Mentorierte Arbeit Fachdidaktik Informatik A (2 KP) und B (2KP)

Rekursive Algorithmen

Autor: Parisi Vincenzo

Datum: 22.12.2023

1 Inhaltsverzeichnis

1 Inhaltsverzeichnis 2
2 Konzeption 5
2.1 Vorwissen 5
2.2 Analyse des Stoffs 5
2.2.1 Definitionen 6
2.3 Ziele 7
2.3.1 Leitidee 7
2.3.2 Dispositionsziel 7
2.3.3 Operationalisierte Ziele 7
3 Einleitung 8
Aufgabe 1 8
Beispiel 1 11
Aufgabe 2 12
4 Rekursion einfach erklärt 13
Beispiel 2 13
Aufgabe 3 16
Aufgabe 4 16
4.1 Rekursive Formeln in der Mathematik 17
Beispiel 3 17
Aufgabe 5 18
Beispiel 4 18
Aufgabe 6 19
Aufgabe 7 19
Beispiel 5 20
Aufgabe 8 21
Beispiel 6 23
Aufgabe 9 24
4.2 Fibonacci-Folge 25
Aufgabe 10 25
Aufgabe 11 26
Aufgabe 12 26
4.2.1 Problemlösestrategie mit Rekursion 28
Aufgabe 13 29
Aufgabe 14 29
4.3 Vertiefung der Rekursion mit zwei rekursiven Aufrufen 30
Aufgabe 15 30
Aufgabe 16 32

4.3.1 Einführung Baumstruktur ... 35
Beispiel 7 ... 35
Aufgabe 17 ... 39
Aufgabe 18 ... 40
Aufgabe 19 ... 41
Aufgabe 20 ... 42
Aufgabe 21 ... 46
4.4 Speichernutzung beim Aufruf einer Funktion ... 51
Aufgabe 22 ... 52
Aufgabe 23 ... 54
Aufgabe 24 ... 54
5 Anwendungen ... 55
5.1 Der euklidische Algorithmus ... 55
5.1.1 Die Berechnung des ggT ... 55
Aufgabe 25 ... 56
Aufgabe 26 ... 57
Aufgabe 27 ... 58
Aufgabe 28 ... 58
5.2 Newton-Methode für die Berechnung von Nullstellen bei Polynomen ... 59
Beispiel 8 ... 60
Beispiel 9 ... 61
Aufgabe 29 ... 62
Aufgabe 30 ... 63
5.3 Transzendente Gleichungen mit Bisektion ... 64
Beispiel 10 ... 66
Analyse des Programmes: ... 68
Aufgabe 31 ... 70
Beispiel 11 ... 71
Beispiel 12 ... 72
Aufgabe 32 ... 75
5.4 Türme von Hanoi ... 76
Aufgabe 33 ... 81
Aufgabe 34 ... 82
5.5 Binärer Suchbaum ... 83
5.5.1 Navigieren ... 83
Aufgabe 35 ... 84
Aufgabe 36 ... 85
Aufgabe 37 ... 86

5.5.2 Suche in einem binären Suchbaum ... 87
Aufgabe 38 ... 87
5.6 Mergesort ... 88
5.6.1 Repetition des Algorithmus ... 88
Beispiel 13 ... 88
5.6.2 Algorithmus ... 90
Aufgabe 39 ... 91
Aufgabe 40 ... 91
Aufgabe 41 ... 93
Aufgabe 42 ... 93
6 Vor- und Nachteile einer Rekursion ... 95
7 Zusammenfassung ... 96
8 Kontrollfragen ... 97
9 Kontrollaufgaben ... 98
Kontrollaufgabe 1 ... 98
Kontrollaufgabe 2 ... 98
Kontrollaufgabe 3 ... 100
Kontrollaufgabe 4 ... 100
Kontrollaufgabe 5 ... 102
Kontrollaufgabe 6 ... 102
Kontrollaufgabe 7 ... 104
Kontrollaufgabe 8 ... 105
10 Abbildungsverzeichnis ... 108
11 Literaturverzeichnis ... 110

2 Konzeption

2.1 Vorwissen

Diese Unterrichtseinheit ist konzipiert für eine 12. Klasse des Gymnasiums.
Die Lernenden haben bereits vertiefte Programmiererfahrung und kennen verschiedene Datenstrukturen wie Listen, Arrays, Bäume, binäre Bäume und binäre Suchbäume. Anhand dieser Datenstrukturen wurden die Sortierverfahren wie Bubblesort, Selectionsort, Quicksort und Mergesort eingeführt.
Die Lernenden wurden darüber hinaus in die Grundlagen der objektorientierten Programmierung eingeführt und wissen, was eine Methode und was ein Attribut ist.
Gewisse Algorithmen wurden mittels Iteration gelöst, aber eine allgemeine Methode für die Lösung von Problemen wurde nicht direkt dargestellt. Somit ist noch kein Wissen hinsichtlich eines Algorithmus-Entwurfs vorhanden.
Im Mathematikunterricht wurde das Newton-Verfahren im Zusammenhang mit der Ableitung als praxisnahe Anwendung behandelt, deshalb wird es in dieser Unterrichtseinheit dies nicht nochmals erklärt, sondern es wird lediglich das Verfahren mittels Rekursion programmiert.

2.2 Analyse des Stoffs

Die Analyse der Fachliteratur führt zu der folgenden Abbildung:

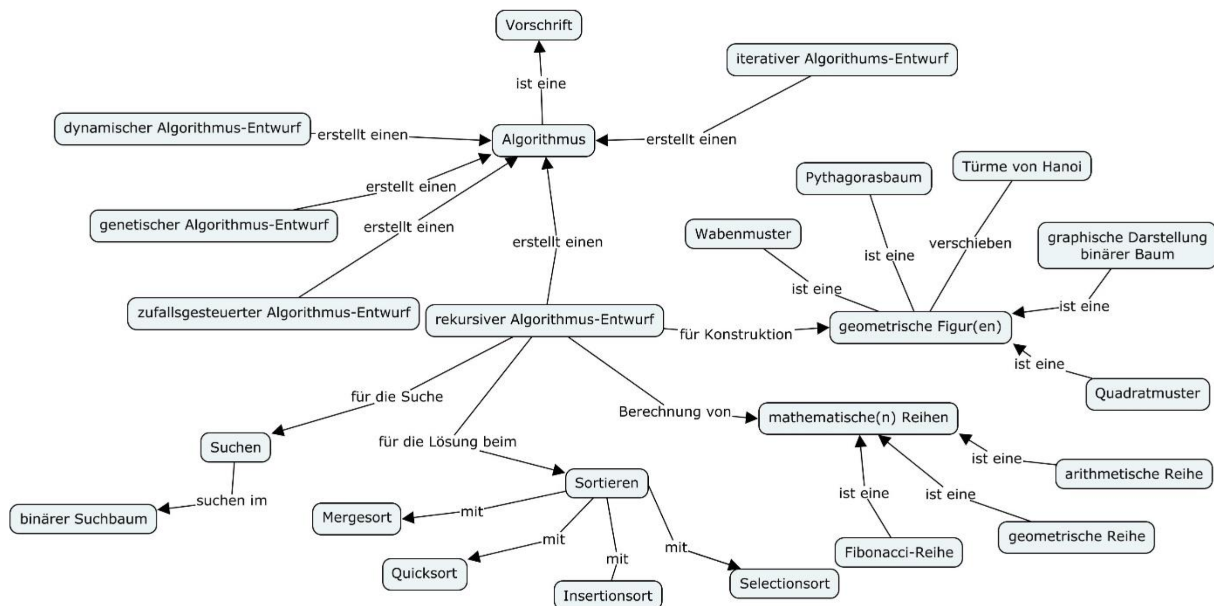


Abbildung 1: Concept Map

Die Focus-Frage lautet:
Welche Probleme eignen sich für rekursive Algorithmen?

Da der Fokus dieser Arbeit die Rekursion ist, wurde auch in der Concept Map nur der rekursive Algorithmus-Entwurf vollständig dargestellt. Die anderen Entwurfsmethoden sind nur erwähnt, aber nicht weiter unterteilt.

2.2.1 Definitionen

Problem

Ein Problem besteht aus einer Menge von Probleminstanzen (siehe unten) und beschreibt allgemein eine Relation zwischen einer Menge von Informationen (Eingang) und einer Lösungsmenge von neuen Informationen (Ausgabe). Sortiert man zum Beispiel eine Liste von Zahlen, so existieren dabei für dieses Problem mehrere Lösungen (verschiedene Sortierverfahren), unabhängig davon wie viele und welche Zahlen in der Liste sind.

Beispiele:

- Sortieren Sie eine Liste von beliebigen Zahlen:
  Eingabe: unsortierte Liste
  Ausgabe: sortierte Liste
  Welche Zahlen in der Liste sind oder ob die Zahlen vorsortiert sind, wird hier nicht definiert.
- Prüfen Sie, ob eine Zahl eine Primzahl ist:
  Eingabe: beliebige natürliche positive Zahl
  Ausgabe: ja oder nein
- Berechnen Sie die Quadratwurzel einer beliebigen positiven Zahl:
  Eingabe: natürliche positive Zahl
  Ausgabe: Quadratwurzel der Zahl
- Finden Sie den kürzesten Pfad zwischen beliebigen Orten auf einer Karte.
  Eingabe: Menge aller Orte
  Ausgabe: Menge aller kürzesten Pfade

Algorithmus

Ein Algorithmus ist eine Beschreibung einer Vorgehensweise, die aus Basisinstruktionen zusammengesetzt ist, und welche jede Eingabe eines Problems in die entsprechende Ausgabe transformiert. Der Algorithmus als eine Lösungsmethode beschreibt eindeutig, was zu tun ist, damit man die Lösung des Problems findet. Diese Methode ist vergleichbar mit den Anweisungen in einem Kochrezept.

Probleminstanz

Eine Probleminstanz ist eine konkrete Eingabe für ein Problem, welche eine bestimmte Ausgabe bestimmt.

Beispiele:

- Sortieren Sie folgende Wörter alphabetisch:
  Eingabe: [Haus, Auto, Baum, Tier, Mensch]
  Ausgabe: [Auto, Baum, Haus, Mensch, Tier]
- Berechnen Sie die Quadratwurzel von 256:
  Eingabe: 256
  Ausgabe: 16
- Primzahlprüfung:
  Eingabe: 15
  Ausgabe: nein
- Was ist der kürzeste Weg vom Schulzimmer zur Mensa.
  Eingabe: Karte mit den Wegen und der Position des Schulzimmers und der Mensa
  Ausgabe: kürzester Pfad Weg vom Schulzimmer zur Mensa

Problemgrösse
Die Grösse einer Probleminstanz wird Problemgrösse genannt, z.B. ist die Problemgrösse bei der Sortierung die Anzahl an Elementen oder bei einer Multiplikation die Anzahl an Operationen. Damit kann die Komplexität und die Laufzeit berechnet werden, was wir aber in dieser Arbeit nicht weiter vertiefen werden. Allgemein und vereinfacht könnte man sagen, je grösser die Problemgrösse ist, desto grösser ist die Komplexität und desto länger ist die Laufzeit.

2.3 Ziele

2.3.1 Leitidee

Viele Probleme können mit dem Top-down oder dem Bottom-up-Ansatz gelöst werden, indem eine Probleminstanz in mehrere kleinere Probleminstanzen aufgeteilt wird und dann die kleineren Probleminstanzen einzeln gelöst werden. Am Ende werden die Lösungen der kleineren Probleminstanzen wieder zusammengeführt, um das ursprüngliche Problem zu lösen. Diese Problemlösestrategie wird auch in der Mathematik und in der Informatik angewendet, um gewisse Probleme zu lösen. Die genannte Strategie möchten wir nun ein wenig abändern, und zwar so, dass es sich nicht nur um eine Reduktion der Problemgrösse (Grösse der Probleminstanz) handelt, sondern dass kleinere Probleminstanzen in gleicher Weise mit demselben Verfahren gelöst werden können. Dadurch entsteht die Rekursion.
Aus diesem Grund ergibt sich folgende **Leitidee**:
Die Methode der Rekursion soll auf verschiedene Problemstellungen angewendet werden und es soll darüber hinaus untersucht werden, welche Probleme nur oder anders gesagt wesentlich einfacher mittels Rekursion gelöst werden können.

2.3.2 Dispositionsziel

- Im Anschluss an diese Unterrichtseinheit werden die Lernenden die Problemlösestrategie mittels Rekursion in Betracht ziehen, dabei werden sie sich aufgrund des Problems für die geeignetste Lösungsstrategie entscheiden.
- Bevor sie direkt mit einer Lösungsvariante beginnen, überlegen sich die Lernenden, ob die iterative oder die rekursive Methode für die vorliegende Probleminstanz geeignet ist.

2.3.3 Operationalisierte Ziele

- Die Lernenden können den Unterschied zwischen einer iterativen und rekursiven Lösungsstrategie für einen Algorithmus-Entwurf erklären, ohne dabei die Unterlagen zu verwenden.
- Sie erkennen die Vor- und Nachteile einer Rekursion und können diese einer Kollegin oder einem Kollegen mit Beispielen wiedergeben, ohne dabei im Skript nachzulesen.
- Den Einsatz einer Rekursion bei der Lösung von verschiedenen Problemen können sie mit dem Begriff «divide et impera» begründen und können die Lösungsstrategie für eine Probleminstanz dieses Problems mit Hilfe eines Rezeptes herleiten.

3 Einleitung

In dem Spiel „Türme von Hanoi“, das der französische Mathematiker Edouard Lucas 1883 erfand, haben wir drei runde Untersetzer und einen Stapel Scheiben, welcher auf dem linken Untersetzer liegt. Die Aufgabe besteht nun darin, den Turm von dem linken auf den mittleren Untersetzer zu verschieben; als Puffer steht uns der rechte Untersetzer zur Verfügung (siehe Abbildung 2).

Die Regeln sind einfach:
• Es darf nur die oberste Scheibe eines Stapels bewegt werden.
• Es darf nie eine grössere Scheibe auf einer kleineren liegen.

Abbildung 2: Türme von Hanoi (Quelle: (Rimscha, 2017))

Die folgende Aufgabe dient der Motivation und soll eine Problemstellung darstellen, die wir im Verlauf der Unterrichtseinheit einführen werden.

Aufgabe 1

a) Überlegen Sie sich, wie Sie einen Turm der Höhe vier (also mit vier Scheiben) von der linken Seite in die Mitte unter Berücksichtigung der obigen Regeln verschieben können. Sie können das Online-Tool https://www.bernhard-gaul.de/spiele/tower/tower.php verwenden, um mit der Maus die Scheiben zu verschieben. Falls Sie nicht weiterkommen oder falls Sie Ihre Lösung prüfen möchten, können Sie den Lösungsknopf drücken.
b) Danach versuchen Sie es mit 10 Scheiben, ohne den Lösungsknopf zu verwenden!

Lösung:

a) Mit vier Scheiben ergeben sich folgende Schritte:
   - Scheibe von links nach rechts schieben
   - Scheibe von links in die Mitte schieben
   - Scheibe von rechts in die Mitte schieben
   - Scheibe von links nach rechts schieben
   - Scheibe von der Mitte nach Links schieben
   - Scheibe von der Mitte nach rechts schieben
   - Scheibe von links nach rechts schieben
   - Scheibe von links in die Mitte schieben
   - Scheibe von rechts in die Mitte schieben
   - Scheibe von rechts nach links schieben
   - Scheibe von der Mitte nach links schieben
   - Scheibe von rechts in die Mitte schieben
   - Scheibe von links nach rechts schieben

- Scheibe von links in die Mitte schieben
- Scheibe von rechts in die Mitte schieben

b) Mit 10 Scheiben oder mehr scheint es fast unmöglich zu sein. Der Versuch diese Probleminstanz zu lösen, sollte als Motivation dienen, um die Rekursion einzuführen, mit der wir versuchen werden, eine «einfache» Lösung zu finden. Im Laufe dieser Unterrichtseinheit werden wir eine Probleminstanz der «Türme von Hanoi» mit einem Algorithmus lösen. Zudem wird in dieser Aufgabe bereits gezeigt, dass, falls die Problemgrösse, d.h. die Anzahl Scheiben grösser wird, die Komplexität auch zunimmt.

Wir alle sind in der Lage, das Spiel regelkonform zu spielen und bei einer kleinen Anzahl Scheiben ist es sogar einfach lösbar. Wir möchten aber das Spiel mit einem Computer lösen. Mit einer iterativen Lösungsstrategie wird es schnell unüberschaubar und kompliziert. Deshalb benötigen wir ein neues Konzept, mit dem diese Probleminstanz einfacher und eleganter beschrieben und gelöst werden kann, sodass wir in der Lage sind den Algorithmus zu programmieren. Genau dieses neue Lösungskonzept wird in dieser Arbeit eingeführt.

Unser Ziel ist es, eine Problemlösungsstrategie zu entwickeln, welche eine Probleminstanz in kleinere Probleminstanzen desselben Problems zerlegt, sodass mit demselben Algorithmus diese kleineren Probleminstanzen gelöst werden können. Die Frage ist nun, was eine kleinere Probleminstanz bedeutet. Zur Klärung dieser Frage wird in Kapitel 2.2.1 der Begriff Problemgrösse eingeführt und beleuchtet, was die Grösse der Probleminstanz bedeutet. In diesem Spiel ist ein Mass für die Problemgrösse die Anzahl Scheiben, also werden wir für die kleinere Probleminstanz eine Scheibe weniger verwenden.
Dies wird wiederholt, bis wir die trivialste Probleminstanz haben, nämlich die Probleminstanz mit nur einer Scheibe. Darauf aufbauend kann nun die Probleminstanz mit zwei Scheiben gelöst werden, dann mit drei Scheiben, bis wir die Probleminstanz mit n Scheiben gelöst haben. Dies nennt man **Rekursion**. Die ausführliche Lösung der Probleminstanz «Türme von Hanoi» wird erst gegen Ende dieser Arbeit gezeigt. Das klingt kompliziert, aber wir werden in Beispiel 2 eine Anwendung aus dem Alltag verwenden, um die Rekursion einfacher zu erklären.

Jedoch bevor wir in das Thema eintauchen, beginnen wir mit einer kleinen Kindergeschichte aus dem Kindergarten auf Schweizerdeutsch:
Es isch emal en Maa gsi, dä hätt en hoole Zah gha. I däm hoole Zaa isch es Trückli gsi und i däm Trückli isch es Briefli gsi. I däm Briefli isch gschtande:
Es isch emal en Maa gsi, dä hätt en hoole Zah gha. I däm hoole Zaa isch es Trückli gsi und i däm Trückli isch es Briefli gsi. I däm Briefli isch gschtande:
Es isch emal en Maa gsi, dä hätt en hoole Zah gha. I däm hoole Zaa isch es Trückli gsi und i däm Trückli isch es Briefli gsi. I däm Briefli isch gschtande:

Übersetzt:
Es war einmal ein Mann, der hatte einen hohlen Zahn. In diesem Zahn war eine kleine Kiste und in dieser Kiste war ein Brief. Im Brief stand:
Es war einmal ein Mann, der hatte einen hohlen Zahn. In diesem Zahn war eine kleine Kiste und in dieser Kiste war ein Brief. Im Brief stand:
Es war einmal ein Mann, der hatte einen hohlen Zahn. In diesem Zahn war eine kleine Kiste und in dieser Kiste war ein Brief. Im Brief stand:

Wo ist hier die Rekursion?
Betrachtet man die Geschichte als Funktion, dann ist ersichtlich, dass sich die Geschichte selbst wieder aufruft. Somit haben wir eine Funktion, welche sich wieder aufruft, was einer Rekursion entspricht.

Was ist aber die Problemgrösse?
Für die Problemgrösse könnten wir die Anzahl Wörter der Teilgeschichte definieren, dann wird sofort klar, dass diese Grösse immer gleichbleibt und wir somit nie eine kleinere Probleminstanz für die Rekursion erhalten und auch nie die trivialste Probleminstanz finden werden. Kurz gesagt, die Funktion (die Geschichte) läuft unendlich lange, da die Rekursion nie beendet werden kann aufgrund der fehlenden trivialsten Probleminstanz, es gibt in dem Fall keine Abbruchbedingung.

Versuchen wir die Geschichte in ein Python-Programm umzuwandeln, dies könnte folgendes Programm sein.

```
def tellStory():
    print('''Es isch emal en Maa gsi, dä hätt en hoole Zah gha.
          I däm hoole Zaa isch es Trückli gsi und i däm Trückli
          isch es Briefli gsi. I däm Briefli isch gschtande:''')
    tellStory()

tellStory()
```

Wie aber bereits erwähnt, bricht das Programm nie ab, was zu einem «Stack Overflow» oder «maximum recursion depth exceeded» führt (probieren Sie es aus). Wir werden in Kapitel 4.4 «Speichernutzung beim Aufruf einer Funktion» auf diesen Fehler zurückkommen. Einfach erklärt: Das Programm läuft unendlich und es verbraucht Ressourcen, welche irgendwann aufgebraucht sind. Nebenbei entspricht dieser Fall einer endlosen Schleife, z.B. `while(True)`.

Dieses Konzept, dass die Funktion `tellStory()` sich selbst wieder aufruft, nennt man **Rekursion**. Hier wurden der Funktion noch keine Parameter mitgegeben und die Aufrufe werden nie terminiert. Also ist es eine **Rekursion ohne Parameter** und **ohne Abbruchbedingung**. Weiter fehlt das Verkleinern der Probleminstanz, was für jede Rekursion zwingend notwendig ist.

Ein weiterer interessanter Punkt ist: Falls man die Ausgabe des Textes mit dem rekursiven Aufruf der Funktion vertauscht, dann wird nichts auf die Konsole geschrieben, da sich die Funktion unendlich viele Male aufruft und nie die Ausgabe-Anweisung (`print`) ausführen kann.

```
def tellStory():
    #Zuerst der Aufruf, dann print
    tellStory()

    print('''Es isch emal en Maa gsi, dä hätt en hoole Zah gha.
          I däm hoole Zaa isch es Trückli gsi und i däm Trückli
          isch es Briefli gsi. I däm Briefli isch gschtande:''')


tellStory()
```

Kennen Sie diesen Effekt?

Abbildung 3: Rekursion als Bild

(Quelle: https://upload.wikimedia.org/wikipedia/commons/6/69/Droste_1260359-nevit%2C_corrected.jpg)

In der Abbildung 3 sitzt das Mädchen mit einem Spiegel vor einem anderen Spiegel, das Spiegelbild wird wieder im Spiegel gespiegelt. Sei das Bild eine Funktion, so ruft sich das Bild wieder selbst auf. Auch hier hat man aus programmtechnischer Sicht eine **Rekursion ohne Parameter**, welche auch endlos läuft. Natürlich möchten wir keine endlos laufenden Programme haben, resp. wir möchten jeweils auf kleinere Probleminstanzen schliessen, bis wir die trivialste erreicht haben. Die obigen zwei Beispiele sind nur als Erläuterung für den Begriff Rekursion gedacht.

Im folgenden Beispiel wird dieser Effekt als Programm geschrieben, aber wir werden es nicht endlos laufen lassen, sondern brechen nach einer bestimmten Abbruchbedingung ab.

Beispiel 1

Um trotzdem den Effekt in Python zu zeigen, wird ein Spiegel gezeichnet, welcher eine Figur (Stern) spiegelt, die Spiegelung der Figur wird jedes Mal 40% kleiner, bis kein erkennbares Bild mehr gezeichnet werden kann, d.h. in unserem Fall, bis die Seitenlänge des Spiegels kleiner als 5 ist, was hier der Abbruchbedingung entspricht. Somit läuft das Programm nicht unendlich lange.
Betrachten wir nun auch hier die Problemgrösse, so könnten wir sie mit der Seitenlänge oder der Bildfläche definieren. An dieser Stelle muss man aber aufpassen, da die kleinere Probleminstanz, d.h. die Probleminstanz mit der kleineren Problemgrösse nicht auch zu einer kleineren Komplexität führt. Aber wie zu Beginn erwähnt, werden wir das Thema Komplexität nicht vertiefen.

```
from gpanel import *

side = 80
offset = 20

def drawImage():
    global side, offset
    setColor('forestgreen')
    fillRectangle(side, side)
    setColor('black')
    rectangle(side, side)
    pt1 = [-side/2, -(side/2 - offset)]
    pt2 = [side/2, -(side/2 - offset)]
    pt3 = [0, side/2]
    pt4 = [-side/2, (side/2 - offset)]
    pt5 = [side/2, (side/2 - offset)]
    pt6 = [0, -side/2]
    setColor('darkorange')
    fillTriangle([pt1, pt2, pt3])
    fillTriangle([pt4, pt5, pt6])
    side = 0.6 * side
    offset = 0.6 * offset
    if side < 5:
        return
    drawImage()

makeGPanel(-50, 50, -50, 50)

lineWidth(5)
drawImage()
```

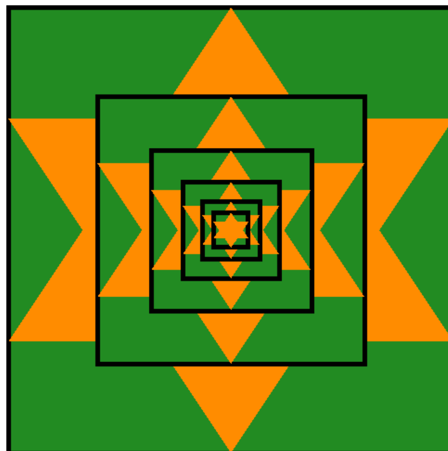


Abbildung 4: Spiegeleffekt mit Python

Aufgabe 2

Überlegen Sie sich, was geschieht, falls die letzte Zeile (`drawImage()`) am Anfang der Funktion wäre. Starten Sie dann das Programm **nicht**.

Lösung:

Es ergibt sich ein endlos laufendes Programm, da die Bedingung `side < 5` nie getestet wird.

4 Rekursion einfach erklärt

Beispiel 2

Stellen wir uns folgende mögliche Alltagssituation vor:
Stellen Sie sich vor, Sie sind auf einer Party und Sie kommen zufällig an der Küche vorbei. In der Küche liegt ein Haufen Teller im Waschbecken, welcher abgewaschen werden soll. Da kommt jemand auf Sie zu und sagt: «Könntest du bitte die Teller abwaschen?». Aus Höflichkeit hilft man, jedoch geht niemand zu einer Party, um abzuwaschen. Deshalb übergeben Sie die Aufgabe, nachdem Sie einen Teller abgewaschen haben, an eine andere Person. Diese macht dasselbe, wäscht einen Teller ab und sucht sich wieder eine andere Person, welche weitermachen soll, bis irgendwann kein Teller mehr im Waschbecken ist.

Diese Tätigkeit als Pseudocode könnte so aussehen:

```
Erledige den Abwasch:
      wenn das Spülbecken leer ist:
           ist nichts zu tun.
      sonst:
           spüle einen Teller und
           finde die nächste Person und
           sage ihr/ihm: Erledige den Abwasch
```

Die Aufrufsequenz stellen wir nun für drei Teller graphisch dar.

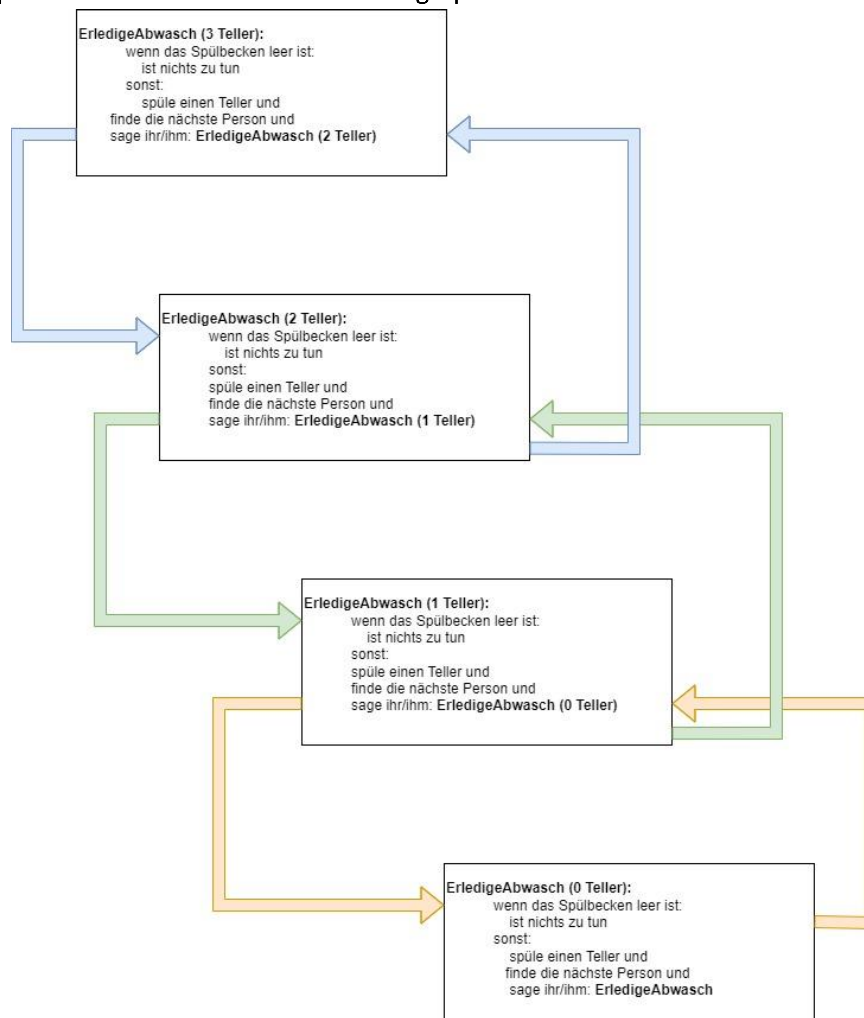


Abbildung 5: Ablauf Teller waschen

Im Gegensatz zur Geschichte auf Schweizerdeutsch und zu dem Spiegelbild terminiert diese Aufgabe, da es nur endlich viele Teller zum Abwaschen gibt.
Was aus der Abbildung 5 weiter ersichtlich ist, ist, dass der Berg an schmutzigen Tellern immer kleiner wird, d.h., die Probleminstanz wird immer kleiner, bis zuletzt kein Teller mehr abgewaschen werden soll, was der trivialsten Probleminstanz entspricht.
An dieser Stelle überlegen wir uns ein Rezept, welches das obige Vorgehen beschreibt:

1. *Beschreiben Sie die Probleminstanz und ermitteln Sie die Problemgrösse N.*
   Wir möchten einen Haufen mit n Tellern abwaschen. Nehmen wir an, dass die Funktion `ErledigeAbwasch(n)` heisst. Somit liegt eine Probleminstanz der Grösse **N = n** vor.
2. *Ermitteln und lösen Sie die trivialste Probleminstanz.*
   Die trivialste Probleminstanz ist, falls das Spülbecken leer ist oder $n = 0$, was zur Problemgrösse $\boldsymbol{N = 0}$ führt, also `ErledigeAbwasch(0)`, dann ist nämlich nichts zu tun.
3. *Reduzieren Sie die Probleminstanz in eine oder mehrere kleinere Probleminstanzen . (divide)*
   Wir reduzieren die Probleminstanz der Grösse $N = n$ zur kleineren Probleminstanz mit der Grösse $N_{red} = (n - 1)$, dies bedeutet, dass bei jedem Durchgang die Menge der Teller um eins kleiner wird.
4. *Lösen Sie die kleinere Probleminstanz desselben Problems mit demselben Algorithmus, d.h., verwenden Sie dazu die Rekursion, solange Sie nicht die trivialste Probleminstanz erreicht haben, welche einfach bestimmt werden kann. (impera)*
   Die kleinere Probleminstanz kann mit `ErledigeAbwasch(n-1)` gelöst werden, wobei diese Probleminstanz mit demselben Algorithmus gelöst werden kann, bis die trivialste Probleminstanz vorliegt. In unserem Beispiel bedeutet dies, dass jeweils pro Reduktion ein Teller weniger im Waschbecken ist und der Algorithmus wird beendet, wenn das Waschbecken leer ist (trivialste Probleminstanz).

Im obigen Beispiel haben wir die Probleminstanz in kleinere Probleminstanzen aufgeteilt, in der lateinischen Sprache heisst dies **divide** (engl.: devide, deutsch: teile), und das Lösen der kleineren Probleminstanzen könnte man als Eroberung oder Herrschen über die anderen Instanzen bezeichnen, was in der lateinischen Sprache **impera** (engl.: conquer, deutsch: herrsche) bedeutet. Somit wird dieses Vorgehen auch als «**divide et impera**» bezeichnet (engl.: devide and conquer, deutsch: teile und herrsche).

«Divide et impera» hat den Ursprung im Römischen Reich, damals wurde das Römische Reich in kleinere Staaten aufgeteilt («divide») und jeder Staat wurde von einem Mann regiert («impera»), welcher mit Rom einen Vertrag hatte. Die unterschiedlichen Staaten hatten verschiedene Wertigkeiten und man konnte sich als Staat hocharbeiten bis zum Freund des römischen Volkes, was für den Regierenden eine Gleichstellung zu einem Römer bedeutete, somit damals die höchste Anerkennung war.

Kehren wir zurück zu unserem Teller- waschen-Beispiel:
Im ersten Beispiel (Spiegeleffekt) wurde die rekursive Funktion ohne Parameter geschrieben, nun haben wir eine gewisse Anzahl Teller, welche wir dem Programm mitgeben möchten, deshalb schreiben wir nun die Funktion mit einem Parameter nämlich mit der Anzahl Teller beim Start.
Um eine bessere Übersicht zu erreichen, wird das Waschbecken im untenstehenden Programm als Rechteck und die Teller werden als hellgraue Kreise gezeichnet. Nach jedem Durchgang wird das Waschbecken mit zufälligen neuen Positionen der Teller neu gezeichnet.

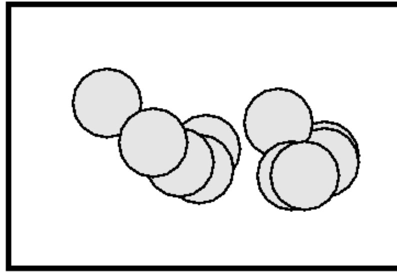


Abbildung 6: Waschbecken beim Start

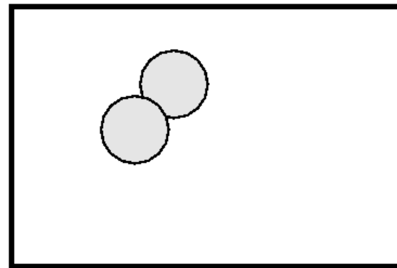


Abbildung 7: Waschbecken fast leer

Die obigen Bilder wurden mit folgendem Programm erstellt.

```
from gpanel import *
import random

sink_x1 = 10
sink_y1 = 20
sink_x2 = 70
sink_y2 = 60
plate_r = 5

def drawPlates(N):
    #clear
    clear()
    lineWidth(5)
    #sink
    rectangle(sink_x1, sink_y1, sink_x2, sink_y2)
    #plates
    for i in range(N):
        plate_x = random.randint(sink_x1 + 10, sink_x2 - 10)
        plate_y = random.randint(sink_y1 + 10, sink_y2 - 10)
        move(plate_x, plate_y)
        setColor('black')
        circle(plate_r)
        setColor('gray90')
        fillCircle(plate_r)
    setColor('black')
    setStatusText('Es hat noch ' + str(N-1) + ' Teller.')
    if N == 0:
        Font('Arial', Font.BOLD, 16)
        text(sink_x1+2, sink_y2+5, 'Das Spülbecken ist leer.',
Font('Arial', Font.BOLD, 16), 'black', 'white')
        setStatusText('Es gibt keine Teller mehr. Abwasch ist
fertig.')
```

```
def tellerWaschen(N):
    drawPlates(N)
    if N == 0:
        return
    delay(500)
    tellerWaschen(N-1)

makeGPanel(0, 100, 0, 100)
addStatusBar(50)
tellerWaschen(10)
```

Dank der Bedingung (Abbruchbedingung) in der Funktion **`tellerWaschen(..)`** und dem folgenden **`return`**

```
if N == 0:
    return
```

wird die Funktion terminiert, sobald das Waschbecken leer ist, und läuft nicht unendlich lange. Die Anzahl der Teller beim Start wurde als Parameter mitgegeben, wir können sagen, dass es sich um eine **Rekursion mit Parameter** handelt.

