

© 2025

Copyright © 2025 Thomas Peters
All rights reserved.

Inhaltsverzeichnis

Vorbemerkungen und Voraussetzungen

Dieser Text bezieht sich auf rekursive Algorithmen. Wir müssen daher definieren, was unter einem solchen Algorithmus zu verstehen ist. *Ein Algorithmus heißt rekursiv, wenn er mindestens eine Funktion enthält, die sich selbst aufruft.* Ein einfaches und bekanntes Beispiel ist die Berechnung von *Fakultäten*. $n!$ steht für das Produkt der ersten n natürlichen Zahlen, z.B. $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$. Normalerweise werden Fakultäten rekursiv definiert, und man setzt $0! = 1$.

$$n! = \begin{cases} n \cdot (n-1)! & \text{falls } n > 0 \\ 1 & \text{falls } n = 0 \end{cases} .$$

Diese Definition kann leicht in eine Python-Funktion umgesetzt werden:

```
def fakrek(n):
    if n == 0: return 1
    else: return n*fakrek(n-1)
```

Allerdings ist es nicht nötig, diese Funktion selbst zu programmieren. Vordefinierte Funktionen für Fakultäten befinden sich in den Modulen *math* und *scipy*.

Einige grundlegende Kenntnisse über das Programmieren in Python werden vorausgesetzt. Wenn Sie Anfängerin oder Anfänger sind, sollten Sie sich vor der Lektüre dieses Textes mit einem Tutorial über Python auseinandersetzen. Dort werden Sie lernen, wie man Schleifen mit `for` und `while` codiert und wie man mit Funktionen in Python umgeht. Die objektorientierte Programmierung mit Klassen wird in diesem Text nicht behandelt, denn es ist die Intention des Autors, die Programmtexte möglichst einfach zu halten.

Mehrere Beispiele in diesem Text benutzen *Turtlegrafik*. Ursprünglich wurde diese Art von Grafik für die Programmiersprache Logo entwickelt [2]. Die

Turtle (Schildkröte) war ein kleiner Roboter. Durch Anweisungen, die auf einer Computertastatur eingetippt wurden, konnte die Turtle umherbewegt werden. Dabei hinterließ sie eine Spur, die etwa von einem Filzstift gezeichnet wurde. So entstand die Turtlegrafik. Dieser Grafikmodus ist auch in Python verfügbar, nämlich im Modul *turtle* [3]. Turtlegrafik gestattet die Entwicklung zahlreicher Beispiele mit Rekursionen. Dabei wird sogar sichtbar, wie sich die Grafik auf dem Monitor entwickelt.
Das Beenden der Turtlegrafik und die anschließende Rückkehr zum benutzten Editor ist leider nicht immer einfach. Im folgenden Text ist die Version für PyCharm [4] dargestellt. Für Spyder [5] wird auf den Anhang verwiesen: Beenden der Turtlegrafik mit Spyder.

Kapitel 1 beginnt mit Beispielen für einfache rekursive Algorithmen. In Kapitel 2 werden mathematische Anwendungen von Rekursionen vorgestellt. Kapitel 3 handelt von rekursiven Algorithmen mit mehr als einem rekursiven Funktionsaufruf. In diesem Kapitel werden wir auch sehen, dass rekursive Algorithmen nicht immer vorteilhaft sind und es unter Umständen besser ist, sie durch iterative Algorithmen zu ersetzen. In Kapitel 4 schließlich werden Beispiele von verschränkt rekursiven Algorithmen behandelt, in denen sich rekursive Funktionen gegenseitig aufrufen.

Es gibt einige allgemeine Konzepte der Programmierung, die mit rekursiven Algorithmen verbunden sind. Bei „Divide and Conquer“ wird die Eingabe so lange in kleinere Unterprobleme aufgeteilt, bis für jedes dieser Mini-Probleme eine triviale Lösung gefunden werden kann. Die Teillösungen ergeben zusammengesetzt die Lösung des originalen Problems. Geläufige Beispiele sind *Binäre Suche*, siehe Abschnitt 1.1.4, und *merge sort*, siehe Abschnitt 3.4. Ein weiteres Konzept ist „Branch and Bound“. Dabei geht es darum, die Anzahl der zu untersuchenden Möglichkeiten zu minimieren. Das geschieht, indem die Menge der möglichen Lösungen aufgeteilt wird und für jede Teilmenge von Lösungen Begrenzungen (Bounds) eingeführt werden. Ein typisches Beispiel ist das ILP-Problem, also das Lösen einer linearen Optimierungsaufgabe, bei der nur ganzzahlige Lösungen zugelassen sind, siehe Abschnitt 3.7. „Backtracking“ ist ein Konzept, das bei solchen Problemen besonders hilfreich ist, in denen Randbedingungen erfüllt werden müssen, wie z.B. das Rucksackproblem, siehe Abschnitt 3.6.

Die in diesem Text vorgestellten Programme sind einfach gehalten. Es dürfte sicher möglich sein, sie kürzer und eleganter zu fassen. Hier wurde jedoch auf Verständlichkeit der größte Wert gelegt. In den letzten Jahren wurden Module für verschiedene Zwecke für Python entwickelt. Viele der hier vorge-

stellten Algorithmen könnten sehr einfach unter Verwendung eines Moduls gelöst werden. In diesem Text kommen jedoch nur solche Module vor, die nicht zusätzlich installiert werden müssen mit Ausnahme von *turtle* und im Falle des ILP-Programms *scipy* (Abschnitt 3.7).
Die Programme wurden in Python 3.9.13 geschrieben. Sie wurden mit den Versionen 3.11.5 und 3.13 getestet [1] . Mittlerweile dürfte es neuere Versionen geben, und es ist möglich, dass Teile der Programmcodes nicht mehr unter den neuen Versionen laufen. Doch wird ein Aktualisieren der Codes in den meisten Fällen nicht schwierig sein.

Kapitel 1

Einfache Rekursionen

1.1 Ein paar Beispiele zum Anfang

1.1.1 Harmonische Reihe

Die harmonische Reihe ist definiert durch

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... = \sum_{n=1}^{\infty} \frac{1}{n}$$

Jeder zu addierende Bruch ist kleiner als der vorangehende. Doch der Wert der Partialsummen H_N

$$H_N = \sum_{n=1}^{N} \frac{1}{n} \tag{1.1}$$

steigt allmählich mit wachsendem N. Schon im 14. Jahrhundert zeigte der französische Philosoph Nicole Oresme [6], dass die Partialsummen über alle Schranken wachsen, also divergieren. Doch schon um eine recht kleine Zahl wie 8 zu übertreffen, braucht man viele Folgenglieder, die alle addiert werden müssen. Wir entwickeln ein kurzes Programm für Partialsummen der harmonischen Reihe (Gl. (1.1)), um das minimale N zu bestimmen, für das $\sum_{n=1}^{N} \frac{1}{n}$ mehr als 8 ergibt. Eine geeignete rekursive Funktion ist sehr einfach:

```
def harmrek(n):
    if n == 1: return 1
    else: return harmrek(n-1) + 1/n
```

Die Berechnung der Partialsumme mit Obergrenze n wird darauf zurückgeführt, dass die Partialsumme mit der Obergrenze n-1 berechnet wird und anschließend $\frac{1}{n}$ addiert wird. Im Falle $n = 1$ gibt die Funktion 1 zurück.

Das Hauptprogramm besteht im Wesentlichen aus nur zwei Zeilen:

```
n = int(input('n = '))
print('H(',n,') = ',harmrek(n))
```

Die Funktion *input* interpretiert jede Eingabe als String, also als eine Zeichenkette. Diese muss in eine Zahl umgewandelt werden, dazu dient `int(...)`. In der zweiten Zeile werden die Obergrenze n und die zugehörige Partialsumme ausgegeben.

Die Partialsummen der harmonischen Reihe können durch $\ln n + \gamma$ approximiert werden. Dabei ist γ die Euler-Mascheroni Konstante mit $\gamma \approx 0.5772$ [7]. Um unsere Ergebnisse mit der Näherung zu vergleichen, müssen wir am Anfang des Programms das Modul *math* importieren. Die Konstante γ muss definiert werden, und eine Ausgabezeile wird hinzugefügt.

```
print('approximiert ',math.log(n)+ gamma)
```

Die Funktion *log* des Moduls *math* berechnet den natürlichen Logarithmus des Arguments.
Die Näherung wird umso besser, je größer n ist. Hier sind zwei Beispiele:

```
H(10) = 2.9289682539682538, approximiert 2.879800757896
H(1000) = 7.485470860550343, approximiert 7.48497094388367
```

Die Anwendung der Näherungsformel zeigt, dass zum Übertreffen der Partialsumme 8 bis etwa 1673 summiert werden muss. Das entwickelte Programm gestattet es, die exakte Lösung zu finden. Versuchen Sie es nicht mit wesentlich größeren Zahlen. Das Programm wird dann nicht funktionieren, und es erfolgt die Fehlermeldung: `RecursionError: maximum recursion depth exceeded in comparison`. Es gibt einige Internetseiten, die sich auf die maximale Rekursionstiefe von Python beziehen. Meist wird behauptet, sie sei 1000. Wie groß die Rekursionstiefe Ihres Computersystems mit Python ist, können Sie leicht selbst herausfinden. Tippen Sie einfach folgenden Zweizeiler ein und starten Sie ihn.

```
import sys
print(sys.getrecursionlimit())
```

Es ist sogar möglich, die Rekursionstiefe Ihres Systems zu verändern. Experten zufolge ist dies aber nicht ratsam.

1.1.2 Sechsecke

Dies ist ein erstes kleines Programm, das einige Funktionen der Turtlegrafik verwendet. Eine Übersicht über alle Möglichkeiten dieser Art von Grafik findet sich hier [3]. Wir wollen regelmäßige Sechsecke erzeugen, die den gleichen

Mittelpunkt haben. Die Kanten jedes folgenden Sechsecks sollen dabei kürzer sein als die des aktuellen. Zunächst aber wollen wir ein einzelnes Sechseck mit Turtlegrafik zeichnen. Dazu müssen wir das Modul „turtle“ importieren:

```
import turtle as tu
```

Damit ist das Modul für uns verfügbar, und wenn wir Funktionen des Moduls verwenden wollen, können wir leicht abgekürzt schreiben `tu.forward(100)` anstelle von `turtle.forward(100)`. Die Turtle ist anfangs so eingestellt, dass sie sich „ostwärts“ bewegt. Um ein Sechseck zu erhalten, muss sich die Turtle nach links drehen, aber um welchen Winkel? Je zwei benachbarte Seiten eines Sechsecks schließen einen Winkel von 120° ein. Solange wir der Turtle keine Anweisung geben, sich zu drehen, würde sie sich geradeaus bewegen. Die Abbildung 1.1 zeigt, dass der richtige Winkel, um den sich die Turtle drehen muss, gleich 60° ist.

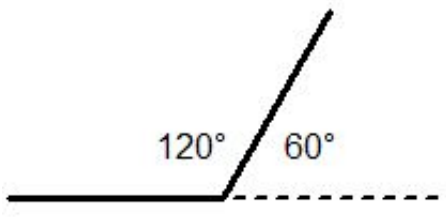


Abbildung 1.1: Die Turtle muss sich um 60° nach links drehen, bevor sie die nächste Seite zeichnet.

Damit können wir ein regelmäßiges Sechseck mit Seitenlänge a konstruieren, wenn wir die beiden folgenden Zeilen in unser Programm einfügen:

```
for i in range(6):
    tu.forward(a); tu.left(60)
```

Wir erhalten auf diese Weise ein einzelnes Sechseck. Aber wir wollen viele Sechsecke bekommen. Die Turtle befindet sich anfangs im Punkt (0,0), das ist die Mitte der Zeichenfläche. Dies soll auch der Mittelpunkt aller Sechsecke sein. Bevor die Turtle zu zeichnen beginnt, müssen wir sie dorthin bewegen, wo sie anfangen soll. Abbildung 1.2 zeigt, dass sich die Turtle um $\frac{\sqrt{3}}{2} \cdot a$, ungefähr $0.866 \cdot a$, nach unten bewegen muss und nach links um $\frac{1}{2}a$. In Python wird dies so codiert:

```
tu.up()
tu.setpos(-0.5*a,-0.866*a)
tu.down()
```

Wenn das Sechseck fertig ist, muss die Turtle zum Punkt (0,0) zurück wandern, ohne zu zeichnen:

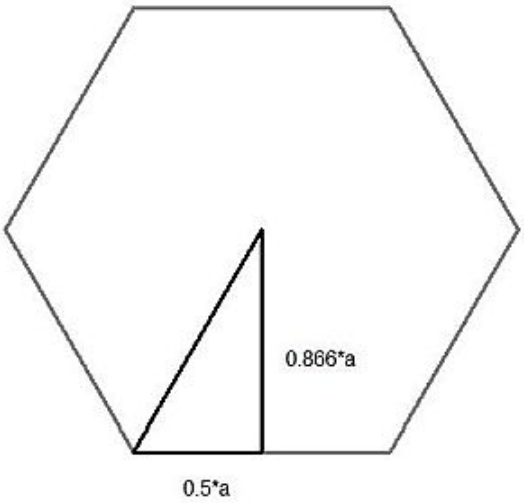


Abbildung 1.2: Ohne zu zeichnen, muss sich die Turtle um $0.866 \cdot a$ nach unten bewegen und um $\frac{1}{2}a$ nach links.

```
tu.up()
tu.setpos(0,0)
```

Wenn alles dieses ausgeführt wurde, kann ein rekursiver Aufruf wie etwa dieser folgen:

```
if a > 25: Sechseck(a*0.8)
```

Die `if`-Anweisung ist sehr wichtig. Wäre sie nicht vorhanden, würde das Programm ewig weiterlaufen. Man sagt, es *terminiere* nicht. Jede rekursive Funktion benötigt eine *Abbruchbedingung*. Die hier gewählten Beispiele sind: minimale Seitenlänge $a = 25$ und Verkleinerungsfaktor 0.8.

Die Sechsecke sollen farbig gezeichnet werden. Das äußere Sechseck soll rot sein, und die Farbe soll kontinuierlich in blau übergehen. Das können wir mit den folgen drei Zeilen erreichen:

```
rot *= 0.8; blau *= 1.25
if blau > 1: blau = 1
tu.pencolor(rot,0,blau)
```

Die Variablen für die Grundfarben rot, grün und blau können Werte von 0 bis 1 annehmen. Die aktuellen Werte für `rot` and `blau` werden mit 0.8 bzw. 1.25 multipliziert. Dadurch vermindert sich von Sechseck zu Sechseck der Anteil von rot, der von blau wird hingegen größer. Die nächste Zeile stellt sicher, dass der Wert von `blau` nicht größer als 1 werden kann. Die nächste Zeile sagt der Turtle, dass sie zum Zeichnen die aus rot und blau zusammengesetzte Farbe benutzen soll; grün wird hier nicht gebraucht und erhält den Wert 0. Hier ist das Listing der ganzen Funktion *Sechseck*:

```
def Sechseck(a, rot, blau):
    rot *= 0.8; blau *= 1.25
    if blau > 1: blau = 1
    tu.pencolor(rot,0,blau)
    tu.up()
    tu.setpos(-0.5*a,-0.866*a)
    tu.down()
    for i in range(6):
      tu.forward(a); tu.left(60)
    tu.up(); tu.setpos(0,0)
    if a > 25: Sechseck(a*0.8, rot, blau)
```

Schauen wir uns jetzt noch das Hauptprogramm an. Die Farben müssen initialisiert werden: `rot = 1; blau = 0.2`, und die Breite des Stifts der Turtle wird so festgelegt, dass sie nicht zu klein ist: `tu.pensize(width = 3)`. Danach wird die wesentliche Funktion aufgerufen: `Sechseck(200, rot, blau)`. Am Ende folgen einige Zeilen, die dem Benutzer mitteilen, dass die Zeichnung fertig ist. Schließlich soll die Zeichenfläche der Turtlegrafik geschlossen werden. Im Listing befindet sich die Version für PyCharm, für Spyder siehe Beenden der Turtlegrafik. Hier ist das ganze Hauptprogramm:

```
rot = 1; blau = 0.2
tu.pensize(width = 3)
Sechseck(200, rot, blau)
tu.up(); tu.hideturtle()
tu.setpos(-300,-250); tu.pencolor((0,0,0))
tu.write('fertig!',font = ("Arial",12, "normal"))
tu.exitonclick() # für PyCharm
try: tu.bye()
except tu.Terminator: pass
```

Die nachstehende Abbildung 1.3 zeigt die ersten drei Schritte der konzentrischen Sechsecke wie oben beschrieben.

1.1.3 Palindrome

Ein Palindrom kann ein Wort sein, das von links nach rechts und von rechts nach links gelesen gleich ist, wie z.B. „Rentner“. Es kann aber auch eine Zahl sein wie etwa 43134. Auch Sätze können Palindrome sein wie „Ein Esel lese nie“. Auf die Groß- und Kleinschreibung wird bei der Kennzeichnung von Palindromen kein Wert gelegt. Wir wollen ein kleines Programm entwickeln, das prüft, ob eine eingegebene Zeichenkette (Wort, Satz oder Zahl) ein Palindrom ist. [8]

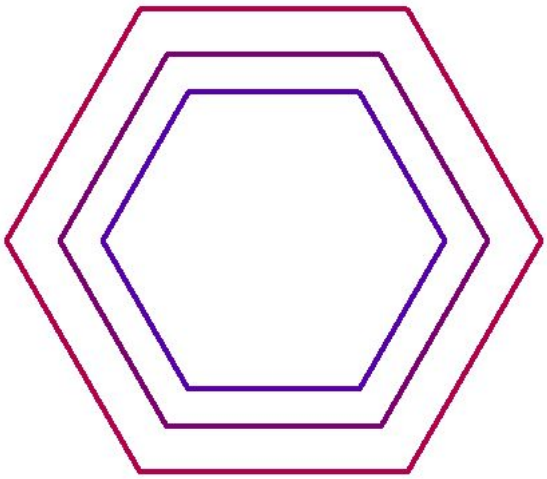


Abbildung 1.3: Die ersten drei Sechsecke, Farben verstärkt

Dazu ist es nützlich, die Kennzeichnung des Begriffs anders zu fassen. Wir definieren:
Eine Zeichenkette P ist ein Palindrom,

- wenn sie höchstens ein Zeichen enthält
- oder wenn die Zeichenkette P' ein Palindrom ist, die man erhält, wenn man von P das erste und das letzte Zeichen streicht.

Diese Kennzeichnung erlaubt es, eine rekursive Funktion `istPalindrom` zu entwickeln, die prüft, ob eine Zeichenkette ein Palindrom ist. In der Funktion bedeutet `p` das Ergebnis der Prüfung, also `True` oder `False`. Dieser Wert wird an die aufrufende Funktion zurückgegeben. Der erste Fall ist sehr einfach. `len(st)` ermittelt die Länge der Zeichenkette. Ist sie kleiner oder gleich 1, liegt ein Palindrom vor.
Der zweite Fall in obiger Definition bedarf der Erläuterung. Die Zeichenkette `st` (von engl. *string*) sei 'RADAR'. Python kann auf jedes einzelne Zeichen zugreifen, dabei werden die Zeichen mit 0 anfangend nummeriert. Z.B. ist st[2] = 'D', st[0] = st[4] = 'R'. Man kann aus einer Zeichenkette auch Teile ausschneiden. Die ersten drei Zeichen von `st` erhält man durch `st[:3]`, also `st[:3] = 'RAD'`. Es werden die Zeichen von `st` bis zu der durch die Zahl angegebenen Länge übernommen. Setzt man eine Zahl vor den Doppelpunkt, werden die Zeichen ab dieser Nummer übernommen, z.B. `st[2:] = 'DAR'`. Unsere Funktion soll die Zeichenkette außer dem ersten Zeichen verwenden, also `st[1:]`. Wie aber sagen wir dem Programm, dass auch das letzte Zeichen entfernt werden soll, wenn wir die Länge der Zeichenkette nicht

kennen? Dafür gibt es in Python eine Besonderheit, nämlich die negativen Indizes. `st[1:-1]` bedeutet die Zeichenkette, von der wir das erste und das letzte Zeichen entfernt haben. (Wollten wir die letzten beiden Zeichen entfernen, lautete der Term `st[1:-2]`.)
Ein rekursiver Aufruf `istPalindrom(st[1:-1])` soll aber nur stattfinden, wenn die um das erste und letzte Zeichen verkürzte Zeichenkette ein Palindrom sein kann. Daher wird im `else`-Zweig der Funktion zuerst abgefragt, ob das erste und letzte Zeichen verschieden sind: `st[0] != st[-1]`. Dann nämlich wird `p = False`. Ist das aber nicht der Fall, wird der rekursive Aufruf durchgeführt. So lautet unsere Funktion:

```
def istPalindrom(st):
    if len(st) <= 1: p = True
    else:
       if st[0] != st[-1]: p = False
       else: p = istPalindrom(st[1:-1])
    return p
```

In unserem Programm gibt es eine zweite Funktion: `checkPalindrom(st)`. Sie soll die eingegebene Zeichenkette ein wenig aufbereiten, die Funktion `istPalindrom` aufrufen und das Ergebnis ausgeben. Durch die Zeile `stu = st.upper()` werden Buchstaben der eingegebenen Zeichenkette in Großbuchstaben umgewandelt. Aus `st = 'Radar'` wird `stu = 'RADAR'`. In den beiden folgenden Zeilen werden Leerzeichen und Kommas entfernt, indem sie jeweils durch `''`, den Leertext, ersetzt werden. (Auf die gleiche Weise könnte man noch andere Sonderzeichen entfernen.) Der Rest der Funktion erklärt sich selbst.

```
def checkPalindrom(st):
    stu = st.upper()
    stu = stu.replace(' ','')
    stu = stu.replace(',','')
    print(st)
    palin = istPalindrom(stu)
    if palin == True: print('ist ein Palindrom.')
    else: print('ist kein Palindrom.')
    print()
```

Das Hauptprogramm besteht im Prinzip nur aus einer Zeile, nämlich dem Aufruf von `checkPalindrom`. Doch kann man diese Zeile auch mehrfach mit unterschiedlichen Zeichenketten schreiben, z.B.

```
checkPalindrom('Reliefpfeiler')
checkPalindrom('12022021')
checkPalindrom('Regenpfeifer')
```

Letztere Eingabe ist natürlich kein Palindrom.

1.1.4 Binäre Suche

Eine Standardaufgabe in der Datenverarbeitung ist die Suche nach bestimmten Daten. Die Eingabe ist ein Schlüsselwort oder eine Zahl. Das Datenverarbeitungsprogramm soll den entsprechenden Datensatz (oder Datensätze) finden oder dem Benutzer melden, dass der gesuchte Datensatz nicht existiert. In dem Falle, dass die Schlüsselwörter nicht sortiert sind, muss das Programm jedes einzelne prüfen. Das ist sehr zeitaufwendig. Für sortierte Schlüsselwörter gibt es jedoch viel effektivere Suchverfahren. Eines davon ist die Binäre Suche, die wir in diesem Abschnitt behandeln wollen. Dazu wird eine Datei mit 1024 Vornamen in die Liste `NamenListe` geladen. Weil dieser Programmteil nichts mit der eigentlichen Binären Suche zu tun hat, betrachten wir den Teil nicht näher.

Die entscheidende Funktion ist `Suche`. Sie benötigt die zwei Parameter `l,r`. Das sind die linke und rechte Grenze des Bereichs, in dem die Suche durchgeführt wird. Die Variable `gefunden` muss durch `False` initialisiert werden, denn beim Aufruf der Funktion wurde noch nichts gefunden. Dann wird der Mittelwert `m` von `l` und `r` bestimmt: `m = round((l+r)/2)`. Der aktuelle Name `NamenListe[m]` wird ausgegeben, sodass man sehen kann, wie das Programm arbeitet. Es gibt drei Möglichkeiten: (1) `na == NamenListe[m]` bedeutet, dass der Name `na` gefunden wurde. In diesem Falle endet die Suche, und `gefunden = True` und `m` werden zurückgegeben. (2) Der Name `na` steht in alphabetischer Reihenfolge vor `NamenListe[m]`. In diesem Falle muss zusätzlich die Bedingung `l < m` geprüft werden. Wenn beide Bedingungen erfüllt sind, wird die Suche im Intervall [*l,m-1*] fortgesetzt:
`if (na < NamenListe[m]) & (l < m): gefunden, m = Suche(l,m-1)`
Wenn jedoch die erste Bedingung erfüllt ist, die zweite aber nicht, ist der Name `na` nicht in der `NamenListe` vorhanden. (3) Der Name `na` steht in alphabetischer Reihenfolge hinter `NamenListe[m]`. Dieser Fall ist ganz analog zum vorigen zu behandeln. Wenn `m < r` gilt, wird die Suche in der rechten Hälfte der `NamenListe` fortgesetzt, anderenfalls befindet sich `na` nicht in der Liste der Vornamen.
Es muss noch erklärt werden, warum `Suche` zwei Werte zurückgeben muss: `m` und `gefunden`. Das liegt daran, dass in Python alle Variablen *lokal* sind. Angenommen, `na` wird in einem bestimmten rekursiven Aufruf von `Suche` gefunden, z.B. `m` = 47 und `gefunden = True`. Dann sind diese Werte nur dem aktuellen Aufruf von `Suche` bekannt. Die Kopie von `Suche`, welche die aktuelle Version aufgerufen hat, „weiß nichts“ von der erfolgreichen Suche. Für sie könnte `m` = 48 sein und `gefunden = False`. Deswegen müssen wir die Werte von `m` und `gefunden` an die aufrufende Kopie von `Suche` übergeben.

Hier ist das vollständige Listing der Funktion `Suche`:

```
def Suche(l,r):
    gefunden = False
    m = round((l+r)/2)
    print(' Name[',m,'] = ',NamenListe[m])
    if na == NamenListe[m]: gefunden = True
    if (na < NamenListe[m]) & (l < m):gefunden, m = Suche(l,m-1)
    if (na > NamenListe[m]) & (m < r):gefunden, m = Suche(m+1,r)
    return gefunden, m
```

Ein paar Worte zum Hauptprogramm: Abgesehen von der Überschrift findet sich darin nur die Eingabe des Namens `na`, nach dem gesucht werden soll. Der Aufruf der Funktion `Suche` folgt: `gefunden, m = Suche(0,n-1)`. Die linke und rechte Grenze sind `0` und `n-1`, der niedrigste und höchste Index von `NamenListe`. Schließlich wird das Ergebnis der Suche ausgegeben.

Warum ist die Binäre Suche vorteilhaft? Wenn auf einer unsortierten Liste von 1024 Einträgen die lineare Suche ausgeführt wird, ist die mittlere Anzahl von notwendigen Vergleichen 512. Angenommen, in einer sortierten Liste sind ebenfalls 1024 Einträge vorhanden. Bei der Binären Suche ist die Anzahl der zu durchsuchenden Einträge nach dem ersten Vergleich nur noch 512, nach dem zweiten 256 usw. Daher sind höchstens $10 = log_2 1024$ Vergleiche nötig, um einen Eintrag zu finden oder festzustellen, dass er nicht vorhanden ist. Das wiederholte Aufspalten der Liste ist ein Beispiel für das allgemeine Programmierkonzept „Divide and Conquer“ (Teile und herrsche).

1.2 Top-of-Stack Rekursionen

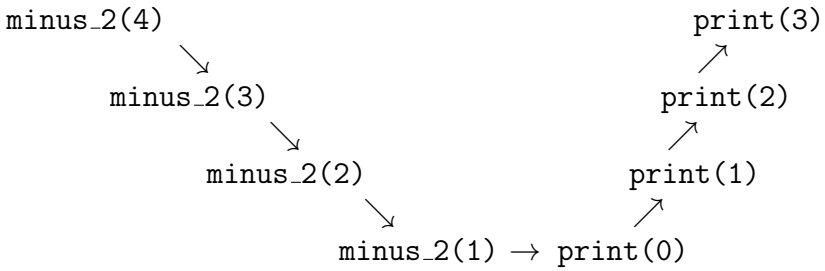
1.2.1 Der kleine Unterschied

Betrachten wir diese beiden kleinen Funktionen:

```
def minus(i):
    i -= 1
    print(i,end= ' ')
    if i > 0: minus(i)

def minus_2(i):
    i -= 1
    if i > 0: minus_2(i)
    print(i,end= ' ')
```

Sie werden von diesem kurzen Hauptprogramm aufgerufen:
`n = int(input('n = '))`
`minus(n); print()`
`minus_2(n); print()`
Angenommen, `n = 4`. Funktion `minus` gibt als erstes `3` aus, dann folgt der rekursive Aufruf. Folglich erscheint `2` auf dem Monitor, danach `1` und schließlich `0`. Funktion `minus_2` bewirkt genau das Umgekehrte. Die Ausgabe beginnt mit 0 und endet mit 3. Woran liegt das? Bevor etwas ausgegeben wird, wird der rekursive Aufruf ausgeführt. Mehrfache Aufrufe der Funktion `minus_2` werden folgendermaßen ausgeführt:



Rekursionen verwenden einen bestimmten Teil des Computerspeichers, den so genannten *Stack*. Dieser ist vergleichbar mit einer Sackgasse, in die nur eine bestimmte Anzahl Autos hineinfahren kann und nur rückwärts wieder hinaus. Der Stack folgt dem Prinzip „LIFO“, die Abkürzung für „Last In First Out“ (Als letztes hinein, als erstes hinaus). Die Skizze oben zeigt, dass jeder rekursive Aufruf Informationen auf den Stack schiebt: den Namen der Funktion und einen oder mehrere Parameter. In Funktion `minus_2` werden diese Informationen erst dann vom Stack heruntergeholt, wenn die Abbruchbedingung erfüllt ist, d.h. `if i > 0` ist nicht mehr `True`. Dann wird `0` ausgegeben, und der Aufruf `minus_2(1)` ist abgearbeitet worden. Danach wird `1` dargestellt, und `minus_2(2)` ist fertig. So wird die Verarbeitung von unten nach oben durchgeführt. Auf diese Weise arbeiten rekursive Funktionen normalerweise.

Aber die Funktion `minus` ist anders. Zuerst erfolgt die Ausgabe, dann erst kommt der rekursive Aufruf. Ausgaben und rekursive Aufrufe wechseln einander ab, bis `i > 0` nicht mehr zutrifft.
`minus(4)` → `print(3)` → `minus(3)` → `print(2)` → `minus(2)`
→ `print(1)` → `minus(1)` → `print(0)`
Funktionen, in denen der rekursive Aufruf die letzte Anweisung ist, nennt man *Top-of-Stack Rekursionen*. Es gibt Programmiersprachen, die einen rekursiven Aufruf vom Stack entfernen, wenn der Aufruf abgearbeitet ist. In

solchen Sprachen kann eine Top-of-Stack Rekursion beliebig lange laufen. Python gehört nicht zu dieser Klasse von Programmiersprachen, und die Anzahl der Aufrufe ist begrenzt.

1.2.2 Rosetten

Obwohl die Möglichkeiten der Top-of-Stack-Rekursionen beschränkt sind, gibt es einige schöne Beispiele. Als erstes betrachten wir eines, in dem Turtlegrafik zum Zeichnen von Rosetten aus Vielecken mit drei, vier, fünf und sechs Ecken angewendet wird. Im Hauptprogramm finden sich keine Besonderheiten. Die Eingabe `n` ist die Anzahl der Ecken eines Vielecks, und `Winkel` erhält den Wert `360/n`. Diese Variable wird für das Zeichnen der Vielecke gebraucht. Die Breite des Zeichenstifts wird auf `2` gesetzt, damit die gezeichneten Strecken nicht zu dünn werden. Dann erfolgt der Aufruf, mit dem die Rosette gezeichnet wird: `Form(0,600/n)`. Wenn das Programm abgearbeitet ist, kommt der aus Abschnitt 1.1.2 bekannte Schlussteil.

Schauen wir uns an, wie `Form(Zaehler,x)` funktioniert. Die Farbe des Zeichenstifts wird als erstes festgelegt. Das erfolgt durch die zufällige Wahl von Werten für den Anteil von rot, grün und blau, die zwischen 0 und 1 liegen müssen, z.B. durch `rot = rd.uniform(0,1)`. Dafür muss das Modul `random` im Programmkopf importiert werden: `import random as rd`. Dann wird das Vieleck gezeichnet. Dafür werden nur zwei Zeilen benötigt:

```
for i in range(n):
  tu.forward(x); tu.right(Winkel)
```

`x` ist die Seitenlänge des Vielecks. Im Hauptprogramm setzten wir `x = 360/n`. `Zaehler` bezeichnet die Anzahl der Aufrufe, die gebraucht werden, damit die Rosette vollständig gezeichnet wird. Jedes Mal, nachdem die Turtle ein Vieleck gezeichnet hat, dreht sie sich um einen kleinen Winkel: `tu.right(7)`. Die letzte Zeile ist der rekursive Aufruf:

```
if Zaehler < 360/7: Form(Zaehler+1,x)
```

Das bedeutet, dass die Seitenlänge immer gleich bleibt, die Variable `Zaehler` hingegen um eins erhöht wird. Auf diese Weise werden viele Vielecke gezeichnet, die jeweils um kleine Winkel verdreht sind. Eine Rosette entsteht. Insgesamt die das Programm sehr kurz und leicht zu verstehen. Dennoch kommt eine Zeichnung wie die von Abbildung 1.4 dabei heraus.

1.2.3 Größter gemeinsamer Teiler

Der *größte gemeinsame Teiler ggT* zweier natürlicher Zahlen ist die größte Zahl, durch die beide Zahlen ohne Rest teilbar sind. Hier betrachten wir eine

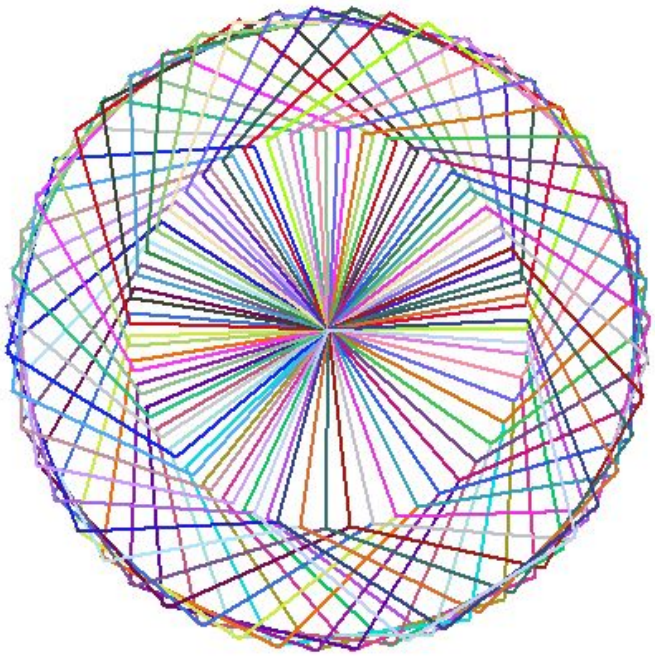


Abbildung 1.4: Diese Rosette besteht aus Fünfecken.

Möglichkeit, den ggT von mehr als nur zwei Zahlen zu bestimmen. Dabei wird der bekannte *Euklidische Algorithmus* angewendet, siehe [9].
Das Hauptprogramm ist sehr kurz. Der Benutzer wird gebeten, mindestens zwei Zahlen einzugeben. Dabei stellen die beiden kleinen `while`-Schleifen sicher, dass die eingegebenen Zahlen nicht 0 sein können. Dann wird die Funktion `ggT` aufgerufen. Hier ist das Listing:

```
a, b = 0, 0
while a == 0: a = int(input(erste Zahl: '))
while b == 0: b = int(input('zweite Zahl: '))
ggT(a,b)
```

In der Funktion selbst sorgen wir dafür, dass auch negative Zahlen verarbeitet werden können. Sie werden erforderlichenfalls in positive umgewandelt.

```
if x < 0: x = -x; if y < 0: y = -y
```

Dann folgen vier Zeilen, die den Euklidischen Algorithmus enthalten.

```
r = 1
while r > 0:
    r = x % y; x = y; y = r
print('Der ggT ist ',x)
```

`r` bezeichnet den Divisionsrest, der anfänglich auf 1 gesetzt wird. Der Rest,

der sich bei Division von x durch y ergibt, wird durch `x % y` ermittelt. x muss nicht größer sein als y, z.B. 5 % 7 = 5. Danach werden die Variablen verschoben. `x` erhält den Wert von `y`, und `y` wird auf `r` gesetzt. Zum Beweis der Korrektheit des Algorithmus siehe [9].
Danach wird der Benutzer aufgefordert, eine weitere Zahl einzugeben, um die Berechnung des ggT fortzusetzen.

```
z = int(input('neue Zahl: '))
if z != 0: ggT(x,z)
```

Wenn als neue Zahl `0` eingegeben wird, endet das Programm. Anderenfalls wird der ggT vom bisherigen ggT und der neuen Zahl berechnet. Beispiel: Gesucht ist der ggT von 525, 1365, 1015 und 749. Die Funktion liefert zunächst ggT(525,1365) = 105. Dann gibt man 1015 ein, und ggT(105,1015) = 35. Nach Eingabe von 749 erhält man schließlich als Ergebnis ggT(35,749) = 7. Weil sich der rekursive Aufruf in der letzten Zeile der Funktion `ggT` befindet, liegt eine Top-of-Stack-Rekursion vor.

1.3 Übungsaufgaben

1. **Alternierende Harmonische Reihe**
   (a) Entwickeln Sie ein Programm zur Berechnung der Partialsummen der alternierenden harmonischen Reihe, die folgendermaßen definiert ist:

   $$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + -... = \sum_{n=1}^{\infty} (-1)^{n+1} \frac{1}{n}$$

   Ihr Programm sollte eine rekursive Funktion enthalten. Zusätzlich soll das Programm die Differenz zu ln 2 berechnen, denn das ist der Grenzwert der Reihe. Hinweis: Um ln 2 zu berechnen, müssen Sie `numpy` importieren: `import numpy as np`. Dann ist der natürliche Logarithmus von 2 durch `np.log(2)` zu berechnen. [10].
   (b) Die alternierende Reihe

   $$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + -... = \sum_{n=0}^{\infty} (-1)^{n} \frac{1}{2n+1}$$

   hat den Grenzwert $\frac{\pi}{4}$. Schreiben Sie ein Programm, das eine Näherung für π anhand von Partialsummen obiger Reihe berechnet.

2. **Schneckenhaus**
   Schreiben Sie ein Programm, das eine Zeichnung ähnlich Abbildung 1.5 erstellt. Das Schneckenhaus besteht aus Quadraten mit abnehmender

Seitenlänge. Nur zwei Funktionsaufrufe sind erforderlich, um ein Quadrat zu färben: `tu.begin_fill()` und `tu.end_fill()` vor und hinter dem Programmtext der Zeichnung, die Sie färben möchten. Hinweis: Rot wird durch das Zahlentripel (1,0,0) und cyan durch (0,1,1) kodiert.

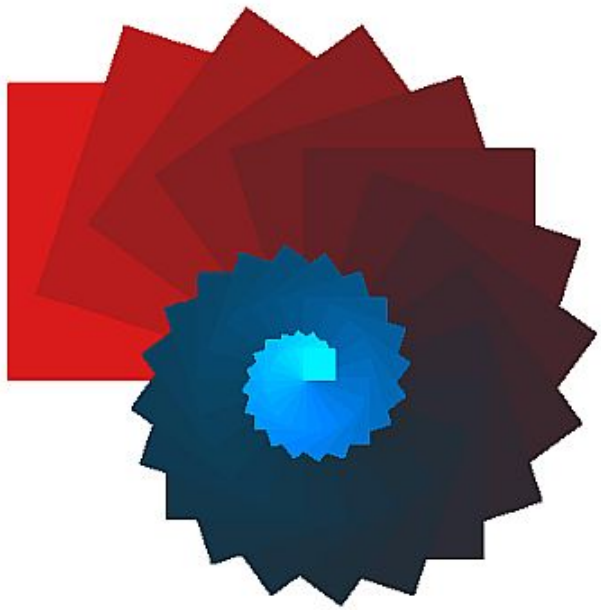


Abbildung 1.5: Schneckenhaus mit Farben, die von rot zu cyan übergehen

3. **Programm „macht_was“**
   Beschreiben Sie, was die Funktionen `macht_was` und `macht_was2` ausgeben und warum sie das tun. Welche Funktion ist eine Top-of-Stack-Rekursion?

```
def macht_was(k):
    print(st[k],end = '')
    if st[k] != ' ': macht_was(k+1)
def macht_was2(k):
    if st[k] != ' ': macht_was2(k+1)
    print(st[k],end = '')
```

4. **Schneeflocken**
   Eine Schneeflocke besteht aus einem Kristall. der mehrere kleinere Teilkristalle enthält, welche die gleiche Form haben wie der erste. Ein Elementarkristall ist ein Stern, der eine bestimmte Anzahl von Strahlen

hat. Erstellen Sie ein Programm zum Zeichnen von Schneeflocken mit Sternen aus mindestens 3 und höchstens 6 Strahlen. Programmieren Sie dazu eine rekursive Funktion mit Turtlegrafik. Ihre Zeichnung könnte so aussehen wie in Abbildung 1.6, siehe auch [**?**].

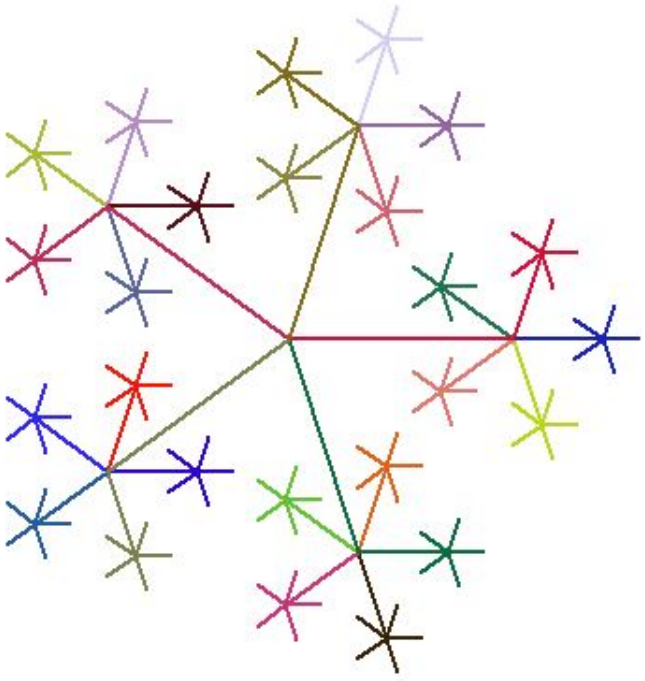


Abbildung 1.6: Schneeflocke - die Sterne haben fünf Strahlen

Hinweise: Vom Mittelpunkt der Zeichnung bis zum aktuellen Stern bewegt sich die Turtle um `x` vorwärts. Dann wird die Funktion `Flocke` aufgerufen mit dem Parameter `k*x`. Wenn die Zahl der Strahlen 3 oder 4 ist, setzen Sie $k = 0.5$, sonst $k = 0.4$. Wenn die Funktion fertig ist, muss sich die Turtle um `x` zurück bewegen. Damit sie den nächsten Strahl zeichnen kann, muss sie sich zuvor um den Winkel $360°/n$ drehen. n ist dabei die Anzahl der Strahlen.

5. **Primfaktoren**
   Schreiben Sie ein Programm mit einer Top-of-Stack-Rekursion, das die Primfaktoren einer gegebenen, nicht zu großen Zahl berechnet.
   Hinweis: x ist durch y teilbar, wenn `x % y == 0` gilt.

6. **Straße**
   Schreiben Sie ein Programm mit einer Top-of-Stack-Rekursion zum Zeichnen einer „Straße“ mit Häusern mit unterschiedlichen Größen, ähnlich der Abbildung 1.7.

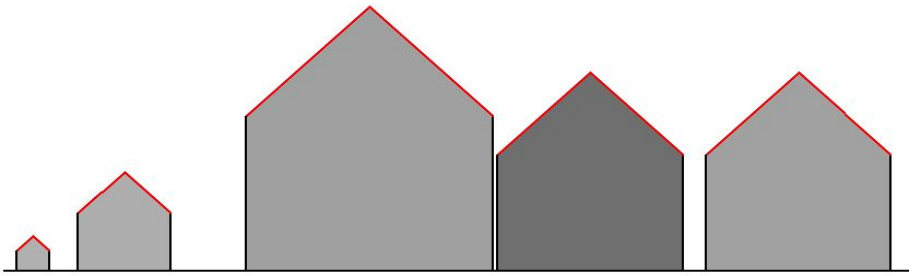


Abbildung 1.7: Straße mit Häusern unterschiedlicher Größe

   Hinweise: Unterschiedliche Schattierungen von grau erhält man durch `gray = rd.uniform(0.4,0.9)`. Achten Sie darauf, das Modul *random* zu importieren. Fragen Sie die x-Koordinate der linken unteren Ecke eines Hauses ab, etwa `if tu.xcor() < 200:`. Daraus können Sie die Abbruchbedingung gewinnen.

Kapitel 2

Rekursive Algorithmen in der Mathematik

Es gibt sehr viele rekursive Algorithmen in der Mathematik; im vorigen Kapitel wurden schon welche behandelt. Einige etwas anspruchsvollere werden in diesem Kapitel besprochen und in Python codiert. Oft werden in diesem Zusammenhang die *Fibonacci-Zahlen* genannt. Doch es gibt darüber schon eine ganze Reihe von Internetseiten, z.B. [11], [12] . Weitere Anwendungen von Rekursionen in der Mathematik finden sich im nächsten Kapitel, wo Algorithmen mit mehr als einem rekursiven Aufruf thematisiert werden.

2.1 Brüche kürzen

Ein Bruch kann gekürzt werden, indem man Zähler und Nenner durch ihren größten gemeinsamen Teiler dividiert. Doch meistens kürzt man nicht auf diese Weise. Statt dessen versucht man, Zähler und Nenner so lange, wie es ohne Rest geht, durch kleine Zahlen zu dividieren. Dieses Vorgehen wollen wir in Python programmieren.

Das Hauptprogramm wird nicht in Einzelheiten besprochen. Zähler und Nenner werden als positive Zahlen eingegeben. Dann erfolgt der rekursive Aufruf: `kuerze(prim, Zaehler, Nenner)`.
`prim` steht für eine Primzahl. Anfangs hat `prim` den Wert 2. Soweit möglich, sollen `Zaehler` und `Nenner` durch `prim` dividiert werden. Wenn das nicht mehr möglich ist, wird die nächst höhere Primzahl gesucht.

Die Funktion `kuerze` ruft als erstes die Funktion `suche_prim` auf. Diese gibt `prim` und `Erfolg` zurück. Wenn `Erfolg == True` gilt, kann der Bruch

gekürzt werden:
`Zaehler = Zaehler // prim` und `Nenner = Nenner // prim`.
Der doppelte Schrägstrich bezeichnet die Ganzzahldivision. Eventuelle Stellen nach dem Komma werden abgeschnitten, und das Ergebnis ist ganzzahlig, z.B. $22 // 5 = 4$. Dann wird das Zwischenergebnis ausgegeben:
`print('=',Zaehler,'/','Nenner',end = '')`.
Dabei bewirkt `end = ''`, dass nach der `print(...)`-Anweisung keine neue Zeile begonnen wird. Schließlich wird `kuerze` mit möglicherweise veränderten Parametern aufgerufen. Es folgt das Listing der ganzen Funktion:

```
def kuerze(prim, Zaehler, Nenner):
    prim, Erfolg = suche_prim(prim, Zaehler, Nenner)
    if Erfolg:
       Zaehler = Zaehler // prim
       Nenner = Nenner // prim
       print(' = ',Zaehler,'/',Nenner,end = ' ')
       kuerze(prim, Zaehler, Nenner)
```

Schauen wir uns jetzt einmal an, was die Funktion `suche_prim` leistet. Ihre Aufgabe ist es, eine Primzahl zu bestimmen, durch die `Zaehler` und `Nenner` ohne Rest geteilt werden können. Zudem wird geprüft, ob der Vorgang des Kürzens fortgesetzt werden kann.
Dazu werden zwei boolesche Variablen gebraucht, also solche, die nur `True` oder `False` sein können. `moeglich` ist solange `True`, wie Kürzen noch möglich sein könnte, also `Zaehler` und `Nenner` größer oder gleich `prim` sind. `Erfolg` wird erst einmal auf `False` gesetzt. Aber die Variable wird `True`, sobald `Zaehler` und `Nenner` ohne Rest durch `prim` teilbar sind. Anfänglich gilt also `moeglich = True; Erfolg = False`.
Was soll geschehen, wenn `Zaehler` und `Nenner` nicht beide durch `prim` dividiert werden können? Zunächst muss `prim` um 1 erhöht werden: `prim += 1`. Dann wird erneut geprüft, ob Kürzen noch möglich ist: `if (Zaehler < prim) or (Nenner < prim): moeglich = False`. Auch wenn Kürzen im Prinzip noch möglich ist, kann es sein, dass `Zaehler` und `Nenner` nicht ohne Rest durch `prim` teilbar sind. In diesem Falle wird `suche_prim` rekursiv aufgerufen: `prim, Erfolg = suche_prim(prim, Zaehler, Nenner)`. Dabei ist zu beachten, dass der Wert von `prim` mittlerweile geändert wurde.
Wenn jedoch `prim` Teiler von `Zaehler` und `Nenner` ist, wird der `else`-Zweig aktiv, und `Erfolg = True`. Endlich werden `prim` und `Erfolg` zurückgegeben. Dies ist die ganze Funktion `suche_prim`:

```
def suche_prim(prim, Zaehler, Nenner):
    moeglich = True; Erfolg = False
```

```
if (Zaehler % prim != 0) or (Nenner % prim != 0):
   prim += 1
   if (Zaehler < prim) or (Nenner < prim): moeglich = False
   if moeglich:
      prim, Erfolg = suche_prim(prim, Zaehler, Nenner)
else: Erfolg = True
return prim, Erfolg
```

Damit ist unser spezielles Programm zum Kürzen fertig; es enthält sogar zwei Funktionen mit rekursiven Aufrufen. Hier ist ein Beispiel für die Ausgabe: `48/84 = 24/42 = 12/21 = 4/7`.

2.2 Heron-Verfahren

Heron lebte um ungefähr 50 n.Chr. in Alexandria, Ägypten [13]. Er erfand ein Verfahren zur näherungsweisen Berechnung von Quadratwurzeln [14]. Der *Radikand* sei a, also die Zahl, deren Quadratwurzel x bestimmt werden soll, so dass $x = \sqrt{a}$. Mit x_0 bezeichnen wir die Ausgangsnäherung, z.B. $x_0 = 1$. Damit berechnen wir

$$x_1 = \frac{1}{2} \cdot (x_0 + \frac{a}{x_0})$$

Offensichtlich ist eine der Zahlen x_0, $\frac{a}{x_0}$ größer als $\sqrt{a}$ und die andere kleiner. x_1 ist eine bessere Näherung von $\sqrt{a}$ als x_0, denn es ist der Mittelwert von x_0 und $\frac{a}{x_0}$. Wieder ist eine der Zahlen x_1 und $\frac{a}{x_1}$ größer als $\sqrt{a}$ und eine kleiner. Wir können also erneut den Mittelwert der beiden Zahlen bilden: $x_2 = \frac{1}{2} \cdot (x_1 + \frac{a}{x_1})$ usw., allgemein

$$x_{i+1} = \frac{1}{2} \cdot (x_i + \frac{a}{x_i}) \tag{2.1}$$

Aber woher wissen wir, dass die Folge $< x_i >$ wirklich gegen $\sqrt{a}$ strebt? Durch Lösen einer quadratischen Gleichung kann man leicht zeigen, dass für alle natürlichen Zahlen $i \in \mathbb{N}$ gilt:

$$x_i > \sqrt{a} \text{ und } \frac{a}{x_i} < \sqrt{a} \tag{2.2}$$

Weil $x_{i+1} < x_i$ für alle i gilt, fällt die Folge $< x_i >$ monoton, und die Folge $< \frac{a}{x_i} >$ steigt monoton. Wegen Gl. (2.2) ist die Folge der Intervalle $[\frac{a}{x_i}, x_i]$ eine Intervallschachtelung und konvergiert gegen $\sqrt{a}$.
Jetzt sind wir bereit, das Heron-Verfahren zu implementieren. Das Hauptprogramm ist kurz und sieht so aus:

```
print('Heron-Verfahren zur Berechnung von Quadratwurzeln')
a = float(input('Radikand = '))
d = int(input('Anzahl von Dezimalen: '))
epsilon = 10**(-d)
Heron(1,0)
```

Die Variable d bezeichnet die gewünschte Genauigkeit der Näherung und wird in *epsilon* umgewandelt. Schließlich wird die Funktion `Heron` mit den Parametern 1 und 0 aufgerufen. Dabei ist 1 die nullte Näherung, also $x_0 = 1$, 0 ist die Nummer der Näherungen.