Aufgabe 3

Testen Sie das obige Programm aus, indem Sie das Programm in TigerJython kopieren und starten.

Aufgabe 4

Schreiben Sie den Code aus Beispiel 1 mit dem Spiegeleffekt als eine Rekursion mit Parameter um. Es soll die Seitenlänge des Quadrates und ein Offset mitgegeben werden. Bei einer Seitenlänge kleiner 5 soll die Rekursion beendet werden.

Lösung:

```
from gpanel import *

side = 80
offset = 20

def drawImage(p_side, p_offset):
    if p_side < 5:
        return
    setColor('forestgreen')
    fillRectangle(p_side, p_side)
    setColor('black')
    rectangle(p_side, p_side)
    pt1 = [-p_side/2, -(p_side/2 - p_offset)]
    pt2 = [p_side/2, -(p_side/2 - p_offset)]
    pt3 = [0, p_side/2]
    pt4 = [-p_side/2, (p_side/2 - p_offset)]
    pt5 = [p_side/2, (p_side/2 - p_offset)]
    pt6 = [0, -p_side/2]
    setColor('darkorange')
    fillTriangle([pt1, pt2, pt3])
    fillTriangle([pt4, pt5, pt6])
```

```
    p_side = 0.6 * p_side
    p_offset = 0.6 * p_offset
    drawImage(p_side, p_offset)

makeGPanel(-50, 50, -50, 50)

lineWidth(5)
drawImage(side, offset)
```

4.1 Rekursive Formeln in der Mathematik

In der Mathematik lassen sich verschiedene Formeln mit rekursiver Definition sehr einfach beschreiben und herleiten. Meistens ist die Findung der direkten Formel zu umständlich, weshalb man eine rekursive Definition verwendet.

Beispiel 3
Wir möchten die Summe der Zahlen von 1 bis N berechnen. Falls $N = 10$ wäre, könnte man $1 + 2 + .. + 10$ in den Taschenrechner eingeben und wir erhalten das Resultat. Bei 1000 Zahlen wird es aber aufwendiger sein. Natürlich existiert dafür eine geschlossene Formel. Aber wir möchten als Beispiel eine rekursive Definition herleiten.

Summe der ersten Zahl

$$summe_1 = 1$$

Summe der ersten 2 Zahlen

$$summe_2 = 1 + 2 = 3$$

Summe der ersten 3 Zahlen

$$summe_3 = 1 + 2 + 3 = 6$$

Summe der ersten 4 Zahlen

$$summe_4 = 1 + 2 + 3 + 4 = 10$$

usw.

Bis jetzt sind wir aber nicht weiter als die Eingabe mit dem Taschenrechner. Wir beenden die Berechnungen hier und überlegen uns eine rekursive Lösung.

Nochmals:

$$summe_2 = 1 + 2 = 2 + 1 = 2 + summe_1 = 3$$
$$summe_3 = 1 + 2 + 3 = 3 + 2 + 1 = 3 + summe_2 = 6$$
$$summe_4 = 1 + 2 + 3 + 4 = 4 + 3 + 2 + 1 = 4 + summe_3 = 10$$

Die Rekursion entspricht hier der Verwendung der zuvor bereits berechneten Summe.

Wir verallgemeinern die obige Berechnung zu,

$$summe_i = i + summe_{i-1}$$

wobei *i* mit dem grössten Index startet, hier startet die Berechnung mit `i = 4`.

Es fehlt aber noch die Abbruchbedingung, die besagt, wann wir die Summe beenden werden?

Dies entspricht der einfachsten Berechnung, in unserem Fall, wenn **`i == 1`** ist, dann ist das Resultat gleich 1.

Nun werden die obigen Schritte in das Rezept eingetragen:

1. *Beschreiben Sie die Probleminstanz und ermitteln Sie die Problemgrösse N.*
   Es soll die Summe der Zahlen von 1 bis $n = 1000$ berechnet werden. Daraus ergibt sich eine Problemgrösse $N = n = 1000$, was der Anzahl Zahlen entspricht.
2. *Ermitteln und lösen Sie die trivialste Probleminstanz (N = 1).*
   Die trivialste Probleminstanz ist: Falls die Anzahl Zahlen $n = 1$ ist und die Problemgrösse $N = 1$ ist, dann ist die $Summe = 1$.
3. *Reduzieren Sie die Probleminstanz in eine oder mehrere kleinere Probleminstanzen. (divide)*
   Die kleinere Probleminstanz ergibt sich, falls man jeweils nur die Summe der Zahlen von 1 bis $(n-1)$ berechnen möchte, die neue Problemgrösse ist in diesem Fall $N_{red} = (n-1)$. Mit der Summe von 1 bis $(n-1)$ Zahlen kann dann die Summe von 1 bis n Zahlen gemäss der Formel
   $$summe_n = n + summe_{n-1}$$
   berechnet werden.
4. *Lösen Sie die kleinere Probleminstanz desselben Problems mit demselben Algorithmus, d.h., verwenden Sie dazu die Rekursion, solange Sie nicht die trivialste Probleminstanz erreicht haben, welche einfach bestimmt werden kann. (impera)*
   Die Berechnung der Summe der kleineren Probleminstanz kann mit derselben Formel gelöst werden. Es sei $k = (n-1)$, dann folgt mit: $summe_k = k + summe_{k-1}$ die Lösung der kleineren Probleminstanz. Die analoge Formel kann nun auch verwendet werden, um die $summe_{k-1}$ zu berechnen, bis wir die trivialste Probleminstanz haben. Mit den Teilergebnissen beginnend bei der Lösung der trivialsten Probleminstanz kann nun die ursprüngliche Probleminstanz mit der Grösse $N = n$ gelöst werden.

Aufgabe 5

Schreiben Sie eine rekursive Funktion **`summe(n)`**, welche die Summe aller Zahlen von 1 bis n berechnet.

Lösung:

```
def summe(a):
    if a == 1:
        return 1
    else:
        return a + summe(a-1)

print(summe(10))
```

Beispiel 4

In diesem Schritt möchten wir das obige Beispiel verallgemeinern. Vorhin war das Inkrement zwischen den Zahlen eins, nun sollen sich die Zahlen um *a* unterscheiden. Z.B. ist $a = 3$, dann würden wir die Summe der Zahlen $1 + 4 + 7 + 10 + 13 + \cdots$ bilden.

Dafür gehen wir analog zum vorherigen Beispiel vor:
Summe der ersten Zahl

$$summe_1 = 1$$

Summe der ersten 2 Zahlen

$$summe_2 = 1 + 4 = 4 + 1 = 4 + summe_1 = 5$$

Summe der ersten 3 Zahlen

$$summe_3 = 1 + 4 + 7 = 7 + 4 + 1 == 7 + summe_2 = 12$$

Summe der ersten 4 Zahlen

$$summe_4 = 1 + 4 + 7 + 10 = 10 + 7 + 4 + 1 = 10 + summe_3 = 22$$

Es stellt sich die Frage, wie man mit dem Index *i* die neu zu addierende Zahl berechnen kann, also wie man bei **`i = 4`** ($summe_4$) den Wert 10 berechnen kann.

Aufgabe 6

Suchen Sie einen mathematischen Zusammenhang zwischen dem Index *i* und der neu zu addierenden Zahl, also den Zusammenhang zwischen dem Index $i = 4$ und dem $Summanden = 10$ in Beispiel 4. Dabei sollten Sie das Inkrement *a* (in unserem Bsp. *a = 3*) benutzen. Versuchen Sie die Zahlen rekursiv zu berechnen.

Lösung:
Aus Index 1 => 1
Index 2 => 4
Index 3 = 7
Index 4 = 10

D.h., der neue Summand ist immer um 3 grösser resp. um *a* grösser als der vorherige.
Daraus schliessen wir, dass der neue Summand S_i folgendermassen berechnet werden kann:

$$S_1 = 1$$
$$S_i = S_{i-1} + a \text{ für } i \geq 2$$

Aufgabe 7

a) Wie lautet nun die rekursive Definition der obigen Summe?
b) Wann brechen Sie die Berechnung ab? Zu beachten ist, dass z.B. die $summe_{10}$ gesucht wird und das Programm mit dem Parameter 10 startet.

Lösung:

a)

$$summe_1 = 1$$
$$S_1 = 1 = summe_1$$

$$summe_i = S_i + summe_{i-1} = S_{i-1} + a + summe_{i-1} \quad \text{für } i \geq 2$$

b) Die Berechnung kann abgebrochen werden, sobald der Index **`i = 1`** ist.

Beispiel 5

In der letzten Aufgabe haben wir uns Gedanken gemacht, wie man rekursiv die Summe von Zahlen, welche jeweils um *a* erhöht werden, berechnen kann. In diesem Beispiel werden wir in Python eine rekursive Funktion dafür schreiben. Für diesen Zweck analysieren wir die Aufgabe.

Analyse:

Ziel: Wir möchten eine rekursive Funktion für die Berechnung der Summe schreiben.

Für die Berechnung der Summe benötigen wir zwei rekursive Berechnungen, die vom ersten Summanden S_i und die der Summe $summe_i$. Beide Formeln haben wir in den zwei vorhergehenden Aufgaben bereits gefunden, sodass wir beide Ergebnisse zusammenfügen können.

$$summe_i = S_i + summe_{i-1}$$

Damit ergibt sich folgender Code:

```
def Summand(Anzahl, incA):
    if Anzahl == 1:
        #Summand 1
        return 1
    else:
        return incA + Summand(Anzahl - 1, incA)

def SummeA(Anzahl, incA):
    if Anzahl == 1:
        #Summe 1
        return 1
    else:
        return Summand(Anzahl, incA) + SummeA(Anzahl - 1, incA)

print(SummeA(4, 3))
```

Grobe Überlegungen zur Laufzeit

Falls wir die Aufgabe so lösen, dann wird zuerst die Funktion für S_i aufgerufen, bis der entsprechende Wert berechnet ist und danach die Summe der Zahlen auch wieder rekursiv ist.

Nehmen wir an, wir möchten, wie im obigen Code, die Summe der ersten 4 Glieder der arithmetischen Folge berechnen, dann ergibt sich folgende Aufrufsequenz:

```
SummeA(4,3)
→Summand(4, 3)
        → Summand(3, 3)
                → Summand(2, 3)
                        → Summand(1, 3)
→SummeA(3,3)
        → Summand(3, 3)
                → Summand(2, 3)
                        → Summand(1, 3)
        →SummeA(2,3)
                → Summand(2, 3)
                        → Summand(1, 3)
                →SummeA(1,3)
                        → Summand(1, 3)
```

Daraus ist ersichtlich, dass diese Berechnung nicht geeignet ist, da für den Summanden S_i mehrfach dieselbe Berechnung ausgeführt wird. Aus diesem Grund analysieren wir die Berechnung nochmals, um eine effizientere rekursive Funktion zu erhalten.

Wir betrachten nochmals die Summenbildung, das Inkrement *a* ist auch wiederum gleich 3, aber wir ordnen die Teilsummen anders an:
Summe der ersten Zahl (Startwert)

$$summe_1 = 1$$

Summe der ersten 2 Zahlen

$$summe_2 = 1 + 4 = 1 + 1 + 3 = summe_1 + summe_1 + a = 5$$

Summe der ersten 3 Zahlen

$$summe_3 = 1 + 4 + 7 = 5 + 1 + 3 + 3 = summe_2 + summe_1 + a + a = 12$$

Summe der ersten 4 Zahlen

$$summe_4 = 1 + 4 + 7 + 10 = 12 + 1 + 3 + 3 + 3 = summe_3 + summe_1 + a + a + a = 22$$

Summe der ersten 5 Zahlen

$$summe_5 = 1 + 4 + 7 + 10 + 13 = 22 + 1 + 3 + 3 + 3 + 3$$
$$= summe_4 + summe_1 + a + a + a + a = 35$$

Die rekursive Formel dazu:

$$summe_i = summe_1 + summe_{i-1} + (i - 1) * a$$

Aufgabe 8

Schreiben Sie eine rekursive Funktion **`SummeA(Start, Anzahl, incA)`** für die neu hergeleitete Summe, der Startwert ($summe_1$) soll als Parameter mitgegeben werden.
Testen Sie ihre Lösung für : **`SummeA(1, 4, 3)`** und erstellen Sie dafür eine Aufrufsequenz.

Lösung:

```
def SummeA(Start, Anzahl, incA):
    if Anzahl == 1:
        #Summe 1
        return Start
    else:
        return SummeA(Start, Anzahl - 1, incA) + \
                   Start + (Anzahl - 1) * incA

print(SummeA(1, 4, 3))
```

Bemerkung:
Mit Backslash (\) kann der Befehl auf der nächsten Zeile weitergeschrieben werden.

Aufrufsequenz:
SummeA(1, 4, 3)
- ➔ SummeA(1, 3, 3) + 1 + 3 * 3
  - ➔ SummeA(1, 2, 3) + 1 + 2 * 3
    - ➔ SummeA(1, 1, 3) + 1 + 1 * 3

Das sieht schon sehr viel besser aus!

Als Abschluss des Beispiels werden wir hier das dazugehörende Rezept aufschreiben, wobei im Normalfall das Rezept zu Beginn der Programmierung erstellt wird und die obigen Überlegungen sich aus den Antworten der einzelnen Rezeptpunkte ergeben.

1. *Beschreiben Sie die Probleminstanz und ermitteln Sie die Problemgrösse N.*
   Es soll die Summe der ersten $n = 4$ Zahlen der Folge mit $Startwert = 1$ und mit dem Inkrement $a = 3$ berechnet werden. Dies ergibt die Problemgrösse $N = 4$, da in der Aufgabe die Summe von $n = 4$ Zahlen berechnet wird.
2. *Ermitteln und lösen Sie die trivialste Probleminstanz (N = 1).*
   Die trivialste Probleminstanz mit der Grösse $N = 1$ ist: Falls wir nur eine Zahl (den Startwert) summieren möchten, dann ist das Resultat der Startwert.
3. *Reduzieren Sie die Probleminstanz in eine oder mehrere kleinere Probleminstanzen. (divide)*
   Die kleinere Probleminstanz ergibt sich, falls man jeweils nur die Summe der Zahlen von 1 bis $(n-1)$ mit Inkrement $a = 3$ berechnen möchte, die neue Problemgrösse ist in diesem Fall $N_{red} = (n-1)$. Mit dieser Summe kann nun die Summe von 1 bis n Zahlen gemäss der Formel
   $$summe_n = summe_1 + summe_{n-1} + (n-1) * a$$
   berechnet werden.
4. *Lösen Sie die kleinere Probleminstanz desselben Problems mit demselben Algorithmus, d.h., verwenden Sie dazu die Rekursion, solange Sie nicht die trivialste Probleminstanz erreicht haben, welche einfach bestimmt werden kann. (impera)*
   Die Berechnung der Summe der kleineren Probleminstanz kann mit derselben Formel gelöst werden. Es sei $k = (n-1)$, dann folgt mit:
   $$summe_k = summe_1 + summe_{k-1} + (k-1) * a$$
   die Lösung der kleineren Probleminstanz. Die analoge Formel kann nun auch verwendet werden um $summe_{k-1}$ zu berechnen, bis wir die trivialste Probleminstanz haben. Mit den Teilergebnissen beginnend bei der Lösung der trivialsten Probleminstanz kann nun die ursprüngliche Probleminstanz mit der Grösse $N = n$ berechnet werden.

Beispiel 6

Stellen wir uns vor, wir hätten ein Startkapital K_0 = 10'000.-CHF auf einer Bank bei einem Zinssatz $p = 2\%$. Eine Bank möchte das Kapital nach $n = 10$ Jahren berechnen.
Bemerkung: Bei dieser Aufgabe vernachlässigen wir die Bankgebühren und Steuern.

Als ersten Schritt gehen wir gemäss unserem Rezept vor.

1. *Beschreiben Sie die Probleminstanz und ermitteln Sie die Problemgrösse N*.
   Es soll das Kapital eines Startkapitals von K_0 = 10'000.-CHF nach $n = 10$ Jahren berechnet werden. Als Problemgrösse kann die Anzahl Jahre verwendet werden, also ist $N = 10$.
2. *Ermitteln und lösen Sie die trivialste Probleminstanz (N = 1).*
   Die trivialste Probleminstanz ist das Kapital nach einem Jahr, die Problemgrösse ist somit $N = 1$ und es berechnet sich mit: $K_1 = K_0 \cdot (1 + p) = K_0 * 1.02$
3. *Reduzieren Sie die Probleminstanz in eine oder mehrere kleinere Probleminstanzen. (divide)*
   Dazu erstellen wir einige Berechnungen:
   K_0 = 10'000 CHF und p = 0.02
   Es ist mit folgendem Kapital in den ersten vier Jahren zu rechnen:
   1. Jahr: $K_1 = K_0$*p = 10'000 CHF * 1.02 = 10'200 CHF
   2. Jahr: $K_2 = K_1$*p = 10'200 CHF * 1.02 = 10'404 CHF
   3. Jahr: $K_3 = K_2$*p = 10'404 CHF * 1.02 = 10'612.08 CHF
   4. Jahr: $K_4 = K_3$*p = 10'612.08 CHF * 1.02 = 10'824.32 CHF
   oder allgemein:
   $$K_n = K_{n-1} \cdot (1 + p)$$
   Die kleinere Probleminstanz ist das Kapital nach $(n - 1)$ Jahren zu berechnen, dann erhalten wir eine kleinere Problemgrösse $N_{red} = (n - 1)$.
4. *Lösen Sie die kleinere Probleminstanz desselben Problems mit demselben Algorithmus, d.h., verwenden Sie dazu die Rekursion, solange Sie nicht die trivialste Probleminstanz erreicht haben, welche einfach bestimmt werden kann. (impera)*
   Die kleinere Probleminstanz ist das Kapital nach $(n - 1)$ Jahren. Mit diesem Kapital können wir das Kapital nach n Jahren berechnen.
   $$K_n = K_{n-1} \cdot (1 + p)$$
   Die kleinere Probleminstanz kann analog berechnet werden wie die Probleminstanz am Anfang, es sei hier $k = (n - 1)$, dann gilt:
   $$K_k = K_{k-1} \cdot (1 + p)$$
   Dies können wir nun für alle $k = (n - 1) \dots 2$ mit demselben Algorithmus berechnen. Mit all diesen Teilergebnissen und der Lösung der trivialsten Probleminstanz können wir die ursprüngliche Probleminstanz berechnen.

Wir haben in Punkt 4 des Rezeptes gesehen, wie man die kleineren Probleminstanzen lösen kann, nämlich indem dieselbe Formel wieder angewendet wird, $K_k = K_{k-1} \cdot (1 + p)$ und K_k ist das Kapital nach k Jahren.
Die Funktion in Python sieht dann folgendermassen aus:

```
def Zinseszinskapital(K0, p, n):
    if n == 1:
        return K0 * (1 + p)

    return (1 + p) * Zinseszinskapital (K0, p, n-1) #(1+p)*Kn-1
```

Das ist das gesamte Programm:

```
#Zinseszins

def Zinseszinskapital (K0, p, n):
    if n == 1:
        return K0 * (1 + p) #Kapital nach einem Jahr

    return (1 + p) * Zinseszinskapital (K0, p, n-1)

print(Zinseszinskapital (10000, 0.02, 10))
```

Aufgabe 9

Berechnen Sie mit der analogen Vorgehensweise wie im letzten Beispiel die Fakultät einer ganzen Zahl *n = 20* (*n! = 20!*). Beginnen Sie mit dem Erstellen des Rezeptes, um die Rekursion herzuleiten. Am Ende erstellen Sie ein Python-Programm für diese Berechnung.

Lösung:

1. *Beschreiben Sie die Probleminstanz und ermitteln Sie die Problemgrösse N*.
   Wir möchten die Fakultät einer ganzen Zahl *n = 20* (**`fakultaet(n)`**) berechnen. Somit liegt eine Problemgrösse $N = 20$ vor.
2. *Ermitteln und lösen Sie die trivialste Probleminstanz (N = 1).*
   Die trivialste Probleminstanz ist: Falls **n=1** ist, also **`fakultaet(1)`**, ist das Ergebnis 1.
3. *Reduzieren Sie die Probleminstanz in eine oder mehrere kleinere Probleminstanzen*. *(divide)*
   Die Fakultät ist folgendermassen definiert:
   $$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$$
   Für die ersten Zahlen gilt:
   $$1! = 1$$
   $$2! = 1 \cdot 2 = 2 \cdot \mathbf{1}!$$
   $$3! = 1 \cdot 2 \cdot 3 = 3 \cdot \mathbf{2}! = 6$$
   $$4! = 1 \cdot 2 \cdot 3 \cdot 4 = 4 \cdot \mathbf{3}! = 24$$
   $$5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 5 \cdot \mathbf{4}! = 120$$

   Dies können wir umformen zu:
   $$n! = n \cdot (n-1)!$$

   So können wir die kleinere Probleminstanz mit derselben Formel lösen, es sei wiederum $k = (n-1)$, dann gilt:
   $$k! = k \cdot (k-1)!$$
4. *Lösen Sie die kleinere Probleminstanz desselben Problems mit demselben Algorithmus, d.h., verwenden Sie dazu die Rekursion, solange Sie nicht die trivialste Probleminstanz erreicht haben, welche einfach bestimmt werden kann*. *(impera)*
   Die kleinere Probleminstanz kann wieder reduziert werden und wir erhalten für jeden Reduktionsschritt:
   $$k! = k \cdot (k-1)!\ für\ k = (n-1) \dots 2$$
   Mit diesen Teilergebnissen und dem Ergebnis der trivialsten Probleminstanz kann nun leicht die ursprüngliche Probleminstanz ausgerechnet werden.

Der letzte Schritt ist, aus dem Rezept eine Python-Funktion zu erzeugen:

```
def fakultaet(n):
    #trivialste Probleminstanz
    if n == 1:
        return 1
    else:
        #Aufteilung in eine kleinere Probleminstanz
        return n*fakultaet(n-1)
```

4.2 Fibonacci-Folge

Aufgabe 10

In dieser Aufgabe betrachten wir einen Kaninchenstall mit einem einzigen Kaninchenpärchen. In unserem vereinfachten Model legen wir fest, dass Kaninchen nie sterben und jedes Kaninchenpaar nach einem Monat geschlechtsreif ist und dann jeden Monat ein weiteres Kaninchenpärchen wirft. Das erste Kaninchenpärchen wurde Ende Januar geboren. Wir wollen die Population der Kaninchen in unserem Stall tabellarisch darstellen.

Diese Abfolge kann als Baum dargestellt werden, wie in der Abbildung 8, jede Tiefe des Baumes entspricht einem Monat.

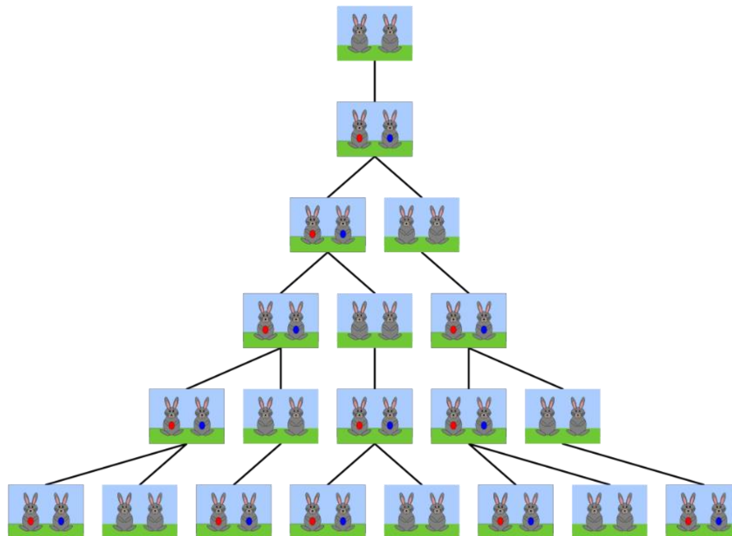


Abbildung 8: Kaninchenpopulation (Quelle: https://erasmus-reinhold-gymnasium.de/info/rekursion/fibonacci.html)

Ergänzen Sie folgende Tabelle mit der Anzahl Kaninchenpärchen bis und mit zum Monat September.

Monat	Jan.	Feb.	März	Apr.	Mai	Jun.	Jul.	Aug.	Sept.
Kaninchenpärchen	1	1	2						

Lösung

Monat	Jan.	Feb.	März	Apr.	Mai	Jun.	Jul.	Aug.	Sept.
Kaninchenpärchen	1	1	2	3	5	8	13	21	34

März: Das erste Kaninchenpärchen wirft das zweite Kaninchenpärchen.
April: Das erste Kaninchenpärchen wirft das dritte Pärchen. Das zweite Kaninchenpärchen ist noch nicht geschlechtsreif.
Mai: Sowohl das erste wie auch das zweite Pärchen werfen. Das dritte Pärchen ist noch nicht bereit.
Juni: Das erste, zweite und dritte Pärchen werfen.
… usw.

Aufgabe 11

Zeichnen Sie zwei nebeneinanderliegende Quadrate der Grösse 1 auf ein Blatt, auf diesen aufbauend ein drittes Quadrat der Grösse 2, danach zeichnen Sie wieder ein Quadrat, welches an der längeren Seite der vorherigen Figur anliegt. Wiederholen Sie den Vorgang sechsmal.

Nun schreiben Sie die erhaltenen Seitenlängen als Folge auf, was fällt ihnen auf?

Lösung

Abbildung 9: Fibonacci-Quadrat

Die einzelnen Zahlen entsprechen der Fibonacci-Folge: 1, 1, 2, 3, 5, 8, 13, 21

Aufgabe 12

Wie Sie festgestellt haben, handelt es sich in Aufgabe 10 und Aufgabe 1 um die gleiche Zahlenfolge. Leonardo da Pisa, später auch Fibonacci genannt, entdeckte sie, als er die Vermehrung von Kaninchen unter die Lupe nahm. Diese Zahlenfolge wird ihm zu Ehren als Fibonacci-Folge bezeichnet. Versuchen Sie die in den vorangehenden Aufgaben gefundene Gesetzmässigkeit der Fibonacci-Folge mit einer rekursiven Definition zu formulieren.

Lösung:

Die obigen Zahlen werden wie folgt berechnet:

$$
\begin{aligned}
&1 \\
&1 \\
2 &= 1 + 1 \\
3 &= 2 + 1 \\
5 &= 3 + 2 \\
8 &= 5 + 3 \\
13 &= 8 + 5 \\
21 &= 13 + 8 \\
&\vdots
\end{aligned}
$$

Es startet mit:

$$fibonacci(1) = 1$$
$$fibonacci(2) = 1$$

Die Zahl zwei, 3. Element in der Folge, ist die Summe der zwei Vorgänger:

$$fibonacci(3) = 2 = 1 + 1 = fibonacci(2) + fibonacci(1)$$

Dann folgt das 4. Element:

$$fibonacci(4) = 3 = 2 + 1 = fibonacci(3) + fibonacci(2)$$

Das 5. Element folgt:

$$fibonacci(5) = 5 = 3 + 2 = fibonacci(4) + fibonacci(3)$$

Nun ist ersichtlich, dass die allgemeine Formel folgendermassen abgeleitet werden kann:

$$
\begin{gathered}
fibonacci(1) = 1 \\
fibonacci(2) = 1 \\
fibonacci(3) = fibonacci(2) + fibonacci(1) \\
\vdots \\
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
\end{gathered}
$$

Bemerkung: Fibonacci-Folgen in der Natur
Die Fibonacci-Folge ist in vielen natürlichen Phänomenen zu beobachten, so zum Beispiel in der Anordnung von Blättern an Pflanzen oder in der Spirale von Schnecken- und Muschelgehäusen:

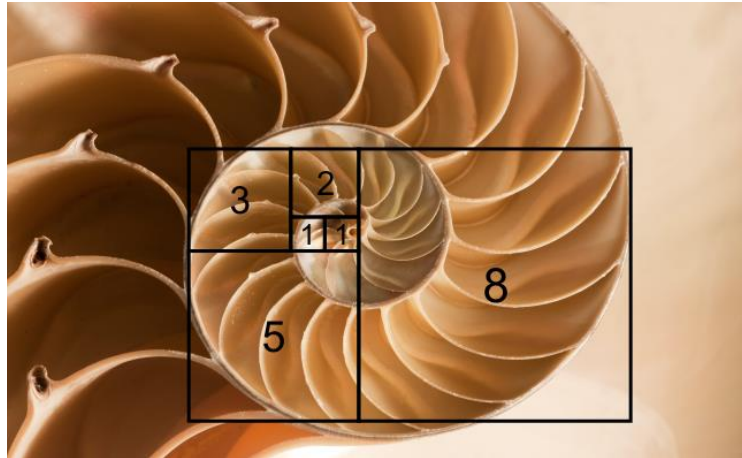


Abbildung 10: Muschelgehäuse

4.2.1 Problemlösestrategie mit Rekursion

Wir wollen nun die in Aufgabe 10, in Aufgabe 11 und in Aufgabe 12 erarbeiteten Lösungen verallgemeinern. Dazu verwenden wir das oben bereits eingesetzte Rezept, mit dem wir viele ähnliche Aufgabenstellungen lösen können.
Angenommen wir möchten die siebte Fibonacci-Zahl berechnen, dann gehen wir, wie auch in den vorherigen Aufgaben, folgendermassen vor:
Bemerkung:
Damit die Herleitung übersichtlicher bleibt, werden die Fibonacci-Zahlen mit F_n bezeichnet, dies entspricht der n-ten Fibonacci-Zahl also fibonacci(n).

1. Um die siebte Fibonacci-Zahl F_7 zu berechnen, benötigen wir F_6 und die F_5, also zwei Vorgänger.
2. Dasselbe gilt für die Fibonacci-Zahlen F_6 und F_5. Für F_6 benötigen wir F_5 und F_4 usw.
3. Die Berechnung endet, falls wir bei F_1 und F_2 angekommen sind. Diese zwei Zahlen sind vorgegeben: $F_1 = 1$ und $F_2 = 1$
4. Nun können wir aus allen Teilschritten F_7 aus den entsprechenden Zahlen berechnen.

Aus dieser Herleitung können wir ein allgemeines Vorgehen in Form des folgenden Rezeptes aufschreiben.

Rezept:
1. *Beschreiben Sie die Probleminstanz und ermitteln Sie die Problemgrösse N*.
2. *Ermitteln und lösen Sie die trivialste(n) Probleminstanz(en) (N = 1, …).*
3. *Reduzieren Sie die Probleminstanz in eine oder mehrere kleinere Probleminstanzen*. *(divide)*
4. *Lösen Sie die kleinere Probleminstanz desselben Problems mit demselben Algorithmus, d.h., verwenden Sie dazu die Rekursion, solange Sie nicht die trivialste Probleminstanz erreicht haben, welche einfach bestimmt werden kann*. *(impera)*

Aufgabe 13

Entwickeln Sie ein Programm zur Berechnung der 10. Fibonacci-Zahl mittels Rekursion.
Beantworten Sie dazu folgende Punkte des Rezeptes, welche vorhin hergeleitet wurden.

Lösung

1. *Beschreiben Sie die Probleminstanz und ermitteln Sie die Problemgrösse N*.
   Wir möchten die 10. Fibonacci-Zahl berechnen, somit ist die Problemgrösse $N = 10$.
2. *Ermitteln und lösen Sie die trivialste(n) Probleminstanz(en) (N = 1, …).*
   Die Lösung der trivialsten Probleminstanzen ist die Berechnung der 1. und 2. Fibonacci-Zahl:
   $$f_1 = 1, f_2 = 1$$
   Daraus kann auch die Abbruchbedingung abgeleitet werden.
3. *Reduzieren Sie die Probleminstanz in eine oder mehrere kleinere Probleminstanzen*. *(divide)*
   Die kleineren Probleminstanzen ergeben sich aus der Formel:
   $$f_n = f_{n-1} + f_{n-2}$$
   Somit können wir nur noch die Probleminstanzen f_{n-1} und f_{n-2} mit der Problemgrösse $N_{red1} = (n-1)$ und $N_{red2} = (n-2)$ betrachten, mit deren Lösung man f_n berechnen kann.
4. *Lösen Sie die kleineren Probleminstanzen desselben Problems mit demselben Algorithmus, d.h., verwenden Sie dazu die Rekursion, solange Sie nicht die trivialste Probleminstanz erreicht haben, welche einfach bestimmt werden kann*. *(impera)*
   Beide kleineren Probleminstanzen können mit derselben Formel berechnet werden:
   $$f_k = f_{k-1} + f_{k-2}\ je\ f\text{ür}\ k = (n-1)\ und\ k = (n-2)$$
   Nun können wir denselben Algorithmus anwenden, um die nächstkleineren Probleminstanzen, d.h. für $k = (n-2)\ und\ k = (n-3)$, zu berechnen, usw., bis die trivialste Probleminstanz erreicht wird. Am Ende ist es nun möglich, mit der Lösung der trivialsten Probleminstanz und den Teillösungen die die Probleminstanz am Anfang zu lösen.

Aufgabe 14

Schreiben Sie einen Pseudocode, welcher die n-te Fibonacci-Zahl berechnet. Die Eingabe ist eine ganze Zahl n (z.B. $n = 10$) und daraus soll die n-te Fibonacci-Zahl berechnet und zurückgegeben werden.
Die Folge beginnt mit **`a1 = 1, a2 = 1, a3 = 2, …, an`**; *an* ist die n-te Zahl.
Die Lösung soll einmal **iterativ** (mit Schleifen) und einmal **rekursiv** (wie vorhin gezeigt) sein.

Lösung

Iterative Lösung

```
int fibonacci(n)
      a1 = 1
      a2 = 1
      wiederhole (n-1) Mal:
            an = a1 + a2 #Vorvorgänger + Vorgänger
            a1 = a2
            a2 = an
      return an
```

Für die rekursive Lösung wird die Rekursion der Lösung der Aufgabe 12 verwendet.

Rekursive Lösung

```
int fibonacci(n)
      falls n <= 2
            return 1
      sonst
            #Vorgänger + Vorvorgänger
            return fibonacci(n-1) + fibonacci(n-2)
```

4.3 Vertiefung der Rekursion mit zwei rekursiven Aufrufen

Im vorherigen Kapitel haben wir die Fibonacci-Reihe mittels Rekursion gelöst. Das Besondere daran war, dass wir in der Funktion `fibonacci(…)` zweimal die Funktion wieder aufgerufen haben. In diesem Kapitel werden wir weitere Beispiele anschauen, welche auch zwei oder mehrere rekursive Aufrufe enthalten. Beginnen wir gleich mit einer Aufgabe.

Aufgabe 15

Teilen Sie ein DIN A4 Blatt in 8 Streifen gleicher Breite. Beschreiben Sie, wie Sie vorgegangen sind.

Lösung:

Es gibt mehrere Möglichkeiten eine davon ist die folgende:
Man teilt das Blatt in der Hälfte, legt die erste Hälfte zur linken Seite und die zweite Hälfte wird wieder geteilt. Nun haben wir ein Viertel des Blattes, deshalb wiederholen wir den Vorgang, um danach zwei Teile von je einem Achtel des Blattes vorliegen zu haben.
Dasselbe wiederholt man mit allen Teilen des Stapels. Das nächste Teilblatt ist ein Viertel des Blattes. Deshalb wird es nur einmal geteilt. Das letzte Blatt halber Grösse wird halbiert und die eine Hälfte auf den Stapel gelegt. Die andere Hälfte wird wieder halbiert. Und zum Schluss wird das vorherige Blatt mit der Grösse ein Viertel genommen und halbiert. Nun haben wir die acht Streifen.

Zum besseren Verständnis wird das Vorgehen mit dem Stapel als Graphik in den Abbildungen 11 bis 17 dargestellt.