Auch die Funktion `Heron` sieht sehr einfach aus. Zuerst erhöhen wir die Zahl der ausgeführten Schritte `i` um eins. Dann wird Gl. (2.1) angewendet: `x_neu = 0.5*(x + a/x)` und das neu berechnete Intervall wird angezeigt: $[\frac{a}{x_i}, x_i]$. Endlich kommt noch der rekursive Aufruf: `if x_neu - a/x_neu > epsilon: Heron(x_neu,i)`. Das ist schon alles, und hier ist das komplette Listing:

```
def Heron(x,i):
    i += 1
    x_neu = 0.5*(x + a/x)
    print('Schritt ',i,'[',a/x_neu,',',x_neu,']')
    if x_neu - a/x_neu > epsilon: Heron(x_neu,i)
```

Das Heron-Verfahren kann leicht so erweitert werden, dass man Näherungen auch für die k-te Wurzel einer gegebenen Zahl berechnen kann. Indem man das Newton-Verfahren für Nullstellen auf die Funktion zu $f(x) = x^k - a$ anwendet, kann man zeigen, dass Gl. (2.1) durch

$$x_{i+1} = (1 - \frac{1}{k}) \cdot x_i + \frac{a}{k \cdot x_i^{k-1}} \tag{2.3}$$

zu ersetzen ist. Schreiben Sie zur Übung auch ein Programm für k-te Wurzeln. Hier können aber nur Näherungen ausgegeben werden, keine Intervalle.

2.3 Catalan-Zahlen

Manch ein Leser mag vielleicht nicht wissen, was Catalan-Zahlen sind. Sie spielen jedoch eine wichtige Rolle in der Kombinatorik. Benannt sind sie nach dem belgischen Mathematiker Eugène Catalan (1814-1894). Sie werden

durch zwei äquivalente Terme definiert [15]:

$$C_n = \frac{1}{n+1} \cdot \binom{2n}{n} = \frac{(2n)!}{(n+1)! \cdot n!} \tag{2.4}$$

Aber wozu sind sie gut? Hier sind ein paar Beispiele:

(1) Klammern: Ein Term wie $18 - 9 - 4 - 1$ ergibt keinen Sinn, wenn keine Klammern gesetzt werden. Dies sind die verschiedenen Möglichkeiten:

$((18 - 9) - 4) - 1 = 4$
$(18 - (9 - 4)) - 1 = 12$
$(18 - 9) - (4 - 1) = 6$
$18 - ((9 - 4) - 1) = 14$
$18 - (9 - (4 - 1)) = 12$

Weil drei Subtraktionen auszuführen sind, gibt es fünf Möglichkeiten, Klammern zu setzen. Mithilfe von Gl. (2.4) bestätigt man , dass $C_3 = 5$ gilt. Allgemein kann man zeigen: Sind n Operationen auszuführen, so gibt es C_n Möglichkeiten, Klammern zu setzen, und das ist die Catalan-Zahl von n.

(2) Die Anzahl der verschiedenen binären Bäume mit n Knoten ist C_n. Abbildung 2.1 zeigt die nicht isomorphen binären Bäume mit drei Knoten. Beachten Sie dabei, dass ein Zweig oder Blatt mit einem Schlüssel kleiner als dem des Elternknotens „linker Kindknoten“ genannt wird. Ist der Schlüssel größer als der des Elternknotens, wird der Zweig oder das Blatt als „rechter Kindknoten“ bezeichnet. Daher sind 1-2-3 und 3-2-1 nicht isomorph. Der Beweis benutzt Gl. (2.5), siehe dazu [16].

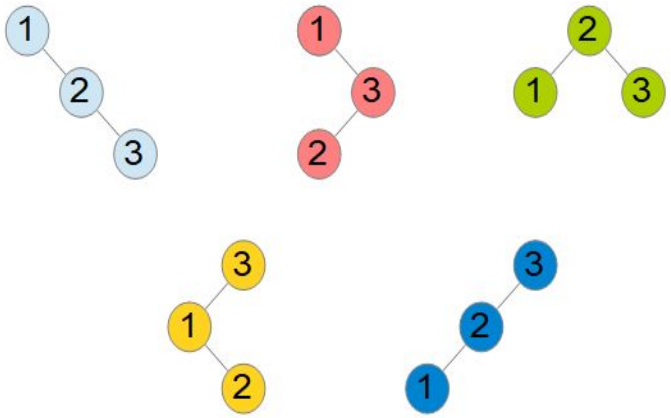


Abbildung 2.1: Fünf nicht isomorphe binäre Bäume mit drei Knoten

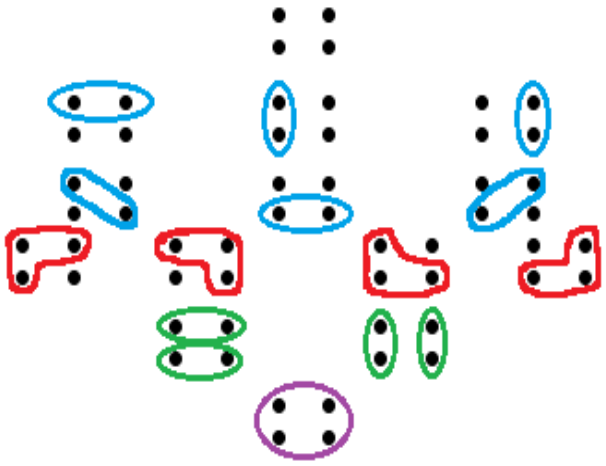


Abbildung 2.2: 14 sich nicht überschneidende Partitionen einer Menge mit vier Elementen

(3) Die Anzahl der sich nicht überschneidenden Partitionen einer Menge mit n Elementen ist C_n [17]. In Abbildung 2.2 sind die $C_4 = 14$ sich nicht überschneidenden Partitionen einer Menge mit vier Elementen dargestellt.

Es gibt noch viel mehr Anwendungen der Catalan-Zahlen. Doch kann man aus obigen Beispielen schon ersehen, dass diese Zahlen nützlich sind. Daher wollen wir drei kurze Programme zur Berechnung der Catalan-Zahlen in Python besprechen. Entsprechend der Definition Gl. (2.4) können wir ein Programm erstellen, das die folgende rekursive Funktion dreimal aufruft:

```
def fakrek(n):
    if n == 0: return 1
    else: return n*fakrek(n-1)
```

Das Hauptprogramm, mit dem man Catalan-Zahlen bis zu einer Obergrenze n berechnen kann, sieht so aus:

```
n = int(input('Obergrenze = '))
for k in range(n+1):
    C = fakrek(2*k)/(fakrek(k+1)*fakrek(k))
    print('C(',k,') = ',round(C))
```

Das ist ein einfaches Verfahren zur Berechnung von Catalan-Zahlen. Aber es gibt noch eine rekursive Formel, die ohne Fakultäten auskommt:

$$C_n = \sum_{j=0}^{n-1} C_j \cdot C_{n-j-1} \text{ where } C_0 = 1 \tag{2.5}$$

Es ist sehr einfach, Gl. (2.5) als Python-Funktion zu implementieren:

```
def Cat(n):
     if n == 0: return 1
     else:
        su = 0
        for j in range(n):
            su += Cat(j)*Cat(n-1-j)
         return su
```

Und dies ist das entsprechende sehr kurze Hauptprogramm:

```
n = int(input('limit = '))
for i in range(n+1):
     print('C(',i,') = ',round(Cat(i)))
```

Zum Testen des Programms sollten Sie kleine Zahlen eingeben, etwa 5 oder 10. Damit funktioniert das Programm gut; bei größeren Zahlen wie z.B. 18 werden Sie sehen, dass das Programm immer langsamer wird. Das liegt daran, dass die Zahl der rekursiven Aufrufe exponentiell ansteigt. Bei einem noch höheren n kann es sein, dass die maximal erlaubte Zahl an Rekursionen überschritten wird. Das Programm bricht ab, und eine Fehlermeldung wird ausgegeben. Das ist nicht das, was wir wollen.

Wie können wir es besser machen? Wir vermeiden die Vielzahl von rekursiven Aufrufen, indem wir eine Liste einführen: *CatList*. Diese muss im Hauptprogramm durch `CatList = [ ]` initialisiert werden. Das modifizierte Programm ist wiederum nicht sehr kompliziert. Wir müssen nur den aktuellen Wert der Funktion `Cat` an unsere Liste `CatList` anfügen. Dann wird das Ergebnis ausgegeben.

```
n = int(input('Obergrenze = '))
CatList = []
for i in range(n+1):
     CatList.append(Cat(i,CatList))
     print('C(',i,') = ',round(CatList[i]))
```

Die Funktion `Cat` muss ebenfalls verändert werden. Statt durch viele rekursive Aufrufe erhält man die Catalan-Zahlen aus der `CatList`, die so weit wie erforderlich gefüllt ist. Der Programmtext der Funktion sieht jetzt so aus:

```
def Cat(n,CatList):
    if n in [0,1]: return 1
    else:
       su = 0
       for j in range(n):
       su += CatList[j]*CatList[n-1-j]
    return su
```

Dieses Programm sieht nicht viel anders aus als das vorherige. Doch unterscheidet es sich in einem wichtigen Aspekt: Es gibt *keinen* rekursiven Aufruf. Programme wie diese nennt man *iterativ*. Die Catalan-Zahlen werden eine nach der anderen berechnet und in einer Liste oder einem Array gespeichert. Sie sind somit sofort verfügbar, wenn sie gebraucht werden.
Beim Testen dieses iterativen Algorithmus zeigt sich der enorme Gewinn an Laufzeit. Es dauert nur ein Augenzwinkern, um Ergebnisse wie dieses zu erhalten: $C(40) = 262212704227649210882 0$. Wenn es problemlos möglich ist, einen rekursiven Algorithmus in einen iterativen umzuwandeln, ist letzterer vorzuziehen. Aber diese Umwandlung ist häufig nicht ohne Schwierigkeiten möglich.

2.4 Binomialverteilung

Angenommen wir haben einen Karton mit w weißen und s schwarzen Kugeln. Es wird n mal eine Kugel aus dem Karton gezogen; die Farbe wird notiert und die Kugel wird wieder in den Karton zurückgelegt. Wir betrachten das Ziehen einer weißen Kugel als „Erfolg“. Die Erfolgswahrscheinlichkeit ist $p := w/(w+s)$. Wird eine spezielle Reihenfolge des Ziehens von weißen und schwarzen Kugeln betrachtet, ist die Wahrscheinlichkeit, genau k weiße Kugeln zu ziehen, gleich $p^k \cdot (1-p)^{n-k}$.

Schauen wir uns ein Beispiel mit drei weißen und fünf schwarzen Kugeln an. Die Wahrscheinlichkeit für die Reihenfolge der Ziehungen *sswswwss* ist $(\frac{3}{8})^3 \cdot (\frac{5}{8})^5 \approx 0.005$. Im Allgemeinen interessiert man sich nicht für die Reihenfolge, in der die weißen und schwarzen Kugeln gezogen werden. Statt dessen möchte man nur die Wahrscheinlichkeit berechnen, k weiße Kugeln zu ziehen. Dann müssen wir das vorige Ergebnis noch mit der Anzahl der Möglichkeiten multiplizieren, k weiße Kugeln bei n Ziehungen zu erhalten. Diese Anzahl ist der *Binomialkoeffizient* $\binom{n}{k}$ [18]. Die Anzahl der Erfolge ist gegeben durch die so genannte *Binomialverteilung* [19]:

$$p(X = k) = \binom{n}{k} \cdot p^k \cdot (1-p)^{n-k} \tag{2.6}$$

Zurück zum Beispiel. Im Karton sind acht Kugeln, von denen drei weiß sind, und wir ziehen acht mal. Die Wahrscheinlichkeit, drei weiße Kugeln zu ziehen, ist nach Gl. (2.6)

$$p(X = 3) = \binom{8}{3} \cdot 0.375^3 \cdot 0.625^5 \approx 0.2816$$

Gleichung (2.6) kann leicht in Python implementiert werden, da im Modul `scipy` eine Funktion enthalten ist, welche Binomialkoeffizienten berechnet. Dies ist das ganze Listing:

```
import scipy.special as sp
     <<< Überschrift und Eingaben >>>
p = w/(w + s)
for k in range(n+1):
     binom = sp.comb(n,k,exact = True)
     binom = binom*(p**k)*((1-p)**(n-k))
     print('Wahrscheinlichkeit für ',k,' weisse Kugeln: ',binom)
```

Dies ist ein kurzes Programm, es braucht keine Rekursion, sondern das Modul `scipy`. Wir aber möchten einen „hausgemachten“ Code mit einer rekursiven Funktion erstellen. Die `for`-Schleife des Hauptprogramms ist sogar kürzer:

```
for k in range(n+1):
     print('Wahrsch. fuer ',k,' weisse Kugeln: ',prob(n,k))
```

Die Funktion `prob` bleibt zu besprechen. Sie beginnt mit einer `if`-Anweisung: `if (weiss > Stufe) or (weiss < 0):return 0`. Das ist so, weil die Anzahl der gezogenen weißen Kugeln nicht größer sein kann als die Zahl der Ziehungen und eine negative Anzahl weißer Kugeln keinen Sinn ergibt. Der erste Teil des `else`-Zweigs enthält eine weitere `if`-Anweisung: `if Stufe == 1:...`. Darin befindet sich eine weitere `if`-Verzweigung:

```
if weiss == 1:return p
 else:return 1-p.
```

Dieser Teil beschreibt den Fall, dass nur eine Kugel gezogen wird. Die Wahrscheinlichkeit, eine weiße Kugel zu ziehen, ist p und die Wahrscheinlichkeit für eine schwarze Kugel ist $1-p$. Der letzte `else`-Zweig ist etwas komplexer: `prob(1,0)*prob(Stufe-1,weiss)+prob(1,1)*prob(Stufe-1,weiss-1)`. Nach dem schon beschriebenen Teil der Funktion kann `prob(1,0)` ersetzt werden durch `1-p`, und `prob(1,1)` ist das Gleiche wie `p`. Zudem sind zwei rekursive Aufrufe dabei: `prob(Stufe-1,weiss)` und `prob(Stufe-1,weiss-1)`.

Um die Richtigkeit dieser Programmzeile zu zeigen, verwandeln wir sie zurück in eine algebraische Formel:

$$P(X = k) = \binom{n}{k} \cdot p^k (1-p)^{n-k} =$$
$$(1-p) \cdot \binom{n-1}{k} \cdot p^k (1-p)^{n-1-k} + p \cdot \binom{n-1}{k-1} \cdot p^{k-1} (1-p)^{n-k}$$

Durch Ausklammern und Zusammenfassen erhalten wir:

$$\binom{n}{k} \cdot p^k (1-p)^{n-k} = p^k (1-p)^{n-k} \cdot \left(\binom{n-1}{k} + \binom{n-1}{k-1} \right)$$

Diese Gleichung ist äquivalent zu

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \tag{2.7}$$

Für den Beweis von Gl. (2.7) siehe [20]. Also ist die letzte Zeile der Funktion `prob` in der Tat richtig. Hier ist das Listing der Funktion:

```
def prob(Stufe,weiss):
    if (weiss > Stufe) or (weiss < 0):return 0
    else:
       if Stufe == 1:
          if weiss == 1:return p
          else:return 1 - p
       else:
          return prob(1,0)*prob(Stufe-1,weiss)+
             prob(1,1)*prob(Stufe-1,weiss-1)
```

Unser „hausgemachtes“ Programm funktioniert gut für kleine Zahlen. Probieren Sie `w` = 3, `b` = 5, `n` = 8. Setzten wir jedoch `w` = 9, `b` = 15, `n` = 24, also das Dreifache der vorherigen Eingaben, wird das Programm sehr langsam. Das liegt daran, dass die Zahl der rekursiven Aufrufe mit der Anzahl der Ziehungen sehr stark ansteigt, siehe Kapitel 3. Um das deutlich zu sehen, geben Sie `w` = 3, `b` = 5, `n` = 30 ein.

2.5 Determinanten

Die Determinante einer quadratischen Matrix ist eine reelle Zahl (Gleitkommazahl), die von den Zahlen in der Matrix abhängt. Es gibt viele Anwen-

dungen von Determinanten. einige werden unten kurz beschrieben. Die *Determinante einer 2 x 2-Matrix* kann leicht berechnet werden:

$$\det \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = a_{11} \cdot a_{22} - a_{12} \cdot a_{21} \tag{2.8}$$

Die Definition der Determinante einer 3 x 3-Matrix ist nicht so einfach, z.B.

$$\begin{aligned} &\det \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \\ &a_{11} \cdot \det \begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix} - a_{21} \cdot \det \begin{pmatrix} a_{12} & a_{13} \\ a_{32} & a_{33} \end{pmatrix} + a_{31} \cdot \det \begin{pmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \end{pmatrix} \end{aligned} \tag{2.9}$$

Dies sieht ziemlich kompliziert aus, doch Gl. 2.9 ist nur ein Spezialfall dessen, was man *Entwicklung einer Determinante nach einer Spalte* nennt.
Programm *Determinante_3* demonstriert die Anwendung von Gl. 2.9. Die allgemeine Formel für n x n-Determinanten ist

$$\det A = \sum_{i=1}^{n} (-1)^{i+j} a_{ij} \cdot \det A_{ij} \tag{2.10}$$

A ist die Matrix, von welcher die Determinante berechnet wird; A_{ij} ist die Matrix, die aus A entsteht, wenn man die i-te Zeile und die j-te Spalte streicht. Die Determinante wird in diesem Falle nach der j-ten Spalte entwickelt. Es ist auch möglich, die Determinante nach einer Zeile zu entwickeln. Doch soll dies hier nicht behandelt werden.
Wie man Gl. 2.9 entnehmen kann, wird die Berechnung einer dreireihigen Determinante rekursiv auf die Berechnung dreier Zweierdeterminanten zurückgeführt. Gl. 2.10 zeigt, dass die Berechnung einer Determinante mit n Zeilen und Spalten auf die Berechnung von n Determinanten mit n-1 Zeilen und Spalten zurückgeführt werden kann. Daher macht es Sinn, eine rekursive Funktion zur Berechnung von Determinanten zu entwickeln. Nebenbei sei bemerkt, dass es im Modul `numpy` eine Funktion `np.linalg.det` gibt, mit der man Determinanten berechnen kann. Sie benötigt als Eingabe einen zweidimensionalen `numpy`-Array. Mit ihr ist also nicht ganz leicht umzugehen, und man sieht den Rechenweg nicht. Dafür ist sie aber sehr schnell.

Unser Code benutzt kein zusätzliches Modul. Matrix `A` wird Zeile für Zeile eingegeben; auf diese Weise erhalten wir eine Liste von Listen. Dieses erfolgt in der Funktion `EingabeMatrix`, die hier nicht im einzelnen besprochen werden soll. Außerdem brauchen wir einen Satz von Indizes {0, 1, ..., n-1},

in Python: `Indizes = {j for j in range(n)}`. Das Hauptprogramm ist kurz:

```
A,n = EingabeMatrix()
Indizes = {j for j in range(n)}
print('Wert der Determinante: ',det(A,n,0,Indizes))
```

Die wichtige Funktion ist `det`; sie nimmt die Parameter `A, Rang, Spalte, ind` auf. `A` bezeichnet die aktuelle Matrix, von der die Determinante berechnet werden soll. `Rang` ist die Anzahl der Zeilen und Spalten von `A`. `Spalte` ist die aktuelle Spalte, nach der die Determinante entwickelt werden soll, und `ind` ist die aktuelle Indexmenge. Das Wort *aktuell* muss betont werden, denn alle Parameter werden sich während der rekursiven Aufrufe wahrscheinlich verändern.
Beispiel: Matrix A ist definiert in Gl. 2.11. Wir wollen $\det(A)$ nach Spalte 0 entwickeln.

$$A = \begin{pmatrix} 1 & 0 & 3 & 4 \\ -2 & 1 & 1 & 3 \\ 0 & 2 & 0 & 0 \\ 1 & 4 & 1 & 5 \end{pmatrix} \tag{2.11}$$

Die Funktion `det` wird mit folgenden Parametern aufgerufen: `Rang = 4`, `Spalte = 0`, `ind = {0,1,2,3}`. Im ersten rekursiven Aufruf muss die folgende Unterdeterminante berechnet werden:

$$\det \begin{pmatrix} 1 & 1 & 3 \\ 2 & 0 & 0 \\ 4 & 1 & 5 \end{pmatrix}$$

Die entsprechenden Parameter in Funktion `det` sind `Rang = 3`, `Spalte = 1`, `ind = {1,2,3}`. Der zweite rekursive Aufruf muss diese zweireihige Determinante berechnen:

$$\det \begin{pmatrix} 0 & 0 \\ 1 & 5 \end{pmatrix}$$

Jetzt sind die Parameter `Rang = 2`, `Spalte = 2`, `ind = {2,3}`. Nunmehr enden die rekursiven Aufrufe, weil eine zweireihige Determinante direkt berechnet werden kann. Das geschieht in dem folgenden Programmteil:

```
if Rang == 2:
   # finde richtige Indizes
   k = list(ind)
   if k[0] > k[1]: x = k[0]; k.pop(0); k.append(x)
   res = A[k[0]][Spalte]*A[k[1]][Spalte+1]- \
         A[k[1]][Spalte]*A[k[0]][Spalte+1]
```

Die Bestimmung der richtigen Indizes sieht ein wenig seltsam aus. Weil wir für die richtige Reihenfolge der Indizes sorgen müssen, brauchen wir eine Liste `k` mit nur zwei Zahlen. Dazu wird zunächst die *Menge* `ind` in die *Liste* `k` umgewandelt. Wenn der erste enthaltene Index größer als der zweite ist, wird der erste Index nach `x` kopiert. Dann wird die erste Zahl `k[0]` aus der Liste entfernt und `x` hinten angehängt. Dann wird die Formel für zweireihige Determinanten Gl. 2.8 angewendet.
Der `else`-Zweig enthält rekursive Aufrufe. Zunächst brauchen wir zwei lokale Variablen. `res` wird den Wert der aktuellen Determinante erhalten. `Zaehler` ist ein Zähler für die Zeilen und bekommt anfangs den Wert 0. Er stellt sicher, dass der Exponent in Gl. 2.10 korrekt bleibt. Da die Entwicklung der Determinante nach der (aktuellen) Spalte 0 erfolgt, liefert `(-1)**Zaehler` das richtige Vorzeichen. `Zaehler` ist nicht das Gleiche wie `i` in der folgenden `for`-Schleife. Schauen wir uns dazu noch einmal die Matrix 2.11 an. Wenn in dieser `i = 1` ist, muss die Unterdeterminante

$$\det \begin{pmatrix} 0 & 3 & 4 \\ 2 & 0 & 0 \\ 4 & 1 & 5 \end{pmatrix}$$

berechnet werden. In letzterer bekommt `i` nacheinander die Werte 0, 2, 3. Aber `Zaehler` ist lokal und bekommt die Werte 0, 1, 2.
Die `for`-Schleife läuft von `0` bis `n-1` und kümmert sich nicht um den aktuellen `Rang`. Wenn der Index `i` in der aktuellen Indexmenge `ind` liegt, muss `i` für den folgenden rekursiven Aufruf aus `ind` entfernt werden: `ind=ind-{i}`. Der Aufruf selbst wendet einfach nur Gl. 2.10 an:
`res += (-1)**Zaehler*A[i][Spalte]*det(A,Rang-1,Spalte+1,ind)`
So erhält `res` den Wert der gerade berechneten Determinante während der Durchläufe der `for`-Schleife. Danach wird `Zaehler` um 1 erhöht, und `i` wird wieder der Indexmenge `ind` hinzugefügt. Dies ist das ganze Listing des `else`-Zweiges:

```
    else:
       res = 0;Zaehler = 0
       for i in range(n):
        if i in ind:
          # Zeile i entfernen!
          ind = ind - {i}
          res+=(-1)**Zaehler*A[i][Spalte]* \
                det(A,Rang-1,Spalte+1,ind)
          Zaehler += 1
          ind = ind.union({i})
```

Es gibt einige einfache Regeln über Determinanten, siehe [21], [22]. Daraus kann man leicht schließen, dass eine Determinante den Wert 0 hat, wenn eine Spalte Linearkombination der übrigen ist. Die Determinante 2.11 hat den Wert -58, nicht 0. Daher sind die Spaltenvektoren *linear unabhängig*, und keiner von ihnen lässt sich als Linearkombination der anderen drei Vektoren darstellen.
Der Absolutbetrag einer dreireihigen Determinante ergibt das Volumen des *Spates*, der von den Spaltenvektoren erzeugt wird [23]. Ein Beispiel für einen Spat ist ein Feldspatkristall in Abbildung 2.3. Wenn jedoch eine Determinante von drei Spaltenvektoren gleich 0 ist, liegen die Vektoren in einer Ebene, man sagt, sie seien *koplanar*.

Abbildung 2.3: Feldspatkristall - Beispiel für einen Spat

Determinanten sind auch nützlich zum Lösen linearer Gleichungssysteme, indem die *Cramersche Regel* angewendet wird [24]. Wenn das System eine eindeutige Lösung hat, muss man eine Determinante mehr berechnen, als das System Variablen hat. Wenn jedoch die Determinante der Matrix der Koeffizienten gleich 0 ist, hat das System keine eindeutige Lösung.

Dies sind nur einige Beispiele für die Anwendung von Determinanten, für weitere Beispiele siehe [22]. Zum Schluss müssen wir noch darauf hinweisen, dass die Entwicklung einer Determinante nach einer Spalte oder einer Zeile kein sehr effizienter Algorithmus ist. Die Laufzeit ist proportional zur Fakultät der Anzahl der Variablen n, also $O(n!)$ [22]. Es gibt effizientere Algorithmen, die auch besser geeignet sind, lineare Gleichungssysteme zu lösen und lineare Unabhängigkeit zu beweisen. Aber der Umgang mit Determinanten ist leicht, und etwa die Cramersche Regel zu programmieren, ist relativ einfach.

2.6 Intervallhalbierungsverfahren

Dieser Abschnitt bezieht sich auf die Bestimmung von Nullstellen einer Polynom-Funktion (auch ganzrationale Funktion). Es gibt keinen Algorithmus, mit dem man die Nullstellen einer solchen Funktion berechnen kann, wenn der Grad der Funktion größer als 4 ist [25]. Die Verfahren für Funktionen vom Grad 3 oder 4 sind kompliziert [26], [27]. Deswegen wollen wir die Nullstellen numerisch näherungsweise bestimmen. Ein geläufiges Verfahren ist das *Intervallhalbierungsverfahren IHV*. Wir wollen zunächst aber die Struktur des Programms vorstellen, das das IHV verwendet.

M sei die Summe der Absolutbeträge der Koeffizienten. Dann liegen alle Nullstellen im Intervall [-M,M]. Wir stellen eine Wertetabelle mit der Schrittweite 1 auf, die bei -M beginnt und mit M endet. Wenn zwischen zwei aufeinander folgenden Funktionswerten ein Vorzeichenwechsel auftritt, liegt zwischen den entsprechenden x-Werten eine Nullstelle (ungerader Ordnung). In diesem Fall wird die Funktion `IHV` aufgerufen. Die Nullstelle liegt entweder in der unteren oder in der oberen Hälfte des Intervalls, folglich tritt der Vorzeichenwechsel in einem der beiden Teilintervalle auf. Dies erlaubt uns, einen rekursiven Algorithmus zu codieren, der endet, wenn die Länge der Intervalle kleiner wird als eine untere Grenze, die wir `epsilon` nennen.
Das Hauptprogramm besteht aus zwei Teilen: der Eingabe und der Verarbeitung. Hier ist das Listing:

```
max = 6
epsilon = 1e-5
print(); print('Intervalhalbierungsverfahren')
ok = False
while not(ok):
     Koeff();Polynom()
     print(); Ant = input('alles richtig? (j/n)')
     if Ant in ['J','j']: ok = True
Grenzen(n,a); WT(n,a)
```

`max = 6` beschränkt den Grad der Funktion. Die Bedeutung von `epsilon` wurde schon erläutert. Die anschließende `while`-Schleife gestattet es, die Eingabe der Koeffizienten, die in der Funktion `Koeff` erfolgt, gegebenenfalls zu korrigieren. `Polynom` gibt das eingegebene Polynom aus. Wenn die Eingabe richtig ist, wird die Funktion `Grenzen` aufgerufen, um das erforderliche Intervall [-M,M] für die Wertetabelle festzulegen. Diese drei genannten Funktionen werden nicht detailliert besprochen, weil sie sehr einfach sind und in ihnen keine Rekursion vorkommt.

Funktion `WT` (steht für „Wertetabelle“) enthält zwar auch keine Rekursion, aber sie soll dennoch besprochen werden. Dies sind die wichtigen Teile des Listings:

```
def WT(n,a):
    for i in range(round(-M),round(M)):
       print(' ',i,' -> ',round(f(i),3))
       if f(i)*f(i+1) < -epsilon:
          IHV(i,i+1)
       if abs(f(i)) < epsilon:
          print(); print('Eine Nullstelle liegt bei ',i)
```

Die Funktion `f` erstellt ein Polynom aus den eingegebenen Koeffizienten, sodass Werte berechnet werden können, siehe unten. `IHV` wird aufgerufen, wenn `f(i)*f(i+1)< -epsilon` gilt. Denn dann tritt ein Vorzeichenwechsel im Intervall [`i`,`i+1`] auf. Sollte das Produkt `f(i)*f(i+1)` nicht kleiner sein als `-epsilon`, tritt kein Vorzeichenwechsel auf, oder eine ganzzahlige Nullstelle wurde gefunden. Dieser Fall wird in den beiden letzten Zeilen behandelt. Leider findet das Programm nicht alle Nullstellen von Polynom-Funktionen. Es kann sein, dass zwei Nullstellen ungerader Ordnung sehr nahe beieinander liegen, z.B. $\frac{1}{4}$ und $\frac{1}{2}$. Daher findet das Programm nicht die Nullstellen der Funktion zu $p(x) = x^2 - \frac{3}{4}x + \frac{1}{8}$. In solchen Fällen muss die Schrittweite in der Wertetabelle verkleinert werden, siehe *IHV_2.py* im Programmteil. Das Programm wird auch bei der Bestimmung von Nullstellen gerader Ordnung versagen, siehe [28]. Nach Eingabe der Funktion zu $p(x) = x^3 - \frac{5}{2}x^2 + \frac{3}{4}x + \frac{9}{8}$ findet das Programm nur $-\frac{1}{2}$, aber nicht die Nullstelle $\frac{3}{2}$, die zugleich ein Minimum ist.

Die Funktion `f` sieht sehr einfach aus, ist es aber nicht. Sie verwendet das *Horner-Schema* [29]. Zurückgegeben werden soll die Variable `y`. Sie wird zuerst gleich dem Koeffizienten des höchsten Exponenten gesetzt: `y = a[n]`. Es folgt eine `for`-Schleife, deren Zähler bis 0 *abnimmt*; dies wird im Parameter `-1` von `range` ausgedrückt. Der bisherige Wert von `y` wird mit dem Argument `x` multipliziert, dann wird der nächst niedrigere Koeffizient addiert: `a[j-1]`. Schließlich steht das Ergebnis `y` fest, das zurückgegeben wird.

```
def f(x):
    y = a[n]
    for j in range(n,0,-1): y = y*x + a[j-1]
    return y
```

Schließlich muss noch die rekursive Funktion `IHV` vorgestellt werden. Die ihr übergebenen Parameter sind die linke und rechte Grenze `l` and `r` des Intervalls, in dem ein Vorzeichenwechsel auftritt. `fl` und `fr` sind die zugehörigen Funktionswerte. Wenn `r-l < epsilon` ist, ist die Abbruchbedingung erreicht. Anderenfalls erfolgt ein rekursiver Aufruf von `IHV`. Die Parameter hängen davon ab, wo der Vorzeichenwechsel auftritt. Hier ist das Listing:

```
def IHV(l,r):
    fl = f(l); fr = f(r)
    print('[',round(l,5),' , ',round(r,5),']
    if r - l < epsilon:
       print(); print('Eine Nullstelle liegt im Intervall')
       print('[',l,',',r,'].'); print()
    else:
       if fl*f((l+r)/2) <= 0: IHV(l,(l+r)/2)
       else: IHV((l+r)/2,r)
```

2.7 Übungsaufgaben

1. **Heron-Verfahren**
   Schreiben Sie ein Programm zur Berechnung der n-ten Wurzel aus einer positiven reellen Zahl. siehe Gl. (2.3).

2. **Pascalsches Dreieck**
   Implementieren Sie ein Programm zur Berechnung der Zahlen im *Pascalschen Dreieck*. Es enthält die Binomialkoeffizienten, siehe [30].

3. **Catalan-Zahlen**
   Es gibt $C_4 = 14$ unterschiedliche (nicht isomorphe) Bäume mit 4 Knoten. Schreiben Sie sie alle auf.

4. **Regula falsi**
   Ein bekanntes Verfahren zur Bestimmung von Nullstellen ungerader Ordnung ist die *regula falsi* [31]. Das Verfahren ist schon seit sehr langer Zeit bekannt; es wurde zuerst im Papyrus Rhind erwähnt (ungefähr 1550 vor Chr.) und detaillierter in der Mathematik des Islam zwischen dem 8. und 13. Jahrhundert.
   Es seien $P_0 = (x_0, f(x_0))$ und $P_1 = (x_1, f(x_1))$ zwei Punkte mit $f(x_0) \cdot f(x_1) < 0$, wobei o.B.d.A. $x_0 < x_1$ angenommen wird. Dann ist $(x_2, 0)$

der Schnittpunkt der Verbindungsgerade von P_0 und P_1 mit der x-Achse, siehe Abbildung 2.4. Es gilt

$$x_2 = \frac{x_0 \cdot f(x_1) - x_1 \cdot f(x_0)}{f(x_1) - f(x_0)} \tag{2.12}$$

x_2 ist eine Näherung der Nullstelle, die zwischen x_0 und x_1 liegt. Wie beim Intervallhalbierungsverfahren gilt entweder $f(x_0) \cdot f(x_2) < 0$ oder $f(x_2) \cdot f(x_1) < 0$. Daher kann dieses Verfahren durch rekursive Aufrufe fortgesetzt werden, bis die gewünschte Genauigkeit erreicht ist.

Entwickeln Sie ein Programm zur näherungsweisen Berechnung von Nullstellen einer gegebenen Funktion unter Anwendung von (2.12). Wenn Sie $f(x) = \sin(2 \cdot x)$ verwenden und als Intervall [2,4] wählen, erhalten Sie einen Näherungswert für π.

Hinweis: Wenn Sie Gleitkommazahlen als Intervallgrenzen eingeben möchten, müssen Sie ihre Eingabe in „float“ umwandeln:
`border = float(input('Geben Sie die Grenze ein. '))`.

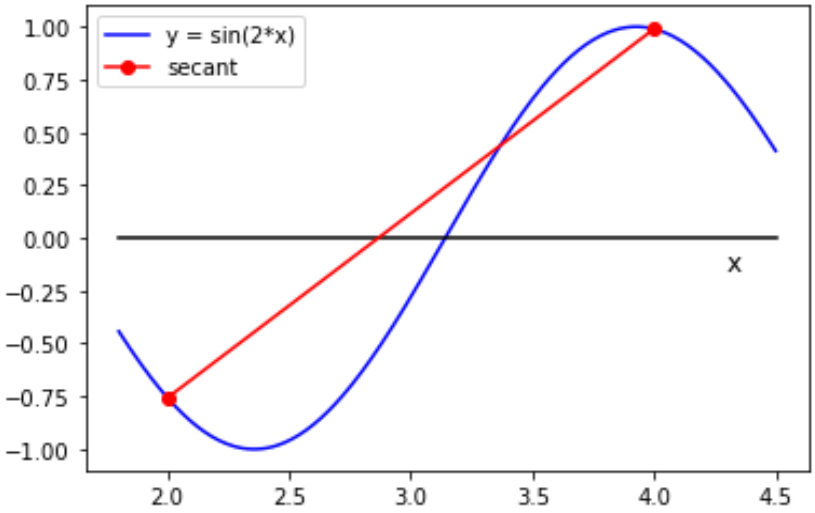


Abbildung 2.4: Ein Vorzeichenwechsel tritt zwischen $x_0 = 2$ und $x_1 = 4$ auf. Nach Gl. (2.12) ist der Schnittpunkt der roten Sekante und der x-Achse (2.87,0). Der blaue Graph schneidet die x-Achse im Punkt $(\pi, 0)$.

5. **Determinanten**

Wir erwähnten, dass die Entwicklung einer Determinante nach einer Spalte oder Zeile kein sehr effizienter Algorithmus ist. Die Umformung

in eine obere Dreiecksmatrix [32] ist besser. Eine Determinante ändert ihren Wert nicht, wenn ein Vielfaches einer Zeile zu einer anderen Zeile addiert wird. Wenn man das $-\frac{a_{k1}}{a_{11}}$-fache von Zeile 1 zu den Zeilen k (k = 2,...,n) addiert, erhält man eine Determinante, die den gleichen Wert hat wie die ursprüngliche, aber in der ersten Spalte stehen nur Nullen, außer a_{11}. Dieses Verfahren kann durch rekursive Aufrufe fortgesetzt werden, bis kein Element $\neq 0$ mehr unter einem Diagonalelement steht [33]. So ergibt sich eine obere Dreiecksmatrix. Der Wert der Determinante ist einfach das Produkt der Diagonalelemente. Leider können sich Rundungsfehler durch die Divisionen ergeben.
Schreiben Sie ein Programmn das den Wert einer Determinante durch Triangulation berechnet. Setzen Sie voraus, dass kein Diagonalelement gleich 0 ist.
Schwierigere Aufgabe: Behandeln Sie auch den Fall, dass ein Diagonalelement 0 ist. Finden Sie dann ein Pivot-Element und vertauschen Sie Zeilen, sodass kein Diagonalelement mehr 0 ist.

Kapitel 3

Rekursive Algorithmen mit mehrfachen Aufrufen

Es gibt viele Funktionen, in denen rekursive Aufrufe zwei- oder mehrfach vorkommen. In diesem Kapitel wird nur eine kleine Auswahl davon vorgestellt. Eine wichtige Anwendung ist die Berechnung von Permutationen. Außergewöhnliche Funktionen sind die Ackermann-Funktion, die Hofstadter-Q-Funktion und der Algorithmus für Fibonacci-Zahlen höherer Ordnung. Mergesort, das Rucksack-Problem und die Lösung von ILP-Aufgaben führen typischerweise auf rekursive Funktionen mit mehrfachen Aufrufen. Starten wollen wir aber mit einer grafischen Anwendung.

3.1 Drachenkurven

Eine Drachenkurve [34] der Stufe n entsteht aus einer Drachenkurve der Stufe n-1, indem jede Strecke der Kurve der Stufe n-1 durch die Katheten eines rechtwinkligen Dreiecks ersetzt wird. Dabei ist die erwähnte Strecke die Hypotenuse des Dreiecks. Der rechte Winkel zwischen den Katheten ist nach außen gerichtet, wenn man die Figur vom Mittelpunkt der Zeichenfläche aus betrachtet. Eine Drachenkurve der Stufe 0 ist nur eine Strecke. Abbildung 3.1 zeigt Drachenkurven der Stufen 3 und 4 und ihre Überlagerung. Diese Kurven sind Drachenkurven, weil die rechten Winkel nach außen abstehen wie die Schuppen einer Drachenhaut.

Weil die Drachenkurven rekursiv definiert sind, liegt es nahe, dass wir die Erstellung einer Drachenkurve mit Turtlegrafik durch eine rekursive Funktion implementieren. Sie hängt von zwei Parametern ab: `Laenge` and `Stufe`. Wenn `Stufe` = 0 ist, besteht die Funktion nur aus der Anweisung `tu.forward(size)`.

Anderenfalls müssen wir dafür sorgen, dass die Aufgabe, eine Drachenkurve der Stufe n zu zeichnen, auf die Aufgabe reduziert wird, dieses mit einer Drachenkurve der Stufe n-1 zu machen.

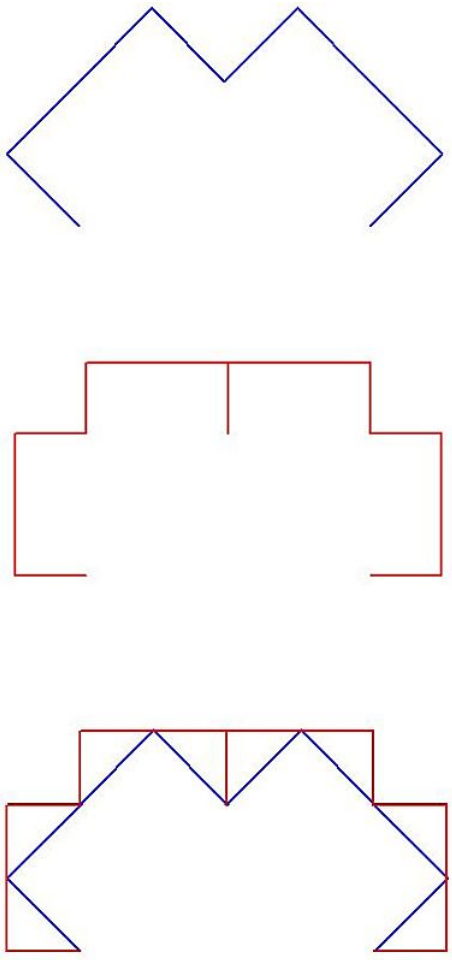


Abbildung 3.1: Drachenkurven: Stufe 3, Stufe 4 und die Überlagerung beider

In einem rechtwinklig gleichschenkligen Dreieck stehen die Längen einer Kathete und der Hypotenuse im Verhältnis $\frac{1}{\sqrt{2}}$, das ist ungefähr 0.707. In der Funktion `Drache` verkürzen wir die Länge: `Laenge = round(0.707*Laenge)`. Dann dreht sich die Turtle um 45° nach links, und `Drache` wird mit `Stufe-1` aufgerufen. Danach muss sich die Turtle um 90° nach rechts drehen, und `Drache(Laenge,Stufe-1)` wird noch einmal aufgerufen. Am Ende dieses kleinen Programmteils soll die Turtle wieder die gleiche Richtung haben wie am Anfang. Daher muss sich die Turtle wieder um 45° nach links drehen. Damit haben wir die ganze Funktion besprochen:

```
def Drache(Laenge,Stufe):
    if Stufe == 0: tu.forward(Laenge)
    else:
       Laenge = round(Laenge*0.707)
       tu.left(45)
       Drache(Laenge,Stufe-1)
       tu.right(90)
       Drache(Laenge,Stufe-1)
       tu.left(45)
```

Der Leser mag jetzt aus folgendem Grunde denken, die Funktion sei falsch: `Stufe` sei anfangs gleich 2. Dann wird `Laenge` durch das 0.707-fache von `Laenge` ersetzt. Danach ruft die Funktion `Drache(0.707*Laenge,1)` auf. Dabei wird die ursprüngliche `Laenge` durch das 0.707^2-fache von `Laenge` ersetzt, also durch die Hälfte der ursprünglichen `Laenge`. Was geschieht, wenn die Funktion `Drache Stufe` 0 verarbeitet? Wird `Laenge` durch mehrere Funktionsaufrufe immer kleiner?
Tatsächlich geschieht dies nicht. Python unterscheidet Variablen, die in verschiedenen Funktionsaufrufen auftreten, obwohl sie alle mit `Laenge` bezeichnet werden. Die `Laenge` im ursprünglichen Aufruf ist nicht das Gleiche wie `Laenge` in `Stufe` 1 oder `Stufe` 0. Nur durch den aktuellen Aufruf wird `Laenge` geändert: `Laenge = round(Laenge*0.707)`. Zur Veranschaulichung werden alle rekursiven Aufrufe von `Drache`(100,2) in der folgenden Tabelle dargestellt:

1	Drache(100,2)
2	Drache(71,1)
3	Drache(50,0)
4	Drache(50,0)
5	Drache(71,1)
6	Drache(50,0)
7	Drache(50,0)

Im Hauptprogramm sollen die Drachenkurven der Stufen 0 bis 5 überlagert werden. Es wäre möglich, auch noch Drachenkurven höherer Ordnung zu zeichnen, doch würden sich Rundungsfehler immer stärker bemerkbar machen und die Figur verderben. Wenn Sie gern eine Drachenkurve höherer Stufe sehen möchten, modifizieren Sie am besten das Programm so, dass nur Ihre Drachenkurve gezeichnet wird und sonst keine.

Die Drachenkurve wird etwas dünn, wenn man nicht die Linienbreite mindestens auf 3 setzt: `tu.width(3)`. Der Hauptteil des Programms besteht aus einer Schleife. Darin wird zuerst die Zeichenfarbe der Turtle bestimmt. Die kleine Funktion `bestimmeFarbe` bestimmt zufällig die `pencolor`. Die Rückkehr zum Editor wird in der Version für PyCharm dargestellt, für Spyder sei auf den Anhang verwiesen. Hier wird zunächst das Hauptprogramm aufgelistet und danach die Funktion `bestimmeFarbe`.

```
tu.width(2)
for Stufe in range(6):
     tu.pencolor(bestimmeFarbe())
     tu.up(); tu.setpos(-160,-50)
     tu.down()
     Drache(270,Stufe)
tu.up(); tu.hideturtle()
tu.setpos(-300,-150)
tu.pencolor((0,0,0))
tu.write('fertig!',font = ("Arial",12, "normal"))
tu.exitonclick() # für PyCharm
try: tu.bye()
except tu.Terminator: pass
```

Damit Sie die Funktion `bestimmeFarbe` benutzen können, müssen Sie `random` importieren, z.B. so: `import random as rd`.

```
def bestimmeFarbe():
     rot = rd.uniform(0,0.9)
     gruen = rd.uniform(0,0.9)
     blau = rd.uniform(0,0.9)
     return (rot, gruen, blau)
```

3.2 Permutationen

Unter einer *Permutation* einer endlichen Menge von natürlichen Zahlen 1,...,n versteht man eine Anordnung dieser Zahlen. Z. B. können die Elemente der Menge {1,2,3} auf sechs verschiedene Weisen angeordnet werden: (123), (132), (213), (231), (312), (321). Man kann leicht zeigen, dass die Anzahl der Permutationen von n Objekten gleich $n!$ ist, siehe [35].

Wir entwickeln ein Programm. das alle Permutationen der Zahlen `1,...,pmax` ausgibt, wobei $2 \leq pmax \leq 6$ gelten soll. Das Hauptprogramm enthält

natürlich die Eingabe der Zahl `pmax`. Daraufhin wird die Liste `p` erstellt, die mit `1,...,pmax` gefüllt wird. Schließlich wird die Funktion `perm` aufgerufen. In ihr werden die Zahlen auf alle möglichen Arten angeordnet und ausgegeben, sodass man die Permutationen von `1,...,pmax` erhält.

Bevor die Funktion `perm` besprochen wird, werfen wir noch einen Blick auf die sehr kurze Hilfsfunktion `swap`. Sie dient dazu, die Listenelemente `p[i]` und `p[j]` zu vertauschen. Das wird einfach durch folgende Python-Spezialität erreicht:

`p[j], p[i] = p[i], p[j]`.

Durch diese Doppelzuweisung erhält `p[j]` den Wert, den zuvor `p[i]` hatte, und umgekehrt.

Nun aber zur Funktion `perm`. Wir betrachten zuerst den Fall `n = 2`.

```
if n == 2:
   print(p)
   swap(p,pmax-2,pmax-1)
   print(p)
   swap(p,pmax-2,pmax-1)
```

Die aktuelle Permutation wird ausgegeben, dann werden die beiden letzten Ziffern vertauscht. Die so erhaltene Permutation wird ausgegeben, und die Vertauschung wird wieder rückgängig gemacht. Beispiel `p=[1,3,2,4]`: Zuerst wird `p=[1,3,2,4]` ausgegeben; dann werden die beiden letzten Ziffern vertauscht. Das führt zur Ausgabe `p=[1,3,4,2]`. Schließlich wird die Vertauschung rückgängig gemacht, und es ist wieder `p=[1,3,2,4]`.

Der `else`-Zweig ist nicht so leicht zu verstehen:

```
else:
   perm(n-1)
   for i in range(pmax-n+1,pmax):
      swap(p,pmax-n,i)
      perm(n-1)
   for i in range(pmax-n+1,pmax):swap(p,i-1,i)
```

Wir betrachten jetzt den Fall `pmax=3`. Wir wissen `p=[1,2,3]`, und `perm(n-1) = perm(2)` wird aufgerufen. Also werden `[1,2,3]` und `[1,3,2]` ausgegeben. Es folgt eine `for`-Schleife, die von `i = pmax-n+1` bis `pmax-1` läuft, d.h. `i` nimmt die Werte 1 und 2 an. Wenn `i=1` ist, geschieht Folgendes: `p[pmax-n]=p[0]` und `p[1]` werden vertauscht, und `perm(n-1) = perm(2)` wird aufgerufen. `[2,1,3]` und `[2,3,1]` werden ausgegeben. Jetzt sei `i=2`. `p[pmax-n]=p[0]` und `p[2]` werden vertauscht, und `perm(2)` wird ausgeführt,

sodass `[3,1,2]` und `[3,2,1]` ausgegeben werden. Die zweite `for`-Schleife dient dazu, die ursprüngliche Anordnung der Zahlen wiederherzustellen. Vor dem ersten Durchlauf ist die Reihenfolge `[3,1,2]`, denn `perm(2)` hat aus `[3,2,1]` wieder `[3,1,2]` gemacht. Wenn `i=1` ist, werden `3` und `1` vertauscht, und wir erhalten `[1,3,2]`. Wenn `i=2` ist, werden `3` und `2` vertauscht, und wir haben die ursprüngliche Reihenfolge zurückbekommen.

Was aber geschieht, wenn `pmax > 3` ist, z.B. `pmax = 4`? An dieser Stelle muss bemerkt werden, dass `pmax` eine globale Variable ist, denn ihr Wert wurde im Hauptprogramm festgelegt. `n` hingegen ist eine lokale Variable; sie verändert sich in jedem Funktionsaufruf. Die erste Anweisung in `perm(4)` ist `perm(n-1) = perm(3)`. In diesem Aufruf ist nach wie vor `pmax = 4`, jedoch `n = 3`. Also ist jetzt die Anfangsbedingung der Schleife `pmax-n+1=4-3+1=2`. Sie reicht jetzt von 2 bis 3, und wir erhalten die Permutationen von `[2,3,4]`, anstelle von `[1,2,3]`. Die Ausgaben werden also sein: (1234), (1243), (1324), (1342), (1423), (1432).
Die erste `for`-Schleife im Aufruf `perm(4)` tauscht nacheinander `2,3,4` in die erste Position, und `perm(n-1)` erzeugt die Permutationen von `[1,3,4]` (bei fester 2), `[1,2,4]` (bei fester 3) und `[1,2,3]` (bei fester 4). So werden alle $4! = 24$ Permutationen gewonnen.

3.3 Ackermann-Funktion

Im Jahre 1926 erfand W. Ackermann eine Funktion, die wir in der Fassung von Rózsa Péter [36] vorstellen. Das ist die Definition:

$$A(m,n) = \begin{cases} n+1 \text{ falls } m = 0 \\ A(m-1,1) \text{ falls } m > 0 \text{ und } n = 0 \\ A(m-1, A(m,n-1)) \text{ sonst} \end{cases} \tag{3.1}$$

Es ist nicht schwierig, diese rekursive Definition in Python zu implementieren:

```
def A(m,n):
    if m == 0:y = n+1
    else:
      if n == 0:y = A(m-1,1)
      else:y = A(m-1,A(m,n-1))
    return y
```

Das Hauptprogramm soll nicht weiter besprochen werden. Es besteht nur aus der Eingabe und dem Aufruf der Funktion A.

y/x	0	1	2	3	4
0					
1					?
2					X
3					X
4					X

Tabelle 3.1: Füllen Sie die Tabelle mit den Werten der Ackermann-Funktion

Versuchen Sie, obige Tabelle auszufüllen. Wahrscheinlich werden Sie den Funktionswert $A(4,1)$ mit obigem Programm nicht bestimmen können. Statt dessen werden Sie eine Fehlermeldung ähnlich dieser erhalten: „maximum recursion depth exceeded“. Sie rührt von der großen Anzahl der rekursiven Aufrufe her. So ergibt sich die Frage: Was ist so merkwürdig an der Ackermann-Funktion? Wozu dient sie?