1. Das Blatt wird halbiert

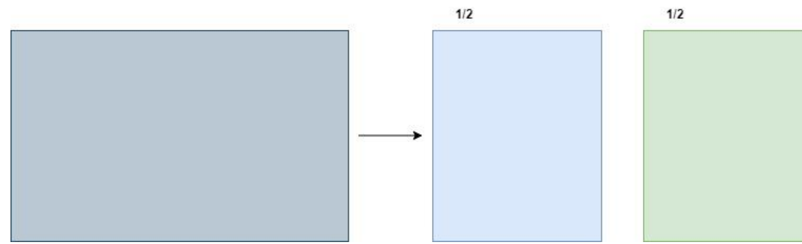


Abbildung 11: Schritt 1 der Teilung

2. Die eine Hälfte geht auf den Stapel, die andere wird wieder halbiert.

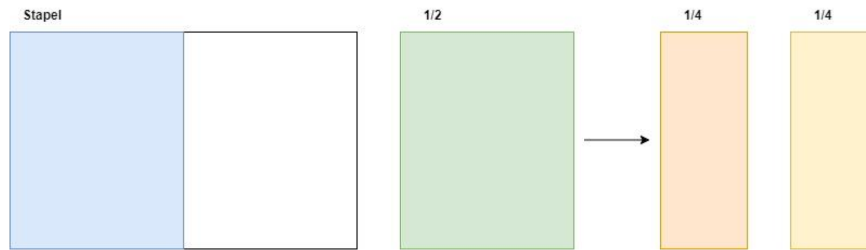


Abbildung 12: Schritt 2 der Teilung

3. Die eine Hälfte der Hälfte wird auf den Stapel gelegt, die andere wird wieder halbiert.

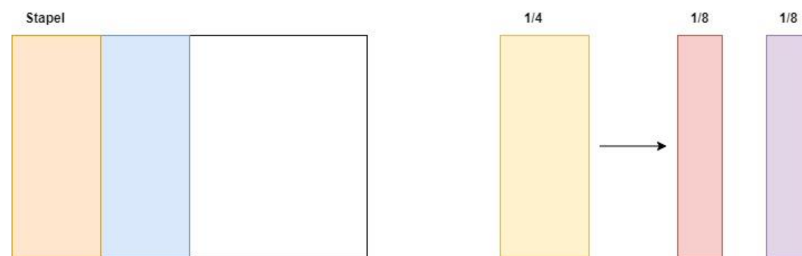


Abbildung 13: Schritt 3 der Teilung

4. Nun wird das oberste Blatt mit der Grösse ein Viertel vom Stapel genommen und wieder halbiert.

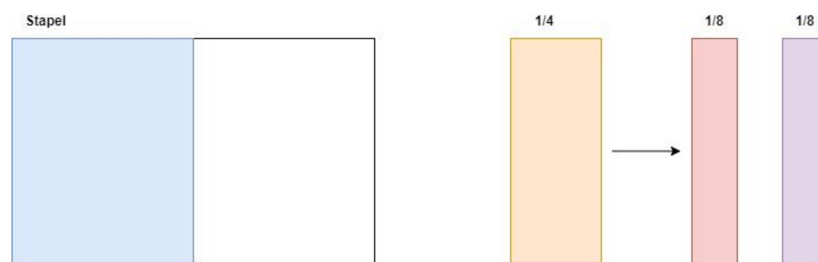


Abbildung 14: Schritt 4 der Teilung

5. Das letzte Blatt hat die Grösse ein Halb. Dieses wird halbiert und die eine Hälfte auf den Stapel gelegt.

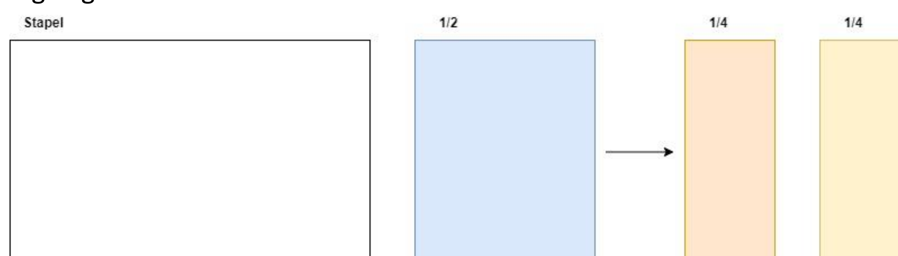


Abbildung 15: Schritt 5 der Teilung

6. Die eine Hälfte wird halbiert.

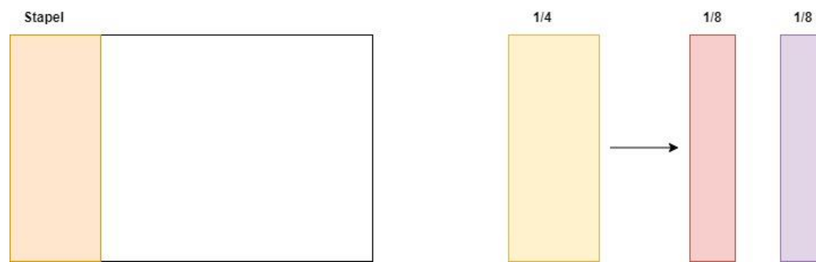


Abbildung 16: Schritt 6 der Teilung

7. Das letzte Blatt des Stapels wird geteilt

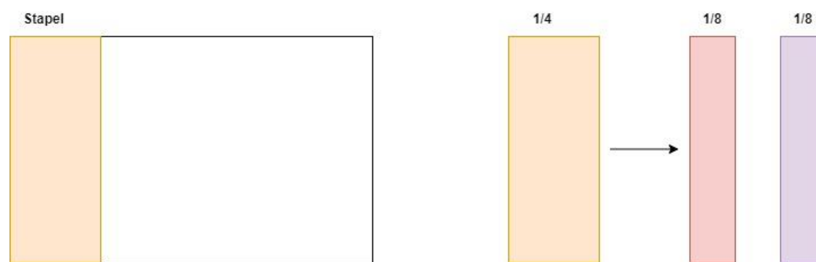


Abbildung 17: Schritt 7 der Teilung

Aus den Abbildungen 11 bis 17 ist die Rekursion nicht klar ersichtlich, sondern lediglich die Lösung von Hand.

Analysieren wir nun dieses Vorgehen:

1. Zuerst wird das Blatt halbiert, die eine Hälfte wird auf den Stapel gelegt, die andere wird weiterverarbeitet. Dies könnte eine Funktion `blattTeilen(anzStreifen)` erledigen, wobei `anzStreifen` die Anzahl Streifen sind, welche wir erstellen möchten.
2. Mit diesem halbierten Blatt erledigen wir dasselbe wie vorhin. Also verwenden wir für diese Hälfte des Blattes wieder die Funktion `blattTeilen(anzStreifen/2)`. Diesmal aber nur mit der Hälfte der Anzahl Streifen, da wir nur ein halbes Blatt haben.
3. Der Vorgang wird wiederholt, solange wir die gewünschte Teilung erreicht haben.

Betrachten wir nochmals den zweiten Punkt. Während wir mit der Funktion `blattTeilen(…)` das Blatt halbieren, benutzen wir wieder die Funktion `blattTeilen(…)`, um die eine Hälfte zu halbieren. Das ist die Rekursion, welche vorhin nicht ersichtlich war.

Aufgabe 16

Erstellen Sie für die rekursiven Aufrufe der Funktion `blattTeilen(8)`, falls wir das Blatt in acht Streifen teilen möchten, eine Aufrufsequenz für den unter Kapitel «Analysieren wir nun das Vorgehen» beschriebenen Ablauf. Beenden Sie die Skizze, sobald ein einziger Streifen der Grösse ein Achtel vorliegt.

Bemerkung:

Beachten Sie weiter, dass in der Beschreibung jeweils nur die linke Hälfte wieder halbiert wird und deshalb die Aufgabe noch nicht fertig ist.

Lösung

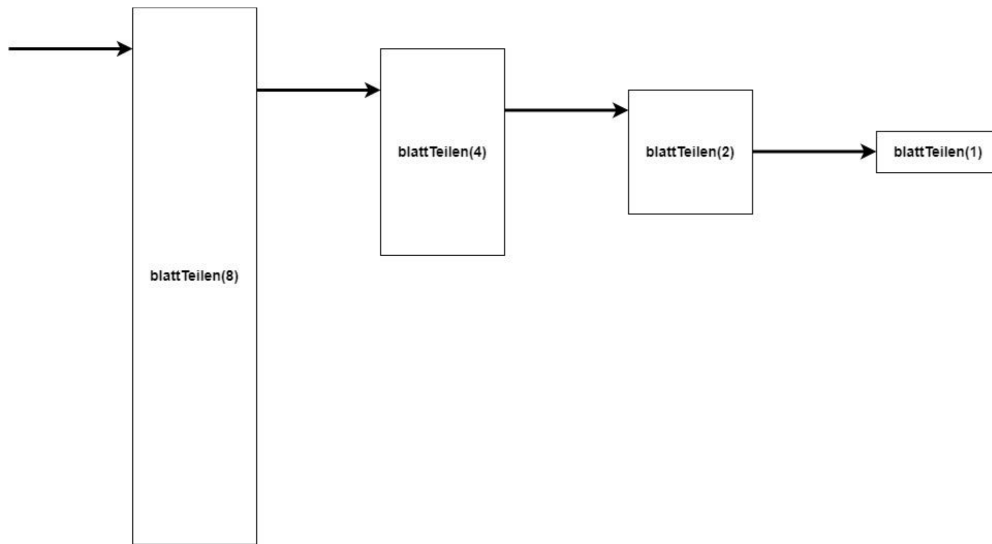


Abbildung 18: Ablauf gemäss der Analyse

Die Lösung in Abbildung 18 zeigt, dass bis jetzt nur ein Streifen erstellt worden ist, was auch in der Beschreibung so geschildert war. Um die Aufgabe zu vervollständigen, müssen wir Folgendes beachten:
Der Aufruf **`blattTeilen(1)`** erstellt nur den Streifen für die linke Hälfte. Sobald dieser Aufruf fertig ist, müssen wir im Aufruf **`blattTeilen(2)`** nochmals **`blattTeilen(1)`** für den rechten Teil aufrufen. Dies gilt wiederum auch für **`blattTeilen(4)`**, auch hier muss die Funktion **`blattTeilen(2)`** für die rechte Hälfte nochmals aufgerufen werden usw.

Dieser Sachverhalt wird nun in der Lösung der Aufgabe 16 ergänzt und wir erhalten folgendes Bild.

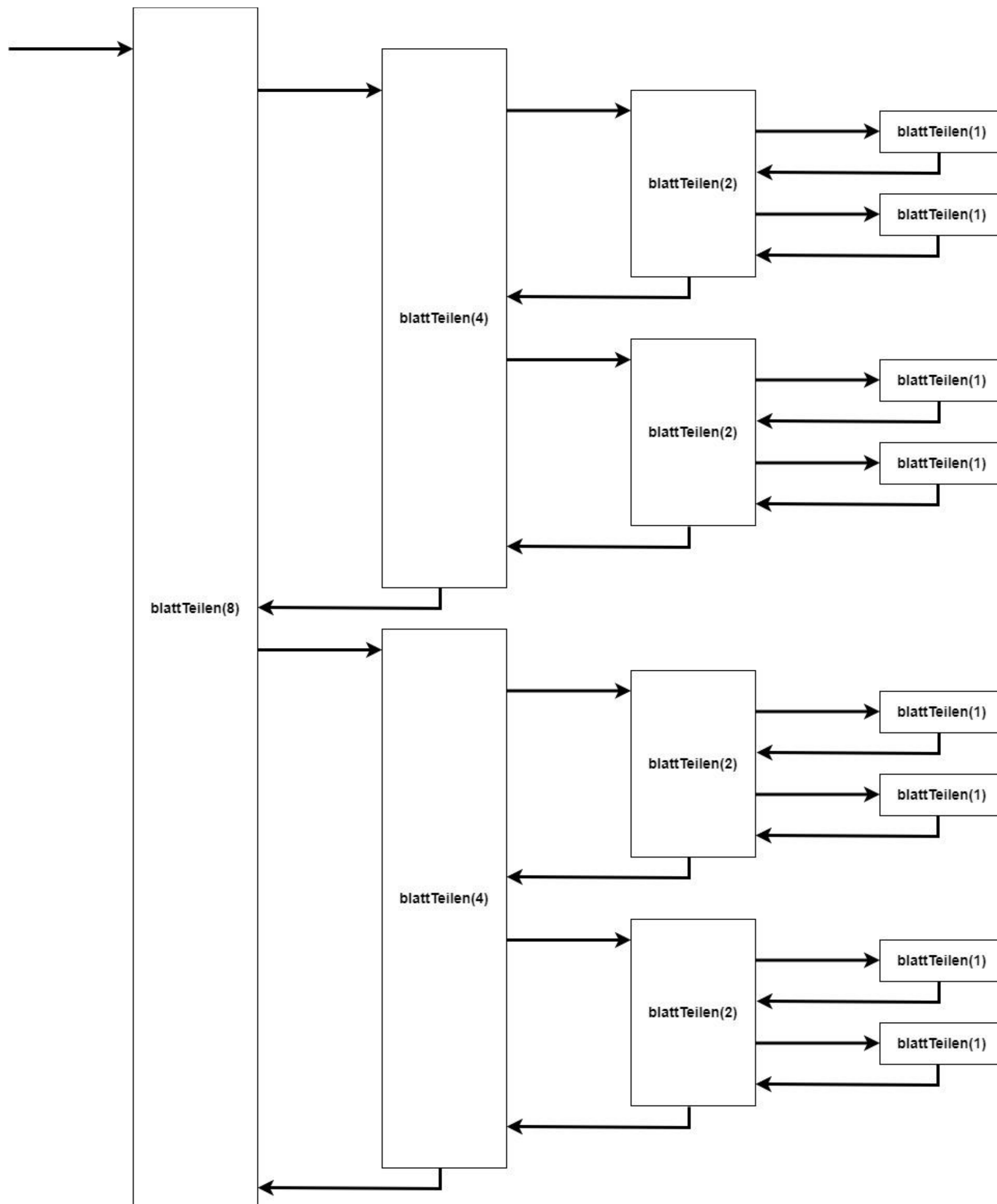


Abbildung 19: gesamte Aufrufsequenz für die acht Streifen

Nun ist ersichtlich, dass wir innerhalb der Funktion **`blattTeilen(…)`** die Funktion **`blattTeilen(…)`** zweimal aufrufen.
Dies ändert aber nichts am Vorgehen, man kann sogar dasselbe Rezept aus der Problemlösestrategie anwenden.

Rezept:

1. *Beschreiben Sie die Probleminstanz und ermitteln Sie die Problemgrösse N*.
   Wir möchten ein Blatt in $n = 8$ Streifen aufteilen, womit eine Problemgrösse $N = n = 8$ vorliegt.
2. *Ermitteln und lösen Sie die trivialste Probleminstanz.*
   Die trivialste Probleminstanz mit der Grösse $N = 1$ ist, falls wir das Blatt nicht teilen müssen, d.h. **`blattTeilen(1)`**.
3. *Reduzieren Sie die Probleminstanz in eine oder mehrere kleinere Probleminstanzen*. *(divide)*
   Die kleineren Probleminstanzen ergeben sich durch das Halbieren des Blattes. Damit reduziert sich die Problemgrösse zu $N_{red} = {}^{n}/_{2}$ und wir haben zwei kleinere Probleminstanzen.
4. *Lösen Sie die kleinere Probleminstanz desselben Problems mit demselben Algorithmus, d.h., verwenden Sie dazu die Rekursion, solange Sie nicht die trivialste Probleminstanz erreicht haben, welche einfach bestimmt werden kann*. *(impera*)
   Jede kleinere Probleminstanz kann gleich gelöst werden wie die ursprüngliche Probleminstanz. Damit können die einzelnen Blatteile mit demselben Algorithmus bearbeitet werden, bis wir die trivialste Probleminstanz erreicht haben. Sind nun alle Hälften bis zur trivialsten Probleminstanz bearbeitet worden, dann haben wir die n Streifen.

Es ist klar, dass die Rekursion v.a. mit zwei Aufrufen komplex ist. Um dem ein wenig entgegenzuwirken, wird der Ablauf der Funktion **`blattTeilen(…)`** als Baumstruktur dargestellt. Dazu erstellen wir zuerst ein Python-Programm, welches die Aufrufsequenz der Abbildung 19 implementiert, natürlich nur anschaulich für die Blattteilung.

4.3.1 Einführung Baumstruktur

Beispiel 7

Wie in der Abbildung 19 dargestellt wird, werden zwei rekursive Aufrufe in der Funktion benötigt. Dies erschwert am Anfang die Vorstellung, deshalb wird das zugehörige Python-Programm angegeben und später als Baum aufgezeichnet:

```
def blattTeilen(anzStreifen):
    if anzStreifen == 1:
        print('Streifen zu 1 / %d vom Blatt' % (AnzahlStreifen))
        return
    #linke Hälfte
    blattTeilen(anzStreifen/2)
    #rechte Hälfte
    blattTeilen(anzStreifen/2)

AnzahlStreifen = 8 #16
#Paramter als 2er Potenz !!
blattTeilen(AnzahlStreifen)
```

Dieses Programm erstellt aus einem Blatt, natürlich nur fiktiv, **`AnzahlStreifen`** Streifen.
Wir haben ein Blatt und möchten es in 8 Streifen teilen, deshalb rufen wir die Funktion mit dem Parameter 8 auf:

`blattTeilen(8)`

Die Aufrufe werden nun graphisch in einer Baumstruktur dargestellt, damit der Ablauf besser dargestellt werden kann. Der erste Aufruf **`blattTeilen(8)`** bildet die Wurzel des Baumes.

Abbildung 20: erster Aufruf blattTeilen(8)

Nun wird in der Funktion **`blattTeilen(8)`** für die linke Hälfte des Blattes die Funktion **`blattTeilen(4)`** rekursiv aufgerufen, **`4`** berechnet sich aus **`AnzahlStreifen /2 = 8/2`**. So können wir die linke Hälfte in 4 Streifen teilen.

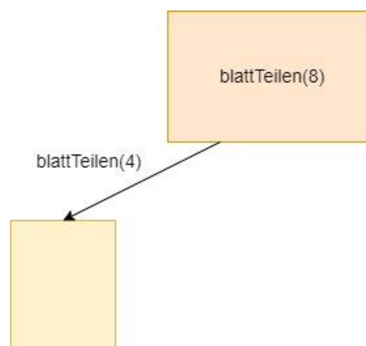


Abbildung 21: blattTeilen(4), erste Rekursionstiefe

Nach diesem Schritt befindet sich das Programm in der Funktion **`blattTeilen(4)`** und es wird wieder zuerst für die linke Hälfte **`blattTeilen(2)`** aufgerufen. Graphisch als Baumstruktur sieht es folgendermassen aus:

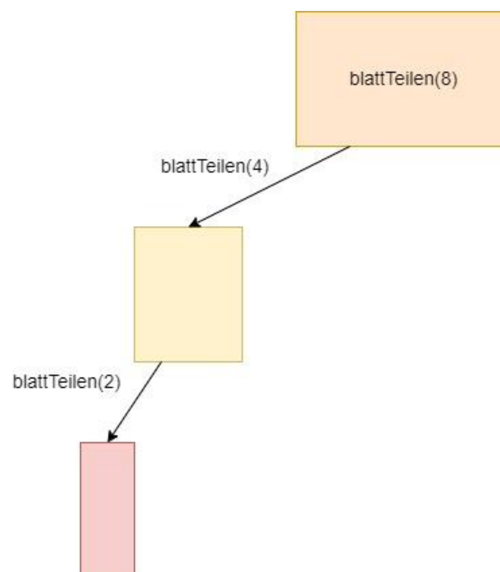


Abbildung 22: blattTeilen(2), zweite Rekursionstiefe

In einem weiteren Schritt werden wir diesen Teil halbieren. Dafür wird für den linken Teil die Funktion **`blattTeilen(1)`** aufgerufen.

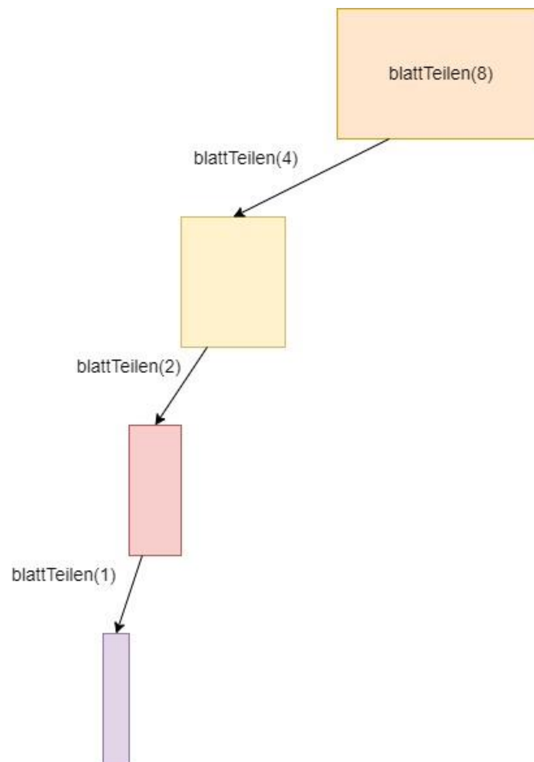


Abbildung 23: blattTeilen(1), dritte Rekursionstiefe

Wir sind am Ende eines Pfades der Rekursion angelangt, **`blattTeilen(1)`**, was die trivialste Probleminstanz ist, bei der man nicht weiter teilen muss. Der Programmablauf befindet sich nun in der Funktion **`blattTeilen(2)`** nach dem ersten Aufruf **`blattTeilen(1)`**. (Abbildung 23) Die nächste Zeile im Programm ruft für den rechten Teil des Blattes die Funktion **`blattTeilen(1)`** nochmals auf. Somit haben wir zwei von den acht Streifen erstellt. (Abbildung 24)

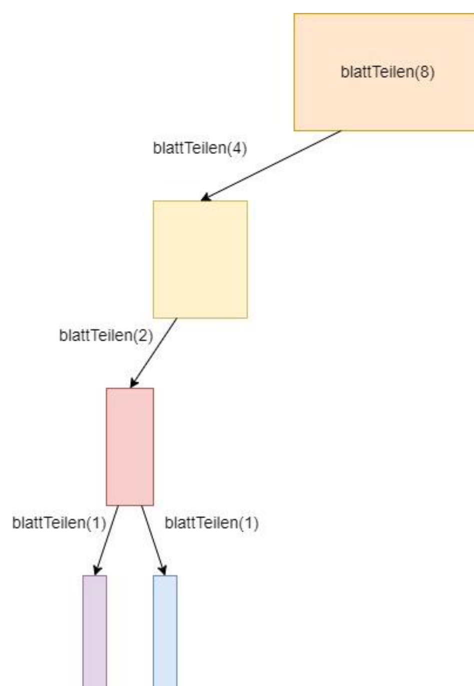


Abbildung 24: blattTeilen(1), dritte Rekursionstiefe, 2 Streifen

Nach diesem zweiten Aufruf von **`blattTeilen(1)`** wird der Aufruf **`blattTeilen(2)`** beendet. Das Programm befindet sich in der Funktion **`blattTeilen(4)`** für die linke Hälfte, was in der Abbildung 19 dem oberen Rechteck mit der Beschriftung **`blattTeilen(4)`** entspricht. Als Nächstes wird der zweite Aufruf von **`blattTeilen(2)`** (Abbildung 25) für den rechten Teil aufgerufen.

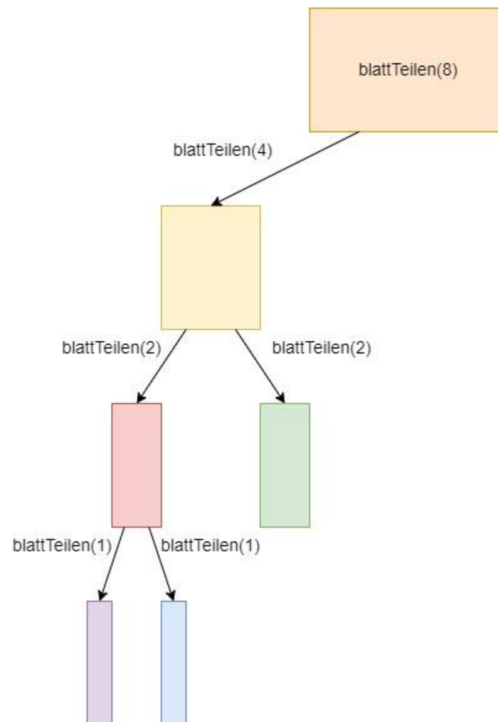


Abbildung 25: blattTeilen(2), rechter Teil, zweite Rekursionstiefe

Die restlichen Streifen werden nun analog zu den vorherigen erstellt.

Aufgabe 17
Erweitern Sie die Graphik in der Abbildung 25, bis alle acht Streifen erstellt sind. Überlegen Sie sich auch, wie die Reihenfolge der Aufrufe bis zum Endergebnis ist.

Lösung

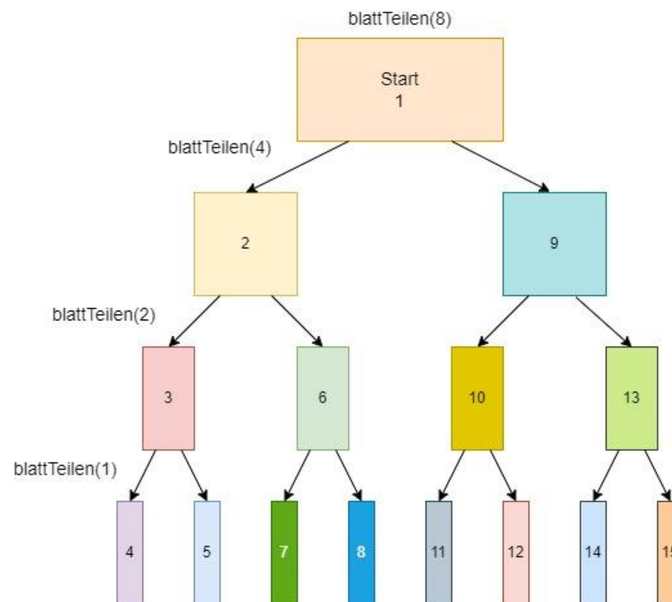


Abbildung 26: Alle acht Streifen sind erstellt.

Die Zahlen in den Blöcken entsprechen der Aufrufreihenfolge der rekursiven Aufrufe. Damit ist jeweils ersichtlich, dass immer zuerst der linke Teil erstellt wird, bis die unterste Ebene erreicht ist, und erst danach wird in der oberen Ebene der rechte Teil erstellt.

Aufgabe 18

Dieser Ablauf ist im folgenden Code als Animation dargestellt.

```
from gpanel import *
import random

offsetX = 3
offsetY = 90
startX = 60
actX = startX
startY = 400
actY = startY
widthX = 160
widthY = 80 #120

def blattTeilen(anzStreifen):
    global actX, widthX, actY
    doIt()
    delay(1000)
    if anzStreifen == 1:
        print('Streifen zu 1 / %d vom Blatt' % (AnzahlStreifen))
        return
    #linke Hälfte
    actX = actX  - offsetX
    actY = actY  - offsetY
    widthX = widthX / 2
    blattTeilen(anzStreifen/2)
    #rechte Hälfte
    actX = actX  + widthX + 2*offsetX
    blattTeilen(anzStreifen/2)
    actX = actX  - widthX + offsetX
    widthX = widthX * 2
    actY = actY  + offsetY


drawLeft = True
def doIt():
    lineWidth(5)
    setColor('black')
    rectangle(actX, actY, widthX+actX, widthY+actY)
    r = random.randint(0, 255)
    g = random.randint(0,255)
    b = random.randint(0, 255)
    c = makeColor(r,g,b)
    setColor(c)
    fillRectangle(actX, actY, widthX+actX, widthY+actY)

makeGPanel(0, 500, 0, 500)

AnzahlStreifen = 8 #16
#Paramter als 2er Potenz !!
blattTeilen(AnzahlStreifen)
```

a) Kopieren Sie den Code in TigerJython und lassen Sie ihn laufen.
b) Vergleichen Sie nun den Ablauf der Rekursion in der Animation mit den Abbildungen aus Ihrer Lösung der vorhergehenden Aufgabe. Sie können das Programm im Debug Modus laufen lassen, damit Sie mehr Zeit haben für Ihre Überlegungen.

Aufgabe 19

Es soll der Aufruf der Funktion **`fibonacci(7)`** mit der rekursiven Implementierung aus der Aufgabe 14 genauer analysiert werden. Überlegen Sie sich mit Hilfe einer Baumstruktur, wie viele Male die Funktion **`fibonacci(…)`** mit demselben Wert aufgerufen wird. Damit Sie nicht zurückblättern müssen, wird die Funktion unten nochmals dargestellt.

```
int fibonacci(n)
    falls n <= 2
        return 1
    sonst
        #Vorgänger + Vorvorgänger
        return fibonacci(n-1) + fibonacci(n-2)
```

Lösung:

```
fibonacci(7) = fibonacci(6) + fibonacci(5) = ...
fibonacci(6) = fibonacci(5) + fibonacci(4) = ...
fibonacci(5) = fibonacci(4) + fibonacci(3) = ...
fibonacci(4) = fibonacci(3) + fibonacci(2) = ...
fibonacci(3) = fibonacci(2) + fibonacci(1) = ...
fibonacci(2) = 1
fibonacci(1) = 1
```

Damit dies übersichtlicher wird, werden alle Aufrufe in einem Baum dargestellt. Die Funktion **`fibonacci(7)`** wird dabei abgekürzt geschrieben **`fib(7)`**.

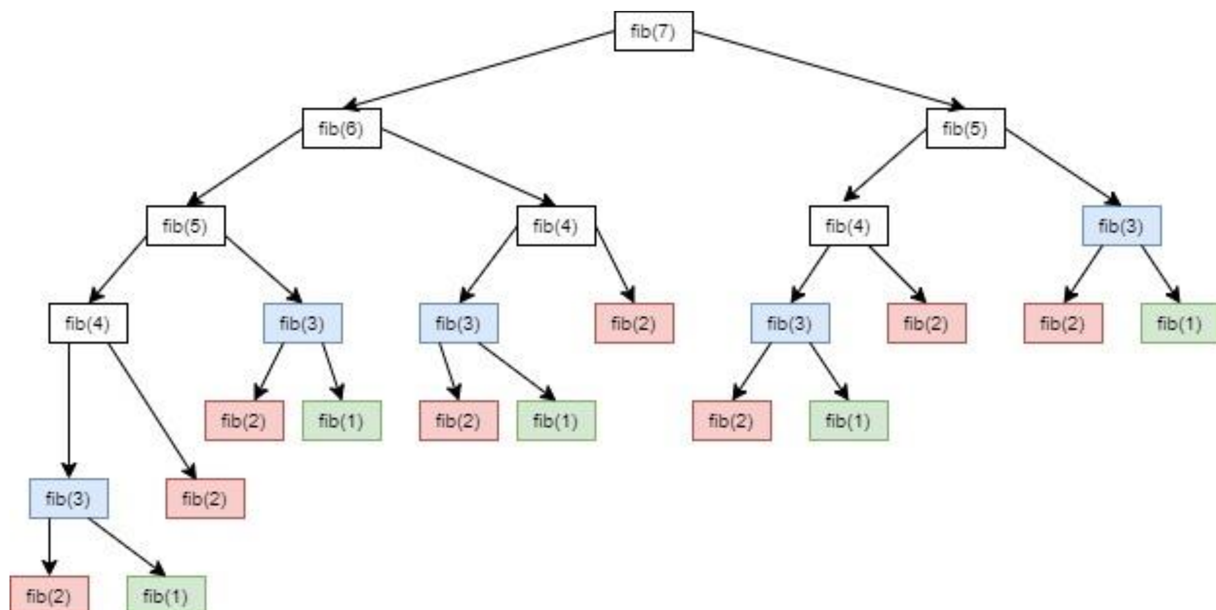


Abbildung 27: Aufruf Baum der Fibonacci-Folge

Nun ist ersichtlich, dass **`fibonacci(3)`** und **`fibonacci(1)`** fünf Mal und **`fibonacci(2)`** acht Mal aufgerufen werden.

Ein kleiner Exkurs:
Dies könnte optimiert werden, indem man bereits berechnete Werte zwischenspeichert. Dieses Verfahren wird auch als **Memoisation** bezeichnet, was in der Problemlösung mit dynamischer Programmierung häufig benutzt wird.

```
memo = []
def fibonacci(n):
    length = len(memo)
    if n < length:
        #da die Zahlen mit a1 beginnen und
        #in der Liste der Index bei 0 beginnt, gilt memo[n-1]
        return memo[n-1]
    elif n <= 2:
        result = 1
    else:
        #Vorgänger + Vorvorgänger
        result = fibonacci(n-1) + fibonacci(n-2)
    # Memoisierung
    memo.append(result)
    return result

print(fibonacci(7))
```

Aufgabe 20
Erstellen Sie einen Baum für das gegebene Programm mit den Aufrufen der Funktion **`wabe(s)`** in den Knoten (analog zu Blatt teilen). Die Funktion **`wabe(s)`** erstellt mit **`s = 8`** folgendes Wabenmuster in TigerJython.

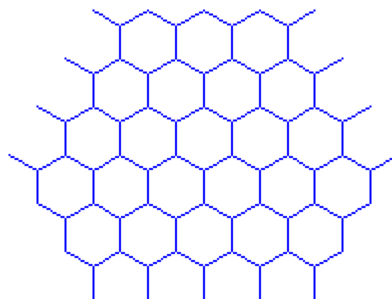


Abbildung 28: Wabenmuster

```
from gturtle import *

def wabe(s):
    if s == 0:
        return
    forward(a)
    left(60)
    wabe(s - 1)
    right(120)
    wabe(s - 1)
    left(60)
    back(a)
```

```
makeTurtle()
hideTurtle()
a = 20
s = 12
wabe(s)
```

a) Testen Sie das Programm aus, z.B. mit s = 1, s=2, s=6 und s=12.
b) Erstellen Sie einen Baum zu den rekursiven Aufrufen, falls das Programm mit **`s = 4`** gestartet wird und zeichnen Sie den Aufbau des Wabenmusters Schritt um Schritt auf.

Lösung:

b)

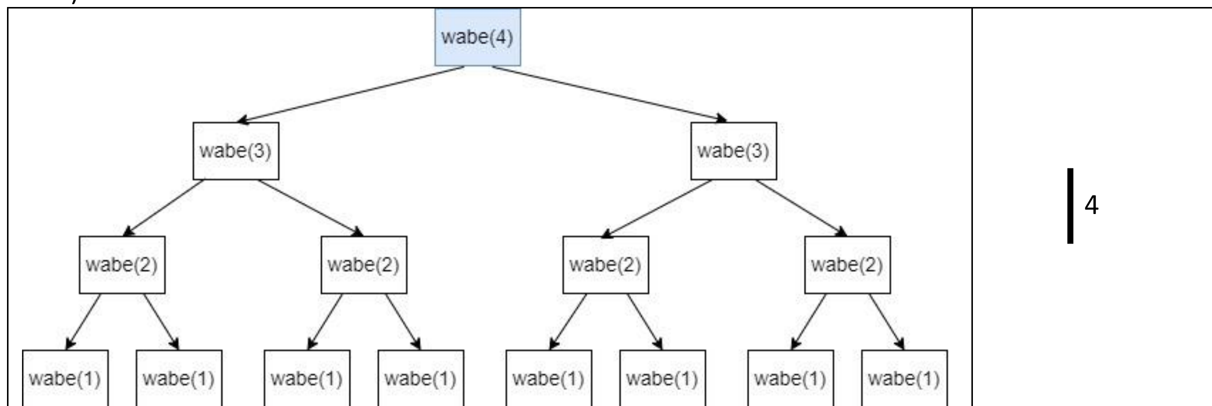


Abbildung 29: Wabe Aufruf 1 mit Ausgabe

Die Zahlen bei den Linien in der rechten Spalte entsprechen dem Wert des Parameters s. In der Abbildung 29 ist **`s=4`** (Aufruf **`wabe(4)`**) und zeichnet zuerst nur eine Linie.