Dies sind Fragen, die in das Gebiet der Theoretischen Informatik führen. Im Jahre 1926 stellte D. Hilbert die Vermutung auf, eine Funktion sei dann und nur dann berechenbar, wenn sie aus bestimmten elementaren Funktionen zusammengesetzt ist. Solche Funktionen bezeichnet man als *primitiv rekursiv* [37]. Im Computerzeitalter kennzeichnet man diese Funktionen häufig dadurch, dass sie mithilfe von Programmen berechnet werden können, die nur `for`-Schleifen verwenden. Ackermann hingegen erfand seine Funktion als ein Beispiel für eine berechenbare Funktion, die nicht primitiv rekursiv ist, sich also nicht, wie man heute sagt, mit `for`-Schleifen berechnen lässt. Die Funktion ist jedoch *WHILE berechenbar* [38]. Es gibt ein Programm, das anstelle von Rekursionen eine `while`-Schleife zur Berechnung von Werten der Ackermann-Funktion verwendet [39]. Es benutzt einen Stack, den wir durch eine Liste in Python emulieren können. Hier ist das Listing:

```
print('Ackermann-Funktion mit WHILE')
x = int(input('m = '))
y = int(input('n = '))
m = x; n = y
stack = []
stack.append(m) # push(m; stack)
stack.append(n) # push(n; stack)
Zaehler = 0
while len(stack) != 1:
```

```
    Zaehler += 1
    if Zaehler % 1000 == 0: print('Durchlaeufe = ',Zaehler)
    n = stack[-1]; stack.pop(-1) # n := pop(stack)
    m = stack[-1]; stack.pop(-1) # m := pop(stack)
    if m == 0:
       stack.append(n+1) # push(n + 1; stack)
    else:
       if n == 0:
          stack.append(m-1) # push(m-1; stack)
          stack.append(1) # push(1; stack)
       else:
          stack.append(m-1) # push(m-1; stack)
          stack.append(m) # push(m; stack)
          stack.append(n-1) # push(n-1; stack)
Ergebnis = stack[-1] # pop(stack)
print(); print(\Anzahl der Durchlaeufe = ",Zaehler)
print(); print(\A(")
```

Dieses Programm benötigt keine weitere Funktion, und es ist nicht rekursiv. Die Kommentare zeigen die entsprechenden Operationen auf dem Stack an. Weil sich die Argumente `m` and `n` während des Programmlaufs ändern, benutzen wir die Variablen `x` and `y`, sodass eine korrekte Ausgabe dargestellt wird. Danach werden die Argumente `m` und `n` auf den Stack gebracht. Im Rumpf der `while`-Schleife wird die Definition 3.1 in Stack-Operationen „übersetzt“. Zusätzlich zählt die Variable `Zaehler` die Durchläufe der `while`-Schleife. Versuchen Sie noch einmal mit diesem Programm, die obige Tabelle 3.1 zu füllen. Der Autor erhielt den korrekten Wert für $A(4,1)$, doch waren dafür 2,86 Milliarden Durchläufe der `while`-Schleife erforderlich.

3.4 Mergesort - Sortieren durch Mischen

Das Sortieren von Daten gehört zu den wichtigsten Anwendungen von Computern. Daher erstaunt es nicht, dass es viele Sortieralgorithmen gibt. Die meisten sind gut dokumentiert, daher behandeln wir sie hier nicht. Nur wenn die Daten sortiert sind, können schnelle Suchverfahren verwendet werden, z.B. die Binäre Suche, siehe Abschnitt 1.1.4. Wir möchten ein kleines Programm entwickeln, das die Funktionsweise eines der ältesten Sortieralgorithmen demonstriert. Schon John von Neumann erfand es im Jahre 1945. Mergesort ist kein „in-place“-Algorithmus, d.h. es benötigt mehr Speicherplatz als die zu sortierenden Daten.

Das Prinzip ist schnell erklärt [40]. Mergesort arbeitet nach dem Programmierkonzept „divide and conquer“ (teile und herrsche). Im ersten Schritt wird die Liste (oder der Array) der zu sortierenden Schlüssel in zwei Hälften gespalten. Dann werden die beiden Teile wieder halbiert. Dieser Prozess wird so lange fortgesetzt, bis die erhaltenen Listen aus nur jeweils einem Element bestehen. Dann gibt es nichts mehr zu sortieren. Wenn dieser Teil fertig ist, werden die Elemente zu Listen aus zwei Elementen zusammengesetzt, diese zu Listen aus vier Elementen usw., wobei das Reißverschlussverfahren angewendet wird. Schließlich erhält man so die fertig sortierte Liste.

Unser Programm soll eine Liste von 40 zufällig bestimmten Großbuchstaben sortieren. Um Mergesort, das Sortieren durch Mischen, zu demonstrieren, verwenden wir Turtlegrafik, obwohl nichts gezeichnet wird. Daher beginnt das Programm mit `import turtle as tu` und `import random as rd`. Das Hauptprogramm enthält Vorbereitungen für Turtlegrafik, und es initialisiert die Liste `a` der Buchstaben. Die entscheidende rekursive Funktion `mergesort` wird aufgerufen, und die Operationen zum Beenden der Turtlegrafik werden ausgeführt, siehe Anhang Beenden der Turtlegrafik.

Die Funktion `update` wird nicht in Einzelheiten besprochen. `update(kk,x,y)` bewegt die Turtle zur Position $(x, y.)$. Ein Buchstabe, der sich dort möglicherweise schon befindet, wird gelöscht. Dann schreibt die Turtle `a[kk]`.

Die wichtigste Funktion ist `mergesort`. Die Parameter `l` und `r` sind die linke und rechte Grenze der Buchstabenliste für den Aufruf von `mergesort`. Die Parameter `h` und `p` haben mit dem Sortiervorgang nichts zu tun. `h` ist ein Maß für die `y`-Position auf der Zeichenfläche der Turtlegrafik, und mithilfe von `p` wird die horizontale Position festgelegt.
Die Teilliste von `a`, die mit `l` beginnt und mit `r` endet, wird dargestellt. Die genaue Position für jeden Buchstaben wurde durch Versuch und Irrtum herausgefunden. Dann wird der Mittelwert `m` von `l` und `r` berechnet, wobei die Ganzzahldivision `//` verwendet wird. Zwei rekursive Aufrufe von `mergesort` folgen. Im ersten Aufruf wird `r` durch `m` ersetzt, im zweiten `l` durch `m+1`. Das bewirkt, dass jeder Buchstabe zwischen `l` und `r` genau einmal in einer der Teillisten vorkommt, die von `l` bis `m` und von `m+1` to `r` reichen. Die Hilfsparameter `h` und `p` werden den rekursiven Aufrufen von `mergesort` mit den nötigen Änderungen übergeben. Schließlich wird die Reißverschluss-Funktion `merge` aufgerufen, die als nächste erläutert wird. Im Sonderfall `l=r` (`else`-Zweig) wird nur ein Buchstabe dargestellt, es gibt dann keine rekursiven Aufrufe. Es folgt das Listing:

```
def mergeSort(l, r, h, p):
     if l < r:
        for i in range(l,r+1):update(i,p+13*i,260-50*h)
        m = (l+r)//2
        mergeSort(l, m, h+1,p-65//(2**h))
        mergeSort(m+1, r, h+1,p+65//(2**h))
        merge(l, m, r, h, p)
     else: update(l,p+13*l,260-50*h)
```

Die Funktion `merge` benötigt ebenfalls alle Parameter von `mergesort` und zusätzlich `m`, den Mittelwert von `l` and `r`. Ferner sind drei Hilfslisten nötig: `L = a[l:m+1]` und `R = a[m+1:r+1]` sind die Teile von `a`, die zusammengefügt werden sollen. `A= []` ist anfangs leer; am Ende enthält `A` den sortierten Teil von `a` vom (aktuellen) `l` bis zum (aktuellen) `r`. Eine `while`-Schleife folgt, die das Reißverschluss-Prinzip umsetzt.
Sollte `L` leer sein, wird einfach `R` an `A` angefügt; entsprechend wird `L` zu `A` hinzugefügt, wenn `R` leer sein sollte. Wenn jedoch keine der beiden Listen leer ist, wird der lexikografisch erste Buchstabe von `L[0]` oder `R[0]` an `A` gehängt. Weiter ist es wichtig, den gerade an `A` gehängten Buchstaben aus der entsprechenden Liste zu löschen. Wenn die `while`-Schleifen abgearbeitet sind, wird die nunmehr sortierte Liste `A` an der richtigen Stelle in `a` kopiert und auf der Zeichenfläche dargestellt. Dies ist das auf den ersten Blick etwas verwirrend aussehende Listing der Funktion:

```
def merge(l, m, r, h, p):
     A = []; L = a[l:m+1]; R = a[m+1:r+1]
        while (L != []) or (R != []):
           if L == []: A += R; R = []
           elif R == []: A += L; L = []
           else:
              while (L != []) & (R != []):
                 if L[0] < R[0]: A.append(L[0]); L.pop(0)
                 else: A.append(R[0]); R.pop(0)

     for i in range(len(A)): a[l+i] = A[i]
     for i in range(l,r+1): update(i,p+13*i,260-50*h)
```

Abbildung 3.2 zeigt die Zeichenfläche der Turtlegrafik, wenn die Sortierung teilweise fertig ist. Im Anhang findet sich ein weiterer Code ohne Turtlegrafik. Dieser kann einfach benutzt werden, wenn die zu sortierenden Schlüssel keine Buchstaben sind.

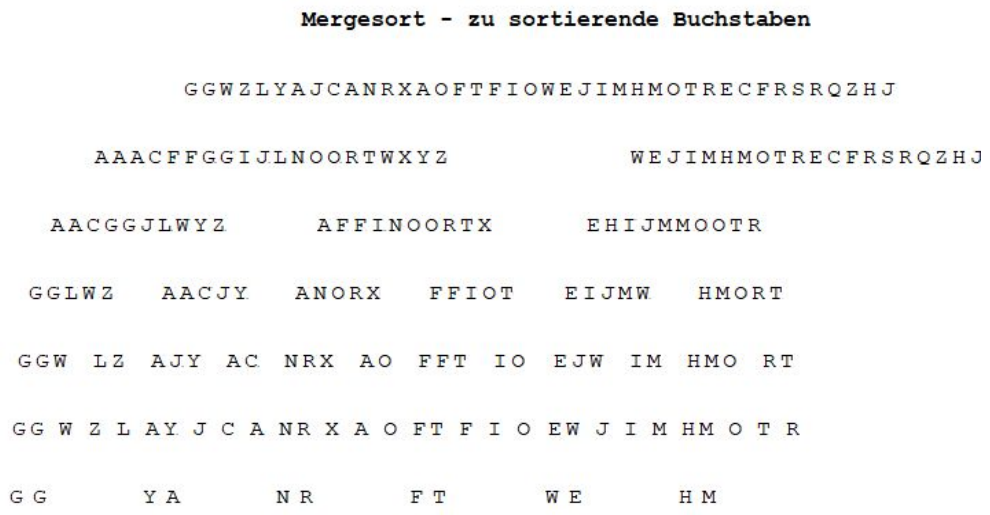


Abbildung 3.2: Fast drei Viertel der gegebenen Liste sind sortiert.

3.5 Türme von Hanoi

„Türme von Hanoi“ oder Brahma ist ein mathematisches Spiel, das 1883 von Édouard Lucas erfunden wurde [41]. Der *Quellturm* besteht aus n Scheiben mit wachsendem Durchmesser, die kleinste Scheibe ist die oberste. Daher hat der Turm die Form eines Kegels. Die Scheiben sollen an einen anderen Ort gebracht werden, dort soll der *Zielturm* entstehen. Bei diesem Transport darf niemals eine größere Scheibe auf einer kleineren liegen. Die größere Scheibe würde zerbrechen. Die Scheiben sind alle so schwer, dass nur eine zur Zeit bewegt werden kann. Desweiteren gibt es nur einen einzigen *Hilfsturm*, auf dem man Scheiben zwischenlagern kann. Auch dort darf nie eine größere Scheibe auf einer kleineren liegen, siehe Abbildung 3.3. Es gibt eine Reihe von Legenden über dieses Spiel, siehe [41], [42]. Im Grunde ist es aber ein eher neues Spiel.

Wir wollen ein kurzes Programm mit einer rekursiven Funktion entwickeln, welches dieses Spiel löst. Im Programmteil befindet sich auch das Programm `TuermeHanoi_Plot`, das Turtlegrafik verwendet. Es demonstriert alle notwendigen Bewegungen von Scheiben, der Code ist jedoch komplizierter.
Wie gewöhnlich ist unser Hauptprogramm sehr kurz. Es besteht nur aus der Eingabe der Anzahl der Scheiben und dem Aufruf der Funktion `schiebe`, die die Anzahl der notwendigen Bewegungen zurückgibt.

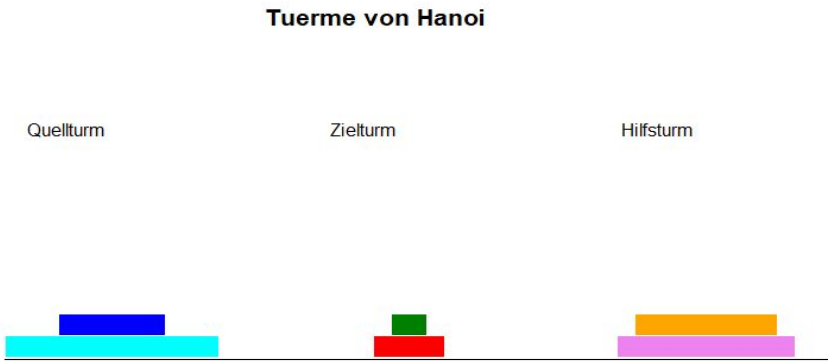


Abbildung 3.3: Türme von Hanoi

`schiebe` ist die wichtige Funktion. Ihre Parameter sind `x,y,z,k,Zaehler`; sie werden durch `x='Quellturm', y='Zielturm', z='Hilfsturm'` initialisiert. `k` ist die relevante Anzahl von Scheiben; es kann mehr geben, aber nur die oberen `k` Scheiben werden betrachtet. `Zaehler` zählt die Anzahl der Bewegungen, die am Ende zurückgegeben wird. Sie ist anfangs gleich 0. Beachten Sie, dass sich alle diese Parameter während der rekursiven Aufrufe verändern. Schauen wir uns das Listing an:

```
def schiebe(x,y,z,k,Zaehler):
    if k == 1:
       Zaehler += 1
       print('Scheibe ',k,' ',x,' -> ',y)
    else:
       Zaehler = schiebe(x,z,y,k-1,Zaehler)
       Zaehler += 1
       print('Scheibe ',k,' ',x,' -> ',y)
       Zaehler = schiebe(z,y,x,k-1,Zaehler)
    return Zaehler
```

Ist `k=1`, wird `Zaehler` um 1 erhöht, und die Scheibe wird von `x` nach `y` verschoben, d.h. vom aktuellen Quellturm zum aktuellen Zielturm. Der `else`-Zweig enthält rekursive Aufrufe und ist komplizierter. Stellen Sie sich vor, dass wir die ersten `k-1` Scheiben auf einmal bewegen könnten. Im ersten rekursiven Aufruf `Zaehler = schiebe(x,z,y,k-1,Zaehler)` werden die Rollen vom Zielturm und Hilfsturm vertauscht. So wird der Turm aus den oberen `k-1` Scheiben zum Hilfsturm bewegt. Die Scheibe `k` wird zum Zielturm geschoben: `print('disk ',k,' ',x,'->',y)`. Der folgende Aufruf `Zaehler = schiebe(z,y,x,k-1,Zaehler)` bewirkt, dass der Turm der oberen `k-1` Scheiben vom Hilfsturm zum Zielturm bewegt wird. Doch wir können nicht

mehr als eine Scheibe zur Zeit bewegen. Daher sind viele rekursive Aufrufe nötig, um alle Scheiben korrekt zum Zielturm zu bewegen. Man kann zeigen, dass die Mindestzahl von notwendigen Bewegungen gleich $2^n - 1$ ist, wobei n die Anzahl der Scheiben ist.

Schauen wir uns noch ein Beispiel an. Wir setzen `n` = 3. `'Quellturm', 'Zielturm', 'Hilfsturm'` werden mit `Q,Z,H` abgekürzt. Im Hauptprogramm wird `schiebe(Q,Z,H,3,0)` aufgerufen. Dann geschieht Folgendes:

```
schiebe(Q,Z,H,3,0)
   schiebe(Q,H,Z,2,0)
      schiebe(Q,Z,H,1,0): Zaehler = 1; Scheibe 1:Q → Z
   Zaehler = 2; Scheibe 2:Q → H
       schiebe(Z,H,Q,1,2): Zaehler = 3; Scheibe 1:Z → H
Zaehler = 4; Scheibe 3:Q → Z
   schiebe(H,Z,Q,2,4)
      schiebe(H,Q,Z,1,4): Zaehler = 5; Scheibe 1:H → Q
   Zaehler = 6; Scheibe 2:H → Z
   schiebe(Q,Z,H,1,6): Zaehler = 7; Scheibe 1:Q → Z
```

Es sollte noch erwähnt werden, dass es auch einen iterativen Algorithmus für das Spiel gibt, siehe [43]. Die Komplexität der Algorithmen (rekursiv oder iterativ) ist in beiden Fällen hoch. Der Aufwand ist proportional zu $2^n - 1$, wobei n die Anzahl der Scheiben ist, siehe oben. Deswegen sollte die Anzahl der Scheiben eines Turms nicht größer als 6 sein.

3.6 Das Rucksack-Problem

Es ist nicht schwierig zu erklären, worin das *Rucksack-Problem* besteht [44], [45]. Angenommen, es gibt mehrere Gegenstände, die Sie in Ihrem Rucksack transportieren möchten. Diese haben ein Gewicht und einen Wert. Sie möchten Ihren Rucksack mit einem möglichst hohen Wert beladen, aber in ihm kann nur ein bestimmtes Höchstgewicht transportiert werden, wir nennen es `maxGewicht`. Doch `maxGewicht` ist kleiner als die Summe der Gewichte aller Gegenstände. Einige müssen zurückbleiben. Die Aufgabe besteht darin herauszufinden, welche Gegenstände Sie mitnehmen können, sodass ihr Wert maximal ist und die Summe der Gewichte `maxGewicht` nicht überschreitet.

Das Rucksack-Problem ist ein so genanntes NP-Problem [46], [47]. Das heißt, dass kein Algorithmus bekannt ist, mit dem man das Problem mit dem Aufwand n^k lösen kann, wobei n die Anzahl der Gegenstände ist und k eine

Zahl. Das Problem kann jedoch mit „dynamischer Programmierung“ gelöst werden, siehe [45]. Dieses Verfahren funktioniert gut, aber es benötigt viel Speicherplatz im Computer. Hier bevorzugen wir das Verfahren *backtracking* ([48], [49]); dieses nimmt den Speicherplatz nicht so stark in Anspruch.

Erläuterungen zum Schema siehe nächste Seite

$$
\begin{array}{l}
[](0) \to [0](4) \to [0,1](8) \to [0,1,2](-) \\
\qquad\qquad\qquad\qquad\qquad\quad \swarrow \\
\qquad\qquad\qquad\qquad\quad [0,1](8) \to [0,1,3](-) \\
\qquad\qquad\qquad\qquad\qquad\quad \swarrow \\
\qquad\qquad\qquad\qquad\quad [0,1](8) \to [0,1,4](-) \\
\qquad\qquad\qquad\qquad\qquad\quad \swarrow \\
\qquad\qquad\qquad\quad \swarrow \\
\qquad\qquad [0](8) \to [0,2](9) \to [0,2,3](-) \\
\qquad\qquad\qquad\qquad\qquad\quad \swarrow \\
\qquad\qquad\qquad\qquad\quad [0,2](9) \to [0,2,4](-) \\
\qquad\qquad\qquad\qquad\qquad\quad \swarrow \\
\qquad\qquad\quad \swarrow \\
\qquad\qquad [0](9) \to [0,3](-) \\
\qquad\qquad\quad \swarrow \\
\qquad\qquad [0](9) \to [0,4](10) \\
\qquad\qquad\quad \swarrow \\
\qquad \swarrow \\
[](10) \to [1](10) \to [1,2](10) \to [1,2,3](-) \\
\qquad\qquad\qquad\qquad\qquad\quad \swarrow \\
\qquad\qquad\qquad\qquad\quad [1,2](10) \to [1,2,4](-) \\
\qquad\qquad\qquad\qquad\qquad\quad \swarrow \\
\qquad\qquad\quad \swarrow \\
\qquad\qquad [1](10) \to [1,3](10) \to [1,3,4](-) \\
\qquad\qquad\qquad\qquad\qquad\quad \swarrow \\
\qquad\qquad\quad \swarrow \\
\qquad\qquad [1](10) \to [1,4](10) \\
\qquad\qquad\quad \swarrow \\
\qquad \swarrow \\
[](10) \to [2](10) \to [2,3](-) \\
\qquad\qquad\quad \swarrow \\
\qquad\qquad [2](10) \to [2,4](11) \\
\qquad\qquad\quad \swarrow \\
[](11) \to [3](11) \to [3,4](-) \\
\qquad\qquad\quad \swarrow \\
\qquad \swarrow \\
[](11) \to [4](11)
\end{array}
$$

Im obigen Schema wird gezeigt, wie die optimale Lösung für ein kleines Beispiel gefunden wird. Die Gewichte und Werte sind durch diese beiden Listen gegeben: `Gewicht = [7,4,6,8,6]` und `Wert = [4,4,5,4,6]`. Das höchstzulässige Gewicht ist `maxGewicht` = 13. Die gerade aktuell gewählten Nummern der Gegenstände werden in `indList` gespeichert. Sie ist ein Teil der Liste `[0,1,2,3,4]`. Im Schema der Vorseite wird die jeweils aktuelle `indList` dargestellt, z.B. [0,1,2]. Die Zahl in runden Klammern im Schema ist der gerade optimale Wert `optiWert`, die entsprechenden Nummern der Gegenstände (beginnend mit 0) werden in `optiList` gespeichert (nicht im Schema dargestellt). Am Anfang gilt natürlich `optiWert = 0` und `optiList = []`. Wenn statt des optimalen Wertes ein Strich in der runden Klammer steht, bedeutet das, dass die entsprechende `indList` nicht zulässig ist, also die Summe der zugehörigen Gewichte `maxGewicht` übersteigt. Ein Pfeil zeigt nach rechts, wenn die nächste mögliche Zahl zu `indList` hinzugefügt wird. Dann muss geprüft werden, ob die so erzeugte `indList` noch zulässig ist. Diagonale Pfeile stehen für Backtracking, und die letzte Zahl in `indList` wird entfernt.

Der Algorithmus betrachtet das erste Element der Liste der Gewichte und prüft, ob `maxGewicht` überschritten wird. Das ist nicht der Fall, also wird `optiWert = 4` gesetzt und `indList = optiList = [0]`. Auch die Summe der ersten beiden Gewichte ist kleiner als `maxGewicht`. Daher wird `optiWert = 8` und `indList = optiList = [0,1]`. Doch ist die Summe der ersten drei Gewichte größer als `maxGewicht`. Nun ist `indList = [0,1,2]`, aber `optiWert` und `optiList` bleiben unverändert, `2` muss aus `indList` entfernt werden. Jetzt versucht der Algorithmus, `3` zu `indList` hinzuzufügen, aber auch `[0,1,3]` ist unzulässig usw.
Die Anzahl der zu prüfenden `indlists` ist in diesem Beispiel gleich 23. Weil wir es mit fünf Gegenständen zu tun haben, ist die Gesamtzahl der Möglichkeiten gleich 32. Die Ersparnis der zu prüfenden Möglichkeiten erscheint in unserem Beispiel nicht allzu groß. Aber betrachten Sie bitte das Beispiel mit elf Gegenständen in den Übungsaufgaben.

Um diesen Algorithmus in Python zu codieren, benötigen wir noch einige Variablen mehr. `WertList` and `GewList` sind Listen, die Teillisten von `Wert` und `Gewicht` sind. `WertList` enthält die Werte der Gegenstände von `indList`, `GewList` ist die entsprechende Liste der Gewichte. Beispiel: Wenn `indList = [0,2,4]` ist, gilt `WertList = [4,5,6]` und `GewList = [7,6,6]`.

`markiert` ist eine Liste von booleschen Variablen, die anfangs alle auf `False` gesetzt werden. `markiert[index]` wird gleich `True`, wenn es nicht möglich

ist, dass ein neuer, besserer `optiWert` mithilfe einer `indList` gefunden werden kann, der `index` enthält, genaueres dazu siehe unten. `n` ist einfach die Anzahl der Gegenstände, und `Zaehler` zählt die zu untersuchenden `indLists`. Dieser Zähler ist für den Algorithmus unnötig, doch er zeigt, um wie viel besser das Backtracking-Verfahren im Vergleich zum Probieren aller Möglichkeiten ist. Hingegen ist `Basis` sehr wichtig. Dieser Wert teilt der Funktion `suche` mit, bei welcher Zahl zu beginnen ist, falls `indList` gerade leer sein sollte. Das kann man auch an obigem Schema erkennen. Nach `indList = [0,4]` werden beide Elemente entfernt, und die Liste ist leer. Die `Basis` wird jetzt 1, auch `indList = [1]`, und das Programm versucht wieder, die Liste weiter zu füllen.

Alle Listen außer `markiert` werden durch `[]` initialisiert und alle Zahlen mit `0`. Abgesehen von diesen Initialisierungen besteht das *Hauptprogramm* nur aus Funktion `suche` und der Ausgabe der Ergebnisse. Das ist das Listing:

```
n = len(Gewicht)
indList, optiList, WertList, GewList = [], [], [], []
markiert = [False for i in range(n)]
Basis, Zaehler, optiWert = 0, 0, 0
suche(-1)
print('optimaler Wert: ',optiWert)
print('optimale Objekte: ',optiList)
print('Anzahl der Checks: ',Zaehler)
```

Die rekursive Funktion `suche` verrichtet die wesentliche Arbeit in diesem Programm. Zuerst werden die Variablen `Zaehler,Basis,optiWert,optiList` als `global` deklariert. Was bedeutet das, und warum ist es erforderlich? Normalerweise sind alle Variablen in Python lokal. Sie sind nur in der Funktion verfügbar, in der sie definiert wurden. Hingegen sind Variablen, die im Hauptprogramm (also nicht in irgendeiner Funktion) definiert wurden, global. Sie können auch in Funktionen verwendet werden. Wir aber wollen die Werte der oben aufgeführten Variablen in der Funktion verändern. In diesem Falle müssen wir sie als `global` deklarieren. Dieses Schlüsselwort ist nicht nötig in Ausgabeanweisungen und für Zugriffe ohne Veränderung der Variablen.

Nun aber wird die Hilfsfunktion `finde` aufgerufen: `y=finde(aktuell)`. Diese Funktion liefert den ersten Index zurück, für den `markiert = False` ist oder `-1`, falls es keinen solchen Index ab `aktuell+1` gibt. Zu Beginn ist `aktuell = -1`, kein Index ist markiert, und `finde` liefert `0`. Im weiteren Verlauf kommt es aber auch vor, dass `y` gleich `-1` wird. In diesem Fall ist Backtracking nötig.

Wir betrachten aber zuerst den Fall `y >= 0`. `aktuell` wird gleich `y` gesetzt und an die Listen `indList`, `GewList` and `WertList` gehängt. Das bedeutet aber noch nicht, dass wir eine neue Lösung für unsere Aufgabe gefunden haben. Die Beschränkung `sum(GewList) <= maxGewicht` und die Bedingung `sum(WertList) > optiWert` müssen überprüft werden. Sind sie beide erfüllt, wurde wirklich eine neue und bessere Lösung gefunden:

```
if (sum(GewList) <= maxGewicht) & (sum(WertList) > optiWert):
    optiWert = sum(WertList); optiList = indList.copy()
```

Zur Vermeidung unnötiger Suche sind die beiden folgenden Zeilen wichtig:

```
if sum(GewList) > maxGewicht:
    for ii in range(aktuell+1,n): markiert[ii] = True
```

Betrachten wir `indList=[0,1,2]` im obigen Schema. Es ist `sum(GewList) = 7+4+6 = 17 > maxGewicht`. Folglich werden die Indizes `3` und `4` markiert, und `indLists [0,1,2,3]` , `[0,1,2,4]` und `[0,1,2,3,4]` werden nicht überprüft. Der rekursive Aufruf `suche(aktuell)` ohne jede Abbruchbedingung ist die letzte Zeile des `if`-Zweiges.

Schauen wir uns jetzt den `else`-Zweig an. Er wird durchlaufen, wenn `y=-1` ist. Dann sind alle Indizes > `aktuell` markiert, oder es gibt keinen. In diesem Falle muss Backtracking ausgeführt werden. Auch `aktuell` wird noch markiert: `markiert[aktuell] = True`. `saeubern` ist eine Hilfsfunktion, die mit dem Parameter `indList[-1]` aufgerufen wird, dem letzten Element von `indList`. Alle Markierungen von `indList[-1]+1` an werden auf `False` gesetzt. Warum ist das nötig? Schauen wir uns noch einmal unser Beispiel an. `indList=[2,3]` ist unzulässig, denn `Gewicht[2]+Gewicht[3]=6+8>maxGewicht`. Daher werden `3` und `4` markiert. Noch wurde aber `indList=[2,4]` nicht überprüft, daher muss die Markierung von `4` wieder entfernt werden. Tatsächlich liefert `indList=[2,4]` die optimale Lösung, und wir möchten sie nicht verpassen. Dann werden die letzten Elemente von `indList`, `GewList` and `WertList` entfernt. Wenn danach `indList` nicht leer ist, erhält `aktuell` den nunmehr letzten Wert von `indList`: `if indList != []:aktuell = indList[-1]`.

Ist aber `indList` leer, wird die Sache etwas komplizierter. In diesem Falle muss `aktuell` den Wert `Basis` erhalten. Am Anfang wurde im Hauptprogramm `Basis` auf `0` gesetzt. Aus dem obigen Schema kann man entnehmen, dass `indList` zum ersten Mal leer wird, wenn alle Möglichkeiten für `0` in `indList` ausgeschöpft sind. Daher muss `Basis` erhöht werden, und die nächste Versuchsreihe beginnt mit `indList = [1]`, und `0` kommt nicht mehr vor. Schließlich folgt der rekursive Aufruf mit der Abbruchbedingung `if Basis<n:suche(aktuell)`. Hier ist das vollständige Listing von `suche`:

```
def suche(aktuell):
    global Zaehler, Basis, optiWert, optiList
    y = finde(aktuell)
    if y >= 0:
       Zaehler += 1; aktuell = y
       indList.append(aktuell)
       GewList.append(Gewicht[aktuell])
       WertList.append(Wert[aktuell])
       if(sum(GewList)<=maxGewicht)&(sum(WertList)>optiWert):
          optiWert = sum(WertList); optiList = indList.copy()
       if sum(GewList) > maxGewicht:
          for ii in range(aktuell+1,n): markiert[ii] = True
       print('aktuell = ',Zaehler')
       print('optimaler Wert=',optiWert,' indList=',indList)
       suche(aktuell)
    else: # backtracking
       markiert[aktuell] = True
       saeubern(indList[-1])
       indList.pop(-1); GewList.pop(-1); WertList.pop(-1)
       if indList != []: aktuell = indList[-1]
       else:
          aktuell = Basis; Basis += 1
       if Basis < n: suche(aktuell)
```

Schließlich notieren wir noch die Listings der kleinen Hilfsfunktionen `finde` und `saeubern`:

```
def finde(x):
    posi = -1
    for j in range(n-1,x,-1):
       if (not markiert[j]): posi = j
    return posi

def saeubern(x):
    y = x + 1
    while y <= n-1:
       markiert[y] = False
       y += 1
```

3.7 ILP mit „Branch and Bound“

ILP steht für **I**nteger **L**inear **P**rogram. Damit sind Aufgaben zur linearen Optimierung [50], [51] gemeint, in der nur ganzzahlige Lösungen ermittelt werden sollen. Derartige Aufgaben sind wichtig, denn häufig ergeben nur ganzzahlige Lösungen einen Sinn. Zum Beispiel können Sie nicht 4,73 Autos kaufen, und es werden nur ganze Sofas produziert, nicht Bruchteile davon. Es ist nicht möglich, eine ILP-Aufgabe durch einfaches Runden zu lösen. Denn gerundete Lösungen könnten Nebenbedingungen verletzen.

Es gibt verschiedene Verfahren, mit denen man optimale Lösungen einer allgemeinen Aufgabe zur linearen Optimierung ermitteln kann. Bekannt ist vor allem das Simplex-Verfahren, siehe [52], [53]. Dieses funktioniert normalerweise sehr gut, aber es ließ sich bisher nicht beweisen, dass es stets eine annehmbar gute Laufzeit hat. Es wurde jedoch bewiesen, dass man solche Aufgaben in *polynomialer* Laufzeit lösen kann, siehe [54]. Das heißt es gibt eine natürliche Zahl k, sodass die Laufzeit proportional zu n^k ist, wobei n die Anzahl der Variablen ist. Für die Lösung allgemeiner Optimierungsprobleme gibt es eine effektive Funktion im Modul *scipy*. Diese werden wir verwenden und uns auf die Lösung von ILP-Aufgaben konzentrieren.

Eine Strategie zur Lösung von ILPs ist *Branch and Bound*. Das ist ein Konzept, das folgendermaßen funktioniert: Die Aufgabe wird in zwei oder mehr Teile aufgeteilt, die leichter als die ursprüngliche Aufgabe gelöst werden können; das ist die Verzweigung (Branch). In den meisten Fällen ergeben aufeinander folgende Verzweigungen einen Baum. Einige Zweige jedoch müssen vielleicht nicht bearbeitet werden, weil im ursprünglichen Problem Nebenbedingungen (Bounds) vorkommen.

Speziell ist diese Strategie für Optimierungsaufgaben nützlich. Sie könnten meinen, nur ganzzahlige Lösungen für ein Optimierungsproblem zu suchen, sei einfacher als die Ermittlung beliebiger Lösungen. Doch das Gegenteil ist der Fall. Kein Algorithmus ist bekannt, der beliebige ILPs in polynomialer Zeit löst. Wenn man Glück hat, kann man jedoch mit Branch and Bound viel Rechenzeit einsparen.

Der erste Schritt besteht in einer Vereinfachung, oft *Relaxation* genannt. Die Beschränkung, dass nur ganzzahlige Lösungen bestimmt werden sollen, wird weggelassen. Die so erhaltene Lösung kann nicht ganzzahlige Ergebnisse enthalten, z.B. $x_3 = 1.5$. Dann wird die ursprüngliche Aufgabe in zwei Teilaufgaben zerlegt. Die erste ist die bisherige Aufgabe mit der Zusatzbe-

dingung $x_3 \leq 1$; die zweite enthält die Zusatzbedingung $x_3 \geq 2$. Bei der Lösung der ersten Aufgabe kann es sein, dass $x_4 = 7.3$ herauskommt. In diesem Fall wird eine weitere Verzweigung vorgenommen. So kommt es zu immer mehr Verzweigungen, bis einer der folgenden Fälle vorliegt: (1) Eine ganzzahlige Lösung wurde gefunden. (2) Das aktuelle Optimierungsproblem ist nicht zulässig in dem Sinne, dass es nicht möglich ist, alle Bedingungen zu erfüllen. (3) Eine bessere Lösung wurde bereits gefunden. Im nächsten Schritt werden dann alle Lösungen gesammelt und die beste ausgewählt.

Jetzt sind wir so weit, dass wir ein ILP in Python kodieren können. Ein Beispiel der Universität Siegen [55] ist dabei hilfreich zur Verdeutlichung der Funktionsweise des Programms. Es ist nicht zu kompliziert, aber auch nicht zu trivial. Hier ist es:
Die zu minimierende *Zielfunktion* ist gegeben durch $c(x) := -x_1 - x_2$.
Die *Nebenbedingungen* sind

$$\begin{aligned} -2x_1 + x_2 &\leq -\frac{1}{2} \\ -x_2 &\leq -\frac{1}{2} \\ 2x_1 + x_2 &\leq \frac{11}{2} \end{aligned}$$

Zuerst müssen die erforderlichen Daten bereitgestellt werden, also die Zielfunktion und die Nebenbedingungen. Es ist ziemlich mühsam, alle benötigten Daten unter Verwendung von Eingabefunktionen einzutippen. Daher speichern wir die Daten in einer Textdatei. Sie kann mit einem Editor für „plain text“ erstellt werden. Wenn Sie ihre eigene Textdatei schreiben, beachten Sie bitte, dass die Einträge jeweils durch eine Leerstelle getrennt sein müssen. Für unser Beispiel sieht die Datei so aus:

$$\begin{array}{rrr} -1 & -1 & 0 \\ -2 & 1 & -0.5 \\ 0 & -1 & -0.5 \\ 2 & 1 & 5.5 \end{array}$$

Die erste Zeile besteht aus den Koeffizienten der Zielfunktion. Am Ende wurde eine 0 hinzugefügt, weil alle Zeilen die gleiche Länge haben müssen. Alle anderen Zeilen enthalten die Koeffizienten der Nebenbedingungen. Schauen wir auf die dritte Zeile. Die entsprechende Ungleichung enthält nicht x_1. In der Textdatei wurde eine 0 an der Position des fehlenden Koeffizienten eingefügt. Die Funktion `lies` besorgt das Einlesen der Daten. Im Hauptprogramm muss die Variable `Dateiname` ersetzt werden, im Beispiel durch

'ILP1.txt'. Mit den Einzelheiten dieser Funktion wollen wir uns nicht befassen. In ihr wird die Funktion `loadtxt` des Moduls *numpy* verwendet. Dieses muss im Programmkopf importiert werden. Wichtiger ist die Ausgabe `obj, A, B`. `obj` ist eine Liste mit den Koeffizienten der Zielfunktion. `A` ist eine Matrix und `B` ist ein Vektor, sodass die Nebenbedingungen in der Form $A \cdot x \leq B$ gegeben sind, wobei x der Vektor der Variablen ist.

Die Funktion `loese_LP` enthält einen Aufruf der Funktion `linprog` des Moduls *scipy*. Dafür wird das Modul `scipy.optimize` importiert. `linprog` liefert eine ganze Reihe von Ausgaben; die meisten von ihnen sind in Bezug auf unsere Aufgabe nicht von Belang. Die Funktion `loese_LP` gibt daher nur `x,y,Erfolg` zurück. `x` ist der Vektor der optimalen Variablen entsprechend der eingegebenen Optimierungsaufgabe. Einige dieser Werte sind möglicherweise nicht ganzzahlig. `y` ist der optimale Wert der Zielfunktion. `Erfolg` ist `True`, wenn die Lösung zulässig ist, sonst `False`.

Die nächsten beiden Funktionen prüfen die Eingabe `x`. `ist_ganz` ergibt `True`, wenn `x` fast ganzzahlig ist. Die Gleichung $x = round(x)$ ist immer falsch, wenn x eine Gleitkommazahl ist. Das liegt an der endlichen Rechengenauigkeit. Daher benutzen wir `(abs(x-xniedrig) < epsilon) or (abs(xhoch - x) < epsilon)`. Dabei ist `epsilon` ein Wert, der im Hauptprogramm festgelegt wird. `xniedrig` $= \lfloor x \rfloor$ und `xhoch` $= \lceil x \rceil$ sind die größte ganze Zahl, die kleiner oder gleich `x` bzw. die kleinste ganze Zahl, die größer oder gleich `x` ist. Während `ist_ganz` eine einzige Zahl als Eingabe braucht, verlangt `alle_ganz` eine Liste. Diese Funktion prüft, ob alle Zahlen der Liste `x`, die als Lösung von `loese_LP` erhalten wird, ganzzahlig sind.

Die bisher behandelten Programmteile sind nur Hilfsfunktionen für die wichtigen. Die Hauptfunktion ist `branch`. Sie wird immer aufgerufen, wenn die vorangehende Optimierung mindestens eine nicht ganzzahlige Variable liefert. Sie gibt eine boolesche Variable zurück, welche den Wert `True` hat, wenn eine Lösung der Optimierungsaufgabe gefunden wurde.
Die benötigten Parameter sind `ind, x, A, B, xList, yList`. `ind` ist der Index der ersten nicht ganzzahligen Variable. `x` ist die Liste der aktuellen Variablen, `A` ist die aktuelle Matrix und `B` der aktuelle Vektor der rechten Seite des Systems der Ungleichungen $A \cdot x \leq B$. Das Wort „aktuell“ muss auch hier betont werden, weil alle diese Variablen während des Programmlaufs ihre Werte verändern können. `xList` ist die Liste von Listen aller Variablen und `yList` die Liste aller entsprechenden optimalen Werte, die in vorangehenden Optimierungen erhalten wurden.

In `branch` werden die lokalen Variablen `gefunden_l` und `gefunden_r` mit `False` initialisiert. Sie erhalten den Wert `True`, sobald eine ganzzahlige Lösung im linken oder rechten Zweig gefunden wurde. Die Zeile `xl,yl,Al,Bl,Erfolg = links(ind,x,A,B)` folgt. Die Funktion `links` fügt eine weitere Nebenbedingung zum bisherigen System hinzu. `A_` ist ein Vektor, der zunächst nur aus Nullen besteht. Seine Dimension ist gleich der Anzahl der Variablen. Dann aber wird 1 an die Position `A_[ind]` gesetzt, denn an der Stelle `ind` befindet sich eine nicht ganzzahlige Variable. Dieser neue Vektor `A_` wird an die bisherige Matrix `A` angehängt, wodurch sich Matrix `A0` ergibt. Vektor `B` wird ergänzt durch $= \lfloor x[ind] \rfloor$, der dadurch entstehende Vektor `B0` enthält also eine Koordinate mehr als B. Dann wird das neue System von Ungleichungen durch einen Aufruf von `loese_LP` gelöst, und zwar mit den Parametern `obj,A0,B0`, also mit der Zielfunktion, der erweiterten Matrix und der erweiterten rechten Seite. Wenn die Lösung nicht zulässig ist, d.h. `Erfolg == False`, wird `y0` auf „unendlich“ gesetzt, was in Python mithilfe von *numpy* durch `np.inf` ausgedrückt wird. Damit ist gesichert, dass eine solche Lösung sicher nicht optimal ist. `links` gibt den Satz der neuen optimalen Variablen `x0`, den neuen optimalen Zielwert `y0` zurück, aber auch die erweiterte Matrix `A0`, den erweiterten Vektor `B0` und schließlich `Erfolg`, womit entscheidbar ist, ob die Lösung des Optimierungsproblems erfolgreich war.

Die Funktion `rechts` funktioniert ganz analog. Es ist aber zu beachten, dass die neue Nebenbedingung $x[ind] \geq \lceil b[ind] \rceil$ durch $-x[ind] \leq -\lceil b[ind] \rceil$ ersetzt werden muss. Deshalb setzen wir `A_[ind] = -1; A1 = A + [A_]`. Der Vektor `B` muss durch `B1 = B + [-np.ceil(x[ind])]` ersetzt werden.

Zurück zur Funktion `branch`. Wenn im linken Zweig eine ganzzahlige und zulässige Lösung gefunden wurde, werden die neue Liste `xl` und der entsprechende optimale Wert `yl` an die Listen `xList` bzw. `yList` gehängt. `gefunden_l` wird dann auf `True` gesetzt. Der zugehörige Code besteht aus den folgenden drei Zeilen:

```
if alle_ganz(xl) & (yl != np.inf):
   xList.append(xl); yList.append(yl)
   gefunden_l = True
```

Dieser Teil wurde für den linken Zweig dargestellt, der entsprechende für den rechten Zweig sieht ganz analog aus.

Im Allgemeinen werden wir aber keine ganzzahligen Lösungen erhalten, und die Verzweigung muss fortgesetzt werden. Wann immer wir eine nicht ganzzahlige Variable durch die Funktionen `links` oder `rechts` erhalten haben,

könnten wir eine neue Verzweigung vornehmen. Genau das wird im Skript `IPL_2` gemacht. Es ist klar, dass der Rechenaufwand exponentiell mit der Anzahl der nicht ganzzahligen Variablen wächst. Aber wir können die Anzahl der Verzweigungen durch „bounding“ reduzieren.

Wir erläutern die Idee anhand des Beispiels von oben. Die Vereinfachung, die auch nicht ganzzahlige Lösungen zulässt, liefert den optimalen Wert $y = -4$, wobei $x_1 = 1.5$, $x_2 = 2.5$ gilt, siehe Programm `IPL_3`. Daher muss eine Verzweigung ausgeführt werden. Die zusätzlichen Nebenbedingungen sind $x_1 \leq 1$ für den linken und $x_1 \geq 2$ für den rechten Zweig. Funktion `links` liefert `yl = -2.5` und `xl = [1, 1.5]`, Funktion `rechts` hingegen `yr = -3.5` und `xr = [2, 1.5]`. Das Minimum von `yl` und `yr` ist eine Schranke für den optimalen Wert des ILP. Er kann nicht kleiner als -3.5 sein. Weil `yr` kleiner als `yl` ist, versuchen wir zunächst, unter Verwendung des rechten Zweiges eine ganzzahlige Lösung zu finden. Sollten wir eine Lösung mit einem optimalen Wert < -2.5 erhalten, ist es nicht notwendig, auch noch den linken Zweig zu betrachten. Das ist nur erforderlich, wenn der rechte Zweig keine zulässige ganzzahlige Lösung erbringt.

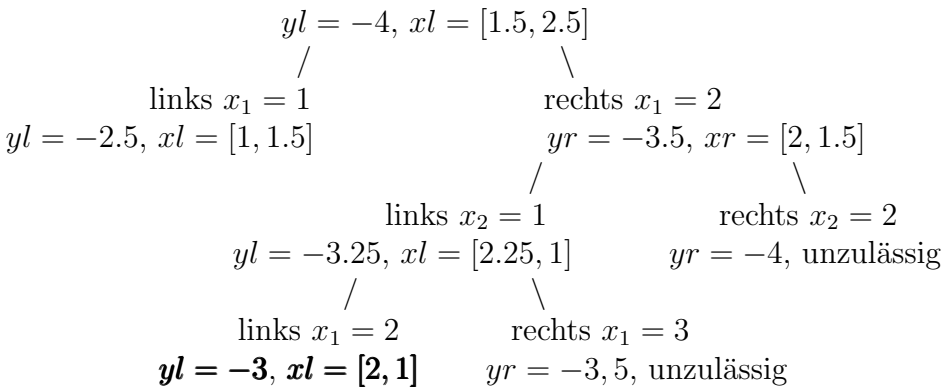
Im Beispiel scheint der rechte Zweig die bessere Wahl zu sein, aber wir haben noch keine ganzzahlige Lösung. Daher muss der folgende Teil der Funktion `branch` ausgeführt werden:

```
if (yr <= bound) & (yr != np.inf) & (gefunden_r == False):
   i = 0
   while (i < len(xr)-1) & ist_ganz(xr[i]) : i += 1
   if i < len(xr):
      gefunden_r = branch(i,xr,Ar,Br,xList,yList)
```

Wir stellen zunächst fest, dass alle Bedingungen der `if`-Verzweigung erfüllt sind. Die beiden nächsten Zeilen dienen dazu, den ersten Index `i` aufzufinden, für den `x[i]` nicht ganzzahlig ist, falls es einen solchen gibt. Wenn das der Fall ist, wird ein rekursiver Aufruf von `branch` ausgeführt. Der entsprechende Teil für den linken Zweig sieht genauso aus und braucht hier nicht behandelt zu werden.

Das Beispiel liefert für den linken Zweig `yl = -3.25` und `xl = [2.25, 1]`. Der rechte Zweig ergibt `yr = -4` und `xr = [2, 2]`. Doch ist diese Lösung nicht zulässig ($2x_1 + x_2 \leq \frac{11}{2}$ ist verletzt) und muss gestrichen werden.
Nach einer neuen Verzweigung erhalten wir `yl=-3`, `xl = [2,1]` und `yr=-3.5`, `xl = [3,0.5]`. Die letztere Lösung ist nicht zulässig; daher ist die optimale Lösung $y = -3$ mit $x_1 = 2, x_2 = 1$. Da dieser optimale Wert kleiner ist als der zuvor für den linken Zweig erhaltene, brauchen wir nicht den linken Zweig

der ersten Verzweigung zu betrachten. Die vollständige Lösung des Beispiels ist in der folgenden Skizze dargestellt.



Beispiel für ein ILP Der vollständige Baum für obiges Beispiel; der Zweig links oben muss nicht weiter verfolgt werden, weil der optimale Zielwert -3 ist, und damit kleiner als yl im linken oberen Zweig.

Funktion `branch` ist beinahe fertig, doch ein Fall fehlt noch. Es ist möglich, dass der vermutlich bessere Zweig (der mit dem niedrigeren Wert in der Zielfunktion) keine zulässige Lösung liefert. Dann muss der verbleibende Zweig weiter behandelt werden.

```
y2 = np.inf; gefunden2 = False
if yl > bound: y2 = yl; x2 = xl; A2 = Al; B2 = Bl
if yr > bound: y2 = yr; x2 = xr; A2 = Ar; B2 = Br
```

Durch die vorangehenden drei Zeilen bekommen die Parameter des verbleibenden Zweiges den Index 2 zugewiesen. So brauchen wir nur einen zusätzlichen Aufruf für eine weitere Verzweigung. Es ist noch zu bemerken, dass dieser Aufruf nur nötig ist, wenn bislang noch keine ganzzahlige Lösung gefunden wurde und `y2` größer ist als die bereits bestimmte Schranke `bound`. Der Rumpf der `if`-Anweisung ist analog zu den bereits behandelten Fällen.

```
if(y2!=np.inf)&(gefunden_l==False)&(gefunden_r==False):
   i = 0
   while (i < len(x2)-1) & ist_ganz(x2[i]) : i += 1
   if i<len(x2):gefunden2=branch(i,x2,A2,B2,xList,yList)
```

Es gibt drei Möglichkeiten, während des Laufs der Funktion `branch` eine ganzzahlige Lösung zu finden: `gefunden_l` oder `gefunden_r` können den Wert `True` angenommen haben und schließlich auch noch `gefunden2`. Daher lautet die letzte Zeile der Funktion
`return gefunden_l or gefunden_r or gefunden2`.

Hier ist das ganze Listing der Funktion `branch`:

```
def branch(ind,x,A,B,xList,yList):
    gefunden_l = False; gefunden_r = False
    xl,yl,Al,Bl,Erfolg = links(ind,x,A,B)
    if Erfolg:
       if alle_ganz(xl) & (yl != np.inf):
          xList.append(xl); yList.append(yl)
          gefunden_l = True
    xr,yr,Ar,Br,Erfolg = rechts(ind,x,A,B)
    if Erfolg:
       if alle_ganz(xr) & (yr != np.inf):
          xList.append(xr); yList.append(yr)
          gefunden_r = True

    bound = min(yl,yr)
    if (yl <= bound) & (yl != np.inf) & (gefunden_l == False):
       i = 0
       while (i < len(xl)-1) & ist_ganz(xl[i]) : i += 1
       if i<len(xl):gefunden_l=branch(i,xl,Al,Bl,xList,yList)
    if (yr<=bound) & (yr!=np.inf) & (gefunden_r==False):
       i = 0
       while (i < len(xr)-1) & ist_ganz(xr[i]) : i += 1
       if i<len(xr):gefunden_r=branch(i,xr,Ar,Br,xList,yList)

    y2 = np.inf; gefunden2 = False
    if yl > bound: y2 = yl; x2 = xl; A2 = Al; B2 = Bl
    if yr > bound: y2 = yr; x2 = xr; A2 = Ar; B2 = Br
    if(y2!=np.inf)&(gefunden_l==False)&(gefunden_r==False):
       i = 0
       while (i < len(x2)-1) & ist_ganz(x2[i]) : i += 1
    if i<len(x2):gefunden2=branch(i,x2,A2,B2,xList,yList)
```

Das *Hauptprogramm* ist wieder recht einfach. Es beginnt mit einigen Initialisierungen: `epsilon = 1e-6`, `xList` und `yList` werden auf `[]` gesetzt. Eine Textdatei (siehe oben) wird gelesen, um die Zielfunktion und die Nebenbedingungen zu definieren. Dann wird das allgemeine Optimierungsproblem gelöst: `x,y, Erfolg = loese_LP(obj,A,B)`. Falls die Lösung dieses Problems schon ganzzahlig ist oder aber unzulässig, ist die Aufgabe bereits gelöst, und das Ergebnis wird dargestellt. Anderenfalls beginnt die Verzweigung. Am Ende werden `xList`, `yList` und das Ergebnis ausgegeben.