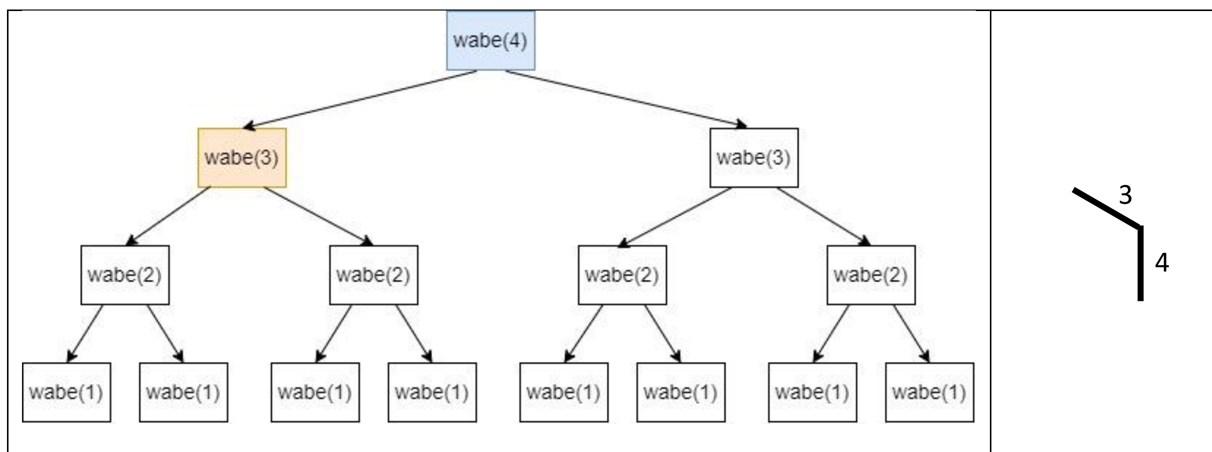


Abbildung 30: Wabe Aufruf 2 mit Ausgabe

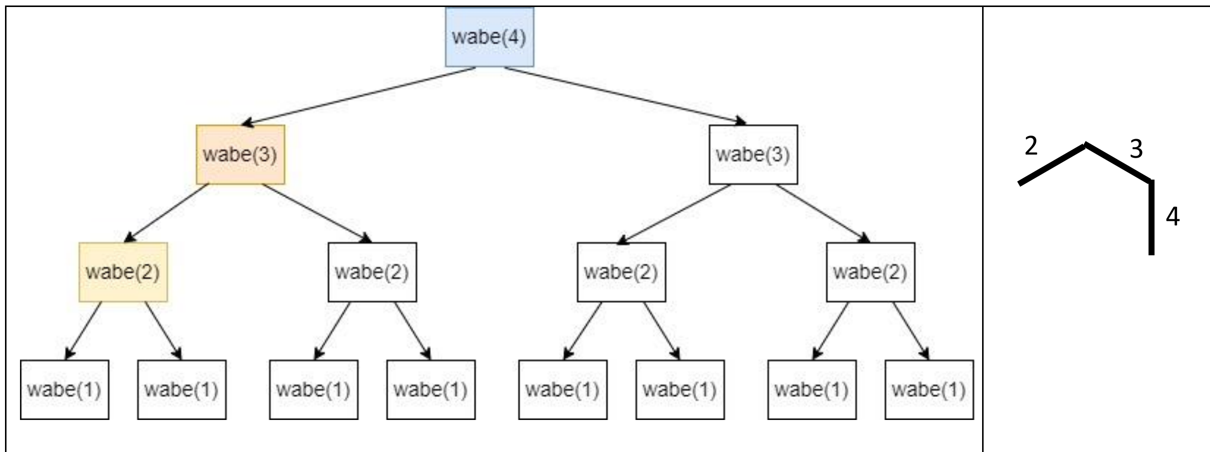


Abbildung 31: Wabe Aufruf 3 mit Ausgabe

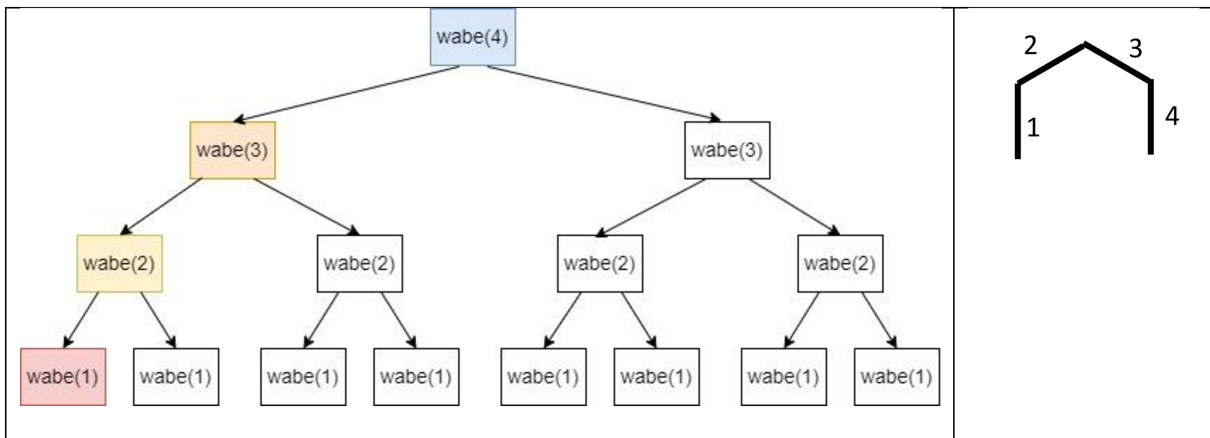


Abbildung 32: Wabe Aufruf 4 mit Ausgabe

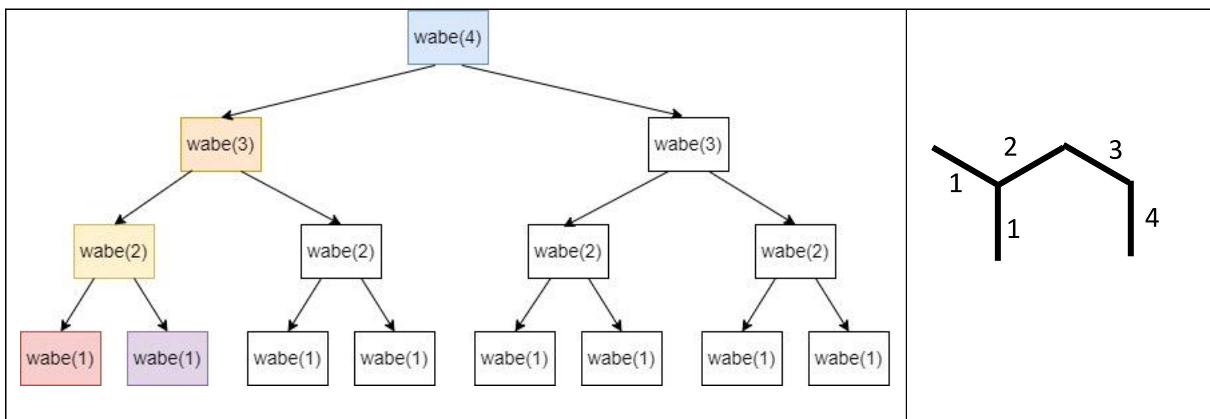


Abbildung 33: Wabe Aufruf 5 mit Ausgabe

Zur besseren Veranschaulichung wird bei den verbleibenden Bildern auf die Baumstruktur verzichtet.

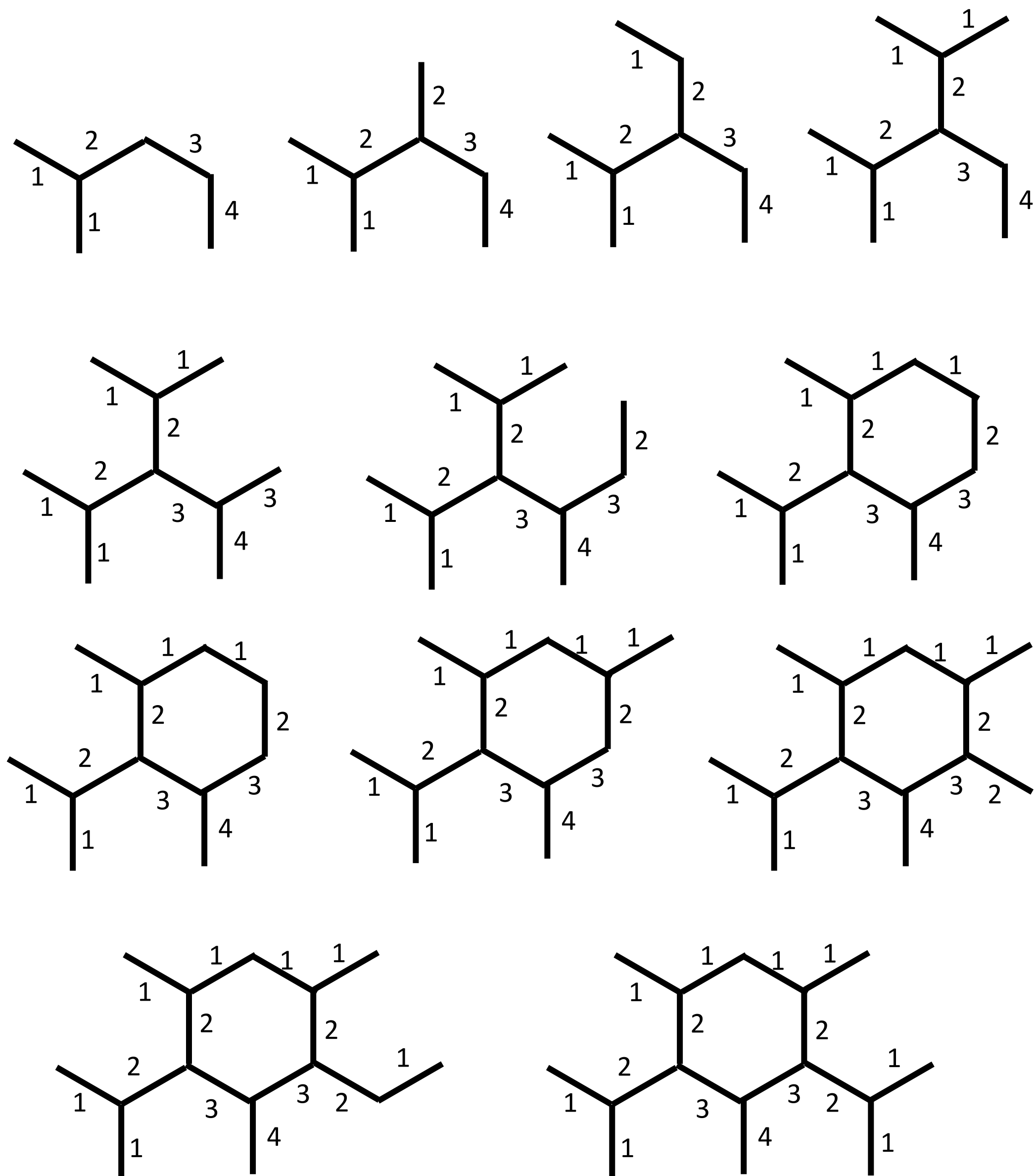


Abbildung 34: Schrittweiser Aufbau der Wabe mit s = 4

Aufgabe 21

Ein bekannter Graph ist der vollständige binäre Baum. Die Abbildung 35 zeigt den Baum für eine bestimmte Rekursionstiefe. Wir wollen eine rekursive Funktion **`tree(length)`** herleiten, die den Baum mit der initialen Länge **`length = 64`** des Baumstammes zeichnet. Bei jedem rekursiven Aufruf der Funktion **`tree(…)`** wird die Länge des Astes resp. des Stammes halbiert.

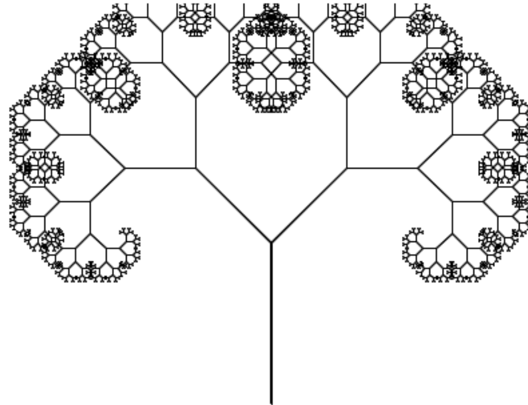


Abbildung 35: Binärbaum

a) In der Aufgabe fehlt eine Aussage über die Abbruchbedingung. Überlegen Sie sich eine plausible Bedingung.
b) Analysieren Sie in einem ersten Schritt die Aufgabenstellung unter Verwendung des im Kapitel 4.2.1 «Problemlösestrategie mit Rekursion» hergeleiteten Rezeptes.
c) Zeichnen Sie von Hand die Entstehung des binären Baums Schritt um Schritt auf. Bei einer Startlänge von 64 und bei einer Länge kleiner als 16 soll das Programm beendet werden. Die Zahlen sind angegeben, um die Rekursionstiefe zu beschränken.
d) Zeichnen Sie die Aufrufsequenz der Funktion **`tree`**(…) für jeden Schritt der Teilaufgabe c) mit einer Startlänge von 64 und Endlänge von 16.
e) Erstellen Sie einen Baum zu den rekursiven Aufrufen der Funktion **`tree(…)`** für die Teilaufgabe c).
f) Leiten Sie aus b), c) und d) den rekursiven Algorithmus ab und implementieren Sie die Funktion **`tree(length)`** in Python.

Lösung

a) Die Abbruchbedingung könnte z.B. sein: `length < 16`

b) Für die rekursive Lösung wird das bekannte Rezept angewendet:

1. *Beschreiben Sie die Probleminstanz und ermitteln Sie die Problemgrösse N.*
   Es soll ein vollständiger Binärbaum mit einer Anfangslänge `length = 64` des Stammes gezeichnet werden. Daraus ergibt sich die Problemgrösse $N = 64$.
2. *Ermitteln und lösen Sie die trivialste Probleminstanz*
   Die trivialste Probleminstanz liegt dann vor, falls die Länge des Stammes kleiner als 16 ist, in diesem Beispiel entspricht dies der Länge `length = 8` , dann wird nichts mehr gezeichnet und die Abbruchbedingung ist erfüllt.
3. *Reduzieren Sie die Probleminstanz in eine oder mehrere kleinere Probleminstanzen. (divide)*
   Der zu zeichnende Baum besteht aus einem Stamm und dann zwei Ästen, welche sich jeweils in einem 45 Grad Winkel nach links resp. rechts verzweigen und nur noch die Hälfte der Länge des Stammes aufweisen. Diese beiden Äste können wiederum als Stämme eines kleineren Baumes mit den gleichen Aufbauregeln interpretiert werden. Die kleineren Probleminstanzen bestehen nun im Zeichnen der jeweiligen zwei kleineren Teilbäume, mit verkürztem Stamm. Die reduzierte Problemgrösse ist $N_{red} = {}^{length}/_{2}$.
4. *Lösen Sie die kleinere Probleminstanz desselben Problems mit demselben Algorithmus, d.h., verwenden Sie dazu die Rekursion, solange Sie nicht die trivialste Probleminstanz erreicht haben, welche einfach bestimmt werden kann. (impera)*
   Zeichne den Baum (ursprüngliche Probleminstanz):
   - Zeichne den Baumstamm mit Länge $length$
   - Zeichne jeweils den linken verkürzten Baum mit der Stammlänge: $length = {}^{length}/_{2^k}$.
   - Zeichne jeweils den rechten verkürzten Baum mit Stammlänge: $length = {}^{length}/_{2^k}$.
   - Wiederhole das Zeichnen bis $length < 16$ ist

c) Baum von Hand zeichnen:
Startlänge 64, Endlänge 16

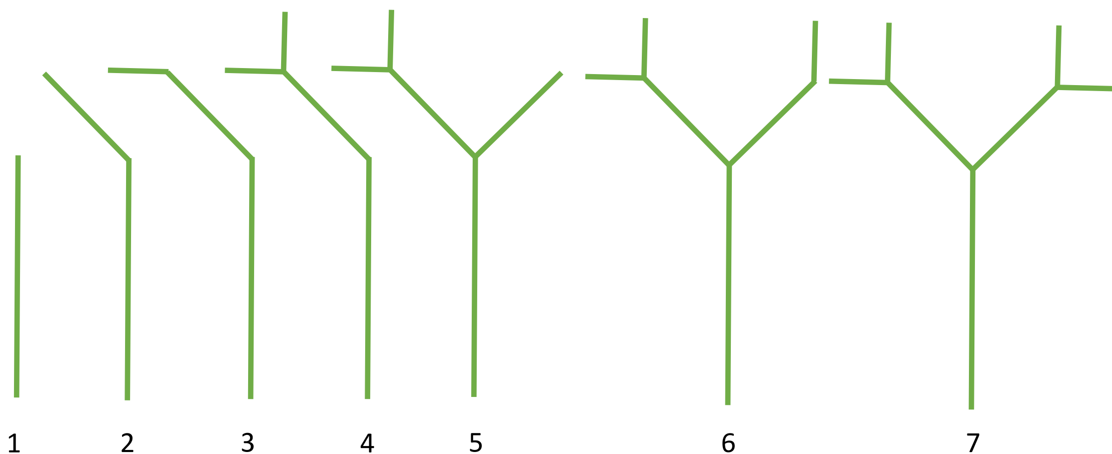


Abbildung 36: schrittweiser Aufbau des Baumes

d) Aufrufsequenz

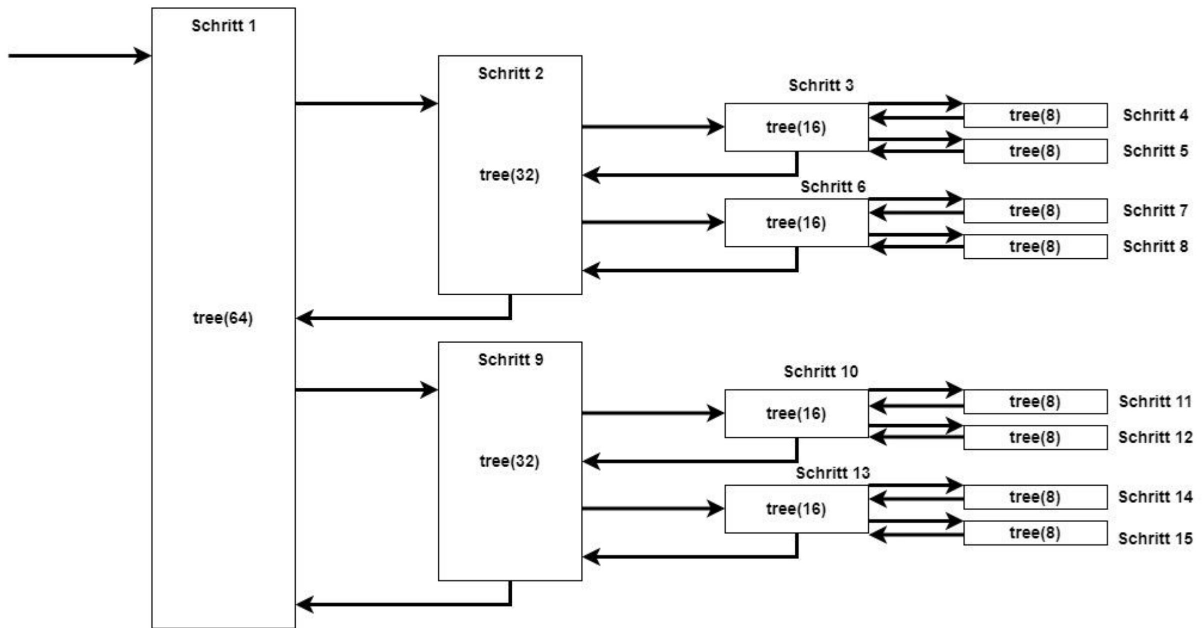


Abbildung 37: Aufrufsequenz der Funktion tree

e) Baumstruktur

Schritt 1:

Abbildung 38: Baumstruktur der Aufrufe in Schritt 1

Schritt 2:

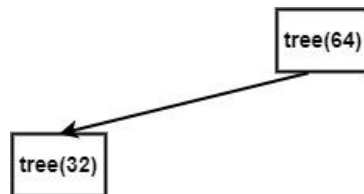


Abbildung 39: Baumstruktur der Aufrufe in Schritt 2

Schritt 3:

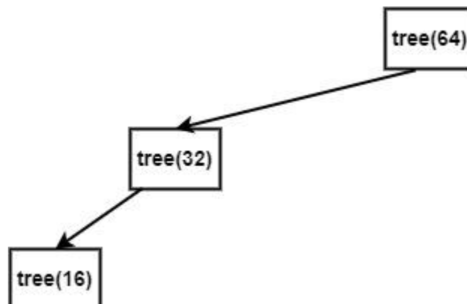


Abbildung 40: Baumstruktur der Aufrufe in Schritt 3

Schritte 4 und 5:

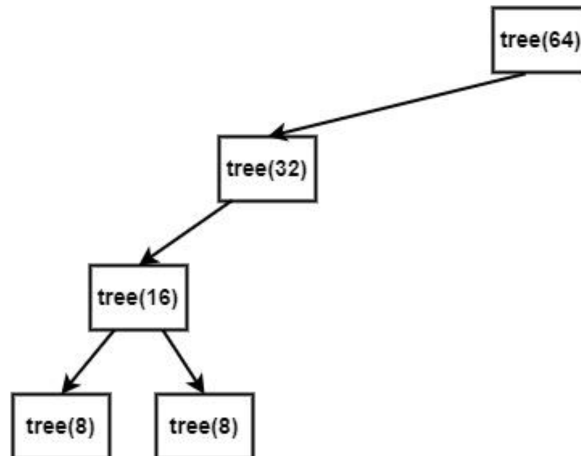


Abbildung 41: Baumstruktur der Aufrufe in den Schritten 4 und 5

Schritte 6, 7 und 8:

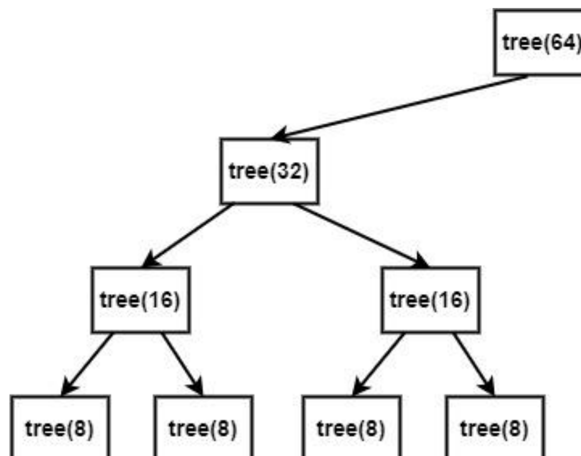


Abbildung 42: Baumstruktur der Aufrufe in den Schritten 6, 7, und 8

Schritt 9:

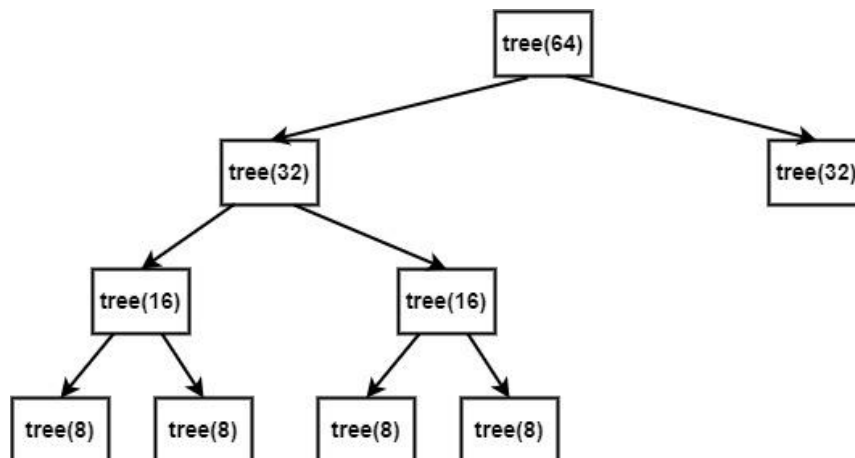


Abbildung 43: Baumstruktur der Aufrufe in Schritt 9

Schritte 10, 11 und 12:

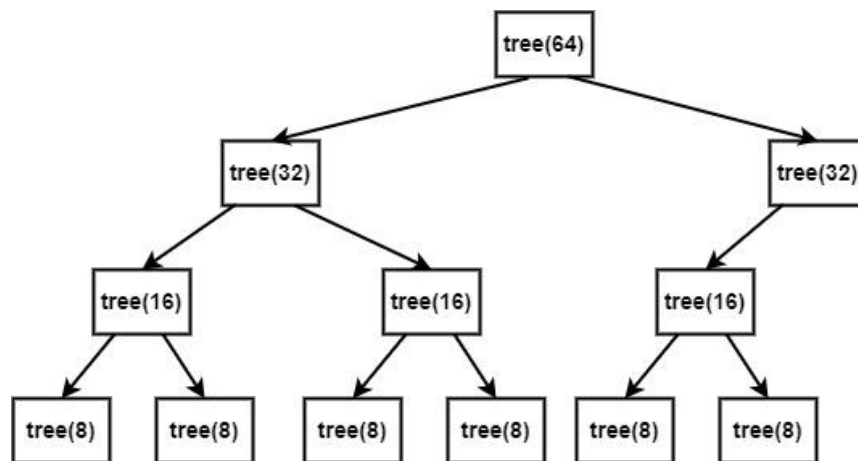


Abbildung 44: Baumstruktur der Aufrufe in den Schritten 10, 11 und 12

Schritte 13, 14 und 15:

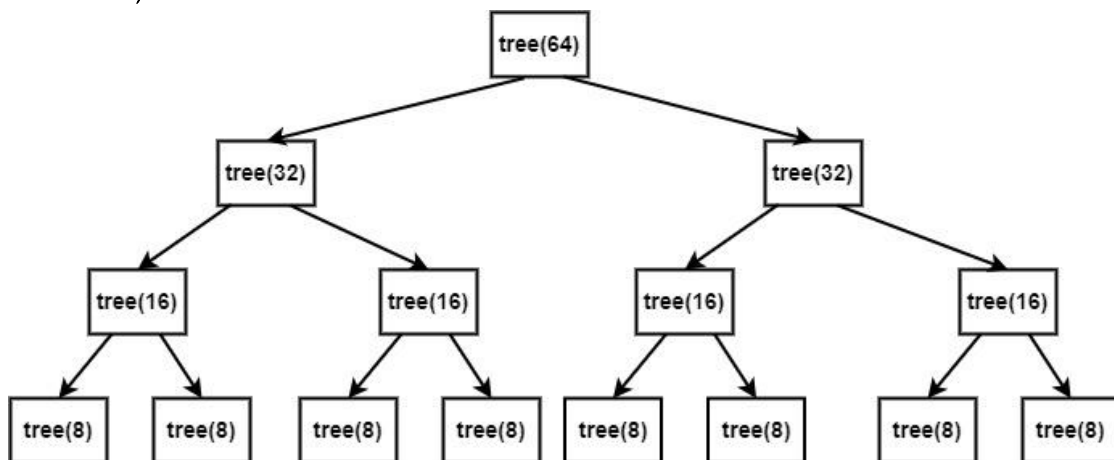


Abbildung 45: Baumstruktur der Aufrufe in den Schritten 13, 14 und 15

f) Python-Programm **tree(length)**

```
from gturtle import *

minLength = 0.1 #16
startLength =  64 #64 #256

def tree(s):
    if s < minLength:
        return
    forward(s)
    left(45)
    tree(s/2)
    right(90)
    tree(s/2)
    left(45)
    back(s)

makeTurtle()
setPos(0, -250)
hideTurtle()
tree(startLength)
```

4.4 Speichernutzung beim Aufruf einer Funktion

Wir befassen uns nun mit den Funktionsaufrufen und möchten erfahren, welche Aufgaben bei einem Funktionsaufruf im Hintergrund durchgeführt werden müssen. Danach betrachten wir die Rekursion und möchten auf eine gewisse Problematik hinweisen.

Betrachten wir zuerst ein einfaches Beispiel:

```
def Mittelwert(a, b):
    m = (a + b) / 2
    return m

x = 8
y = 13
a = 7
b = 9
mxy = Mittelwert(x, y)
print('Der Mittelwert ist: ' + str(mxy))
summe = a + b
print('summe: ' + str(summe))
```

Im Programm wird die Funktion **`Mittelwert(…)`** aufgerufen, nachdem die Funktion den Mittelwert berechnet hat, wird dieser der Variable **`mxy`** zugewiesen und der Wert wird ausgegeben. Danach wird die Summe von **`a + b`** berechnet und ausgegeben. Analoge Beispiele haben wir schon etliche Male ausgeführt und uns nie darum gekümmert, weshalb dies funktioniert. Wir werden gewisse Überlegungen vereinfacht darstellen, damit wir die Erkenntnisse in der Rekursion nutzen können.

Jedes Programm läuft nach der Übersetzung in einen Mikrocode schlussendlich auf einem Prozessor, dieser hat einen sogenannten Programm-Counter (PC), mit dem der Prozessor weiss, welche Zeile des Mikrocodes gerade ausgeführt wird. Bei einer Verzweigung (oder einem Sprung) in eine Funktion wird dem PC ein neuer Wert zugewiesen, nämlich derjenige der Zeilennummer der Funktion. Irgendwie wird nach dem Verlassen der Funktion wieder das Hauptprogramm am ursprünglichen Ort weiter ausgeführt, d.h., der PC muss wieder mit dem ursprünglichen Wert belegt werden.

Die Werte können als Parameter der Funktion übergeben werden, damit sie für die weitere Verarbeitung benutzt werden können.

Aufgabe 22

Wie könnte man sicherstellen, dass der Prozessor, nachdem die Funktion verlassen wird, in der richtigen Zeile im Hauptprogramm weitermacht?
Wie können die Werte der Variablen a und b aus dem obigen Beispiel als Parameter in der Funktion benutzt werden? Oder anders gesagt: Wie kennt die Funktion die mitgegebenen Werte?

Lösung:

Das Programm muss den PC der nächsten Code-Zeile speichern, damit dieser Wert nach Beendigung der Funktion wiedergeholt werden kann.
Die Variablen werden auf einen gemeinsamen Speicher (ein Register) gelegt, auf den das Hauptprogramm und die Funktion zugreifen können und welcher nicht verändert werden darf.

Um die obigen Lösungen zu ermöglichen, benötigen wir einen Speicher. Dafür ist ein sogenannter Stack in jedem Prozessor, dies ist ein bestimmter Bereich im üblichen Speicher (Abbildung 46). Über diesem Stack (auf Deutsch: Stapel) werden alle wesentlichen Daten vor dem Funktionsaufruf gespeichert und danach wieder gelesen.

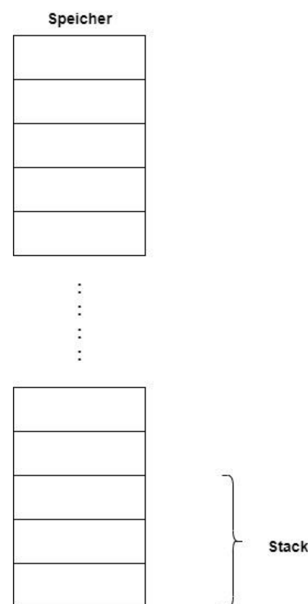


Abbildung 46: Speicher mit Stack im unteren Bereich

Welche Daten genau gespeichert werden müssen, werden wir hier nicht genau erläutern, es müssen alle notwendigen Daten und Register gesichert werden, damit das Programm nach der Rückkehr aus der Funktion an der ursprünglichen Zeile weitermachen kann, d.h., alle Register, der Program-Counter usw. müssen wiederhergestellt werden können. Dieses Datenpaket nennen wir Programmkontext.
Der Begriff Stack (oder Stapel) zielt darauf, dass man sich das Ganze als eine Art Stapel vorstellen kann, ein neuer Behälter wird daraufgelegt (Abbildung 47) und der Behälter enthält alle Daten (Programmkontext), welche gesichert werden müssen.

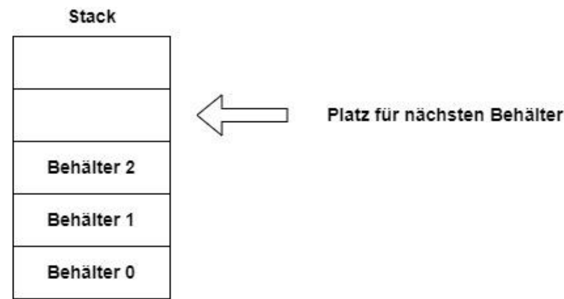


Abbildung 47: Stack befüllt mit Behälter

Betrachten wir nun wieder das kleine Programm mit dem Mittelwert. Zu Beginn ist der Stack leer (Abbildung 48).

Abbildung 48: Stack zu Beginn des Programmes

Bevor die Funktion Mittelwert aufgerufen wird, wird der Programmkontext des Hauptprogrammes auf dem Stack gesichert (Abbildung 49). Die Kopien der Variablen x und y werden in spezielle Register kopiert, was wir hier aber nicht weiter begründen werden.

Abbildung 49: Stack unmittelbar vor dem Aufruf der Funktion

Danach holt sich die Funktion die Parameter aus den Registern, führt die Code-Zeilen aus, stellt das Resultat wieder in ein Register und kehrt wieder zum Hauptprogramm zurück. Dieses holt sich wieder den Programmkontext vom Stack und hat nun alle notwendigen Daten, um die nächsten Programmzeilen auszuführen zu können.

Aufgabe 23

Nun spielen wir das Ganze für eine Rekursion durch, z.B. **`fakultaet(3)`**. Als Erinnerung wird der Code hier nochmals angegeben:

```
def fakultaet(n):
    #trivialste Probleminstanz
    if n == 1:
        return 1
    else:
        #Aufteilung in kleinere Probleminstanz
        return n*fakultaet(n-1)
```

Zeichnen Sie bei jedem rekursiven Aufruf den Stack auf, bis die Abbruchbedingung eintrifft.

Lösung:

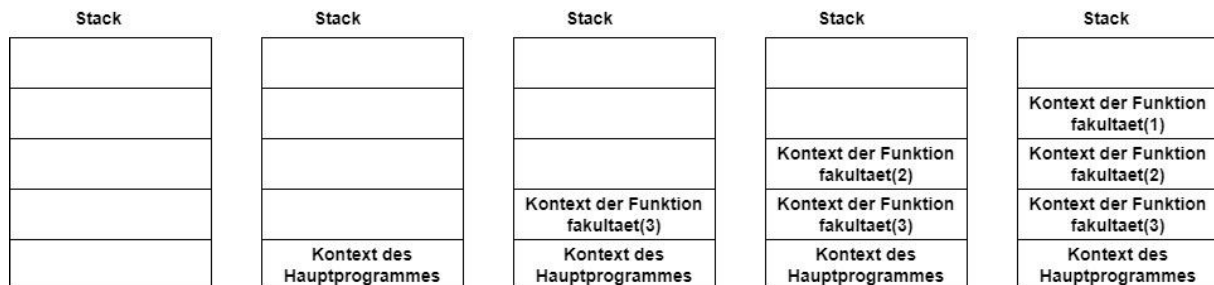


Abbildung 50: Aufbau Stack bei der Rekursion von fakultaet(3)

Stellen wir uns vor, dass wir einen Stack haben mit Platz für nur 10 Behälter und wir würden die **`fakultaet(15)`** aufrufen, dann geht aus der obigen Aufgabe klar hervor, dass der Platz im obigen dargestellten Stack nicht für die Speicherung aller Kontexte ausreicht. Dies wird Stack Overflow (oder auch: Stack-Überlauf, Stapelüberlauf, Buffer Overflow) genannt. Falls das Programm hier nicht abbrechen würde, würde der Speicher einer anderen Funktion oder eines anderen Programmes überschrieben werden, was einen fatalen Fehler zur Folge hätte. Aus diesem Grund erlaubt Python standardmässig maximal 1000 rekursiver Aufrufe, danach erscheint eine Fehlermeldung.