Der Programmteil enthält drei Versionen: `ILP` gibt nur die Ergebnisse aus, während `ILP_3` auch viele Zwischenergebnisse anzeigt, sodass man sehen kann, wie der Algorithmus arbeitet. Zudem gibt es die Version `ILP_2`, in der nur die Verzweigungen implementiert sind, nicht aber das „Bounding“. Der Unterschied in den Laufzeiten mag unbedeutend erscheinen, aber in Übungsaufgabe 6 wird er deutlicher.

3.8 Übungsaufgaben

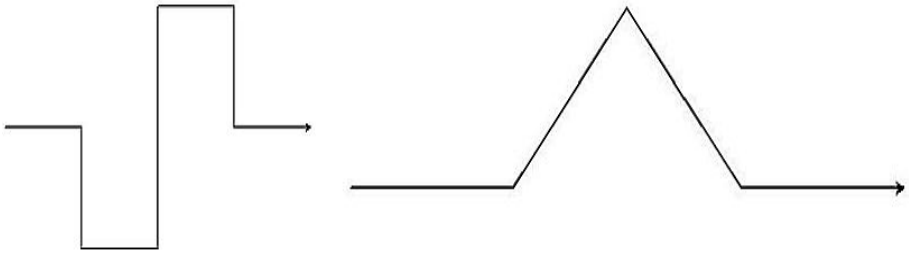


Abbildung 3.4: Hakenkurve und Schneeflockenkurve

1. **Hakenkurve [56] und Schneeflockenkurve [57]** Entwickeln Sie Programme zum Zeichnen dieser beiden Kurven. Stufe eins von ihnen ist in der Abbildung 3.4 dargestellt. Die rekursive Funktion `zeichne` beginnt mit `Laenge /= 4` bei der *Hakenkurve*, bei der *Schneeflockenkurve* genügt es, `Laenge` durch 3 zu dividieren. Im Falle `Stufe = 1` bedeutet `zeichne(Laenge,Stufe-1)` einfach nur `tu.forward(Laenge)`.
   Das Hauptprogramm ist für beide Kurven gleich. Hier ist es:

```
for Zaehler in range(3):
     tu.up(); tu.setpos(-300,0); tu.down()
     tu.width(2)
     zeichne(800,Zaehler)
     time.sleep(3);tu.clearscreen()
tu.up(); tu.hideturtle()
tu.setpos(-300,-250)
tu.pencolor((0,0,0))
tu.write('fertig!',font = ("Arial",12,"normal"))
tu.mainloop()
tu.done()
```

   Hinweis: Bitte am Anfang des Programms `import time` nicht vergessen.

2. **Fibonacci-Funktionen höherer Ordnung**
   Die *Fibonacci-Funktion fibop der Ordnung p* einer Zahl n ist definiert durch die Summe der $p+1$ Vorgänger von n [58]. Im Falle $n = p$ ist $fibop(p) = 1$, im Falle $n < p$ gilt $fibop(n) = 0$.
   Schreiben Sie ein Programm zur Berechnung von Werten der Funktion *fibop*, wobei wahlweise ein rekursiver oder iterativer Algorithmus angewendet werden soll.

3. **Q-Funktion**
   Die Q-Funktion von D. R. HOFSTADTER [59] ist folgendermaßen definiert:

$$Q(k) = \begin{cases} 1, \text{ falls } k \leq 2 \\ Q(k - Q(k-1)) + Q(k - Q(k-2)) \text{ sonst} \end{cases}$$

   Schreiben Sie ein Programm zur Berechnung von Werten dieser Funktion, wobei wahlweise ein rekursiver oder iterativer Algorithmus angewendet wird.
   Bemerkung: Es wurde noch nicht bewiesen, dass die Q-Funktion wohldefiniert ist. Vielleicht benötigt die Rekursion negative Argumente, für welche Q nicht existiert. Doch gilt dies als unwahrscheinlich [59].

4. **Permutationen**
   Der folgende Code gehört zum HEAP-Algorithmus zur Berechnung von Permutationen [60], [61]. Führen Sie einen Schreibtischtest für diesen Code mit n=3, A = [1,2,3] aus. Notieren Sie, in welcher Reihenfolge die Permutationen ausgegeben werden.

```
def Perm(m):
    if m == 1:
       print(A)
    for i in range(m):
       Perm(m-1)
       if m % 2 == 1:
          A[m-1], A[0] = A[0], A[m-1]
       else:
          A[m-1], A[i] = A[i], A[m-1]
```

5. **Rucksack Problem**
   (1) Führen Sie das Programm `Rucksack_Problem` mit folgenden Parametern aus: `Gewicht` = [6,8,7], `Wert` = [5,6,8] und `maxGewicht` = 13.
   (2) Führen Sie das Programm aus mit `Gewicht` = [5, 7, 6, 8, 10, 11, 12, 15, 17, 30, 18] , `Wert` = [8, 9, 6, 5, 10, 5, 10, 17, 19, 20, 13] und

`maxGewicht` = 33.
(3) Verändern Sie das Programm so, dass es alle Möglichkeiten prüft, die Gegenstände anzuordnen.

6. **ILP**
(1) Testen Sie alle vorhandenen Textdateien mit den Programmversionen `ILP_3` und `ILP_2`.
(2) Suchen Sie Beispiele für jeden möglichen Fall und testen Sie Ihre Beispiele:
   (a) das LP ist nicht zulässig,
   (b) die LP-Vereinfachung ergibt schon eine ganzzahlige Lösung,
   (c) das LP ist zulässig, das ILP jedoch nicht.
(3) Lösen Sie das folgende ILP (von der Universität Münster [62]).
Zielfunktion: max $-2x_1 + x_2 - 8x_3$
Nebenbedingungen:

$$\begin{aligned} 2x_1 - 4x_2 + 6x_3 &\leq 3 \\ -x_1 + 3x_2 + 4x_3 &\leq 2 \\ 2x_3 &\leq 1 \\ x_1, x_2, x_3 &\geq 0 \end{aligned}$$

Benutzen Sie `ILP_3` und `ILP_2`. Zählen Sie die unterschiedlichen Aufrufe der Funktion `branch`. Lohnt es sich, *bounding* zu verwenden?

Kapitel 4

Einige verschränkt rekursive Algorithmen

4.1 Collatz-Problem

Das Problem ist sehr einfach zu formulieren: Man nehme eine natürliche Zahl. Ist sie ungerade, wird 1 zu ihrem Dreifachen addiert. Ist die Zahl gerade, wird sie durch 2 dividiert. Collatz stellte schon vor 1950 die Vermutung auf, dass jede so gebildete Folge von Zahlen in der Periode ...,4,2,1,4,2,1,... enden müsse [63], [64]. Ein Beispiel ist: $17 \to 52 \to 26 \to 13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1 \to 4 \to 2 \to 1$ ...
In den vergangenen Jahren wurden immer wieder Beweise für die Gültigkeit der Collatz-Vermutung vorgelegt, z.B. [65], [66]. Diese wurden jedoch stets angezweifelt [67], [68]. Es wurden auch Preise für einen Beweis oder ein Gegenbeispiel ausgelobt [69]. In diesem Beitrag wollen wir bescheidener sein. Es geht uns hier nur um ein einfaches Beispiel für zwei Funktionen, die sich *verschränkt rekursiv* aufrufen. Die Folge soll auch nicht mit ewigen Wiederholungen von 4,2,1 weitergehen, sondern das Programm soll enden, wenn 1 erreicht ist. Die Funktion `gerade` lautet:

```
def gerade(n):
    print(n,end = ' ')
    n = n //2
    if n % 2 == 0: gerade(n)
    else: ungerade(n)
```

Einige Erläuterungen hierzu: Nach der Ausgabe von `n` wird die Zahl mit Ganzzahldivision durch 2 dividiert, gekennzeichnet durch `n // 2`. Ob das Ergebnis gerade ist, stellt man durch `n % 2` fest. Diese Operation ermittelt den Divisionsrest. Ist er 0, ist die Zahl gerade. In diesem Falle wird erneut

`gerade` aufgerufen, anderenfalls `ungerade`. Eine Abbruchbedingung gibt es hier nicht, denn die Folge endet nicht mit einer geraden Zahl.

Die Funktion `ungerade` sieht ähnlich einfach aus:

```
def ungerade(n):
    if n == 1: print(1)
    else:
       print(n,end = ' ')
       gerade(3*n+1)
```

Hier gibt es eine Abbruchbedingung: `if n == 1: print(1)`. Anderenfalls folgt nach der Ausgabe von `n` ein rekursiver Aufruf von `gerade`. Im *Hauptprogramm* wird der Benutzer nur aufgefordert, eine ungerade Zahl einzugeben. Es folgt der Aufruf von `ungerade`. Damit haben wir zwei Funktionen, die einander verschränkt rekursiv aufrufen: `gerade` ruft sich selbst oder `ungerade` auf, und `ungerade` ruft `gerade` auf. Nur eine der beiden Funktionen benötigt eine Abbruchbedingung. Es ist auch möglich, in jeder rekursiven Funktion eine Abbruchbedingung zu verwenden.

Der Reiz der COLLATZ-Vermutung liegt unter anderem darin, dass die entstehenden Folgen sehr unterschiedliche Längen haben. Ob die Folge lang oder kurz ist, lässt sich bei Eingabe der Zahl kaum abschätzen. Folgende Eingaben seien empfohlen: 27, 97, 113, 491, 497. Die Länge der Folgen hat offenbar nichts mit Primzahlen zu tun.

4.2 Mondrian

Piet MONDRIAN (1872 - 1944) war ein berühmter niederländischer Maler [70], [71]. In vielen Museen, speziell in den Niederlanden, werden seine Bilder ausgestellt [72]. Arbeiten, die zur künstlerischen Stilrichtung „Neuplastizismus“ gehören, sind häufig aus schwarzen horizontalen und vertikalen Balken zusammengesetzt, die ein Gitternetz bilden. Einige der rechteckigen Flächen davon sind mit kräftigen Farben ausgefüllt, z.B. rot, blau und gelb [73], [74].

Wir wollen ein Programm entwickeln, das Bilder nach dem Vorbild dieser Art von Gemälden erzeugt. Natürlich ist die Qualität der Computerbilder nicht mit der von originalen Werken vergleichbar. Es ist unschwer zu erkennen, dass ihnen die Ausgewogenheit fehlt. Ein Grund dafür ist die Verwendung von Zufallszahlen im Programm. Wir benutzen nur die Technik der oben beschriebenen Bilder, haben aber keinen künstlerischen Anspruch. Ein mögliches Ergebnis wird in Abbildung 4.2 gezeigt.

Die bedeutendsten Funktionen sind `horizontal` und `vertikal`. Sie sind *verschränkt rekursiv*, `vertikal` ruft `horizontal` auf, und `horizontal` ruft `vertikal` auf, bis eine Abbruchbedingung erfüllt ist. Beide Funktionen benötigen die vier Parameter `links`, `rechts`, `unten` und `oben`. Durch sie wird der Bereich markiert, in welchem der nächste horizontale oder vertikale Balken gezeichnet werden soll.

Aber als erstes brauchen wir zwei globale Variable: `minSpanne` and `halb`. `minSpanne` ist ein Kriterium für die Differenz von `oben` und `unten` in Funktion `vertikal` bzw. von `rechts` und `links` in Funktion `horizontal`. Ist `minSpanne` unterschritten, wird kein neuer schwarzer Balken mehr gezeichnet, allenfalls könnte der vorliegende Bereich gefärbt werden. `halb` ist die halbe Breite eines schwarzen Balkens. Zum Beispiel kann man `minSpanne = 120` und `halb = 5` vorgeben.

Nun aber zur Funktion `vertikal`. Wir definieren die lokale Variable `Spanne` als Differenz von `rechts` und `links`. Der interessante Teil ist der `if`-Zweig des Listings unten. Dort werden gesetzt `li = Spanne//4+links` und `re = -Spanne//4+rechts`. Dadurch liegen die Positionen `li` and `re` ein wenig vom linken und rechten Rand des Bereichs entfernt, wo ein vertikaler Balken gezeichnet werden könnte. Dann wird `Mitte` durch `links` oder `-1` initialisiert. Das ist in jedem Falle außerhalb des Bereiches für einen vertikalen Balken, sodass die folgende `while`-Schleife mindestens einmal durchlaufen werden muss. Der Schleifenrumpf sorgt dann aber dafür, dass `Mitte` zufällig einen Wert zwischen `li` und `re` erhält:
`Mitte = links + rd.randint(1,Spanne)`.

Nunmehr wird die Funktion `fuellen` aufgerufen:
`fuellen(Mitte-halb,Mitte+halb,unten,oben,0)`.
Sie zeichnet einen neuen vertikalen schwarzen Balken. Die Grenzen sind `Mitte-halb` und `Mitte+halb`, wobei `unten` und `oben` unverändert bleiben. Der letzte Parameter `0` steht für die Zeichenfarbe, also schwarz. Zum Schluss wird die Funktion `horizontal` zweimal aufgerufen. Dadurch wird versucht, zuerst einen horizontalen Balken zwischen `links` und `Mitte-halb` zu zeichnen und dann zwischen `Mitte+halb` und `rechts`. Das erfolgt aber nur, wenn dafür genug Platz vorhanden ist.

Wenn die Bedingung `Spanne >= minSpanne` nicht erfüllt ist, kann das Rechteck mit den Grenzen `links`, `rechts`, `unten` und `oben` mit rot, blau oder gelb gefärbt werden, indem die Funktion `faerben` aufgerufen wird.

Listing der Funktion vertikal:

```
def vertikal(links,rechts,unten,oben):
    Spanne = rechts - links
    if Spanne >= minSpanne:
       li = Spanne//4 + links
       re = -Spanne//4 + rechts
       if links < 0: Mitte = links
       else: Mitte = -1
       while (Mitte < li) or (Mitte > re):
          Mitte = links + rd.randint(1,Spanne)
       fuellen(Mitte-halb,Mitte+halb,unten,oben,0)
       horizontal(links,Mitte-halb,unten,oben)
       horizontal(Mitte+halb,rechts,unten,oben)
    else: faerben(links,rechts,unten,oben)
```

Die Funktion `horizontal` läuft ganz analog zu `vertikal` ab. Abbildung 4.1 demonstriert, wie die beiden Funktionen einander aufrufen.

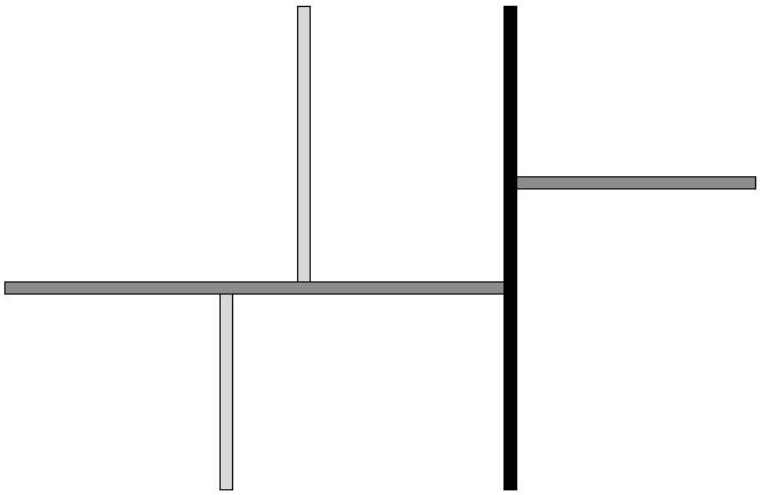


Abbildung 4.1: Zuerst wird der schwarze vertikale Balken durch einen Aufruf von `vertikal` gezeichnet. Die Funktion `horizontal` wird daraufhin zweimal aufgerufen. Folglich werden zwei dunkelgraue horizontale Balken gezeichnet, einer links und einer rechts des vertikalen Balkens. Jeder Aufruf von `horizontal` ruft zweimal `vertikal` auf. Aber nur auf der linken Seite des zuerst gezeichneten schwarzen Balkens werden hellgraue vertikale Balken gezeichnet, einer über und einer unter dem dunkelgrauen horizontalen Balken. Auf der rechten Seite jedoch ist nicht genug Platz für weitere vertikale Balken vorhanden.

Jetzt schauen wir uns das Hauptprogramm an. In ihm werden drei globale Konstanten definiert. `minSpanne = 120` und `halb = 5` wurden oben schon erwähnt. Die dritte ist `Farben = {1,2,3}`. Sie wird gebraucht, um eine Fläche zu färben. Die kleine Funktion `faerben` erzeugt Zufallszahlen von 1 bis 18. Wenn die gezogene Zahl höchstens 3 ist, wird die durch `links`, `rechts`, `unten` und `oben` begrenzte Fläche mit einer Farbe gefüllt. Wird eine höhere Zahl gezogen, wird die Fläche nicht gefärbt. In der Funktion `faerben` selbst wird nur `fuellen` aufgerufen, wobei der zusätzliche Parameter `z` die Farbe kennzeichnet: 1 für rot, 2 für blau, 3 für gelb.

Zurück zum Hauptprogramm. Die Geschwindigkeit der Turtle wird auf `8` gesetzt, damit ist sie sehr schnell. Zudem wird sie durch `tu.hideturtle()` im ganzen Programm unsichtbar. Dennoch wird die Turtle zur korrekten Startposition bewegt, und Funktion `vertikal` wird aufgerufen. Der Schlussteil des Programms ist schon bekannt, siehe 1.1.2 bzw. Anhang. Dies ist das ganze Hauptprogramm:

```
minSpanne = 120;halb = 5;Farben = {1,2,3}
tu.speed(8)
# Bewege Turtle zur Startposition
tu.up(); tu.hideturtle(); tu.left(90)
vertikal(-300,300,-200,200)
# Schlussteil
tu.up();tu.setpos(-300,-250)
tu.pencolor((0,0,0))
tu.write('fertig!',font = ("Arial",12,"normal"))
tu.exitonclick() # Version für PyCharm
try: tu.bye()
except tu.Terminator: pass
```

Es bleibt noch zu erklären, wie die unsichtbare Turtle Balken und Flächen zeichnet. Der erfolgt in Funktion `fuellen`. Die benötigten Parameter sind schon bekannt. Das Füllen selbst wird durch Zeichnen eines Rechtecks bewirkt. `tu.begin_fill()` und `tu.end_fill()` sorgen dafür, dass die Rechtecksfläche mit der ausgewählten Farbe gefüllt wird.

```
    tu.begin_fill()
    tu.setpos(links,unten)
    for i in range(2):
       tu.forward(oben-unten); tu.right(90)
       tu.forward(rechts-links); tu.right(90)
    tu.end_fill()
```

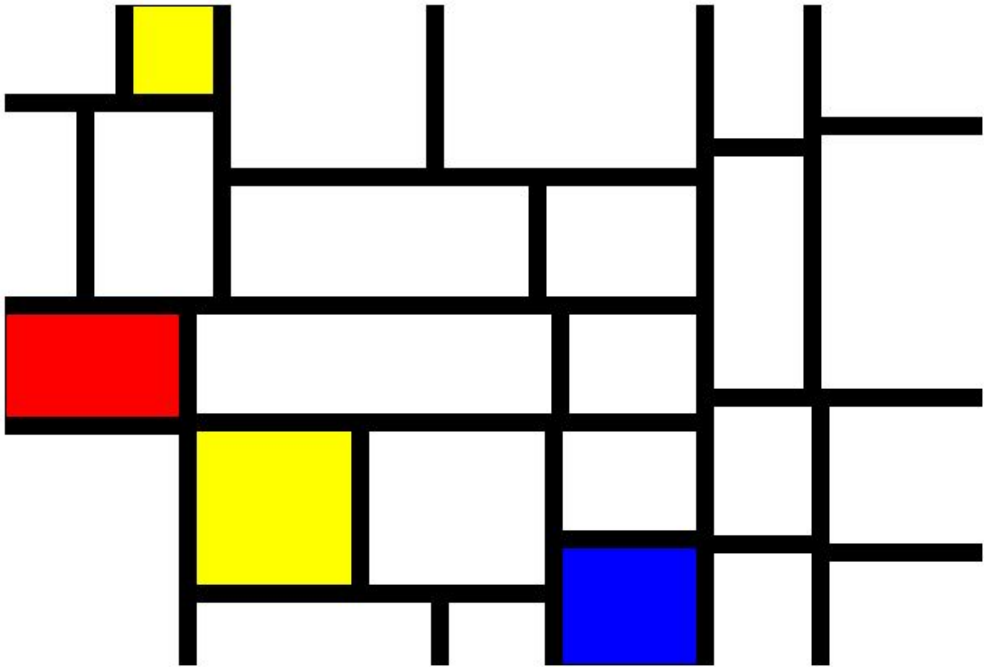


Abbildung 4.2: Beispiel für das Ergebnis eines Durchlaufs des Programms „Mondrian“

4.3 Shakersort

Bubblesort ist ein Sortierverfahren, das sehr leicht zu verstehen ist, doch arbeitet es außerordentlich langsam. Ein ähnliches Verfahren ist *Shakersort* [75], obwohl man vermuten könnte, dass es schneller sei. Die Idee ist diese: Wie in Bubblesort [76] durchläuft der Algorithmus die zu sortierenden Elemente, wobei mit dem letzten angefangen wird. Das aktuelle Element wird mit dem Vorgänger verglichen und mit ihm gegebenenfalls vertauscht, siehe linke Spalte des Schemas unten. Wenn nun der erste Durchlauf fertig ist und sich das Element mit kleinstem Schlüssel an Position 1 befindet, wird die Sortierrichtung umgekehrt und die Sortierung beginnt bei Position 2, siehe mittlere Spalte im Schema. Dadurch wird erreicht, dass das Element mit dem größten Schlüssel an die letzte Position gelangt. Dann wird die Sortierrichtung wieder umgekehrt, die Sortierung beginnt an der vorletzten Position, siehe rechte Spalte im Schema. Diese Änderungen der Richtung des Sortierens werden solange vorgenommen, bis die Sortierung fertig ist. Im Beispiel werden nur 8 Großbuchstaben alphabetisch sortiert. Dort ist die Sortierung nach zwei Durchläufen von hinten nach vorn und einem in umgekehrter Richtung fertig. Meist sind aber mehr Durchläufe erforderlich.

Sortierung rückwärts	Sortierung vorwärts	Sortierung rückwärts
E C B F A H G **D**		
E C B F A H **D** G	A **E** C B F D H G	A C B E D F **G** H
E C B F A **D** H G	A C **E** B F D H G	A C B E D **F** G H
E C B F **A** D H G	A C B **E** F D H G	A C B E **D** F G H
E C B **A** F D H G	A C B E **F** D H G	A C B **D** E F G H
E C **A** B F D H G	A C B E D **F** H G	A C **B** D E F G H
E **A** C B F D H G	A C B E D F **H** G	A **B** C D E F G H
A E C B F D H G	A C B E D F G **H**	

Wir demonstrieren dieses Sortierverfahren auf ähnliche Weise wie mergesort, siehe Abschnitt 3.4. Dazu benutzen wir 60 zufällig gezogene Großbuchstaben. Das Ziehen erfolgt im Hauptprogramm. Die Liste `a` wird durch `[]` initialisiert. Dann werden die zufälligen Buchstaben hinzugefügt.

```
for i in range(n): # setze n = 60
     a.append(rd.randint(65, 90))
     update(a,i)
```

Die Buchstaben sollen alphabetisch sortiert werden. Immer wenn einer mit einem seiner Nachbarn vertauscht wird, wird die Funktion `update` aufgerufen. Diese Funktion soll hier aber nicht im einzelnen behandelt werden. Ein Aufruf wie `update(a,kk)` bewegt die Turtle zur richtigen Position. Sollte dort schon ein Buchstabe vorhanden sein, wird er gelöscht. Dann schreibt die Turtle `a[kk]` dorthin.

Die wesentlichen Funktionen sind `rueckwaerts` und `vorwaerts`. Sie benötigen drei Parameter: die Liste `a`, sowie `l` und `r`, welche die linke und rechte Grenze des Bereichs markieren, in dem die Sortierung noch stattfinden muss. Am Anfang ist `l = 0` und `r = n-1`, denn Python beginnt die Zählung bei 0. Wenn 60 Buchstaben zu sortieren sind, haben ihre Positionen innerhalb der Liste die Nummern 0 bis 59. Die Funktion `shakersort` selbst besteht nur aus dem Aufruf `rueckwaerts(0,n-1)`.

Die beiden Funktionen `rueckwaerts` und `vorwaerts` sehen sehr ähnlich aus; dennoch gibt es wichtige Unterschiede. Wir behandeln zuerst `rueckwaerts`. Die lokale Variable `k` soll die linke Grenze des folgenden Sortiervorgangs markieren. Sie wird anfangs auf `r` gesetzt und bliebe auch `r`, wenn die Liste bereits sortiert wäre. Die folgende `for`-Schleife beginnt bei `r` und endet bei `l`. Der dritte Parameter in `range` ist `-1`. Das bedeutet, dass `j` abwärts zählt, z.B. 8,7,6,...,0. Wenn ein Element mit einem größeren Schlüssel links vom aktuellen Element steht, werden die beiden vertauscht, und es folgt `update`. In

diesem Falle wird `k` auf `j` gesetzt, denn das nach rechts getauschte Element muss möglicherweise nochmals vertauscht werden. Folglich ist nicht sicher, dass der Sortiervorgang von Position `j` an schon fertig ist. Schließlich folgt der rekursive Aufruf `if k < r:voewaerts(a,k,r)`. Die Sortierung in der umgekehrten Richtung ist natürlich nur sinnvoll, wenn `k < r` gilt, denn anderenfalls ist die Liste schon sortiert. Hier ist das Listing von `rueckwaerts`:

```
def rueckwaerts(a,l,r):
    k = r
    for j in range(r,l,-1):
       if a[j-1] > a[j]:
          a[j], a[j-1] = a[j-1], a[j]
          update(a,j-1); update(a,j)
          k = j
    if k < r: vorwaerts(a,k,r)
```

In `vorwaerts` wird `k` anfänglich auf `0` gesetzt. Sollte die Liste schon sortiert sein, änderte sich `k` nicht. Die folgende `for`-Schleife zählt aufwärts. Man beachte, dass die rechte Grenze `r+1` ist. Warum ist das nötig? Möglicherweise müssen `a[r]` und `a[r-1]` vertauscht werden. Deshalb muss der Index `r` von der Schleife erreicht werden. Aber in `range(l,r)` ist der letzte erreichte Index nur `r-1`. Das ist eine Eigenart von Python. Die letzte Zeile enthält den rekursiven Aufruf `if l < k:rueckwaerts(a,l,k)`. Denn im Falle `l >= k` wäre die Liste bereits sortiert. Dies ist das Listing:

```
def vorwaerts(a,l,r):
    k = 0
    for j in range(l,r+1):
       if a[j-1] > a[j]:
          a[j], a[j-1] = a[j-1], a[j]
          update(a,j-1); update(a,j)
          k = j
    if l < k: rueckwaerts(a,l,k)
```

Ist Shakersort signifikant besser als Bubblesort? Dazu haben wir ein kleines Experiment gemacht. Mit `n` = 60 und `tu.speed(6)` haben wir jeweils 10 Durchläufe von Shakersort und Bubblesort gemacht. Der Mittelwert der Laufzeiten war 113.5 ± 14.5 s für Shakersort und 119.5 ± 10.7 s für Bubblesort. Shakersort war also ein wenig besser als Bubblesort, jedoch nicht signifikant. Die Komplexität beider Verfahren ist $O(n^2)$ ist, siehe [75]. Wenn die Anzahl der zu sortierenden Elemente mit 2, 3 usw. multipliziert wird, wächst der Zeitbedarf auf das 4-fache, 9-fache usw. Daher wird Shakersort nicht für das Sortieren sehr vieler Elemente empfohlen.