Aufgabe 24

Testen Sie, bei welchem Parameter der Funktion **`fakultaet(n)`** Sie einen solchen Fehler erhalten.

Lösung:
Bei **`fakultaet(1001)`** haben wir die 1000 Aufrufe überschritten und das Programm bricht ab mit der Fehlermeldung:
RuntimeError: maximum recursion depth exceeded

5 Anwendungen

5.1 Der euklidische Algorithmus

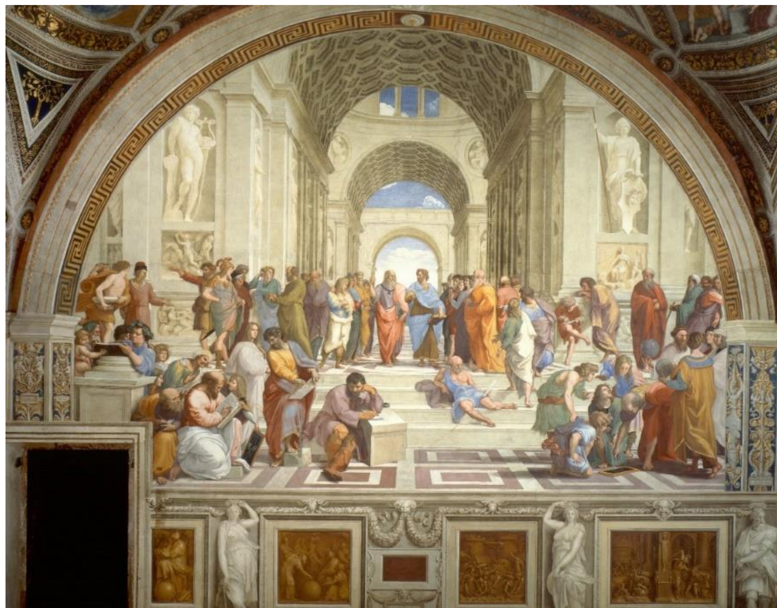


Abbildung 51: Die Schule von Athen (Quelle: https://commons.wikimedia.org/wiki/File:La_scuola_di_Atene.jpg)

Das Fresko trägt den Namen «Die Schule von Athen» und wurde von Raffael gemalt. Es zeigt einen Sommertag in Athen. Achtundfünfzig griechische Männer sind in einer prächtigen Halle und diskutieren eifrig miteinander. Sie beschäftigen sich mit den Dingen, welche sie lieben: Philosophie, Mathematik und Astronomie. Unter ihnen soll sich vorne rechts der ehrwürdige Mathematiker Euklid befinden. Er lebte vermutlich im 3. Jahrhundert vor Christus in Alexandria. Umringt von einer Menschengruppe neigt sich dieser gen Boden und schreibt etwas auf eine Tafel, was die Schar um ihn herum sehr zu interessieren scheint. Wer weiss, vielleicht ist er gerade dabei, seinen Algorithmus zur Berechnung des grössten gemeinsamen Teilers (ggT) zweier ganzer Zahlen zu erklären?

5.1.1 Die Berechnung des ggT

In diesem Kapitel wollen wir uns den euklidischen Algorithmus zur Berechnung des ggT's genauer anschauen. Wir zeigen zuerst den chinesischen Algorithmus, um dann zum Verfahren von Euklid zu gelangen.
Zur Erinnerung: Der grösste gemeinsame Teiler (ggT) zweier ganzer Zahlen ist die grösste Zahl, die beide gegebenen Zahlen ohne Rest teilt. Zum Beispiel ist der ggT von 66 und 18 gleich 6, da 6 beide Zahlen ohne Rest teilt und keine grössere Zahl diese Eigenschaft hat. Es gibt verschiedene Ansätze, den ggT zu berechnen. Neben dem euklidischen Algorithmus, welcher vom chinesischen Algorithmus hergeleitet wurde, sind die Faktorisierung und Primfaktorzerlegung die häufigsten Methoden:

1. Faktorisierung: Man faktorisiert die gegebenen Zahlen und bestimmt dann den grössten gemeinsamen Faktor, der in allen Faktorisierungen vorkommt.
2. Primfaktorzerlegung: Man zerlegt die gegebenen Zahlen in ihre Primfaktoren. Der grösste gemeinsame Teiler ist dann das Produkt aller Primfaktoren, welche in beiden Zerlegungen vorkommen.

Diese beiden Methoden können jedoch für grosse Zahlen sehr aufwendig werden. Der chinesische Algorithmus bietet eine elegante und effiziente Alternative. Er beruht auf der Tatsache, dass $ggT(a,b) = ggT(a-b,b) für\ a >= b$, wobei a und b ganze positive Zahlen sind. Der Algorithmus lässt sich damit wie folgt zusammenfassen:

1. Wenn a < b, vertausche a und b.
2. Wiederhole so lange b > 0: a = a – b. Wenn a < b, vertausche a und b.
3. Der grösste gemeinsame Teiler ist a.

Die folgende Abbildung illustriert die Berechnung des grössten gemeinsamen Teilers der beiden Zahlen 66 und 18 mittels des oben aufgeführten Verfahrens, welches auch als chinesischer Algorithmus bezeichnet wird:

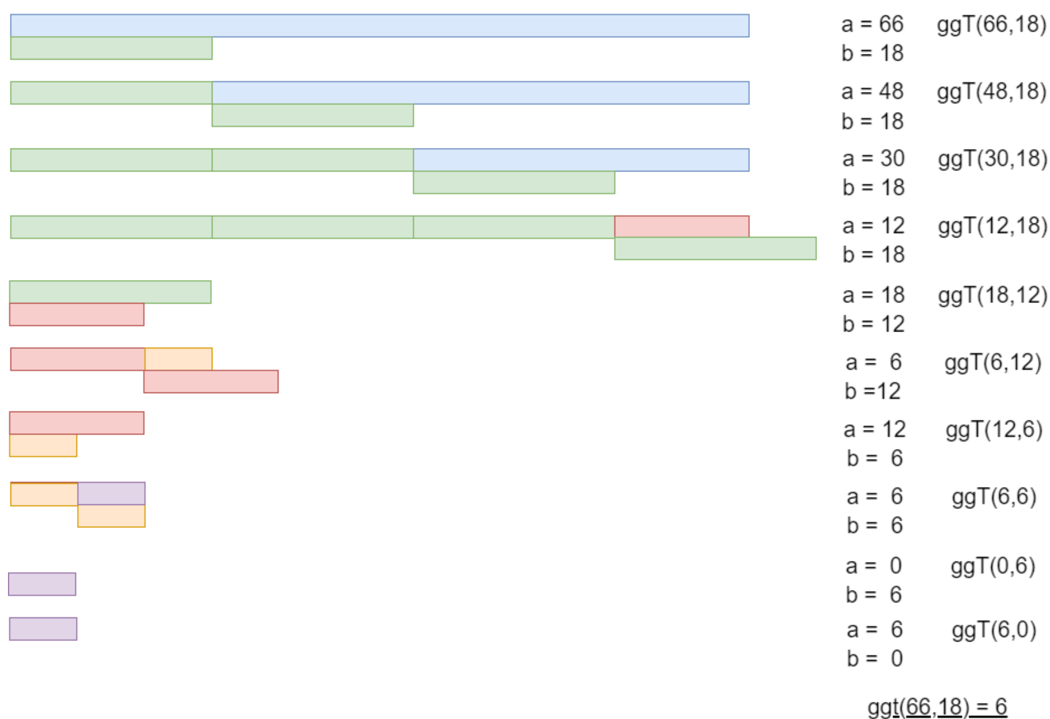


Abbildung 52: Berechnung ggT(66,18)

Aufgabe 25

Berechnen Sie, ohne eine Python-Funktion anzuwenden, den grössten gemeinsamen Teiler der beiden Zahlen 1802 und 1054 mit dem chinesischen Algorithmus.

Lösung:

ggT(1802,1054) = ggT(748,1054) = ggT(1054,748) = ggT(306,748) = ggT(748,306) = ggT(442,306) = ggT(136,306) = ggT(306,136) = ggT(170,136) = ggT(34,136) = ggT(136,34) = ggT(102,34) = ggT(68,34) = ggT(34,34) = ggT(0,34) = ggT(34,0)

Der grösste gemeinsame Teiler von 1802 und 1054 ist 34.

Nun möchten wir die Schritte aus der Aufgabe 25 genauer betrachten und versuchen eine Rekursion für diesen Algorithmus zu finden. Dabei wird die im Kapitel «Problemlösestrategie mit Rekursion» beschriebene Problemlösestrategie verwendet:

1. *Beschreiben Sie die Probleminstanz und ermitteln Sie die Problemgrösse N*.
   Gesucht ist der ggT zweier positiver ganzer Zahlen a und b. Für die Problemgrösse N werden wir die Summe der beiden Zahlen verwenden, dann gilt: $N = a + b$.
2. *Ermitteln und lösen Sie die trivialste Probleminstanz.*
   Die trivialste Probleminstanz ist der ggT einer Zahl x und 0.
   Es gilt: $ggT(x, 0) = x \; und \; ggT(0, x) = x$. Sobald wir die trivialste Probleminstanz gefunden haben, haben wir auch die Lösung des ggT von a und b berechnet.
3. *Reduzieren Sie die Probleminstanz in eine oder mehrere kleinere Probleminstanzen*. *(divide)*
   Der chinesische Algorithmus zeigt, dass
   $$ggT(a, b) = ggT(a - b, b) \; falls \; a \geq b$$
   und
   $$ggT(a, b) = ggT(a, b - a) \; falls \; b > a.$$
   Damit haben wir kleinere Zahlen und somit eine kleinere Probleminstanz und können nun mit demselben ggT-Algorithmus den ggT für diese zwei Zahlen berechnen. Die reduzierte Problemgrösse ist, falls $a \geq b$: $N_{red} = (a - b) + b = a$ oder falls $b > a$: $N_{red} = a + (b - a) = b$.
4. *Lösen Sie die kleinere Probleminstanz desselben Problems mit demselben Algorithmus, d.h., verwenden Sie dazu die Rekursion, solange Sie nicht die trivialste Probleminstanz erreicht haben, welche einfach bestimmt werden kann*. *(impera)*
   Die kleinere Probleminstanz kann nun mit demselben ggT-Algorithmus gelöst werden, bis wir die trivialste Probleminstanz erreicht haben, was uns den ggT der Zahlen a und b liefert.

Aufgabe 26

Implementieren Sie mit Rekursion die Berechnung des ggT's zweier ganzer Zahlen a und b unter Verwendung des chinesischen Algorithmus.

Lösung:

```
def ggTRecursive(a, b):
    if b == 0:
        return a
    elif a == 0:
        return b
    else:
        if a >= b:
            return ggTRecursive(a - b, b)
        else:
            return ggTRecursive(a, b - a)

print(ggTRecursive(1802, 1054))
```

Aufgabe 27
Berechnen Sie den grössten gemeinsamen Teiler der beiden Zahlen 127'500'170 und 12'750, indem Sie das Programm aus Aufgabe 26 laufen lassen. Was können Sie beobachten?

Lösung:
Der rekursive Algorithmus bricht nach einigen Sekunden ab, da die maximale Anzahl rekursiver Aufrufe in Python überschritten wird. Es erscheint die Fehlermeldung: *RuntimeError: maximum recursion depth exceeded.* Was diese Meldung genau bedeutet, wurde im Kapitel 4.4 Speichernutzung beim Aufruf einer Funktion behandelt.
Um den grössten gemeinsamen Teiler der Zahlen 127'500'170 und 12'750 zu berechnen, wird eine Rekursionstiefe von 10'075 benötigt. Die Standardeinstellung in Python erlaubt jedoch nur 1000 Rekursionsaufrufe.

Aufgrund der Erkenntnis aus Aufgabe 27 suchen wir eine bessere Berechnung, resp. Euklid hat dies für uns bereits erledigt.
Betrachten wir die Abbildung 52 für die Visualisierung des chinesischen Algorithmus. Es wird von der Zahl 66 immer 18 abgezogen, solange das Ergebnis positiv ist. Danach wird dieser Rest von 18 abgezogen usw. Der Rest, der hier entsteht, ist der Rest einer ganzzahligen Division, also die Modulo-Operation (in Python %). Nun können wir diesen Rest mit einer Modulo-Operation direkt finden und sparen uns die mehrmaligen Subtraktionen - vor allem, falls eine Zahl viel grösser ist als die andere, wie in der Aufgabe 27.
Anstatt 12'750 etliche Male von 127'500'170 abzuziehen, können wir auch Folgendes schreiben:

```
a % b = 12750070 % 12750 = 170
```

Das Ergebnis der Modulo-Operation (`Rest`) ist immer kleiner als die Zahl b, deshalb kann danach, falls `b != 0` ist, der `ggT(b, Rest)` berechnet werden und falls `b == 0` ist, dann ist der `Rest` der ggT.

Aufgabe 28
Schreiben Sie nun den verbesserten Algorithmus als rekursive Funktion.
Berechnen Sie wiederum den ggT der zwei Zahlen: 127'500'170 und 12'750

Lösung:
Da das Rezept sehr ähnlich ist wie in Aufgabe 25, wird es hier weggelassen. Es ändert sich nur die Berechnung der kleineren Probleminstanz.

```
def ggT(a, b):
    if b == 0:
        return a
    elif a == 0:
        return b
    else:
        return ggT(b, a % b)

print(ggT(127500170, 12750))
```

Für die Berechnung des ggT(127'500'170, 12'750) brauchen wir neu nur noch zwei Rekursionsschritte.

5.2 Newton-Methode für die Berechnung von Nullstellen bei Polynomen

Wie Sie aus dem Mathematikunterricht erfahren haben, können mit der Newton-Methode die Nullstellen von Polynomen berechnet werden. Man beginnt mit einem Startpunkt x_1, dann im nächsten Schritt wird die Nullstelle x_2 der Tangente an der Stelle x_1 an die Funktion als neuer Punkt definiert. Dieses Verfahren kann nun mit dem neuen Punkt weitergeführt werden, bis wir einen Wert erreichen, der sehr nahe an der Nullstelle des Polynoms liegt.

Die folgenden Bilder aus https://commons.wikimedia.org/wiki/File:NewtonIteration_Ani.gif zeigen als Repetition graphisch den Ablauf:

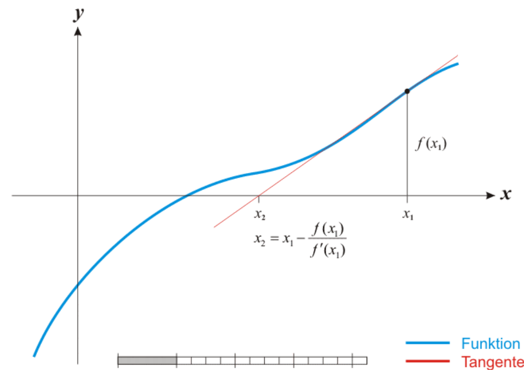


Abbildung 53: Newton-Verfahren erster Schritt (Quelle: https://commons.wikimedia.org/wiki/File:NewtonIteration_Ani.gif)

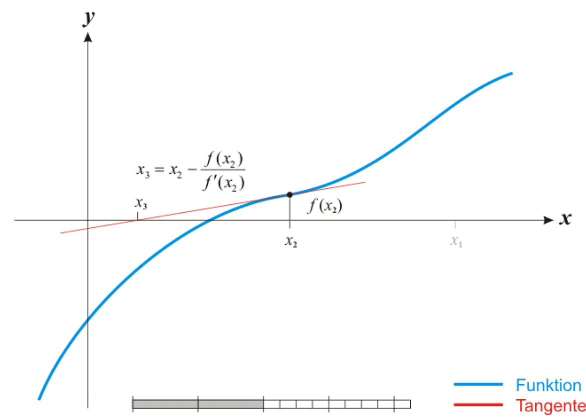


Abbildung 54: Newton-Verfahren zweiter Schritt (Quelle: https://commons.wikimedia.org/wiki/File:NewtonIteration_Ani.gif)

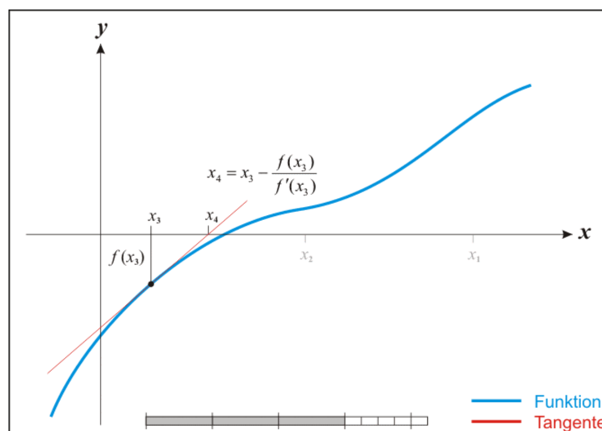


Abbildung 55: Newton-Verfahren dritter Schritt (Quelle: https://commons.wikimedia.org/wiki/File:NewtonIteration_Ani.gif)

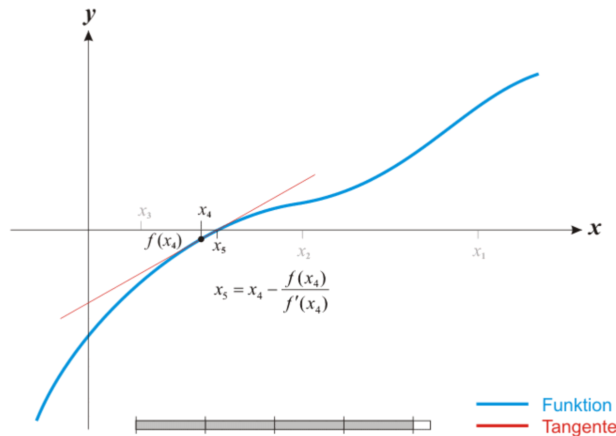


Abbildung 56: Newton-Verfahren vierter Schritt (Quelle: https://commons.wikimedia.org/wiki/File:NewtonIteration_Ani.gif)

Für die Berechnung der neuen Annäherung an die Nullstelle wird folgende Formel verwendet:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Dies ist eine rekursive Definition der neuen Annäherung x_{n+1}, deshalb werden wir nun das Newton-Verfahren in Python auch mit Rekursion lösen. Wir beschränken uns hier auf Polynome der Form:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Dann gilt für die Ableitung $f'(x)$:

$$f'(x) = n \cdot a_n x^{n-1} + (n-1) \cdot a_{n-1} x^{n-2} + \cdots + a_1$$

Beispiel 8

Wir erstellen eine Funktion in Python, welche den Funktionsgraphen eines Polynoms dritter Ordnung zeichnet. Die Parameter eines Polynoms können als Liste (Array) übergeben werden, wobei der Wert an dem Index 0 dem Koeffizienten a_n (a_3) entspricht und somit a_0 am Index n (3) liegt. Die Länge der Liste ist ***n+1*** (= **4**), falls das Polynom die Ordnung ***n*** (**3**) hat.

Also falls wir das Polynom

$$f(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0 = 3x^3 + 4x^2 + 2x + 5$$

der Plot-Funktion übergeben möchten, werden wir die Liste mit den untenstehenden Koeffizienten erstellen:

```
coeff = [3, 4, 2, 5]
```

Für die Berechnung des Funktionswertes im Programm wird es einfacher, falls wir das Horner-Schema verwenden:

$$f(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0 = 3x^3 + 2x^2 + x + 5$$
$$= \big((a_3 x + a_2)x + a_1\big)x + a_0$$

Diese Umformung führt direkt zum untenstehenden Code:

```
from gpanel import *

def polynom(coeff, x):
    summe = 0
    length = len(coeff)
    for a in coeff:
        #mit Horner-Schema berechnen
        summe += a
        if a != coeff[length - 1]:
            summe *= x
    return summe

start = -4.0
end = 4.0
makeGPanel('Graphen', (start)*1.2, (end)*1.2, -(start**4)*1.2,
(end**4)*1.2)
drawGrid(start, end, -start**4, end**4, 16, 16, 'darkgray')

x = start
coeffs = [3, 4, 2, 5]
setColor('blue')
lineWidth(3)
while x < end:
    y = polynom(coeffs, x)
    if x == start:
        move(x, y)
    else:
        draw(x, y)
    x = x + 0.01
```

Beispiel 9
Als Nächstes wird eine Funktion in Python implementiert, welche den Funktionsgraphen der ersten Ableitung eines Polynoms dritter Ordnung zeichnet. Auch hier werden die Koeffizienten als Liste, wie vorhin beschrieben, übergeben.

Die Ableitung eines Polynoms ist:

$$f'^{(x)} = n \cdot a_n x^{n-1} + (n-1) \cdot a_{n-1} x^{n-2} + \cdots + a_1$$

Der letzte Koeffizient a_0 fällt weg und die anderen werden mit dem Exponenten von x multipliziert.
Nehmen wir die Funktion aus dem vorherigen Beispiel:

$$f(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0 = 3x^3 + 4x^2 + 2x + 5$$

Dann ist die Koeffizienten-Liste von f(x):

```
coeff = [3, 4, 2, 5]
length = len(coeff) = 4
```

Daraus können wir mit der Länge die Koeffizienten-Liste der Ableitung berechnen:

```
coeffDerivation = [(length-1)*a3, (length-2)*a2, [(length-3)*a1] =
[3*3, 2*4, 1*2] = [9, 8, 2]
```

a_0 fällt weg, da es eine Konstante ist.

Also können wir die Python-Funktion **`polynom(…)`** aus dem vorherigen Beispiel mit den neuen Koeffizienten der Ableitung wieder verwenden, um den Funktionsgraphen darzustellen und erhalten die folgende Python-Funktion:

```
def polynomDerivation(coeff, x):
    length = len(coeff)
    coeffDerivation = []
    for i in range(length-1):
        coeffDerivation.append(coeff[i]*(length-1-i))
    return polynom(coeffDerivation, x)
```

Aufgabe 29

Berechnen Sie die Nullstelle eines Polynoms dritter Ordnung

$$f(x) = a_3x^3 + a_2x^2 + a_1x + a_0 = 3x^3 + 4x^2 + 2x + 5,$$

indem Sie das Newton-Verfahren anwenden:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Implementieren Sie die Python-Funktion mit Rekursion.

Lösung:

Bevor wir mit dem Programm beginnen, überlegen wir uns die Punkte der Problemlösestrategie.

1. *Beschreiben Sie die Probleminstanz und ermitteln Sie die Problemgrösse N.*
   Wir möchten mit dem Newton-Verfahren die Nullstellen des Polynoms *f(x)* annähern. Dabei wird mit einem beliebigen Startwert x_1 gestartet. Als Problemgrösse N definieren wir hier den Abstand des neu berechneten x-Wertes (x_{n+1}) zum eigentlichen Nullpunkt.
2. *Ermitteln und lösen Sie die trivialste(n) Probleminstanz(en) (N = 1, …).*
   Die trivialste Probleminstanz liegt dann vor, wenn der neu berechnete Wert und die Nullstelle übereinanderliegen und der Abstand somit Null ist, dann gilt auch $f(x_{n+1}) = 0$. Es kann sein, dass die Nullstelle nicht genau getroffen wird, deshalb werden wir die trivialste Probleminstanz leicht anders definieren, nämlich dass der Abstand vom neuen Punkt zur Nullstelle sehr klein ist, also: $|f(x_{n+1})| < \varepsilon; z.B. \varepsilon = 10^{-6}$
3. *Reduzieren Sie die Probleminstanz in eine oder mehrere kleinere Probleminstanzen. (divide)*
   Wir berechnen einen Punkt, der näher an der Nullstelle liegt. Dieser Punkt ist die Nullstelle x_{n+1} der Tangente durch den Punkt x_n an der Funktion *f(x)* (Abbildung 53). Damit reduziert sich die Problemgrösse, da der Abstand zur Nullstelle des Polynoms kleiner wird.
4. *Lösen Sie die kleinere Probleminstanz desselben Problems mit demselben Algorithmus, d.h., verwenden Sie dazu die Rekursion, solange Sie nicht die trivialste Probleminstanz erreicht haben, welche einfach bestimmt werden kann. (impera)*
   Wir berechnen den neuen x-Wert für die kleinere Probleminstanz mit derselben Formel wie für die ursprüngliche Probleminstanz, bis wir die trivialste Probleminstanz erreicht haben, dann wurde die Nullstelle gefunden.

Damit können wir die rekursive Funktion **`calcZero(…)`** herleiten.

```
from gpanel import *

def polynom(coeff, x):
    summe = 0
    length = len(coeff)
    for a in coeff:
        #mit Horner-Schema berechnen
        summe += a
        if a != coeff[length - 1]:
            summe *= x
    return summe

def polynomDerivation(coeff, x):
    length = len(coeff)
    coeffDerivation = []
    for i in range(length-1):
        coeffDerivation.append(coeff[i]*(length-1-i))
    #print(coeffDerivation)
    return polynom(coeffDerivation, x)

def calcZero(coeff, xVal):
    yVal = polynom(coeff, xVal)
    if abs(yVal) < eps:
        return xVal, yVal
    else:
        yValDeriv = polynomDerivation(coeff, xVal)
        xVal = xVal - yVal/yValDeriv
        return calcZero(coeff, xVal)

coeffs = [3, 4, 2, 5]
xStart = 3.0
eps = 0.000001
xZero, yZero = calcZero(coeffs, xStart)
print(xZero, yZero)
```

Aufgabe 30
Testen Sie den Algorithmus für die Parabel

$$f(x) = a_2 x^2 + a_1 x + a_0 = x^2 + 8x - 3$$

und überprüfen Sie, ob das Resultat stimmt.

Lösung:
Das Resultat stimmt, man findet aber nur eine Lösung, welche abhängig vom Startpunkt x_1 ist.

5.3 Transzendente Gleichungen mit Bisektion

Wir betrachten ein zweites Verfahren, die Bisektion oder auch Intervallhalbierungsverfahren genannt, um numerisch eine Nullstelle zu finden.
Mit diesem Verfahren können transzendente Gleichungen gelöst werden, welche keine analytischen Lösungen haben. Dies ist vor allem in den Ingenieurdisziplinen von Interesse, um z.B. in der Regelungstechnik die Polstellen zu berechnen.

Beispiele transzendenter Gleichungen sind:

$$e^x - x - 1 = 0$$
$$\sin(2x) + \ln(x) = 0$$
$$1 + x \cdot \cos(x) = 0$$

Das Verfahren startet mit einem Intervall, in dem sich eine Nullstelle befindet. In der Abbildung 57 wird mit einem Startintervall $[x_1, x_2]$ begonnen, danach wird der Funktionswert *f(x₃)* des x-Wertes (x_3) in der Mitte des Intervalls berechnet.

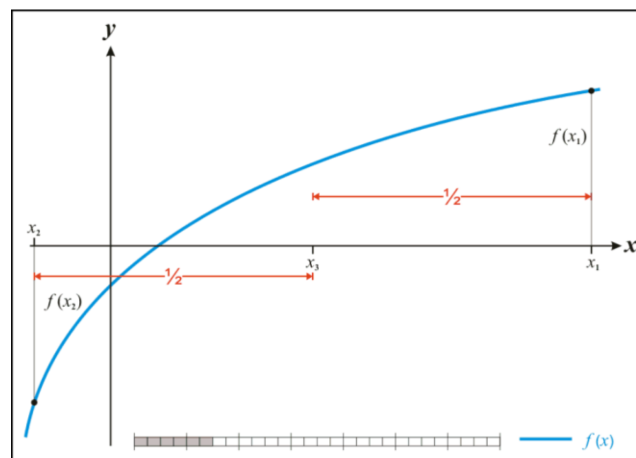


Abbildung 57: Startintervall, welches eine Nullstelle beinhaltet. (Quelle: https://commons.wikimedia.org/wiki/File:Bisektion_Ani.gif)

Nun muss überprüft werden, in welchem Teilintervall sich die Nullstelle befindet. Angenommen wir haben ein Intervall $[l,\ r]$, dann befindet sich in diesem Intervall eine Nullstelle, falls die Funktionswerte an den Intervallgrenzen $f(l)$ und $f(r)$ unterschiedliche Vorzeichen haben.

Der x-Wert in der Mitte des Intervalls wird mit folgender Formel berechnet:

$$x_{new} = x_{left} + \frac{x_{right} - x_{left}}{2}$$

Wir sehen in der Abbildung 57, dass x_3 in der Mitte des Intervalls ist und dass $f(x_1) > 0, f(x_2) < 0$ und $f(x_3) > 0$, somit müssen die neuen Intervallgrenzen x_2 und x_3 sein. Das neue kleinere Intervall ist nun: $[x_2, x_3]$.

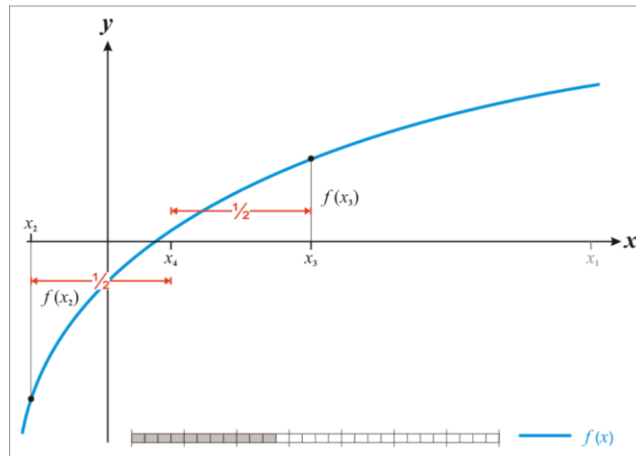


Abbildung 58: Intervall verkleinern - Schritt 1 (Quelle: https://commons.wikimedia.org/wiki/File:Bisektion_Ani.gif)

Das analoge Verfahren wird wieder angewendet (Abbildung 58) und wir erhalten x_4 in der Mitte mit $f(x_3) > 0$, $f(x_2) < 0$ und $f(x_4) > 0$, womit das nächstkleinere Intervall $[x_2, x_4]$ ist, wie in der Abbildung 59 dargestellt.

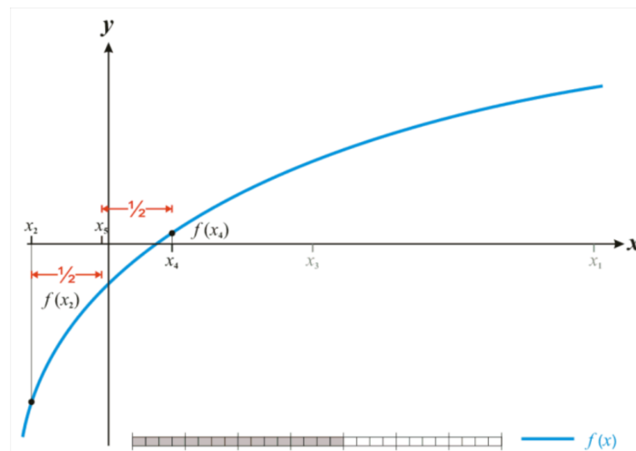


Abbildung 59: Intervall verkleinern - Schritt 2 (Quelle: https://commons.wikimedia.org/wiki/File:Bisektion_Ani.gif)

Dieses Verfahren wird so lange durchgeführt, bis der Wert in der Mitte nahe an oder gleich der Nullstelle ist. Da wir die Nullstelle unter Umständen nicht exakt treffen werden, wählen wir eine Abweichung der Nullstelle z.B. von 10^{-6}.

Nach einigen Schritten finden wir die Lösung (Abbildung 60). Es gilt für dieses Beispiel $f(x7) = 0$.

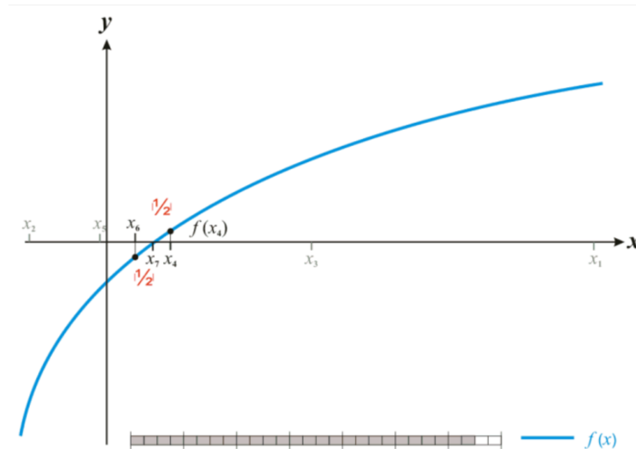


Abbildung 60: Nullstelle gefunden (Quelle: https://commons.wikimedia.org/wiki/File:Bisektion_Ani.gif)

Der Nachteil dieses Verfahrens ist, dass man ungefähr wissen muss, in welchem Bereich die Nullstelle liegt, d.h., wir müssen zuerst die Funktion graphisch darstellen.
Da das Intervall immer wieder halbiert wird und mit demselben Verfahren die Nullstelle im neuen Intervall gesucht wird, werden wir eine Lösung mit Rekursion anstreben.

Beispiel 10
Wir betrachten für unser Beispiel die Gleichung

$$e^x - x - 2 = 0,$$

um die Bisektion mit Rekursion herzuleiten.

Als Erstes müssen wir das Intervall definieren. Dies geht am einfachsten, indem wir den Funktionsgraphen plotten.

$$f(x) = e^x - x - 2$$

Mit folgendem Programm erhalten wir den Plot:

```
from gpanel import *
import math

start = -2.0
end = 2.0
makeGPanel('Graphen', -2.5, 2.2, -2.5, 5)
drawGrid(start, end, -2.0, 4.0, 'darkgray')

x = start
setColor('blue')
lineWidth(3)
while x < end:
    y = math.exp(x) - x - 2
    if x == start:
        move(x, y)
    else:
        draw(x, y)
    x = x + 0.01
```

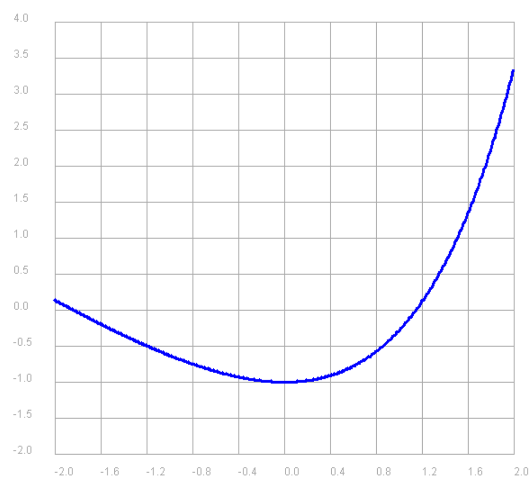


Abbildung 61: Funktionsgraphen

Für eine Nullstelle mit positivem x-Wert starten wir mit dem Intervall $[0.8, 1.6]$.
Um eine Lösung mit Rekursion herzuleiten, wenden wir das Rezept aus dem Kapitel 4.2.1 «Problemlösestrategie mit Rekursion» an.

Rezept:

1. *Beschreiben Sie die Probleminstanz und ermitteln Sie die Problemgrösse N.*
   Mit Hilfe der Bisektion möchten wir eine Nullstelle der Funktion $f(x) = e^x - x - 2$ finden. Dabei beginnen wir mit einem Intervall $[0.8, 1.6]$ und lassen diesen schrumpfen. Als Problemgrösse können wir die Intervallbreite verwenden, in diesem Fall ist somit N = 0.8.
2. *Ermitteln und lösen Sie die trivialste Probleminstanz.*
   Falls die Nullstelle in der Mitte des Intervalls liegt, also falls der Funktionswert beim x-Wert in der Mitte des Intervalls gleich 0 ist, sind wir am Ziel. In andere Worte gefasst heisst das: Da wir die Nullstelle nicht genau treffen werden, soll der Betrag des Funktionswertes *f(x)* beim x-Wert der Intervallmitte kleiner als z.B. $\varepsilon = 10^{-6}$ sein:
   $$|f(x_n)| < \varepsilon$$
3. *Reduzieren Sie die Probleminstanz in eine oder mehrere kleinere Probleminstanzen. (divide)*
   Durch das Halbieren des Intervalls kommen wir immer näher an die Nullstelle und wir können denselben Algorithmus auf dieser Hälfte des Intervalls anwenden. Die Problemgrösse, welche der Intervallbreite entspricht, wird bei jedem Rekursionsschritt halbiert.
4. *Lösen Sie die kleinere Probleminstanz desselben Problems mit demselben Algorithmus, d.h., verwenden Sie dazu die Rekursion, solange Sie nicht die trivialste Probleminstanz erreicht haben, welche einfach bestimmt werden kann. (impera)*
   Die kleinere Probleminstanz wird durch eine Halbierung des Intervalls (Bisektion) erreicht. Diese kleinere Probleminstanz kann wieder mit demselben Algorithmus gelöst werden, was wiederum eine Halbierung des Intervalls bedeutet, bis die trivialste Probleminstanz vorliegt, was uns die Nullstelle liefert.

Bemerkung:

Da wir zur Vereinfachung ein Intervall suchen, in dem es eine Nullstelle gibt, muss an dieser Stelle beachtet werden, dass zwar die Intervallbreite bei jedem Rekursionsschritt kleiner wird, womit die Problemgrösse auch kleiner wird, aber sobald der Wert in der Mitte Nahe beim Nullpunkt ist, wird die Rekursion abgebrochen. Die Problemgrösse spielt in diesem speziellen Fall keine Rolle beim Abbruch. Ein allgemeiner Fall wäre: Falls die Nullstelle in einem beliebigen Intervall gesucht wird, dann müsste die Intervallbreite resp. die Problemgrösse bei der Abbruchbedingung berücksichtigt werden, um den Fall abzufangen, bei dem es keine Nullstelle im Intervall gibt.

Die Lösung der Probleminstanz mittels untenstehendem Code ergibt eine Nullstelle bei $x = 1.14619293213 \; und \; y = -6.19156401704e - 07$ (roter Punkt in Abbildung 62).

```
import math

eps = 1e-6
def bisection(intervalLeft, intervalRight):
    #Mitte
    m = intervalLeft + (intervalRight - intervalLeft) / 2
    yLeft = math.exp(intervalLeft) - intervalLeft - 2
    yRight = math.exp(intervalRight) - intervalRight - 2
    if (yLeft < 0 and yRight < 0) or (yLeft > 0 and yRight > 0):
        raise RuntimeError('invalid Range')
    yM = math.exp(m) - m - 2
    if abs(yM) < eps:
        return m
    elif (yM < 0 and yLeft < 0) or (yM > 0 and yLeft > 0):
        return bisection(m, intervalRight)
    elif (yM < 0 and yRight < 0)or (yM > 0 and yRight > 0):
        return bisection(intervalLeft, m)

print(bisection(0.8, 1.6))
```

Die Zeilen

```
    if (yLeft < 0 and yRight < 0) or (yLeft > 0 and yRight > 0):
        raise RuntimeError('invalid Range')
```

wurden eingefügt, damit das Programm stabil läuft. Dies war aber noch nicht Gegenstand des Unterrichts und es soll auch an dieser Stelle nicht thematisiert werden.

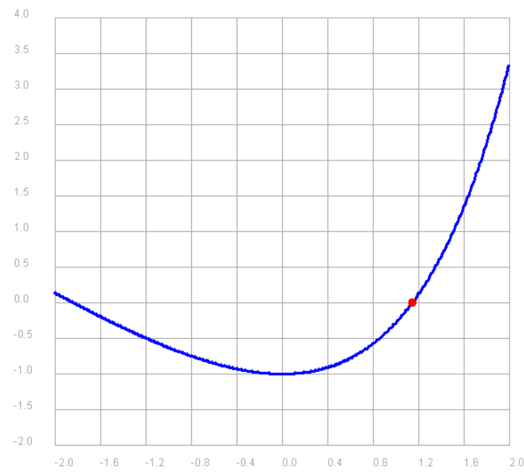


Abbildung 62: Nullstelle

Analyse des Programmes:

Wir starten mit einem Intervallbereich $[0.8, 1.6]$, in dem eine Nullstelle liegt:

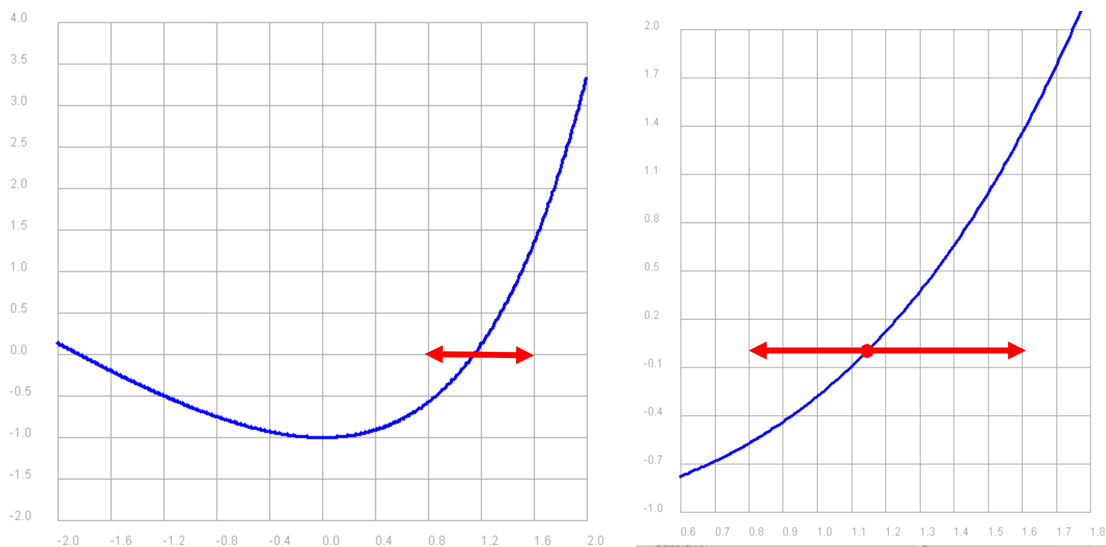


Abbildung 63: Startintervall, rechts ist der Bereich vergrössert.

Danach wird das Intervall halbiert und wir haben zwei Teilintervalle: $[0.8, 1.2]$ und $[1.2, 1.6]$
Nun müssen wir das Intervall aussuchen, in dem die Nullstelle liegt. Um dies zu überprüfen, vergleichen wir die Vorzeichen der Funktionswerte am Intervallbeginn und am Intervallende. Sind beide unterschiedlich, so liegt die Nullstelle darin. Graphisch kann dies sehr schnell gefunden werden, aber im Programm müssen wir die Funktionswerte an den Intervallgrenzen, in der Mitte des Intervalls berechnen und deren Vorzeichen vergleichen, damit dasjenige Teilintervall ausgewählt werden kann, bei welchem die Nullstelle enthalten ist.

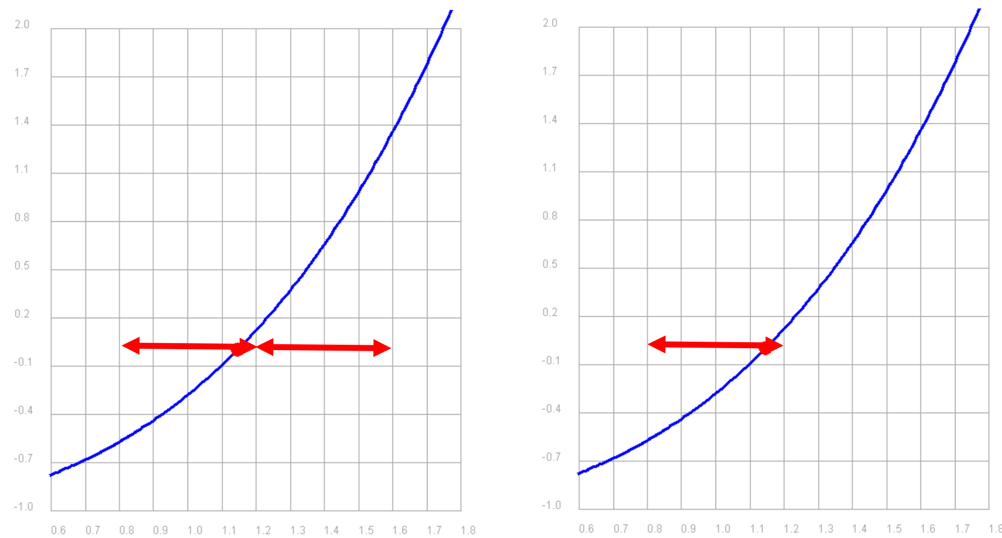


Abbildung 64: Intervall Halbierung

Nun haben wir dieselbe Aufgabe wie zu Beginn, nur mit einem kleineren Intervall, dies wird wieder halbiert usw., bis wir die Lösung gefunden haben.

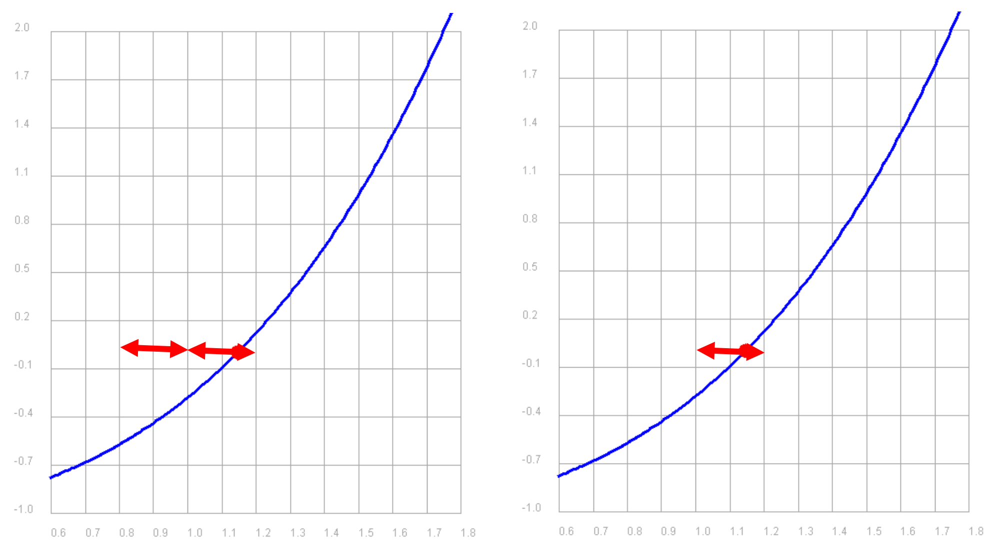


Abbildung 65: Nächste Halbierung

Wir sehen, dass das Intervall immer kleiner wird und somit auch die Problemgrösse. Am Ende liegt ein sehr kleines Intervall vor. Als Abbruchbedingung wurde geprüft, ob der Funktionswert des x-Wertes in der Mitte des Intervalls kleiner als ein ε ist. Man hätte zur Vollständigkeit auch die Intervallbreite in der Abbruchbedingung verknüpfen können, falls der Fall eintritt, dass es keine Nullstelle im Intervall gibt. Dies wurde hier zur Vereinfachung absichtlich weggelassen.

Aufgabe 31
Lösen Sie folgende Gleichung.

$$x + 2 \cdot \cos(x) = 0$$

Gesucht ist eine negative Nullstelle, also auf der linken Halbebene.

Lösung:
Wir gehen analog wie im Beispiel vor.

Zuerst plotten wir die Funktion:

$$f(x) = x + 2 \cdot \cos(x)$$

Der Code ist ähnlich wie im Beispiel, deshalb wird hier nur die Ausgabe dargestellt.

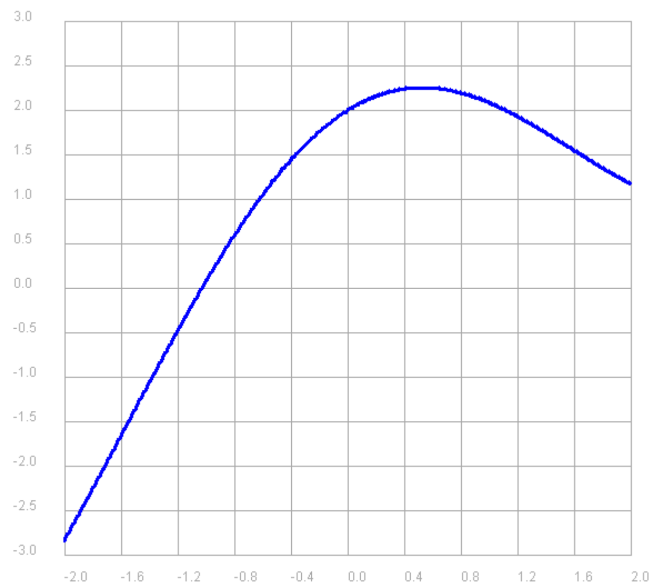


Abbildung 66: Plot der Funktion: f(x) =x+2cos(x)

Danach finden wir mit der Bisektion die Nullstelle in der linken Halbebene.
Das Startintervall definieren wir mit Hilfe des Plots mit $[-1.6, -0.4]$, damit starten wir die Rekursion:

```
import math

eps = 1e-6
def bisection(intervalLeft, intervalRight):
    #Mitte
    m = intervalLeft + (intervalRight - intervalLeft) / 2
    yLeft = intervalLeft + 2*math.cos(intervalLeft)
    yRight = intervalRight + 2*math.cos(intervalRight)
    if (yLeft < 0 and yRight < 0) or (yLeft > 0 and yRight > 0):
        raise RuntimeError('invalid Range')
    yM = m + 2*math.cos(m)
```

```
    if abs(yM) < eps:
        return m
    elif (yM < 0 and yLeft < 0) or (yM > 0 and yLeft > 0):
        return bisection(m, intervalRight)
    elif (yM < 0 and yRight < 0)or (yM > 0 and yRight > 0):
        return bisection(intervalLeft, m)

print(bisection(-1.6, -0.4))
```

Das ergibt eine Nullstelle bei x = -1.02986679077, y = -7.096896113e-07.

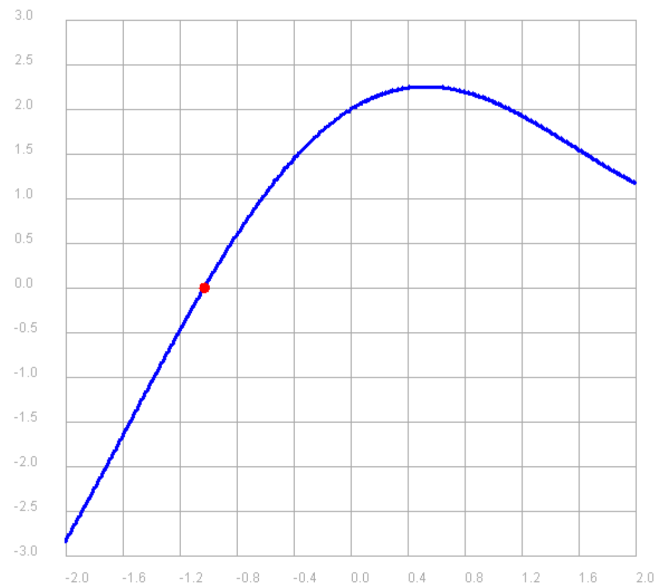


*Abbildung 67: Nullstelle von f(x)=x+2*cos(x)*

Exkurs: `eval(…)`

Wie Sie sicherlich bemerkt haben, ist das Programm der Aufgabe fast identisch wie das in Beispiel 10. Nun werden wir hier eine Möglichkeit zeigen, wie die Funktion **`bisection(intervalLeft, intervalRight)`** für verschiedene Gleichungen angewendet werden kann, ohne immer den Code minimal anzupassen. Dazu verwenden wir in Python den Befehl **`eval(…)`**.
Um zu zeigen, wozu **`eval(..)`** nützlich ist, schreiben wir ein kleines Beispiel.

Beispiel 11

```
import math

a = 4
b = 5

#fkt = 'a * b'
#fkt = 'a**b'
fkt = 'math.cos(a) + math.sqrt(a+b)'

res = eval(fkt)
print(res)
```

eval(…) wertet den Ausdruck, welcher als Text im Parameter steht, aus. Natürlich müssen alle Variablen und Funktionen zuvor definiert resp. importiert werden.
Obiges Programm berechnet den Funktionswert der Funktion:

$$f(a,b) = \cos(a) + \sqrt{a+b}$$

Damit passen wir nun unsere Plot- und unsere Bisektionsfunktion an.

Beispiel 12
Wir nehmen nun eine Anpassung unseres obigen Beispiels mit der **eval(…)** Funktion vor.
Zuerst erstellen wir einen Plot einer Funktion. Den Wertebereich für das GPanel setzen wir von Hand ausserhalb der Funktion, da es sonst zu aufwendig wird, den höchsten Punkt in y-Richtung zu bestimmen. Mit eval(..) geht die einzige Einschränkung einher, dass der Parameter der Funktion danach immer gleich heissen muss. Falls wir den String **fkt = 'x + 2*math.cos(x)'** definieren, so muss der Parameter mit dem aktuellen Wert auch **x** heissen.
Dies ist im untenstehenden Codeausschnitt zu sehen:

```
x = startx
starty = eval(fkt)
```

Die Funktion in **eval(…)** kann nur den Parameter **x**, welcher im String **fkt** definiert ist, auswerten, deshalb muss zuvor **x = startx** angegeben werden, falls man für die Berechnungen sinnvolle Variablennamen verwenden möchte. Sonst hätte man direkt x im Programm verwenden können.

Gesamtes Programm für den Plot:

```
from gpanel import *
import math

startx = -2.0
endx = 2.0
fktNumber = 2
if fktNumber == 1:
    fkt = 'x + 2*math.cos(x)'
elif fktNumber == 2:
    fkt = 'math.exp(x) - x - 2'
x = startx
starty = eval(fkt)
x = endx
endy = eval(fkt)

if fktNumber == 1:
    #für f(x) = x + 2*cos(x)
    makeGPanel('Graphen', -2.5, 2.2, -3.5, 4.0)
    drawGrid(startx, endx, -3.0, 3.0, 10, 12, 'darkgray')
elif fktNumber == 2:
    #für f(x) = e^x - x- 2
    makeGPanel('Graphen', -2.5, 2.2, -2.5, 5)
    drawGrid(startx, endx, -2.0, 4.0, 10, 12, 'darkgray')
```

```
def plotFunction(functionString, start, end):
    x = start
    setColor('blue')
    lineWidth(3)
    while x < end:
        y = eval(functionString)
        if x == start:
            move(x, y)
        else:
            draw(x, y)
        x = x + 0.01
```

Dies ergibt den entsprechenden Plot.
Die Auswahl der Funktion kann im obigen Programm mit dem Flag **`fktNumber`** gewählt werden.

Danach führen wir die Bisektion mit Hilfe von **`eval(…)`** aus:

```
eps = 1e-6
def bisection(functionString, intervalLeft, intervalRight):
    #für eval muss der Parameter jeweils x heissen !!
    #Mitte
    m = intervalLeft + (intervalRight - intervalLeft) / 2
    #für eval muss der Parameter jeweils x heissen !!
    x = intervalLeft
    yLeft = eval(functionString)
    #für eval muss der Parameter jeweils x heissen !!
    x = intervalRight
    yRight = eval(functionString)
    if (yLeft < 0 and yRight < 0) or (yLeft > 0 and yRight > 0):
        raise RuntimeError('invalid Range')
    #für eval muss der Parameter jeweils x heissen !!
    x = m
    yM = eval(functionString)
    if abs(yM) < eps:
        return m
    elif (yM < 0 and yLeft < 0) or (yM > 0 and yLeft > 0):
        return bisection(functionString, m, intervalRight)
    elif (yM < 0 and yRight < 0) or (yM > 0 and yRight > 0):
        return bisection(functionString, intervalLeft, m)
```

Das **vollständige** Programm:

```
from gpanel import *
import math

startx = -2.0
endx = 2.0
fktNumber = 2
if fktNumber == 1:
    fkt = 'x + 2*math.cos(x)'
elif fktNumber == 2:
    fkt = 'math.exp(x) - x - 2'
x = startx
starty = eval(fkt)
x = endx
endy = eval(fkt)

if fktNumber == 1:
    #für f(x) = x + 2*cos(x)
    makeGPanel('Graphen', -2.5, 2.2, -3.5, 4.0)
    drawGrid(startx, endx, -3.0, 3.0, 10, 12, 'darkgray')
elif fktNumber == 2:
    #für f(x) = e^x - x- 2
    makeGPanel('Graphen', -2.5, 2.2, -2.5, 5)
    drawGrid(startx, endx, -2.0, 4.0, 10, 12, 'darkgray')


def plotFunction(functionString, start, end):
    x = start
    setColor('blue')
    lineWidth(3)
    while x < end:
        y = eval(functionString)
        if x == start:
            move(x, y)
        else:
            draw(x, y)
        x = x + 0.01
```

```
eps = 1e-6
def bisection(functionString, intervalLeft, intervalRight):
    #für eval muss der Parameter jeweils x heissen !!
    #Mitte
    m = intervalLeft + (intervalRight - intervalLeft) / 2
    #für eval muss der Parameter jeweils x heissen !!
    x = intervalLeft
    yLeft = eval(functionString)
    #für eval muss der Parameter jeweils x heissen !!
    x = intervalRight
    yRight = eval(functionString)
    if (yLeft < 0 and yRight < 0) or (yLeft > 0 and yRight > 0):
        raise RuntimeError('invalid Range')
    #für eval muss der Parameter jeweils x heissen !!
    x = m
    yM = eval(functionString)
    if abs(yM) < eps:
        return m
    elif (yM < 0 and yLeft < 0) or (yM > 0 and yLeft > 0):
        return bisection(functionString, m, intervalRight)
    elif (yM < 0 and yRight < 0) or (yM > 0 and yRight > 0):
        return bisection(functionString, intervalLeft, m)

plotFunction(fkt, startx, endx)
if fktNumber == 1:
    #für f(x) = x + 2*cos(x)
    xNull = bisection(fkt, -1.6, -0.4)
elif fktNumber == 2:
    #für f(x) = e^x - x- 2
    xNull = bisection(fkt, 0.8, 1.6)
print(xNull)

##Nullstelle:
#für eval muss der Parameter jeweils x heissen !!
x = xNull
yNull = eval(fkt)
print(xNull, yNull)
move(xNull, yNull)
setColor('red')
fillCircle(0.04)
```

Aufgabe 32

Testen Sie das Beispiel aus.

5.4 Türme von Hanoi

Blicken wir nochmals auf die Regeln des Knobelspiels „Türme von Hanoi“:
• Es darf immer nur die oberste Scheibe eines Stapels bewegt werden.
• Es darf nie eine grössere Scheibe auf einer kleineren liegen.

Die Idee ist «divide et impera», welche auch im Rezept vorgeschlagen wird:

1. *Beschreiben Sie die Probleminstanz und ermitteln Sie die Problemgrösse N*.
   Wir möchten einen Turm mit **n** Scheiben und drei Untersetzer vom linken Untersetzer in die Mitte verschieben, unter Einhaltung der obigen Spielregeln. Die Problemgrösse ist die Anzahl Scheiben, also $N = n$.
2. *Ermitteln und lösen Sie die trivialste(n) Probleminstanz(en) (N = 1, …).*
   Ist nur eine Scheibe vorhanden, so ist dies die trivialste Probleminstanz, d.h. hanoi (1) - die Scheibe wird in diesem Fall von links in die Mitte verschoben. Die Problemgrösse ist dann N = 1.
3. *Reduzieren Sie die Probleminstanz in eine oder mehrere kleinere Probleminstanzen*. *(divide)*
   Die Reduktion der Probleminstanz ist, den Turm mit einer Scheibe weniger zu verschieben. Somit haben wir eine Problemgrösse $N_{red} = (n - 1)$.
4. *Lösen Sie die kleinere Probleminstanz desselben Problems mit demselben Algorithmus, d.h., verwenden Sie dazu die Rekursion, solange Sie nicht die trivialste Probleminstanz erreicht haben, welche einfach bestimmt werden kann*. *(impera)*
   Mit der Lösung der kleineren Probleminstanz können wir den gesamten Turm verschieben, indem der obere Teil mit einer Scheibe weniger zum Puffer und die unterste Scheibe zur Mitte verschoben werden. Diese kleinere Probleminstanz kann nun mit demselben Verfahren gelöst werden, sei $k = (n - 1)$, so verschieben wir dann den oberen Teil des Turmes mit $(k - 1)$ Scheiben zum linken Untersetzer und die unterste wieder in die Mitte. Dieses Vorgehen wenden wir an, bis die trivialste Probleminstanz erreicht ist und somit die Probleminstanz gelöst wurde.

Die Punkte, v.a. den Punkt 4, aus dem Rezept werden nun zur Veranschaulichung graphisch dargestellt.

Beispiel der Idee mit n = 5
Ausgangslage:

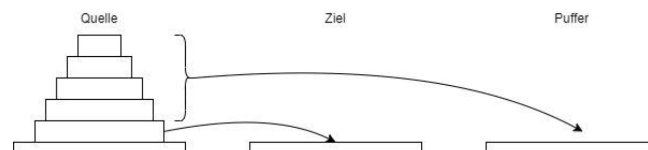


Abbildung 68: Türme von Hanoi - Ausgangslage

Wir tun so, als würden wir wissen, wie man den oberen Teil des Turmes, bestehend aus $n - 1$ Scheiben, unter Einhaltung der Regeln verschieben kann und erreichen die erste Verschiebung der oberen $n - 1$ Scheiben zum Puffer, wie in Abbildung 68 angedeutet ist, und in Abbildung 69 die Situation nach dem Schieben.

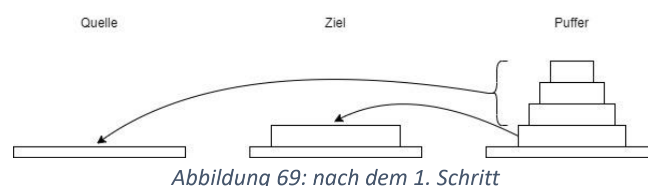


Abbildung 69: nach dem 1. Schritt

In der Abbildung 69 sieht es so aus, als hätten wir die Spielregeln verletzt, aber wir gehen davon aus, dass wir wissen, in welchen Schritten unter Einhaltung der Regeln verschoben werden kann. Somit ist dies ein gültiger Spielzug.

Danach verschieben wir, wie in Abbildung 69 dargestellt, die oberen Scheiben ausser der untersten wieder zurück zur Quelle und erhalten die Darstellung von Abbildung 70.

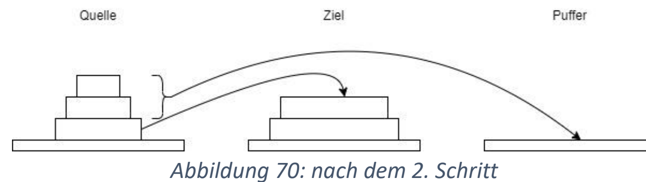


Abbildung 70: nach dem 2. Schritt

Jetzt wiederholt sich das Ganze, der obere Teil geht vom Stapel links nach rechts und die unterste Scheibe in die Mitte. Wir erhalten die Darstellung in der Abbildung 71.

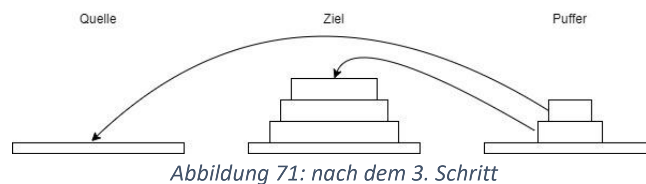


Abbildung 71: nach dem 3. Schritt

Und wieder geht die oberste Scheibe geht nach links und die unterste in die Mitte.
Nach diesem Schritt müssen wir nur noch eine Scheibe, dies ist die trivialste Probleminstanz, in die Mitte schieben und wir haben nun alle Scheiben im Ziel.

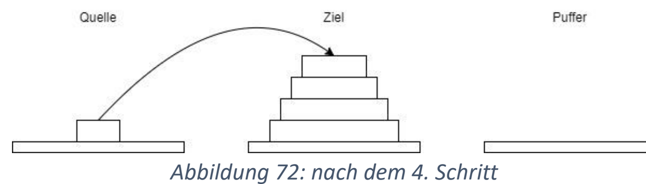


Abbildung 72: nach dem 4. Schritt

Ist das die Lösung?
Es sieht so aus, als hätten wir die Probleminstanz nicht gelöst, doch in der Tat haben wir nun die Lösung vorliegen. Wir sollten die Idee, die hier der Rekursion zugrunde liegt, genau durchdenken. Wir reduzieren das Spiel in jedem Schritt um eine Scheibe und lösen am Ende die trivialste Probleminstanz, dies entspricht der Idee «divide et impera».

Betrachten wir die Idee genauer, damit wir daraus einen Pseudocode erstellen können. Dafür werden wir der Vereinfachung halber das Spiel mit drei Scheiben (`anz = 3`) durchdenken. Der Funktionsaufruf für dieses Spiel lautet:

```
hanoi(von, nach, puffer, anz)
```

oder mit unseren Bezeichnungen aus der Abbildung 73:

```
hanoi(«Quelle», «Ziel», «Puffer», 3)
```

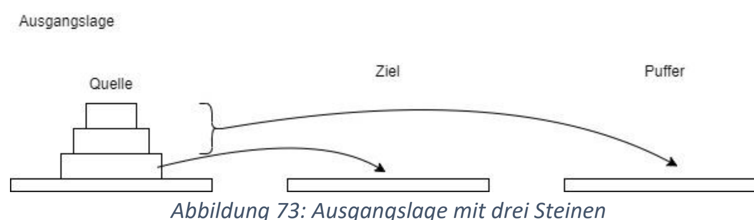


Abbildung 73: Ausgangslage mit drei Steinen

In diesem Aufruf werden zuerst die oberen zwei Scheiben zum Puffer verschoben und die unterste in die Mitte verschoben. Genauer betrachtet möchten wir eine Probleminstanz der «Türme von Hanoi» mit $anz = anz - 1 = 2$ Scheiben lösen, aber der Stapel soll zum Puffer und nicht in die Mitte verschoben werden (siehe Abbildung 73). Mit der obigen Funktion und den obigen Parametern entspricht dies dem folgenden Aufruf, wobei hier direkt die Bezeichnungen der Untersetzer verwendet werden:

```
hanoi(«Quelle», «Puffer», «Ziel», 2)
```

Geschrieben mit den Parametern des Aufrufes:

```
hanoi(von, puffer, nach, anz-1)
```

Der zweite Schritt innerhalb des Aufrufes der Funktion

```
hanoi(«Quelle», «Ziel», «Puffer», 3)
```

wird die unterste Scheibe in die Mitte verschoben. Dies kann mit dem Aufruf

```
hanoi(«Quelle», «Ziel», «Puffer», 1)
```

oder mit den Parametern des ersten Aufrufes geschrieben

```
hanoi(von, nach, puffer, 1)
```

geschehen.
Also ruft die ursprüngliche Funktion **hanoi(...)** für das Lösen dieser kleineren Probleminstanz sich selbst wieder zweimal auf, da sehen wir bereits die Rekursion.

Nach diesen Schritten erhalten wir die Situation wie in Abbildung 74.

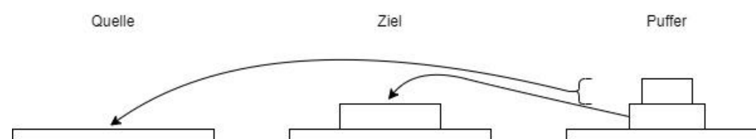


Abbildung 74: nach dem ersten Schritt

Dies sieht ähnlich aus, wie in der Ausgangslage, allerdings mit drei Unterschieden. Erstens sind die Scheiben im «Puffer» und müssen, mit «Quelle» als Puffer zum «Ziel» geschoben werden, zweitens haben wir nur noch $anz - 1 = 2$ Scheiben, welche verschoben werden müssen und drittens ist eine Scheibe bereits am Ziel platziert. Der erste und zweite Punkt kann mit dem folgenden Aufruf erledigt werden:

```
hanoi(«Puffer», «Ziel», «Quelle», 2)
```

Verwendet man die Parameter aus dem ursprünglichen Aufruf der Funktion, so kann man Folgendes schreiben:

```
hanoi(puffer, nach, von, anz-1)
```

Damit die obige Erklärung zusammenhängend dargestellt werden kann, wird das Fragment der Funktion als Pseudocode nochmals eingefügt:

```
hanoi(«Quelle», «Ziel», «Puffer», 3)
            hanoi(«Quelle», «Puffer», «Ziel», 2)
            hanoi(«Quelle», «Ziel», «Puffer», 1)
            hanoi(«Puffer», «Ziel», «Quelle», 2)
```

Nach diesen Rekursionsschritten haben wir wieder folgende Situation:

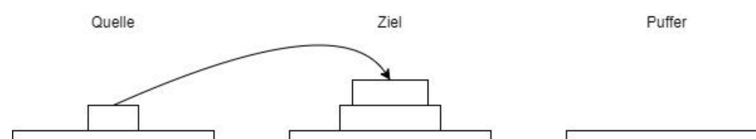


Abbildung 75: nach dem zweiten Schritt

Wir müssen nur noch eine Scheibe von der «Quelle» zum «Ziel» schieben, was der trivialsten Probleminstanz entspricht.

```
hanoi(«Quelle», «Ziel», «Puffer», 1)
```

Mit den Parametern des ersten Aufrufs geschrieben gilt:

```
hanoi(von, nach, puffer, 1)
```

Fassen wir nun die Überlegungen zu einem **Pseudocode** zusammen:
`von, nach, puffer` sind Strings,
z.B. ist `von = «Quelle»`, `nach = «Ziel»` und `puffer = «Puffer».`
Der Algorithmus, welcher in der Funktion

```
hanoi(von, nach, puffer, anz)
```

implementiert wird, kann die Probleminstanz «Türme von Hanoi» mit `anz` Scheiben lösen. Dafür werden die oberen Scheiben bis auf die unterste zum Puffer mit dem folgenden Funktionsaufruf verschoben:

```
hanoi(von, puffer, nach, anz-1)
```

Und die unterste Scheibe wird mit folgendem Aufruf ins Ziel verschoben:

```
hanoi(«Quelle», «Ziel», «Puffer», 1)
```

oder mit den Parametern:

```
hanoi(von, nach, puffer, 1)
```

Bleibt die kleinere Probleminstanz mit ($anz - 1 = 2$) Scheiben im Puffer, welche ins Ziel geschoben werden sollen, lösen wir mit

```
hanoi(«Puffer», «Ziel», «Quelle», anz-1)
```

oder mit:

```
hanoi(puffer, nach, von, anz-1)
```

Die trivialste Probleminstanz liegt vor, falls wir nur eine Scheibe schieben müssen.
Somit werden wir eine rekursive Funktion schreiben, welche sich dreimal aufruft.

Der **Pseudocode**:

```
hanoi(von, nach, puffer, anz)
     falls anz == 1:
          Ausgabe: «Scheibe von «von» nach «nach» verschieben
     sonst falls anz >= 2:
          # oberer Teil von Quelle (links) nach Puffer (rechts)
          hanoi(von, puffer, nach, anz-1)

          # unterste Scheibe von Quelle (links)
          # nach Ziel (Mitte)
          hanoi(von, nach, puffer, 1)

          # Quelle und Puffer tauschen und
          # dasselbe nochmals für den Rest
          hanoi(puffer, nach, von, anz-1)
```

Aufruf der Funktion:

```
hanoi(«Quelle, «Ziel», «Puffer», 3)
```

Dann das **Python-Programm**

```
def hanoi(von, nach, puffer, n):
    if n==1:
        print('Scheibe von ' + von + ' nach ' + nach + ' schieben.')
    else:
        hanoi(von, puffer, nach, n-1)
        hanoi(von, nach, puffer, 1)
        #puffer und quelle vertauschen gegenüber Funktionskopf
        hanoi(puffer, nach, von, n-1)


von = 'Quelle'
nach = 'Ziel'
puffer = 'Puffer'

hanoi(von, nach, puffer, 3)
```

Ausgabe:

```
Scheibe von Quelle nach Ziel schieben.
Scheibe von Quelle nach Puffer schieben.
Scheibe von Ziel nach Puffer schieben.
Scheibe von Quelle nach Ziel schieben.
Scheibe von Puffer nach Quelle schieben.
Scheibe von Puffer nach Ziel schieben.
Scheibe von Quelle nach Ziel schieben.
```

Aufgabe 33

Erstellen Sie daraus graphisch die Aufrufsequenz der rekursiven Aufrufe der Funktion:

```
hanoi(von, nach, puffer, 3)
hanoi(«Quelle», «Ziel», «Puffer», 3)
```

Um die Analyse zu vereinfachen, können Sie das Python-Programm im Debugger laufen lassen.