4.4 Sierpiński-Kurven

Im Jahre 1912 entwickelte der polnische Mathematiker Wacław Sierpiński eine Folge fraktaler Kurven, die wir in Python codieren möchten. Die Kurven dieser Folge sind geschlossen, und sie liegen alle innerhalb einer begrenzten Fläche, etwa dem Einheitsquadrat. Ihre Längen jedoch wachsen exponentiell mit der Stufe der Kurve n. Der Grenzwert für $n \to \infty$ ist eine Kurve, die durch jeden Punkt einer Fläche verläuft. Man nennt sie „flächenfüllend“.

Eine Sierpiński-Kurve [78] [79] besteht aus Buchten und Passagen. Im Falle `n=1` ist eine Passage nur eine Strecke der Länge `h2`; eine von vier Buchten ist in Abbildung 4.3 rot gefärbt.

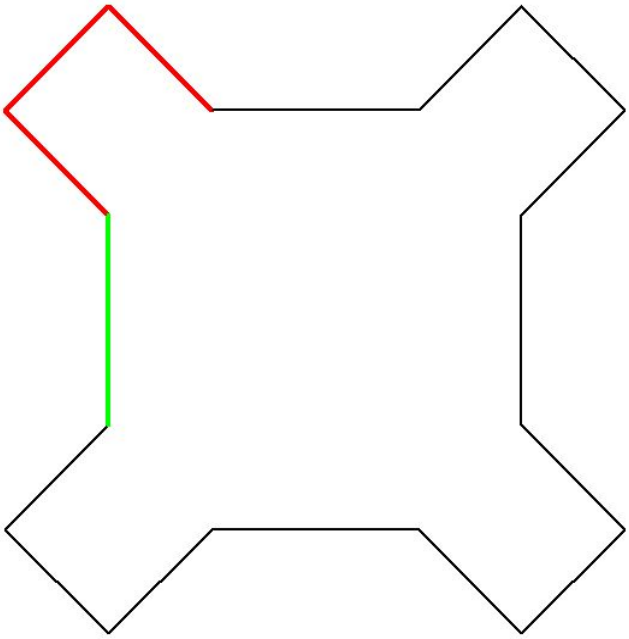


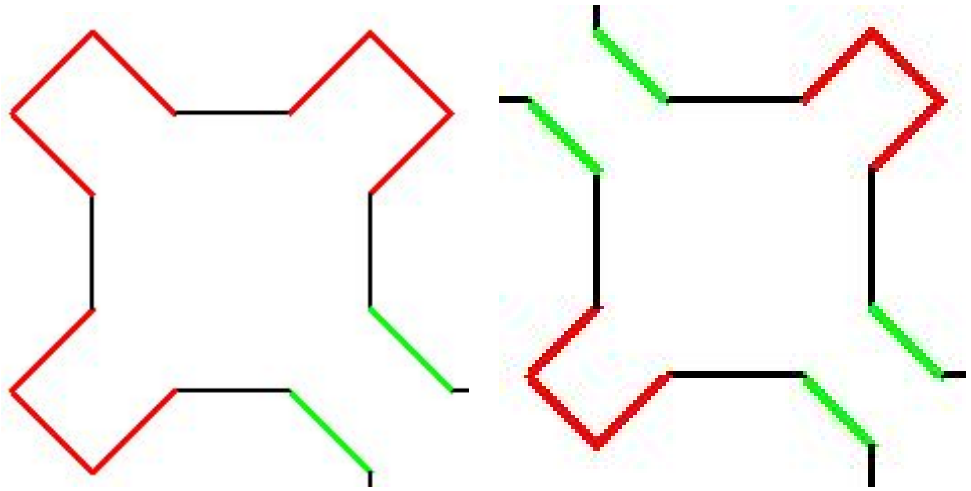
Abbildung 4.3: Sierpiński-Kurve der Stufe 1; die grüne Strecke ist eine Passage, eine Bucht ist rot gefärbt.

In der Funktion `Bucht` sind der Fall `Stufe = 1` und der rekursive Fall `Stufe > 1` unterschiedlich zu behandeln. Der Fall `Stufe = 1` erklärt sich selbst. Offensichtlich wird die Turtle beim Durchlaufen einer Bucht um 90° gedreht.

```
if Stufe == 1:
  tu.left(45);tu.forward(h2)
  tu.right(90);tu.forward(h2)
  tu.right(90);tu.forward(h2)
  tu.left(45)
```

In Abbildung 4.4 ist zu sehen, wie der Code einer Bucht für `Stufe > 1` zu ändern ist. Es genügt nicht, einfach eine Bucht durch drei kleinere Buchten zu ersetzen. Zusätzlich muss sich zuvor die Turtle um 45° nach links drehen und sich dann entlang einer Passage bewegen. Dann dreht sie sich nochmals um 45° nach links und bewegt sich um eine weitere Strecke vorwärts. Danach folgen die Buchten, nach jeder der drei muss sich die Turtle um eine Strecke vorwärts bewegen. Schließlich wird `Passage` noch einmal aufgerufen, zuvor und danach erfolgt eine 45°-Drehung nach links. Dies ist der `else`-Zweig:

```
else:
  tu.left(45);Passage(h,h2,Stufe-1)
  tu.left(45);tu.forward(h2)
  for i in range(3):
    Bucht(h,h2,Stufe-1);tu.forward(h2)
  tu.left(45); Passage(h,h2,Stufe-1)
  tu.left(45)
```



a **b**

Abbildung 4.4: (a) zeigt eine Bucht der Stufe n=2. Die Passagen (n=1) sind grün gefärbt und die Buchten der Stufe n=1 sind rot. **(b)** zeigt eine Passage mit n=2. Die nötigen Buchten mit n=1 sind rot gefärbt, die Passagen mit n=1 sind grün.

Im Falle `Stufe > 1`, in `Passage` muss die Bewegung `tu.forward(h2)` durch die folgenden in Abbildung 4.4b dargestellten ersetzt werden:

```
if Stufe > 1:
  Passage(h,h2,Stufe-1)
  tu.left(45); tu.forward(2*h)
```

```
    Bucht(h,h2,Stufe-1)
    tu.forward(2*h); tu.left(45)
    Passage(h,h2,Stufe-1)
```

`Passage` wird anfangs mit dem Parameter `Stufe-1` aufgerufen und grün gefärbt. Die Turtle dreht um 45° nach links und führt eine Vorwärtsbewegung aus, schwarz in Abbildung 4.4b. Als nächstes wird `Bucht(h,h2,Stufe-1)` aufgerufen, in Abbildung 4.4b rot dargestellt. Danach führt die Turtle wieder eine Linksdrehung von 45° aus und bewegt sich vorwärts. Schließlich wird nach einer weiteren Linksdrehung um 45° noch einmal `Passage` (grün gefärbt) aufgerufen, und die Turtle dreht sich nochmals um 45° links herum. Die Funktionen `Bucht` und `Passage` rufen einander gegenseitig auf, sie sind verschränkt rekursiv.

Das Hauptprogramm ist nicht so kurz wie gewöhnlich. Zwischen der Eingabe der Stufe `n` und dem Aufruf der Funktion sind einige Vorbereitungen nötig. Die anfängliche Länge `h0 = 200` hängt von der Auflösung des Monitors ab und sollte angepasst werden. Zum Zeichnen einer Sierpiński-Kurve müssen zwei unterschiedliche Längen verwendet werden: `h=round(h0)` und `h2`$= \sqrt{2}\cdot$`h0`. Abbildung 4.3 kann man entnehmen, dass der Abstand der rechten und linken Seiten der Kurve gleich $(4+\sqrt{2})\cdot h$ ist. Aus Abbildung 4.4a ist ersichtlich, dass dort der Abstand $(8+3\cdot\sqrt{2})\cdot h$ ist. Daher muss für jede Stufe die ursprüngliche Länge mit

$$\frac{4+\sqrt{2}}{8+3\cdot\sqrt{2}}$$

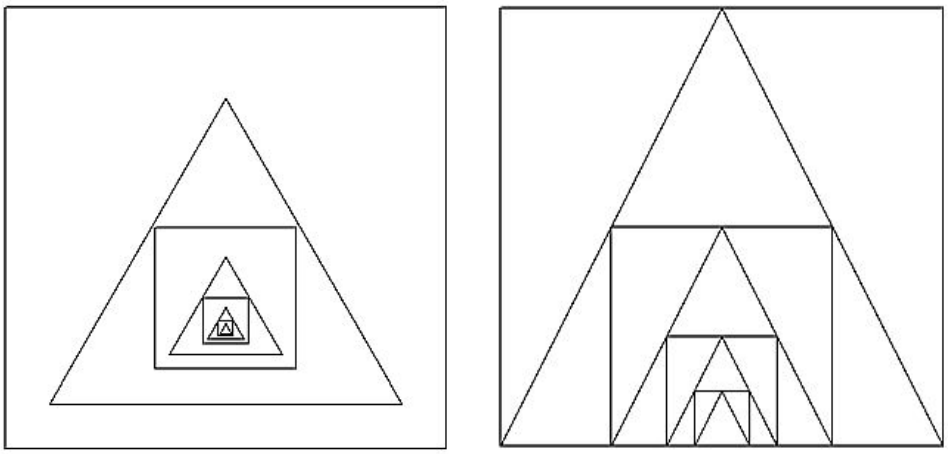
multipliziert werden. Für den Schlussteil des Programms sei auf den Anhang verwiesen. Hier ist fast das ganze Hauptprogramm:

```
n=int(input('n = (1,...,6) '))
h0=200
for i in range(n):
    h0=h0*(4+math.sqrt(2))/(8+3*math.sqrt(2))
h=round(h0);h2=round(math.sqrt(2)*h0)
tu.width(2);tu.speed(5)
tu.up();tu.setpos(-100,60);tu.down();tu.left(90)
for i in range(4):
    Bucht(h,h2,n);tu.forward(2*h)
tu.write('fertig!',font = ("Arial",12,"normal"))
tu.exitonclick() # Version für PyCharm
try: tu.bye()
except tu.Terminator: pass
```

Dieses war nur ein Beispiel für eine Folge von Kurven, deren Grenzwert flächenfüllend ist. Es gibt viel mehr von ihnen. Einige davon sind geschlossene Kurven, andere sind es nicht. Für weitere Informationen sei auf [80] verwiesen. Es sollte möglich sein, mithilfe von Turtlegrafik Hilbert-Kurven zu modellieren, siehe [81]. Dazu sind mehr als zwei verschränkt rekursive Funktionen erforderlich.

4.5 Übungsaufgaben

1. **Verschränkt rekursive Figuren**
   (a) Schreiben Sie ein Programm mit zwei verschränkt rekursiven Funktionen `Quadrat` und `Dreieck`. Das Ergebnis soll Abbildung 4.5a ähneln.
   (b) Schreiben Sie noch ein Programm mit zwei verschränkt rekursiven Funktionen `Rechteck` und `Dreieck`. Das Ergebnis sollte Abbildung 4.5b ähneln.



a **b**

Abbildung 4.5: (a) Die Funktionen `Quadrat` und `Dreieck` rufen einander auf. **(b)** Die Funktionen `Dreieck` und `Rechteck` rufen einander auf.

2. **Einfache Figuren**
   Schreiben Sie ein Programm, das Sechsecke, Rechtecke und Dreiecke mit zufällig erzeugten Farben erzeugt. Die Figuren sollten möglichst innerhalb der Zeichenfläche liegen. Schreiben Sie das Programm so, dass die Funktion `Sechseck` die Funktion `Rechteck` aufruft und diese `Dreieck`. Letztere soll wiederum `Sechseck` aufrufen. Das Zeichnen soll nach einer bestimmten Zeit, etwa 25 Sekunden, angehalten werden.

Importieren sie dazu das Modul `time`. Die Zeit des Computer-Systems erhält man durch `time.time()`. Eine mögliche Lösung wird in Abbildung 4.6 dargestellt. Aber Ihre Lösung kann ganz anders aussehen.

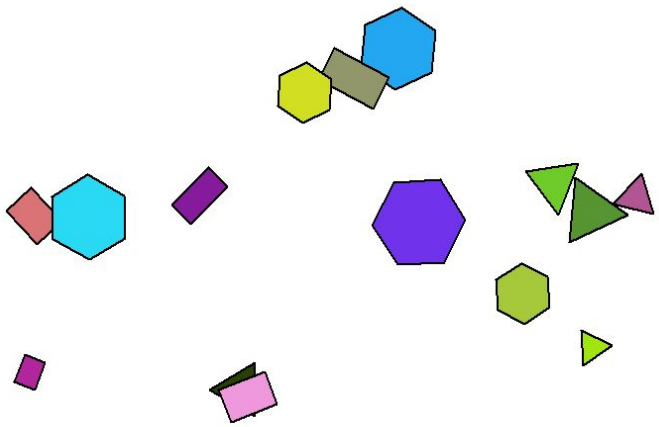


Abbildung 4.6: Eine mögliche Lösung von Übungsaufgabe 2

3. **Sierpiński-Kurven**
   Färben Sie die Buchten und Passagen, z.B. könnten die Buchten rot und die Passagen grün gezeichnet werden. Testen Sie `n=6`, und erhalten Sie ein nettes Muster.

4. **Wurzeln und Quadrate**
   Schreiben Sie ein Programm mit den verschränkt rekursiven Funktionen `Wurzel` und `Quadrat`. Dieses soll nach Eingabe einer Zahl `n` einen Term berechnen, der aus Wurzeln und Quadraten besteht. Zum Beispiel soll für `n=5` Folgendes herauskommen:

$$\sqrt{5+(4+\sqrt{(3+(2+1)^2)}\,^2} = 7.7918427$$

5. **Catalan-Reihe**
   Schreiben Sie ein Programm mit zwei verschränkt rekursiven Funktionen zur Berechnung von Teilsummen der Catalan-Reihe, definiert durch

$$\frac{1}{\sqrt{2}-1} - \frac{1}{\sqrt{2}+1} + \frac{1}{\sqrt{3}-1} - \frac{1}{\sqrt{3}+1} + \frac{1}{\sqrt{4}-1} - \frac{1}{\sqrt{4}+1} \cdots$$

   Strebt diese Reihe einem Grenzwert zu?

Anhang

Beenden der Turtlegrafik

Auch wenn es noch so viel Freude bereitet mit Turtlegrafik zu arbeiten, gibt es doch ein generelles Problem. Wenn ein Programm mit Turtlegrafik (Kap. 1 bis 4) beendet ist, lautet die nächste Programmzeile
`tu.write('fertig!',font = ("Arial",12,"normal"))`.
Wie aber erreicht man, dass die Zeichenfläche geschlossen wird und Python für neue Aufgaben bereit ist? Diese Frage kann nicht allgemein beantwortet werden. Sie hängt vom Betriebssystem des Computers ab, aber auch vom verwendeten Python-Editor.
Zwei häufig benutzte Editoren sind PyCharm [4] und Spyder [5].

Mit **PyCharm** ist die Sache ziemlich einfach. Fügen Sie unter der Zeile mit „fertig!“ folgende Zeilen an:

```
tu.exitonclick()
try:
    tu.bye()
except tu.Terminator:
    pass
```

Dann genügt ein Klick auf die Zeichenfläche der Turtlegrafik. Sie wird geschlossen, und das Programm kehrt zum Editor zurück.

Mit **Spyder** ist die Sache anders. In [82] werden die beiden nachstehenden Zeilen empfohlen:

```
turtle.done()
turtle.bye()
```

Mit einem Klick auf die Zeichenfläche ist es in diesem Falle nicht getan. Man muss auf das Kreuz zum Schließen des Fensters klicken. Dann wird die Zeichenfläche geschlossen. Aber was dann passiert, ist unterschiedlich und nicht vorhersagbar.
(1) Man kehrt zurück zur IPython-Konsole.

(2) Man erhält in der IPython-Konsole eine Fehlermeldung folgender Art:

```
runfile('D:/myPython/xyz.py', wdir='D:/myPython')
Traceback (most recent call last):
File D:\Anaconda3\Lib\site-packages\spyder_kernels\py3compat.py:356
in compat_exec
exec(code, globals, locals)
File d:\mypython\xyz.py:25
tu.done()
File <string>:5 in mainloop
Terminator
```

Das ist zwar ärgerlich, aber immerhin landet man wieder in der IPython-Konsole.

(3) Man erhält eine heftige Fehlermeldung, die hier nicht im einzelnen wiedergegeben werden soll. Sie beginnt mit

```
Tcl_AsyncDelete: async handler deleted by the wrong thread
Windows fatal exception: code 0x80000003
```

und endet mit

```
Restarting kernel... .
```

Nach [82] sollte man die Fehlermeldungen unterdrücken können, indem man diese Zeilen verwendet:

```
tu.done()
try:
    tu.bye()
except tu.Terminator:
    pass
```

Dies gelingt jedoch nicht immer.

Quellenverzeichnis

[1] Python 3.9 und neuere Versionen können heruntergeladen werden von https://www.python.org/downloads

[2] Das Konzept der Programmiersprache Logo wird kurz beschrieben in https://de.wikipedia.org/wiki/Logo_(Programmiersprache)

[3] https://docs.python.org/3/library/turtle.html

[4] https://www.jetbrains.com/de-de/pycharm/download/?section=windows

[5] https://www.spyder-ide.org/download/

[6] https://www.spektrum.de/wissen/nicole-oresme-1323-1382/1458053

[7] https://www.spektrum.de/kolumne/von-weltrekorden-zur-mysterioesen-euler-masch
2233498

[8] https://de.khanacademy.org/computing/computer-science/algorithms/recursive-algorithms/a/using-recursion-to-determine-whether-a-word-is-a-palindrome

[9] https://de.wikipedia.org/wiki/Euklidischer_Algorithmus

[10] https://www.aleph1.info/?call=Puc&permalink=analysis1_2_4_Z1

[11] https://studyflix.de/mathematik/fibonacci-folge-5511

[12] http://www.wilke-j.de/Python_3/fibonacci/

[13] https://de.wikipedia.org/wiki/Heron_von_Alexandria

[14] https://studyflix.de/mathematik/heron-verfahren-8229

[15] https://de.wikipedia.org/wiki/Catalan-Zahl

[16] https://www2.informatik.uni-stuttgart.de/fmi/fk/lehre/ss02/ti3/ThIII,SS01,CatalanscheZahlen,S_48.ppt.prn.pdf

[17] https://de.wikipedia.org/wiki/Nicht-kreuzende_Partition

[18] https://studyflix.de/statistik/binomialkoeffizient-einfach-erklaert-4115

[19] https://de.serlo.org/mathe/1587/binomialverteilungl

[20] https://vhm.mathematik.uni-stuttgart.de/Vorlesungen/Mathematische_Grundlagen/Folien_Binomialkoeffizient.pdf

[21] https://netmath.vcrp.de/downloads/Skripte/Schell/Mathe1/15-Determinanten/rechenregeln.html

[22] https://de.wikipedia.org/wiki/Determinante

[23] https://studyflix.de/mathematik/spatprodukt-2255

[24] https://de.wikipedia.org/wiki/Cramersche_Regel

[25] https://de.wikipedia.org/wiki/Satz_von_Abel-Ruffini

[26] https://www.lernhelfer.de/schuelerlexikon/mathematik/artikel/cardanische-formel

[27] https://imsc.uni-graz.at/baur/lehre/SS2013-LAK-Seminar/V1.pdf

[28] https://de.serlo.org/mathe/272069/vielfachheit-von-nullstellen

[29] https://de.wikipedia.org/wiki/Horner-Schema

[30] https://studyflix.de/mathematik/pascalsches-dreieck-2736

[31] https://de.wikipedia.org/wiki/Regula_falsi

[32] https://www.justmaththings.de/de/reference/TriangularMatrix

[33] https://de.wikipedia.org/wiki/Gau%C3%9Fsches_Eliminationsverfahren

[34] https://de.wikipedia.org/wiki/Drachenkurve Die dort entwickelte Drachenkurve ist der von abschnitt 3.1 sehr ähnlich, aber es ist nicht ganz die gleiche.

[35] https://de.wikipedia.org/wiki/Permutation

[36] https://de.wikipedia.org/wiki/Ackermannfunktion

[37] https://www.tcs.ifi.lmu.de/lehre/ss-2024/fsk_de/vl-10a-f-primitiv-und-my-rekursive-funktionen.pdf

[38] https://www.ruhr-uni-bochum.de/lmi/lehre/materialien/ti/vorlesung/ackermann.pdf

[39] https://www.tcs.ifi.lmu.de/lehre/ss-2024/fsk_de/vl-09c-f-die-ackermannfunktion.pdf

[40] https://studyflix.de/informatik/mergesort-1324

[41] https://mathematikalpha.de/turm-von-hanoi

[42] https://www.spektrum.de/kolumne/freistetters-formelwelt-die-apokalyptischen-1952467

[43] https://www.klinger.nrw/wp-content/uploads/2021/01/306_Tu%CC%88rme-von-Hanoi-2.pdf

[44] https://de.wikipedia.org/wiki/Rucksackproblem

[45] https://www.proggen.org/doku.php?id=algo:knapsack

[46] https://hwlang.de/algorithmen/np/pnp.htm

[47] https://www.pms.ifi.lmu.de/NFModule/TheoretischeInformatik/NPProbleme.pdf

[48] https://de.wikipedia.org/wiki/Backtracking

[49] https://www.gm.th-koeln.de/~hk/lehre/ala/ws0506/Praktikum/Projekt/C_blau/Backtracking_final.pdf

[50] https://www.westermann.de/landing/schmolke-deitermann/lineare-optimierung

[51] https://studyflix.de/mathematik/lineare-optimierung-318

[52] https://studyflix.de/mathematik/primaler-simplex-744

[53] https://www5.in.tum.de/lehre/vorlesungen/perlen2/ss2010/perlen_simplex.pdf

[54] https://de.wikipedia.org/wiki/Innere-Punkte-Verfahren

[55] aus OPTI2.dvi(uni-siegen.de), ist leider nicht mehr im Internet verfügbar

[56] https://de.wikipedia.org/wiki/Z-Kurve Die Hakenkurve ähnelt der Lebesgue-Kurve oder Z-Kurve.

[57] https://de.wikipedia.org/wiki/Koch-Kurve

[58] Leider wurde keine deutschsprachige Quelle gefunden.
http://www.modernscientificpress.com/Journals/ViewArticle.aspx?XBq7Uu+HD/8eRjFUGMqlRSziRNLeU2RixB4ndiJghEnTfTElvDRYUuwGofG1d+5T

[59] https://de.wikipedia.org/wiki/Hofstadter-Folge

[60] https://de.wikipedia.org/wiki/Heap-Algorithmus

[61] https://www.geeksforgeeks.org/heaps-algorithm-for-generating-permutations/

[62] https://www.uni-muenster.de/AMM/num/Vorlesungen/maurer/Uebungen09.pdf

[63] https://de.wikipedia.org/wiki/Collatz-Problem

[64] https://home.mathematik.uni-freiburg.de/didaktik/material_download/collatzproblem.pdf

[65] https://figshare.com/articles/journal_contribution/Beweis_der_Collatz-Vermutung/27822657?file=50577633

[66] https://vixra.org/pdf/2408.0100v1.pdf

[67] https://www.spiegel.de/wissenschaft/mensch/zahlenraetsel-mathematiker-zweifeln-am-beweis-der-collatz-vermutung-a-768289.html

[68] https://www.reddit.com/r/Collatz/comments/1b2vrhi/the_unofficial_proof_of_the_collatz_conjecture/

[69] https://mathprize.net/posts/collatz-conjecture/

[70] https://de.wikipedia.org/wiki/Piet_Mondrian

[71] https://blog.artsper.com/de/ein-naeherer-einblick-de/10-dinge-piet-mondrian/

[72] https://www.holland.com/de/tourist/suchen?keyword=Mondrian

[73] https://de.wikipedia.org/wiki/Piet_Mondrian#/media/File:Tableau_I,_by_Piet_Mondriaan.jpg

[74] https://de.wikipedia.org/wiki/Piet_Mondrian#/media/File:Piet_Mondriaan,_1939-1942_-_Composition_10.jpg

[75] https://de.wikipedia.org/wiki/Shakersort

[76] https://de.wikipedia.org/wiki/Bubblesort

[77] https://medical-dictionary.thefreedictionary.com/Fractal+curve

[78] https://de.wikipedia.org/wiki/Wac%C5%82aw_Sierpi%C5%84ski

[79] https://de.wikipedia.org/wiki/Sierpi%C5%84ski-Kurve

[80] https://de.wikipedia.org/wiki/Peano-Kurve

[81] https://www5.in.tum.de/lehre/seminare/oktal/SS03/ausarbeitungen/oberhofer.pdf

[82] https://github.com/spyder-ide/spyder/wiki/How-to-run-turtle-scripts-within-Spyder