Lösung:

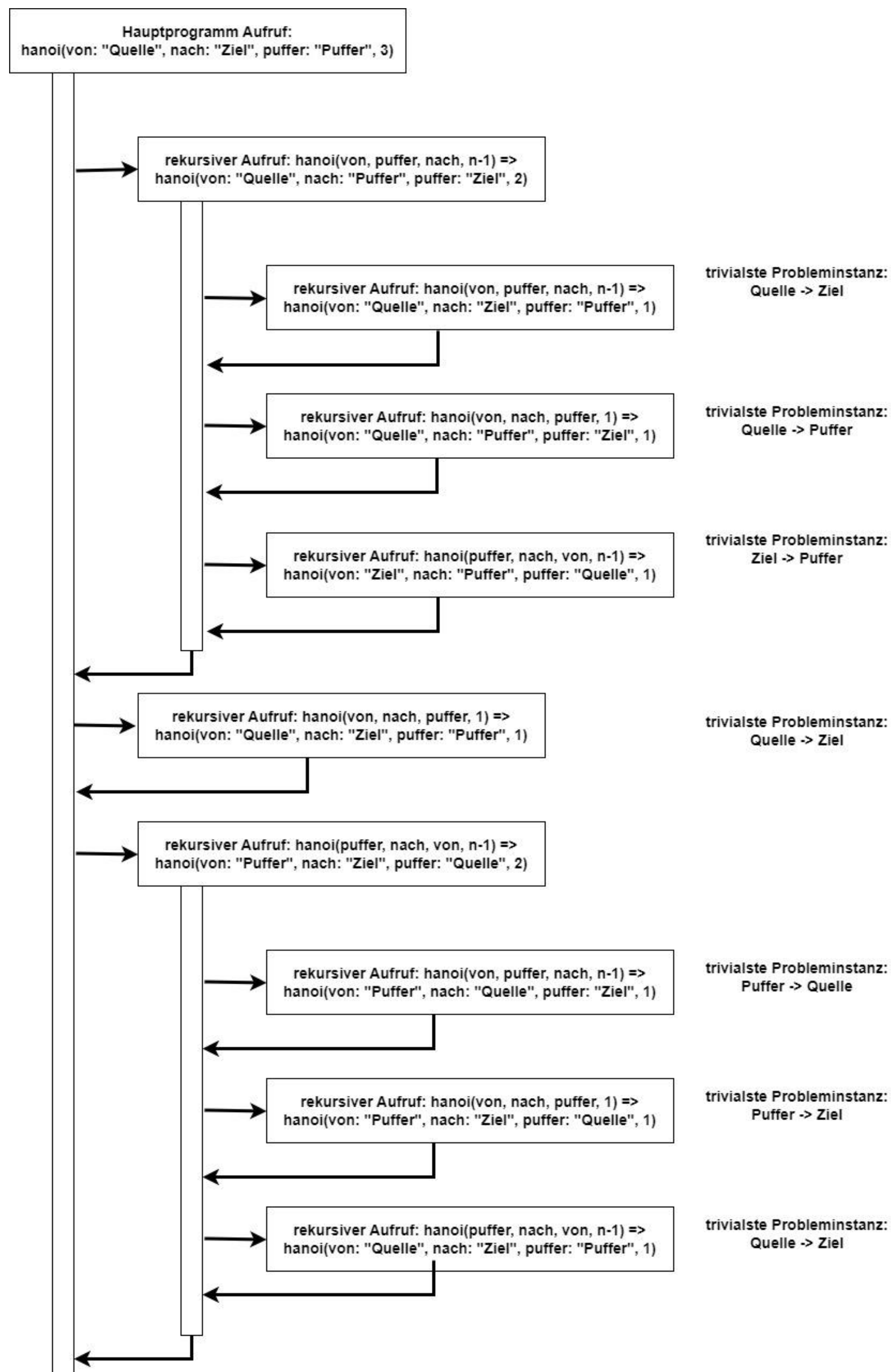


Abbildung 76: rekursive Aufrufe der Funktion hanoi(…) für die Lösung der Türme von Hanoi mit 3 Scheiben

Aufgabe 34

1. Lassen Sie das Python-Programm mit 4 Scheiben ($anz = 4$) laufen und führen Sie alle Schritte auf der untenstehenden Webseite von Hand aus. Man kann dort die Steine mit der Maus verschieben.
2. Vergleichen Sie danach, ob die Lösung, welche mit dem Knopf Lösung gezeigt wird, dieselbe ist wie die Variante des Python-Programms.

Das Knobelspiel «Türme von Hanoi» ist als Graphik online verfügbar:
https://www.bernhard-gaul.de/spiele/tower/tower.php

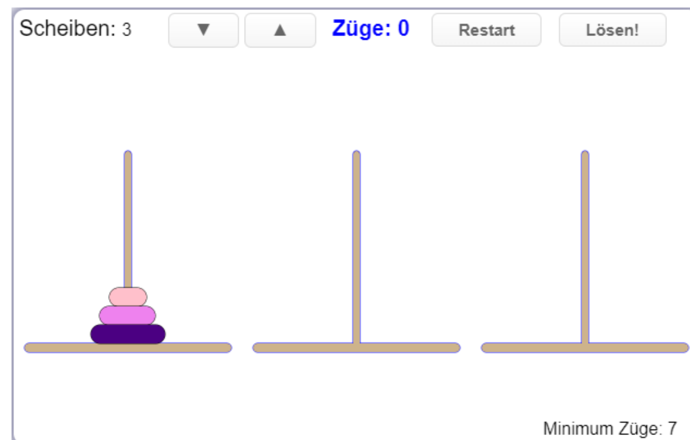


Abbildung 77: Türme von Hanoi (Quelle: https://www.bernhard-gaul.de/spiele/tower/tower.php)

Bemerkung:
Auf den ersten Blick sieht es so aus, als würden wir die Probleminstanz nicht richtig lösen, aber durch die Aufteilung in kleinere Probleminstanzen wird die Rekursion sehr hilfreich. Dabei wurde die grössere Probleminstanz als Blackbox betrachtet, bis wir die trivialste Probleminstanz erhalten haben. Die Blackbox wurde dabei nie direkt programmiert, sondern die Probleminstanz wurde immer verkleinert.
Das Besondere an der obigen Lösung ist, dass diese Probleminstanz nur sehr umständlich mittels Iteration gelöst werden kann. Hingegen ergibt sich mittels Rekursion ein einfacher eleganter Code.

5.5 Binärer Suchbaum

In diesem Kapitel gehen wir davon aus, dass wir einen binären Baum und einen binären Suchbaum als Datenstruktur in den vorgängigen Lektionen eingeführt haben.
Zur Erinnerung sagen wir zu den Werten in den Knoten Schlüssel, damit dies allgemein gültig ist. Bei einem Schlüssel sind die Beziehungen grösser, kleiner und gleich jeweils definiert und wir müssen uns nicht darum kümmern.
Der binäre Suchbaum ist so aufgebaut, dass links von der Wurzel nur Knoten vorhanden sind mit kleinerem Schlüssel als der Schlüssel in der Wurzel selbst und rechts der Wurzel alle Knoten mit grösserem oder gleichem Schlüssel als der Schlüssel in der Wurzel. Dieser Aufbau gilt auch für jeden Knoten, dabei wird der obere Knoten Vater genannt und die Unterknoten linkes und rechtes Kind (left and right child). Ein Algorithmus kann herausfinden, wann er einen untersten Knoten besucht, indem er die Kanten prüft. Hat ein Knoten keine Kanten und somit keine Kinder, so ist dies ein Knoten der untersten Ebene.
Wir gehen davon aus, dass wir einen bereits gefüllten binären Suchbaum vorliegen haben, durch den navigiert werden soll, z.B. sollen alle Knoten besucht werden und als Folge aller Schlüssel in der besuchten Reihenfolge ausgegeben werden. Danach betrachten wir ein zweites Beispiel, indem wir einen Schlüssel im Baum suchen werden. Zur Erinnerung ist in der Abbildung 78 ein binärer Suchbaum für die Zahlen: 12, 4, 17, 15, 8, 23, 3 aufgebaut.

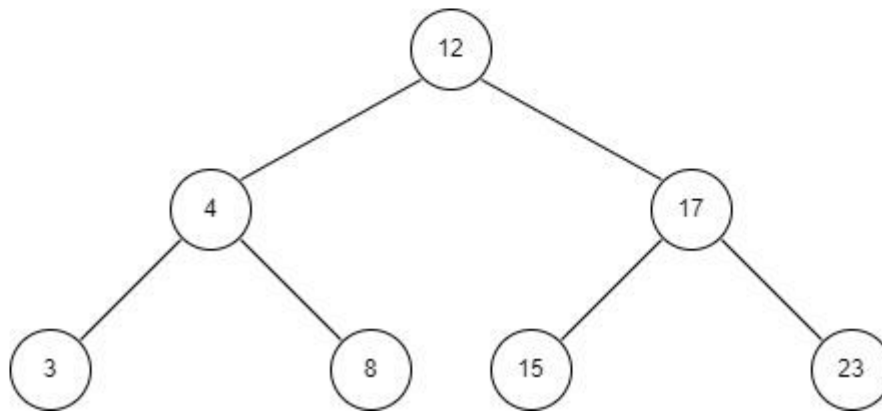


Abbildung 78: binärer Suchbaum

5.5.1 Navigieren

Unsere erste Aufgabe ist, durch den Baum so zu navigieren, sodass wir jeden Knoten besuchen. Dabei gibt es drei verschiedene Methoden, den Baum zu durchqueren: die **Preorder-**, **Inorder-** und **Postorder**-Reihenfolge.

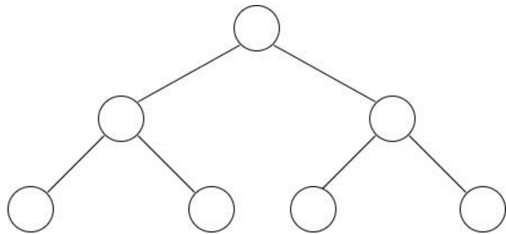
Preorder-Reihenfolge
Bei dieser Reihenfolge besucht der Algorithmus den obersten Knoten, die Wurzel. Der Algorithmus sieht die Kanten der Wurzel und kann damit den linken Teilbaum mit dem linken Kind als Wurzel holen und darauf rekursiv denselben Algorithmus anwenden, bis alle Knoten besucht worden sind. Danach wird analog der rechte Teilbaum durchsucht.

Postorder-Reihenfolge
Der Algorithmus sieht die Kanten der Wurzel und kann damit den linken Teilbaum mit dem linken Kind als Wurzel und darauf rekursiv denselben Algorithmus anwenden, bis alle Knoten besucht worden sind. Danach wird analog der rechte Teilbaum durchsucht. Und zuletzt wird die Wurzel des Baumes besucht.

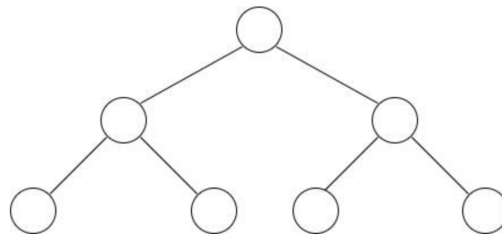
Inorder-Reihenfolge
Der Algorithmus sieht die Kanten der Wurzel und kann damit den linken Teilbaum mit dem linken Kind als Wurzel und darauf rekursiv denselben Algorithmus anwenden, bis alle Knoten besucht worden sind. Danach wird die Wurzel und anschliessend analog zum linken Teilbaum der rechte Teilbaum besucht.

Aufgabe 35
In den obigen Beschreibungen wurde auf eine Graphik verzichtet, nun ist es Ihre Aufgabe, die entsprechenden Grafiken zu den obigen Beschreibungen zu erstellen und sie kurz zu dokumentieren. Sie können die Reihenfolge der besuchten Knoten mit aufsteigenden Zahlen beginnend bei 1 im Baum markieren.
Nehmen Sie an, dass es sich um den in Abbildung 78 gezeigten Baum handelt, welcher für die drei Reihenfolgen ohne Schlüssel unten nochmals kopiert ist.

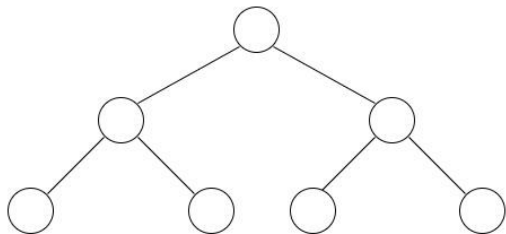
Preorder:



Postorder

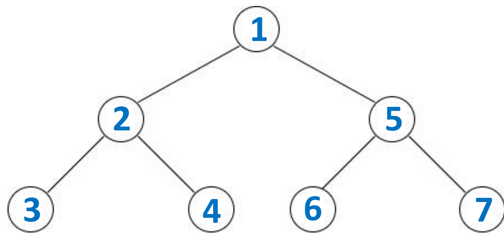


Inorder



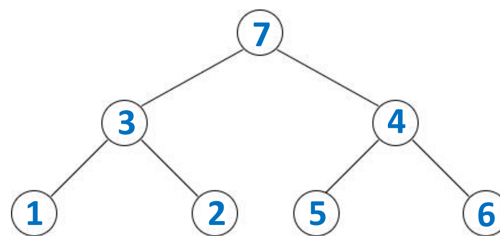
Lösung
Den Knoten, bei dem wir starten, nennen wir **r**. Falls wir zuoberst beginnen, ist das die **Wurzel**, auf Englisch **root**, weshalb die Bezeichnung r für den Baum und für die jeweilige „Wurzel“ der Teilbäume verwendet wird.

Preorder:



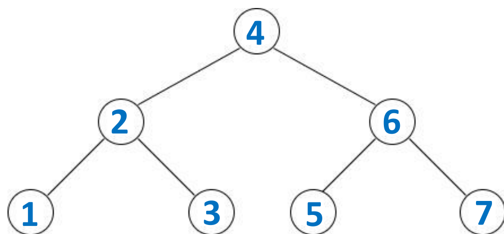
1. Besuchen Sie den Knoten **r**.
2. Durchlaufen Sie den linken Teilbaum des Knotens **r** in der Preorder-Reihenfolge.
3. Durchlaufen Sie den rechten Teilbaum des Knotens **r** in der Preorder-Reihenfolge.

Postorder



1. Durchlaufen Sie den linken Teilbaum des Knotens **r** in der Postorder-Reihenfolge.
2. Durchlaufen Sie den rechten Teilbaum des Knotens **r** in der Postorder-Reihenfolge.
3. Besuchen Sie den Knoten **r**.

Inorder



1. Durchlaufen Sie den linken Teilbaum des Knotens **r** in der Inorder-Reihenfolge.
2. Besuchen Sie den Knoten **r**.
3. Durchlaufen Sie den rechten Teilbaum des Knotens **r** in der Inorder-Reihenfolge.

Die Ähnlichkeit der Schritte aller Reihenfolge ist gross, d.h., es würde ausreichen, einen Algorithmus für eine Methode zu überlegen, die anderen können dann praktisch kopiert werden.

Aufgabe 36

Erstellen Sie für die Aufgabe 35 das Rezept für die Rekursion. Die Punkte des Rezeptes sind unten aufgeführt, als Probleminstanz können wir einen Baum der Tiefe drei annehmen, analog zur Aufgabe 35.

1. *Beschreiben Sie die Probleminstanz und ermitteln Sie die Problemgrösse N.*
2. *Ermitteln und lösen Sie die trivialste(n) Probleminstanz(en) (N = 1, …).*
3. *Reduzieren Sie die Probleminstanz in eine oder mehrere kleinere Probleminstanzen. (divide)*
4. *Lösen Sie die kleinere Probleminstanz desselben Problems mit demselben Algorithmus, d.h., verwenden Sie dazu die Rekursion, solange Sie nicht die trivialste Probleminstanz erreicht haben, welche einfach bestimmt werden kann. (impera)*

Lösung:

1. *Beschreiben Sie die Probleminstanz und ermitteln Sie die Problemgrösse N.*
   Es soll durch den binären Suchbaum aus der Aufgabe 35 navigiert werden. Wir verwenden die Tiefe des Baumes als Problemgrösse, somit gilt: N = 2, da die Wurzel die Tiefe Null hat.
2. *Ermitteln und lösen Sie die trivialste Probleminstanz (N = 0).*
   Die trivialste Probleminstanz liegt dann vor, wenn der Knoten kein linkes und kein rechtes Kind hat. In diesem Fall ist die Tiefe des Teilbaumes gleich Null und die Problemgrösse: $N = 0$. D.h., wir müssen nur diesen Knoten besuchen.
   Bemerkung: In diesem Fall liefern `left(r)` und `right(r)` Null.
3. *Reduzieren Sie die Probleminstanz in eine oder mehrere kleinere Probleminstanzen. (divide)*
   Die Reduktion besteht darin, dass jeweils nur durch den linken oder den rechten Teilbaum navigiert wird, was wiederum mit demselben Algorithmus gelöst werden kann. Diese Probleminstanz hat nun die Problemgrösse $N_{red} = N - 1 = 1$.
4. *Lösen Sie die kleinere Probleminstanz desselben Problems mit demselben Algorithmus, d.h., verwenden Sie dazu die Rekursion, solange Sie nicht die trivialste Probleminstanz erreicht haben, welche einfach bestimmt werden kann. (impera)*
   Falls der Algorithmus die Kanten sieht und damit eine Referenz auf die Teilbäume geholt werden kann, kann das Navigieren durch den Teilbaum mit demselben Algorithmus gelöst werden, man kann den obersten Knoten des Teilbaumes als neue Wurzel verwenden. Dieses Vorgehen wird rekursiv wiederholt, bis die trivialste Probleminstanz gefunden wird und wir mit der Navigation in einem Teilbaum fertig sind.

Aufgabe 37

Schreiben Sie eine Funktion mit Pseudocode, um die Preorder-, Postorder- und Inorder-Reihenfolge zu implementieren. Die Funktion `left(r)` liefert eine Referenz auf den linken Teilbaum, die Funktion `right(r)` liefert eine Referenz auf den rechten Teilbaum vom Knoten r aus gesehen.

Lösung

```
def preorder(root r)
#Durchläuft alle Knoten des Binärbaumes mit Wurzel (root) r
#in der Preorder-Reihenfolge
     if r != 0:
          Besuche den Knoten r
          preorder(left(r))
          preorder(right(r))
```

```
def postorder(root r)
#Durchläuft alle Knoten des Binärbaumes mit Wurzel (root) r
#in der Preorder-Reihenfolge
     if r != 0:
          postorder (left(r))
          postorder (right(r))
          Besuche den Knoten r
```

```
def inorder(root r)
#Durchläuft alle Knoten des Binärbaumes mit Wurzel (root) r
#in der Inorder-Reihenfolge
     if r != 0:
          inorder (left(r))
          Besuche den Knoten r
          inorder (right(r))
```

5.5.2 Suche in einem binären Suchbaum

Wir suchen einen bestimmten Schlüssel in einem binären Suchbaum. Ist der Suchbaum korrekt aufgebaut, so können wir wie folgt vorgehen:

1. Wir starten mit der Wurzel (root). Ist der zu suchende Schlüssel gleich dem Schlüssel in der Wurzel, so haben wir den Schlüssel gefunden und sind fertig.
2. Ansonsten, falls der Schlüssel kleiner ist als derjenige in der Wurzel, suchen wir im linken Teilbaum.
3. Falls er grösser ist als der Schlüssel in der Wurzel, so suchen wir im rechten Teilbaum.
4. In den Teilbäumen gehen wir genau gleich die Punkte 1. bis 3. durch, bis wir den Schlüssel gefunden haben oder bis wir alle Knoten des Baumes besucht haben und der Schlüssel nicht im Baum enthalten ist.
5. Ein unterster Knoten ist erreicht, falls es keinen Teilbaum mehr gibt.

Aufgabe 38

Nun sollen Sie den obigen Ablauf mit Pseudocode in einer Funktion **`searchBinTree(…)`** implementieren. Dabei können Sie wiederum den binären Suchbaum aus der Abbildung 78 mit der Tiefe N = 2 verwenden. Der Algorithmus ist aber auch gültig für Bäume mit anderen Tiefen. In dieser Aufgabe müssen Sie das Rezept nicht herleiten, da es sehr ähnlich ist wie das für die Navigation.

Hinweis:

Ein Objekt **`node`** besteht aus einem Schlüssel, welcher mit der Methode **`getKey()`** gelesen werden kann, und mit **`left()`** resp. **`right()`** erhält man als Rückgabewert eine Referenz auf den linken resp. rechten Knoten.

Wir halten fest:

`node.getKey()` gibt den Schlüssel zurück.
`node.left()` gibt eine Referenz auf den linken Knoten zurück oder NULL, falls es links keinen Knoten gibt.
`node.right()` gibt eine Referenz auf den rechten Knoten zurück oder NULL, falls es rechts keinen Knoten gibt.

Lösung

```
searchBinTree(node, searchVal):
     if node == NULL:
          return False
     if node.getKey() == searchVal:
          return True
     elif searchVal < node.getKey():
          searchBinTree(node.left(), searchVal)
     elif searchVal > node.getValue():
          searchBinTree(node.right(), searchVal)
```

5.6 Mergesort

5.6.1 Repetition des Algorithmus

Zur Auffrischung des Gedächtnisses wird das Mergesort-Verfahren in wenigen Worten beschrieben. Eine ausführliche Beschreibung finden Sie unter: (Ottmann & Widmayer, 2012, S. 112)
Beim Mergesort soll eine Liste sortiert werden, indem die Liste halbiert wird, dann jeweils diese Hälften wieder mit Mergesort sortiert werden. Sobald wir bei der Halbierung eine Liste mit nur einem Element haben, werden die Listen in der richtigen Reihenfolge wieder verschmolzen (merge). Diese sehr kurze Beschreibung wird anhand eines Beispiels erläutert.

Beispiel 13
Es soll folgende Folge mit Mergesort sortiert werden:

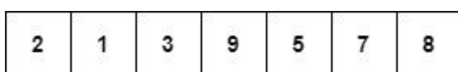


Abbildung 79: unsortierte Liste

In dem ersten Schritt wird die Liste halbiert. Da die obige Liste eine ungerade Anzahl hat, wird die linke Hälfte die Länge 4 haben und die rechte die Länge 3. Wir werden die Mitte mit den Indexen berechnen; in unserer Liste haben wir die Indexe **`start = 0`** bis **`end = 6`**, somit berechnet sich der Index des mittleren Elementes:

$$m = start + \frac{end - start}{2} = start - \frac{start}{2} + \frac{end}{2} = \frac{start + end}{2}$$

Für diesen ersten Fall gilt dann:

$$m = \frac{start + end}{2} = \frac{0 + 6}{2} = 3$$

Index 0 bis 3 gehören zur linken Liste, Index 4 bis 6 zur rechten.

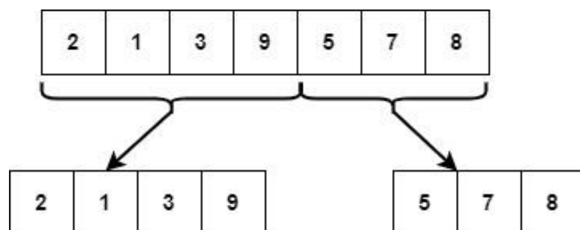


Abbildung 80: Liste aufteilen

Danach wird dasselbe Verfahren mit den Teillisten durchgeführt:

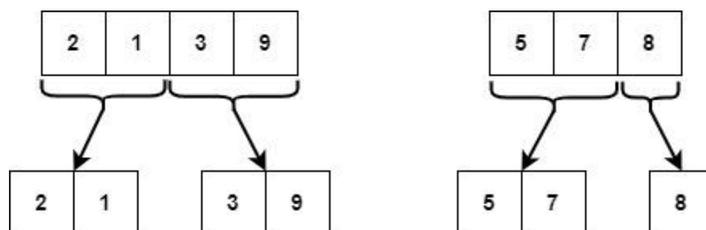


Abbildung 81: Teillisten wieder aufteilen

Das geschieht so lange, bis wir Listen haben mit nur einem Element.

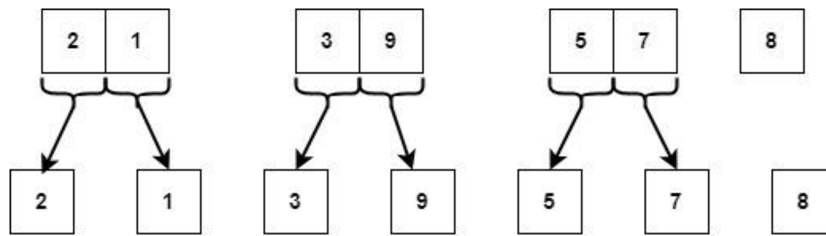


Abbildung 82: Listen mit nur noch einem Element

Nun haben wir sieben «sortierte» Listen, welche wir in die vorgängigen Teillisten sortiert vereinigen (merge) werden. Beim Vereinigen beginnen wir mit den Teillisten mit nur einem Element und fügen zuerst das kleinere Element der nächstoberen Teilleiste hinzu. In Abbildung 83 ist dies veranschaulicht.

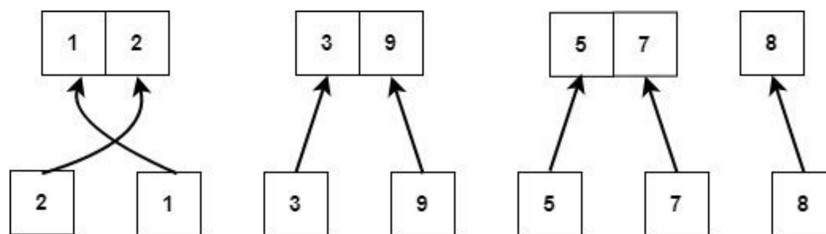


Abbildung 83: Listen sortiert vereinigen

Danach führen wir wieder dasselbe durch mit der nächsten Stufe, wir fügen wiederum zuerst das kleinere Element in die nächstobere Teilliste ein.

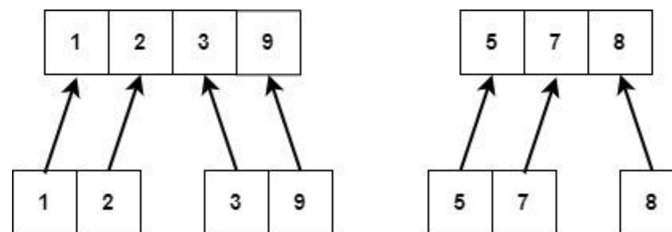


Abbildung 84: weiter vereinigen

Wir wiederholen den Vorgang des Zusammenführens:

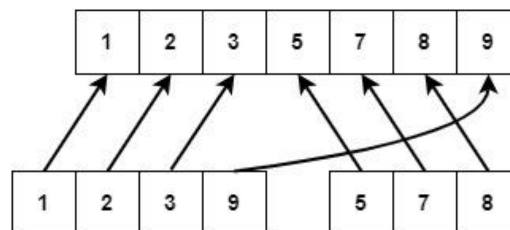


Abbildung 85: Liste ist sortiert

5.6.2 Algorithmus

Das obige Beispiel zeigt, dass man zuerst die Liste teilen muss und danach sortiert verschmelzen. Deshalb beginnen wir zuerst mit dem Teilen und danach fahren wir mit der Vereinigung resp. mit dem Verschmelzen fort.

Liste aufteilen:
Die Liste wird in mehrere gleiche Schritte aufgeteilt. Dies zeigt, dass die Lösung mittels Rekursion möglich ist. Also wenden wir das Rezept an:

1. *Beschreiben Sie die Probleminstanz und ermitteln Sie die Problemgrösse N*.
   Es soll eine Liste mit n Elementen aufgeteilt werden. Somit liegt eine Problemgrösse $N = n$ vor.
2. *Ermitteln und lösen Sie die trivialste(n) Probleminstanz(en) (N = 1, …).*
   Die trivialste Probleminstanz ist eine Liste mit einem Element, also $N = 1$.
3. *Reduzieren Sie die Probleminstanz in eine oder mehrere kleinere Probleminstanzen. (divide)*
   Die kleineren Probleminstanzen ergeben sich durch das Halbieren der Liste. Dann liegt eine Probleminstanz für die linke Hälfte und eine für die rechte Hälfte vor, jeweils mit der Problemgrösse $N_{red} = {}^{n}/_{2}$.
4. *Lösen Sie die kleinere Probleminstanz desselben Problems mit demselben Algorithmus, d.h., verwenden Sie dazu die Rekursion, solange Sie nicht die trivialste Probleminstanz erreicht haben, welche einfach bestimmt werden kann. (impera)*
   Jede kleinere Probleminstanz kann gleich gelöst werden wie die ursprüngliche Probleminstanz und die Problemgrösse wird jeweils halbiert. Damit können die einzelnen Teillisten mit demselben Verfahren bearbeitet werden, bis wir nur noch ein Element pro Liste haben, was der trivialsten Probleminstanz entspricht.

Obwohl die oben beschriebene Probleminstanz kein Sortieren einer Liste ist, nennen wir die Funktion trotzdem **`mergeSort(…)`**.

Erste Variante noch **ohne** Verschmelzen (Abbildung 80 bis Abbildung 82):

```
def mergeSort(theList, resultLeft, resultRight):
    length = len(theList)
    if length > 1:
        resLeft_L = []
        resLeft_R = []
        resRight_L = []
        resRight_R = []
        #Mitte der Liste
        m = (length-1) // 2
        for i in range(m+1):
            resultLeft.append(theList[i])
        #wieder teilen zuerst linker Teil
        mergeSort(resultLeft, resLeft_L, resLeft_R)

        for i in range(m+1, length):
            resultRight.append(theList[i])
        #dann rechter Teil
        mergeSort(resultRight, resRight_L, resRight_R)
        print(resultLeft)
        print(resultRight)
#    else: # 1 oder kein Element
#        return
```

```
myList = [2, 1 ,3, 9, 5, 7, 8]
resultLeft = []
resultRight = []

mergeSort(myList, resultLeft, resultRight)
print(myList)
```

In diesem Code wird genau das durchgeführt, was wir im obigen Beispiel gezeigt haben. Es ist auch klar ersichtlich, dass die Rekursion immer wieder auf die halbierte Liste angewendet wird. Im Vergleich zu den vorherigen Beispielen haben wir keinen `return` (dies ist im Code auskommentiert), da Listen als Referenzen übergeben werden, können sie in der Funktion direkt modifiziert werden. Die Abbruchbedingung ist hier der `else`-Teil, in dem nichts erledigt werden muss, deshalb kann er weggelassen werden und ist deswegen auskommentiert.

Aufgabe 39

Lassen Sie das Programm laufen, es zeigt alle Teillisten an.
Überlegen Sie sich ferner, ob dies eine gute Lösung ist, resp. was Sie ändern würden.

Lösung:
Das Programm gibt die korrekten Teillisten aus.
Aber die Erzeugung von neuen Listen bei jeder Rekursion verschwendet Speicher und Zeit, da sie immer wieder neu abgefüllt werden.

Aufgabe 40

Da wir nun wissen, dass man so wenig wie möglich neue Listen erzeugen soll, sollten Sie sich überlegen, wie der Algorithmus anders implementiert werden kann. Überprüfen Sie, ob eine Lösung nur durch Verwenden der Indexe in Frage kommen könnte. Stellen Sie ihre Lösung graphisch dar.

Lösung:
Bei der Aufteilung der Listen genügt es zu wissen, wie die Indexe der Grenzen sind, welche die Teillisten bilden. Dies wird hier anhand des einführenden Beispiels zu Mergesort gezeigt. Die Abbildungen 80 bis 82 werden nun hier mit den entsprechenden Indexen dargestellt.

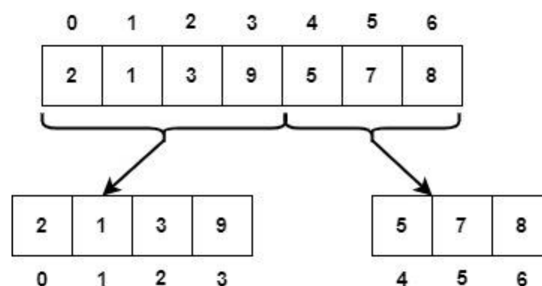


Abbildung 86: Aufteilung mit Index

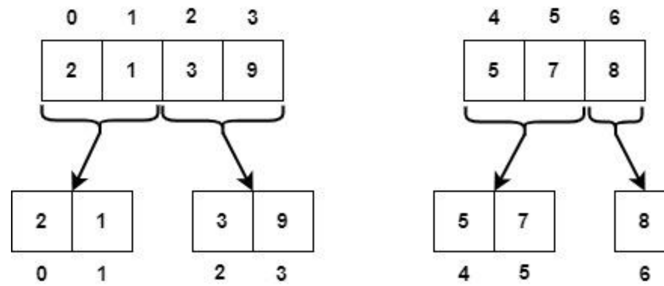


Abbildung 87: nächste Aufteilung mit Index

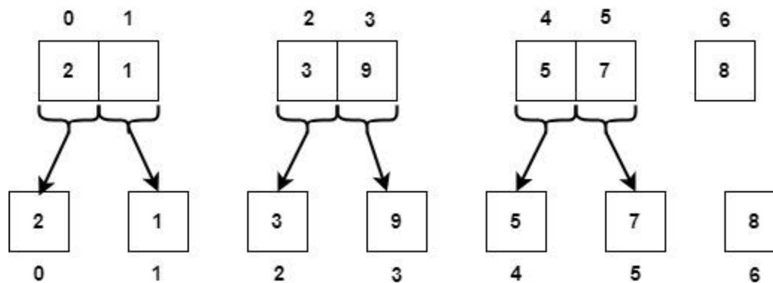


Abbildung 88: letzte Aufteilung mit Index

Daraus ist ersichtlich, dass es genügt, nur die Indexe zu verwenden, um die Liste rekursiv aufzuteilen.

Zweite Variante mit Optimierung (auch wieder ohne Verschmelzung):

```
def mergeSort(theList, start, end):
    if start < end:
        #Mitte der Liste
        #m= start + (end - start) / 2 = start + end / 2 - start / 2
        #m = (start + end) / 2
        m = (start + end) // 2
        #wieder teilen zuerst linker Teil
        mergeSort(theList, start, m)
        #+1 im print wegen Slice Operator
        print(theList[start:(m+1)])
        #dann rechter Teil
        mergeSort(theList, m + 1, end)
        #+1 im print wegen Slice Operator
        print(theList[m+1:(end+1)])
        #Liste ist aufgeteilt, nun verschmelzen
        #merge(theList, start, m, end)

myList = [2, 1 ,3, 9, 5, 7, 8]
length = len(myList)
mergeSort(myList, 0, (length-1))
print(myList)
```

Dieser Code sieht auch viel eleganter aus und wir allozieren keine zusätzlichen Teillisten im Speicher.

Aufgabe 41
Testen Sie diesen Code und prüfen Sie, ob die Listen korrekt geteilt werden.

Aufgabe 42
Sobald die beiden rekursiven Aufrufe von: `mergeSort(…)` in der obigen Funktion beendet sind, können die Teillisten zusammengeführt werden. Der optimierte Code enthält ausgeklammert den Aufruf von
`merge(theList, start, m, end)`
`theList`: Das ist die Liste, welche zu Beginn die Anfangsdaten und nach der Ausführung das Resultat enthält, also das sortierte Verschmelzen der «linken» und «rechten» Teillisten enthält.
`start` bis `m`: Das sind die Indexe der «linken» Teilliste
`m+1` bis `end`: Das sind die Indexe der «rechten» Teillisten

Implementieren Sie nun die `merge(…)` -Funktion.
Hinweis: Sie benötigen eine Hilfsliste, um die Resultate der sortierten Teillisten zwischenzuspeichern, damit das Tauschen implementiert werden kann.

Lösung:
Das **vollständige** Programm mit allen Funktionen und dem Hauptprogramm lautet:

```
#theList: ist die Liste, welche zu Beginn die Anfangsdaten und
#         nach der Ausführung das Resultat, also das
#         Verschmelzen der linken und rechten Teillisten enthält.
#start bis m: Sind die Indexe der «linken» Teilliste
#m+1 bis end: sind die Indexe der «rechten» Teillisten
def merge(theList, start, m, end):
    tmpList = []
    indLeft = start
    indRight = m + 1
    lengthLeft = m - start + 1
    lengthRigth = end - m
    while (indLeft <= m) and (indRight <= end):
        if theList[indLeft] < theList[indRight]:
            tmpList.append(theList[indLeft])
            indLeft += 1
        else:
            tmpList.append(theList[indRight])
            indRight += 1

    if indLeft > m and indRight <= end:
        #Rest übernehmen aus der rechten Hälfte
        for i in range(indRight, end+1):
            tmpList.append(theList[i])
    elif indRight > end and indLeft <= m:
        #Rest übernehmen aus der rechten Hälfte
        for i in range(indLeft, m+1):
            tmpList.append(theList[i])

    #alle Daten aus tmpList nach theList kopieren
    for i in range(start, end+1):
        theList[i] = tmpList[i-start]
```

```
#Aufsteigend sortieren
def mergeSort(theList, start, end):
    if start < end:
        #Mitte der Liste
        #m= start + (end - start) / 2 = start + end / 2 - start / 2
        #m = (start + end) / 2
        m = (start + end) // 2
        #wieder teilen zuerst linker Teil
        mergeSort(theList, start, m)
        #+1 im print wegen Slice Operator
        #print(theList[start:(m+1)])
        #dann rechter Teil
        mergeSort(theList, m + 1, end)
        #+1 im print wegen Slice Operator
        #print(theList[m+1:(end+1)])
        #Liste ist aufgeteilt, nun verschmelzen
        merge(theList, start, m, end)


myList = [2, 1 ,3, 9, 5, 7, 8]
length = len(myList)
print(myList)
mergeSort(myList, 0, (length-1))
print(myList)
```

6 Vor- und Nachteile einer Rekursion

Vorteile:

- Die Idee der Rekursion zur Lösung bestimmter Probleminstanzen, z.B. «Türme von Hanoi» mit 10 Scheiben, hat einen enormen Vorteil gegenüber der iterativen Lösung. Wir lösen die grössere Probleminstanz mit Hilfe ihrer Teilung in kleinere Probleminstanzen, befassen uns also nicht direkt mit der ursprünglichen Probleminstanz. Diese Probleminstanz der «Türme von Hanoi» ist z.B. mittels Iteration fast nicht lösbar resp. sehr umständlich lösbar.
- Wir können mit Rekursion gewisse Probleme oder Definitionen sehr einfach beschreiben.
- Das Navigieren oder Suchen in einem Baum resp. in einer Verzeichnisstruktur kann sehr elegant und effizient mittels Rekursion gelöst werden. Das heisst aber nicht, dass jeder Algorithmus mit Rekursion effizient ist, das Gegenbeispiel dazu ist die Fibonacci-Reihe.

Nachteile:

- Die Rekursion ist schwierig zu programmieren, da unsere Denkweise anders ist.
- Bei einem rekursiven Algorithmus muss sichergestellt werden, dass es eine Abbruchbedingung gibt und dass die Funktion sich nicht zu häufig aufruft. Sonst kann dies zu Stack-Überläufen führen, da der Programmkontext bei jedem Funktionsaufruf auf dem Stack abgelegt wird.
- Manchmal führt die Rekursion zu einem nicht effizienten Algorithmus und man muss andere, effizientere Lösungsstrategien suchen (z.B. die dynamische Programmierung).
- Da bei einem rekursiven Aufruf der Programmfluss in eine Unterfunktion verzweigt, benötigt dies meistens mehr Zeit als die iterative Version. Bei zeitkritischen Anwendungen, wie z.B. in Steuerungen oder Regelungen, sollte man genau überprüfen, ob eine Rekursion sinnvoll ist.

7 Zusammenfassung

In dieser Unterrichtseinheit haben wir gesehen, dass gewisse Effekte und gewisse Geschichten Rekursionen erkennbar sind, es gibt sicher viele solcher Beispiele. Vertieft haben wir die Rekursion als eine Problemlösestrategie für bestimmte Probleme.
Gewisse Probleme können mit diesem Algorithmus-Entwurf elegant resp. effizient gelöst werden, andere wiederum nicht. Der Grund liegt darin, dass nicht alle Probleminstanzen in ähnliche oder gleiche kleinere Probleminstanzen unterteilt werden können.
Am Beispiel des Knobelspiels «Türme von Hanoi» ist es uns einwandfrei gelungen diese Unterteilung durchzuführen, wir haben die grössere Probleminstanz so weit unterteilt, bis wir nur die trivialste Probleminstanz zu lösen hatten, und haben die ganze Zeit so getan, als könnten wir den Turm mit einer Scheibe weniger verschieben. Dies ist der grosse Vorteil der Rekursion.
Mit dieser Idee lassen sich viele Probleme lösen, jedoch nicht bei allen ist diese Methode wegen der Effizienz geeignet. Betrachten wir die Fibonacci-Folge, welche wir mit Rekursion gelöst haben. Da ist uns aufgefallen, dass wir gewisse Berechnungen mehrfach ausführen, was keine effiziente Programmierung ist.
Eine weitere Anwendung ist das Navigieren durch einen binären Baum oder in einem binären Suchbaum. Da wir links und rechts jeweils auch einen Baum haben, ist die Unterteilung der Probleminstanz von der Struktur her bereits gegeben, sodass wir mittels Rekursion schnell ans Ziel gelangen.
Weiter können die meisten Annäherungen bei der Suche einer Lösung, z.B. Nullstellen eines Polynoms mittels Rekursion gelöst werden, da jeweils mit jedem Schritt ein Intervall oder ein Wert immer näher zur Lösung rückt.
Das Ziel dieser Unterrichtseinheit ist es, die Methode der Rekursion als Algorithmus-Entwurf zu vertiefen. Es sollte nun klar sein, was Rekursion bedeutet und welche Probleme damit gelöst werden können.

8 Kontrollfragen

1. Begründen Sie, weshalb bei der Implementierung einer Operation bei einem binären Suchbaum die Lösung mit einer Rekursion bevorzugt werden sollte.
   Lösung:
   Bei einem binären Suchbaum wird pro Teilbaum dasselbe ausgeführt wie für den gesamten Baum, dies entspricht einer Rekursion.
2. Beschreiben Sie kurz, weshalb es zu einem Speicher-Überlauf (Stack Overflow) kommen kann.
   Lösung:
   Bei jedem Aufruf einer Funktion wird der Programmkontext in den Stack, das ist ein spezieller Bereich im Speicher, gespeichert. Sind es zu viele Aufrufe, so wird irgendwann der Speicher voll und es können keine weiteren Programmkontexte gespeichert werden. Dies ist dann ein Stack-Überlauf (Stack Overflow).
3. Würden Sie immer eine rekursive Lösungsstrategie anwenden? Begründen Sie ihre Wahl.
   Lösung:
   Eine Rekursion macht dann Sinn, wenn eine Probleminstanz in fast gleiche kleinere Probleminstanzen aufgeteilt werden kann, dann kann nämlich die kleinere Probleminstanz mit demselben Algorithmus gelöst werden. Bei Zahlenfolgen ist das Problem, dass mit Rekursion derselbe Wert häufig neu berechnet wird.
4. Was könnte der Grund sein, dass eine Programmiersprache die Rekursion nicht erlaubt.
   Lösung:
   Bei Rekursionen muss man sicherstellen, dass es eine Abbruchbedingung gibt, dass diese auch ausgeführt wird und dass es somit nicht zu einem Stack Overflow kommt.

9 Kontrollaufgaben

Kontrollaufgabe 1
Wandeln Sie folgende iterative Funktion **`count(zahl, max)`**, welche im Pseudocode geschrieben ist, in eine rekursive Variante um.

```
count(zahl, max):
  wiederhole so lange zahl ≤ max :
    Ausgabe: zahl
    zahl := zahl + 1
  # Ende der Schleife
# Ende von count
```

Lösung:

```
count(zahl, max) :
  wenn zahl > max :
    fertig
  sonst:
    Ausgabe: zahl
  count(zahl + 1, max)
  # Ende von wenn
# Ende von count
```

Kontrollaufgabe 2
Die *Hofstadter Q-Folge* ist rekursiv definiert durch:

$$hof(1) = hof(2) = 1$$
$$hof(n) = hof\big(n - hof(n-1)\big) + hof\big(n - hof(n-2)\big);\ f\text{ür}\, n\ >\ 2$$

Formel 1: Hofstadter Q-Folge

a) Berechnen Sie einige Glieder dieser Folge von Hand.
b) Schreiben Sie ein rekursives Programm in Python, welches die ersten n (z.B. n = 10, 20, 30 oder 40) Glieder der Hofstadter Q- Folge berechnet.

Lösung

a) n = 4: Start: 1, 1,

$$\begin{aligned} hof(3) &= hof\big(3 - hof(3-1)\big) + hof\big(3 - hof(3-2)\big) \\ &= hof\big(3 - hof(2)\big) + hof\big(3 - hof(1)\big) = hof(3-1) + hof(3-1) \\ &= hof(2) + hof(2) = 1 + 1 = 2 \\ hof(4) &= hof\big(4 - hof(4-1)\big) + hof\big(4 - hof(4-2)\big) \\ &= hof\big(4 - hof(3)\big) + hof\big(4 - hof(2)\big) = hof(4-2) + hof(4-1) \\ &= hof(2) + hof(3) = 1 + 2 = 3 \\ hof(5) &= hof\big(5 - hof(5-1)\big) + hof\big(5 - hof(5-2)\big) \\ &= hof\big(5 - hof(4)\big) + hof\big(5 - hof(3)\big) = hof(5-3) + hof(5-2) \\ &= hof(2) + hof(3) = 1 + 2 = 3 \\ hof(6) &= hof\big(6 - hof(6-1)\big) + hof\big(6 - hof(6-2)\big) \\ &= hof\big(6 - hof(5)\big) + hof\big(6 - hof(4)\big) = hof(6-3) + hof(6-3) \\ &= hof(3) + hof(3) = 2 + 2 = 4 \end{aligned}$$

Folge: 1, 1, 2, 3, 3, 4, …
Ohne Rechnung im Detail ergibt dies folgende Folge:
1, 1, 2, 3, 3, 4, 5, 5, 6, 6, 6, 8, 8, 8, 10, 9, 10, 11, 11, 12, …

b)

```
def hof(n):
    if n == 1 or n == 2:
        return 1
    else:
        return hof(n - hof(n-1)) + hof(n - hof(n-2))
```

Da die Folge rekursiv definiert ist, kann man im Programm direkt die Definition angeben.
Das Programm läuft bereits mit n = 30 sehr lange, dies ist bei Folgen vielfach ein Nachteil der Rekursion, da bereits berechnete Glieder mehrmals neu berechnet werden.

Kontrollaufgabe 3

Gegeben ist folgender Code:

```
from gturtle import *

def peano(n, s, w):
    if n == 0:
        return
    left(w)
    peano(n - 1, s, -w)
    forward(s)
    right(w)
    peano(n - 1, s, w)
    forward(s)
    peano(n - 1, s, w)
    right(w)
    forward(s)
    peano(n - 1, s, -w)
    left(w)

makeTurtle()
setPos(185, -185)
peano(2, 20, 90)
```

Zeichnen Sie die Ausgabe der Turtle mit n = 2 auf. Prüfen Sie danach durch Ausführen des Programmes, ob das Resultat stimmt.

Kontrollaufgabe 4

Schreiben Sie einen Algorithmus, welcher den Sortieralgorithmus Selectionsort mit Rekursion implementiert. Überlegen Sie sich zuerst das Rezept, um die Aufgabe zu lösen.
Selectionsort:
Dieser Algorithmus sucht sich als Erstes das kleinste Element in der Liste, merkt es sich und tauscht es gegen das Element am Anfang aus, sodass sich dann das kleinste Element ganz am Anfang befindet. Als Nächstes wird das zweitkleinste Element in der Liste gesucht und wird gegen das an zweiter Stelle platzierte Element der Liste ausgetauscht usw.
Auf diese Weise haben immer die Elemente auf der linken Seite bis zur aktuellen Position einen festen Platz und werden nicht mehr geändert.

Lösung

Wir wenden hier das Rezept an:

1. *Beschreiben Sie die Probleminstanz und ermitteln Sie die Problemgrösse N.*
   Selectionsort soll für die Sortierung einer Liste mit **n** Elementen angewendet werden, dabei wird das kleinste Element in der Liste gesucht und mit dem ersten vertauscht. Die Problemgrösse ist $N = n$.
2. *Ermitteln und lösen Sie die trivialste Probleminstanz (N = 1)*
   Die trivialste Probleminstanz ist eine Liste mit einem Element, dann ist die Liste sortiert und die Problemgrösse ist N = 1.
3. *Reduzieren Sie die Probleminstanz in eine oder mehrere kleinere Probleminstanzen*. *(divide)*
   Nachdem das kleinste Element gefunden wurde, kann Selectionsort für die Sortierungen der Restliste mit $(n - 1)$ Elementen angewendet werden, was zu einer kleineren Probleminstanz führt mit der Grösse $N_{red} = n - 1$.
4. *Lösen Sie die kleinere Probleminstanz desselben Problems mit demselben Algorithmus, d.h., verwenden Sie dazu die Rekursion, solange Sie nicht die trivialste Probleminstanz erreicht haben, welche einfach bestimmt werden kann*. *(impera)*
   Man kann die kleinere Probleminstanz mit der reduzierten Problemgrösse mit demselben Algorithmus lösen. Danach kann man mit demselben Algorithmus die restlichen Elemente sortieren, bis die trivialste Probleminstanz vorliegt. In diesem Fall ist dann die Liste sortiert.

Die Lösung in Python sieht dann folgendermassen aus:

```
def swap(dataList, e1, e2):
    tmp = dataList[e1]
    dataList[e1] = dataList[e2]
    dataList[e2] = tmp

def selectionSort(dataList, startIndex):
    #kleinstesElement in Listesuchen
    itIndex = startIndex
    minIndex = itIndex
    minimum = dataList[itIndex]
    while itIndex < listLen:
        if minimum > dataList[itIndex]:
            minimum = dataList[itIndex]
            minIndex = itIndex
        itIndex += 1
    swap(dataList, minIndex, startIndex)
    startIndex += 1
    if startIndex < listLen:
        selectionSort(dataList, startIndex)



myList = [3, 6, 1, 45, 23, 19, 2, 37,12]
listLen = len(myList)
selectionSort(myList, 0)
print(myList)
```

Kontrollaufgabe 5

Die Verzeichnisstruktur auf einer Festplatte ist auch eine Baumstruktur. Erstellen Sie einen Algorithmus (mit Pseudocode oder in Python), der alle Dateinamen und Verzeichnisnamen auf der Konsole ausgibt. Nach jedem Verzeichnis sollen die Dateien eingerückt ausgegeben werden.
Eine mögliche Ausgabe könnte die folgende sein:

```
Dokumente
    Aufgabe1.txt
    Loesungen
        LoesungAufgabe1.txt
    Noten.xlsx
```

Lösung:

```
from pathlib import Path

def listDir(file, depth):
    print('    ' * depth + file.name)
    if file.is_dir():
        for f in file.iterdir():
            listDir(f, depth + 1)

listDir(Path('C:\\Sonst\\Dokumente'), 0)
```

Kontrollaufgabe 6

Sie sollen eine Funktion
`def blattTeilen(anzStreifen, x, y, breite, hoehe)`
schreiben, welche die Blattteilung aus Beispiel 7 mit einem Rechteck (Breite: 64, Höhe: 50) als Blatt mit der Turtlegraphik darstellt. Das Blatt wird in der Breite halbiert und mit untenstehendem Aufruf können Sie dann die linke und die rechte Seite darstellen.
linke Seite:

```
blattTeilen(anzStreifen/2, x-breite, y+breite/2+hoehe, breite/2, hoehe)
```

rechte Seite:

```
blattTeilen(anzStreifen/2, x+3*breite/2, y+breite/2 + hoehe, breite/2, hoehe)
```

Die Graphik mit **`anzStreifen = 8`** sieht wie in Abbildung 89 aus.

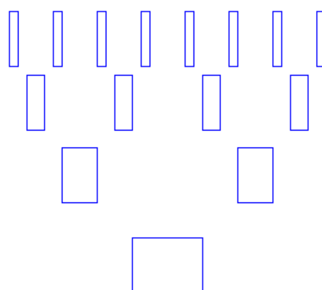


Abbildung 89: Blattteilung mit Rechteck

Lösung:

```
#Kontrollaufgabe_blattTeilen.py
from gturtle import *

def rectangle(x, y, breite, hoehe):
    setPos(x, y)
    H = True
    repeat 4:
        if H:
            s = hoehe
            H = False
        else:
            s = breite
            H = True
        forward(s)
        right(90)


def blattTeilen(anzStreifen, x, y, breite, hoehe):
    rectangle(x, y, breite, hoehe)
    if anzStreifen == 1:
        print('Streifen zu 1 / {} vom Blatt'.format(AnzahlStreifen))
        return
    #linke Hälfte
    blattTeilen(anzStreifen/2, x-breite, y+breite/2 + hoehe,
breite/2, hoehe)
    #rechte Hälfte
    blattTeilen(anzStreifen/2, x+3*breite/2, y+breite/2 + hoehe,
breite/2, hoehe)

makeTurtle()
hideTurtle()
#Paramter als 2er Potenz !!
AnzahlStreifen = 8 #16
x = 0
y = 0
Breite = 64
Hoehe = 50
blattTeilen(AnzahlStreifen, x, y, Breite, Hoehe)
```

Kontrollaufgabe 7

Sortieren Sie aufsteigend eine Liste mit Bubblesort unter Verwendung der Rekursion.
Zur Erinnerung: Bubblesort vergleicht paarweise die Elemente der Liste und vertauscht sie, falls das erste Element des Paares grösser ist als das zweite Element. Nach einem Durchlauf ist das letzte Element in der Liste das grösste und man muss nur noch die gekürzte Liste mit **n-1** Elementen sortieren.
Falls im Durchgang nie vertauscht wurde, ist die Liste sortiert.
Das Rezept ist ähnlich wie das für den Selectionsort, deshalb wird es hier nicht noch einmal verlangt.

Schreiben Sie das Programm.

Lösung:

```
def bubbleSort(theList, endIndex):
    wasSwapped = False
    for i in range(endIndex):
        if theList[i] > theList[i+1]:
            #tauschen
            tmp = theList[i]
            theList[i] = theList[i+1]
            theList[i+1] = tmp
            wasSwapped = True

    if not wasSwapped:
        return

    bubbleSort(theList, endIndex - 1)


myList = [15, 2, 43, 17, 4, 8, 47]
print(myList)
end = len(myList)-1
bubbleSort(myList, end)
print(myList)
```

Kontrollaufgabe 8
Das Verfahren von Quicksort wurde in den letzten Monaten erklärt, deshalb wird es hier nur kurz beschrieben. Betrachten wir dazu eine unsortierte Liste.
Man definiert das erste Element als Pivotelement (am `pivotIndex`). Man geht mit einem Zeiger (zum Beispiel mit einem grünen Zeiger) von links durch die Liste und vergleicht die Elemente mit dem Pivotelement, bis man auf das erste Element stösst, das grösser oder gleich ist als das Pivotelement. Dann sucht man mit einem anderen Zeiger (zum Beispiel mit einem blauen Zeiger) von rechts nach dem ersten Element, das kleiner oder gleich ist als das Pivotelement und vertauscht die beiden Elemente. Nun geht die Suche mit demselben Vorgehen weiter, bis sich die zwei Zeiger kreuzen. In dieser Aufgabe sind die Zeiger z.B. zwei Indexe (`startIndex`, `endIndex`), zuerst wird das erste Element von links gesucht, welches grösser ist als das Pivotelement, danach wird von rechts dasjenige Element gesucht, welches kleiner ist als das Pivotelement. Nun werden diese beiden Elemente vertauscht, bis der `startIndex`, welcher links beginnt, grösser als der `endIndex` ist, welcher rechts beginnt.
In diesem Fall wird das Pivotelement mit dem Element an der Position `endIndex` vertauscht. Nun sind alle Werte der Liste, welche kleiner sind als das Pivotelement, links und die grösseren rechts vom Pivotelement.
Dasselbe wird nun für den Teil links vom Pivotelement und rechts vom Pivotelement durchgeführt.

a) Zeichnen Sie zuerst eine Skizze des Ablaufes, um das Verfahren von Quicksort darzustellen.
b) Wenden Sie das Rezept an.
c) Schreiben Sie den Code.

Lösung:
a)

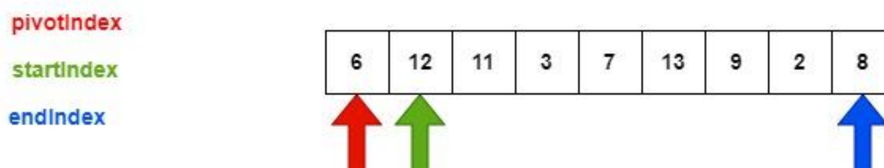


Abbildung 90: Start der Sortierung

Das erste Element (roter Pfeil) ist das Pivotelement und man beginnt nun links beim Element nach dem Pivotelement (grüner Pfeil) und rechts beim letzten Element (blauer Pfeil) wie in Abbildung 90. Nun wird von links der Wert gesucht, welcher grösser ist als das Pivotelement, und von rechts der Wert, welcher kleiner ist.

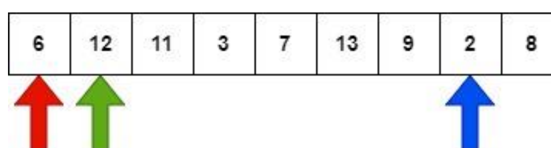


Abbildung 91: Werte gefunden

Dann wird der Wert der blauen Position mit dem der grünen vertauscht.

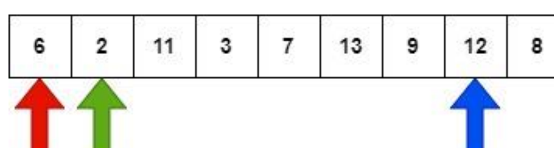


Abbildung 92: Tauschen

Der nächste Schritt sucht wieder die entsprechenden Werte:

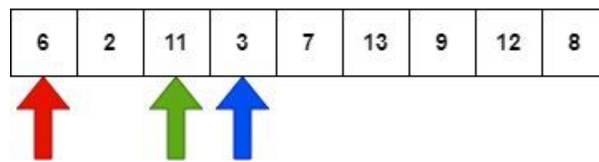


Abbildung 93: nächste Werte suchen

Nun wird wieder getauscht:

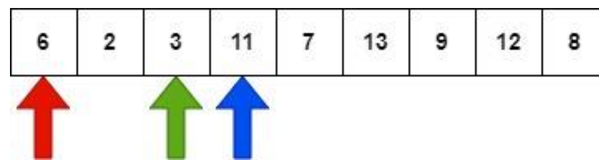


Abbildung 94: tauschen

Die Suche wird fortgesetzt:

Abbildung 95: nächste Werte suchen

Nach dieser Suche haben sich die Zeiger überkreuzt, d.h., der **`startIndex`** ist grösser als der **`endIndex`**, deshalb werden wir das Pivotelement mit dem Wert beim blauen Zeiger tauschen. Das Pivotelement selbst ist nun am Index **`endIndex`** (blauer Pfeil) und **`pivotIndex`** ist immer noch an der alten Position.

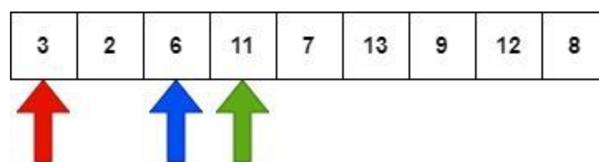


Abbildung 96: Pivot mit endIndex tauschen

Nach diesem Schritt ist das Pivotelement an der richtigen Stelle und wir können die zwei Teillisten (links und rechts vom Pivotelement) wieder mit Quicksort sortieren. Die linke Liste mit Index **`pivotIndex`** (roter Pfeil) bis **`endIndex-1`** und die rechte Liste mit Index **`endIndex+1`** bis Ende der Liste (**`listLen-1`**). **`endIndex`** ist die Position des Pivotelementes an der richtigen Stelle in der Liste.

b) Rezept

1. *Beschreiben Sie die Probleminstanz und ermitteln Sie die Problemgrösse N.*
   Es soll eine Liste bestehend aus **n** Elementen mit Quicksort sortiert werden. Die Problemgrösse ist $N = n$.
2. *Ermitteln und lösen Sie die trivialste Probleminstanz (N = 1).*
   Dies liegt vor, falls nur noch ein Element zu sortieren ist, d.h., wenn der **startIndex == endIndex** ist, dann haben wir eine Problemgrösse $N = 1$.
3. *Reduzieren Sie die Probleminstanz in eine oder mehrere kleinere Probleminstanzen. (divide)*
   Die kleineren Probleminstanzen ergeben sich, sobald das Pivotelement an der richtigen Stelle ist, dann müssen wir nur noch die linken und rechten Teillisten sortieren. Da die Teillisten kleiner sind als die Liste selbst, haben wir eine kleinere Problemgrösse $N_{red} < N$. Die rechten und linken Teillisten können dann analog sortiert werden.
4. *Lösen Sie die kleineren Probleminstanzen desselben Problems mit demselben Algorithmus, d.h., verwenden Sie dazu die Rekursion, solange Sie nicht die trivialste Probleminstanz erreicht haben, welche einfach bestimmt werden kann. (impera)*
   Die Teillisten können wiederum mit demselben Verfahren gelöst werden, bis die trivialste Probleminstanz vorliegt, dann ist wiederum die Liste sortiert.

c)

```
def swap(dataList, e1, e2):
    tmp = dataList[e1]
    dataList[e1] = dataList[e2]
    dataList[e2] = tmp

def quickSort(dataList, startIndex, endIndex):
    if startIndex < endIndex:
        pivotIndex = startIndex
        pivot = dataList[pivotIndex]
        startIndex += 1
        while startIndex < endIndex:
            print(startIndex, endIndex)
            while startIndex < endIndex and dataList[startIndex] <= pivot:
               #weiterschalten, falls Element kleiner pivot ist, sonst stopp
                startIndex += 1
            while endIndex >= startIndex and dataList[endIndex] > pivot:
               #weiterschalten, falls Element grösser pivot ist, sonst stopp
                endIndex -= 1
            if startIndex < endIndex:
                swap(dataList, startIndex, endIndex)
        #pivot tauschen
        if dataList[endIndex] < dataList[pivotIndex]:
            swap(dataList, pivotIndex, endIndex)
        #pivot ist richtig platziert, an Index endIndex.
        #sortieren vom linken Teil
        quickSort(dataList, pivotIndex, endIndex-1)
        #sortieren vom rechten Teil
        quickSort(dataList, endIndex+1, listLen-1)

myList = [3, 6, 1, 45, 23, 19, 37,12]
listLen = len(myList)
quickSort(myList, 0, listLen-1)
print(myList)
```

10 Abbildungsverzeichnis

Abbildung 1: Concept Map ... 5
Abbildung 2: Türme von Hanoi (Quelle: (Rimscha, 2017)) ... 8
Abbildung 3: Rekursion als Bild ... 11
Abbildung 4: Spiegeleffekt mit Python ... 12
Abbildung 5: Ablauf Teller waschen ... 13
Abbildung 6: Waschbecken beim Start ... 15
Abbildung 7: Waschbecken fast leer ... 15
Abbildung 8: Kaninchenpopulation (Quelle: https://erasmus-reinhold-gymnasium.de/info/rekursion/fibonacci.html) ... 25
Abbildung 9: Fibonacci-Quadrat ... 26
Abbildung 10: Muschelgehäuse ... 28
Abbildung 11: Schritt 1 der Teilung ... 31
Abbildung 12: Schritt 2 der Teilung ... 31
Abbildung 13: Schritt 3 der Teilung ... 31
Abbildung 14: Schritt 4 der Teilung ... 31
Abbildung 15: Schritt 5 der Teilung ... 31
Abbildung 16: Schritt 6 der Teilung ... 32
Abbildung 17: Schritt 7 der Teilung ... 32
Abbildung 18: Ablauf gemäss der Analyse ... 33
Abbildung 19: gesamte Aufrufsequenz für die acht Streifen ... 34
Abbildung 20: erster Aufruf blattTeilen(8) ... 36
Abbildung 21: blattTeilen(4), erste Rekursionstiefe ... 36
Abbildung 22: blattTeilen(2), zweite Rekursionstiefe ... 36
Abbildung 23: blattTeilen(1), dritte Rekursionstiefe ... 37
Abbildung 24: blattTeilen(1), dritte Rekursionstiefe, 2 Streifen ... 37
Abbildung 25: blattTeilen(2), rechter Teil, zweite Rekursionstiefe ... 38
Abbildung 26: Alle acht Streifen sind erstellt. ... 39
Abbildung 27: Aufruf Baum der Fibonacci-Folge ... 41
Abbildung 28: Wabenmuster ... 42
Abbildung 29: Wabe Aufruf 1 mit Ausgabe ... 43
Abbildung 30: Wabe Aufruf 2 mit Ausgabe ... 43
Abbildung 31: Wabe Aufruf 3 mit Ausgabe ... 44
Abbildung 32: Wabe Aufruf 4 mit Ausgabe ... 44
Abbildung 33: Wabe Aufruf 5 mit Ausgabe ... 44
Abbildung 34: Schrittweiser Aufbau der Wabe mit s = 4 ... 45
Abbildung 35: Binärbaum ... 46
Abbildung 36: schrittweiser Aufbau des Baumes ... 47
Abbildung 37: Aufrufsequenz der Funktion tree ... 48
Abbildung 38: Baumstruktur der Aufrufe in Schritt 1 ... 48
Abbildung 39: Baumstruktur der Aufrufe in Schritt 2 ... 48
Abbildung 40: Baumstruktur der Aufrufe in Schritt 3 ... 48
Abbildung 41: Baumstruktur der Aufrufe in den Schritten 4 und 5 ... 49
Abbildung 42: Baumstruktur der Aufrufe in den Schritten 6, 7, und 8 ... 49
Abbildung 43: Baumstruktur der Aufrufe in Schritt 9 ... 49
Abbildung 44: Baumstruktur der Aufrufe in den Schritten 10, 11 und 12 ... 50
Abbildung 45: Baumstruktur der Aufrufe in den Schritten 13, 14 und 15 ... 50
Abbildung 46: Speicher mit Stack im unteren Bereich ... 52
Abbildung 47: Stack befüllt mit Behälter ... 53

Abbildung 48: Stack zu Beginn des Programmes 53
Abbildung 49: Stack unmittelbar vor dem Aufruf der Funktion........ 53
Abbildung 50: Aufbau Stack bei der Rekursion von fakultaet(3) 54
Abbildung 51: Die Schule von Athen (Quelle:
https://commons.wikimedia.org/wiki/File:La_scuola_di_Atene.jpg)........ 55
Abbildung 52: Berechnung ggT(66,18) 56
Abbildung 53: Newton-Verfahren erster Schritt (Quelle:
https://commons.wikimedia.org/wiki/File:NewtonIteration_Ani.gif)........ 59
Abbildung 54: Newton-Verfahren zweiter Schritt (Quelle:
https://commons.wikimedia.org/wiki/File:NewtonIteration_Ani.gif)........ 59
Abbildung 55: Newton-Verfahren dritter Schritt (Quelle:
https://commons.wikimedia.org/wiki/File:NewtonIteration_Ani.gif)........ 59
Abbildung 56: Newton-Verfahren vierter Schritt (Quelle:
https://commons.wikimedia.org/wiki/File:NewtonIteration_Ani.gif)........ 60
Abbildung 57: Startintervall, welches eine Nullstelle beinhaltet. (Quelle:
https://commons.wikimedia.org/wiki/File:Bisektion_Ani.gif)........ 64
Abbildung 58: Intervall verkleinern - Schritt 1 (Quelle:
https://commons.wikimedia.org/wiki/File:Bisektion_Ani.gif)........ 65
Abbildung 59: Intervall verkleinern - Schritt 2 (Quelle:
https://commons.wikimedia.org/wiki/File:Bisektion_Ani.gif)........ 65
Abbildung 60: Nullstelle gefunden (Quelle:
https://commons.wikimedia.org/wiki/File:Bisektion_Ani.gif)........ 65
Abbildung 61: Funktionsgraphen 66
Abbildung 62: Nullstelle 68
Abbildung 63: Startintervall, rechts ist der Bereich vergrössert. 68
Abbildung 64: Intervall Halbierung 69
Abbildung 65: Nächste Halbierung........ 69
Abbildung 66: Plot der Funktion: f(x) =x+2cos(x) 70
Abbildung 67: Nullstelle von f(x)=x+2*cos(x)........ 71
Abbildung 68: Türme von Hanoi - Ausgangslage........ 76
Abbildung 69: nach dem 1. Schritt 76
Abbildung 70: nach dem 2. Schritt 77
Abbildung 71: nach dem 3. Schritt 77
Abbildung 72: nach dem 4. Schritt 77
Abbildung 73: Ausgangslage mit drei Steinen........ 77
Abbildung 74: nach dem ersten Schritt........ 78
Abbildung 75: nach dem zweiten Schritt 78
Abbildung 76: rekursive Aufrufe der Funktion hanoi(…) für die Lösung der Türme von Hanoi mit 3
Scheiben 81
Abbildung 77: Türme von Hanoi (Quelle: https://www.bernhard-gaul.de/spiele/tower/tower.php). 82
Abbildung 78: binärer Suchbaum........ 83
Abbildung 79: unsortierte Liste........ 88
Abbildung 80: Liste aufteilen........ 88
Abbildung 81: Teillisten wieder aufteilen 88
Abbildung 82: Listen mit nur noch einem Element........ 89
Abbildung 83: Listen sortiert vereinigen 89
Abbildung 84: weiter vereinigen 89
Abbildung 85: Liste ist sortiert 89
Abbildung 86: Aufteilung mit Index........ 91
Abbildung 87: nächste Aufteilung mit Index........ 92
Abbildung 88: letzte Aufteilung mit Index 92
Abbildung 89: Blattteilung mit Rechteck........ 102

Abbildung 90: Start der Sortierung 105
Abbildung 91: Werte gefunden 105
Abbildung 92: Tauschen 105
Abbildung 93: nächste Werte suchen 106
Abbildung 94: tauschen 106
Abbildung 95: nächste Werte suchen 106
Abbildung 96: Pivot mit endIndex tauschen 106

11 Literaturverzeichnis

Erdag, T., & Steffen Björn. (2008). *Binäre Suchbäume.* Zürich: EducETH.

Gallenbacher, J. (2021). *Abenteuer Informatik.* Mainz: Springer.

Jarka Arnold, A. P. (12. April 2023). *jython.ch*. Von https://jython.ch/ abgerufen

Jarka Arnold, T. K. (3. September 2022). *programmierkonzepte*. Von https://programmierkonzepte.ch/ abgerufen

Kalista, H. (2018). Python 3 Einsteigen und Durchstarten. (Hanser, Hrsg.) München.

Ottmann, T., & Widmayer, P. (2012). *Algorithmen und Datenstrukturen.* Neu-Ulm: Springer.

Parisi, V., & Travanov, A. (2023). Rekursive Algorithmen.

Rimscha, M. v. (2017). *algorithmen kompakt und verständlich.* Fürth: Springer.