

Algorithmen III



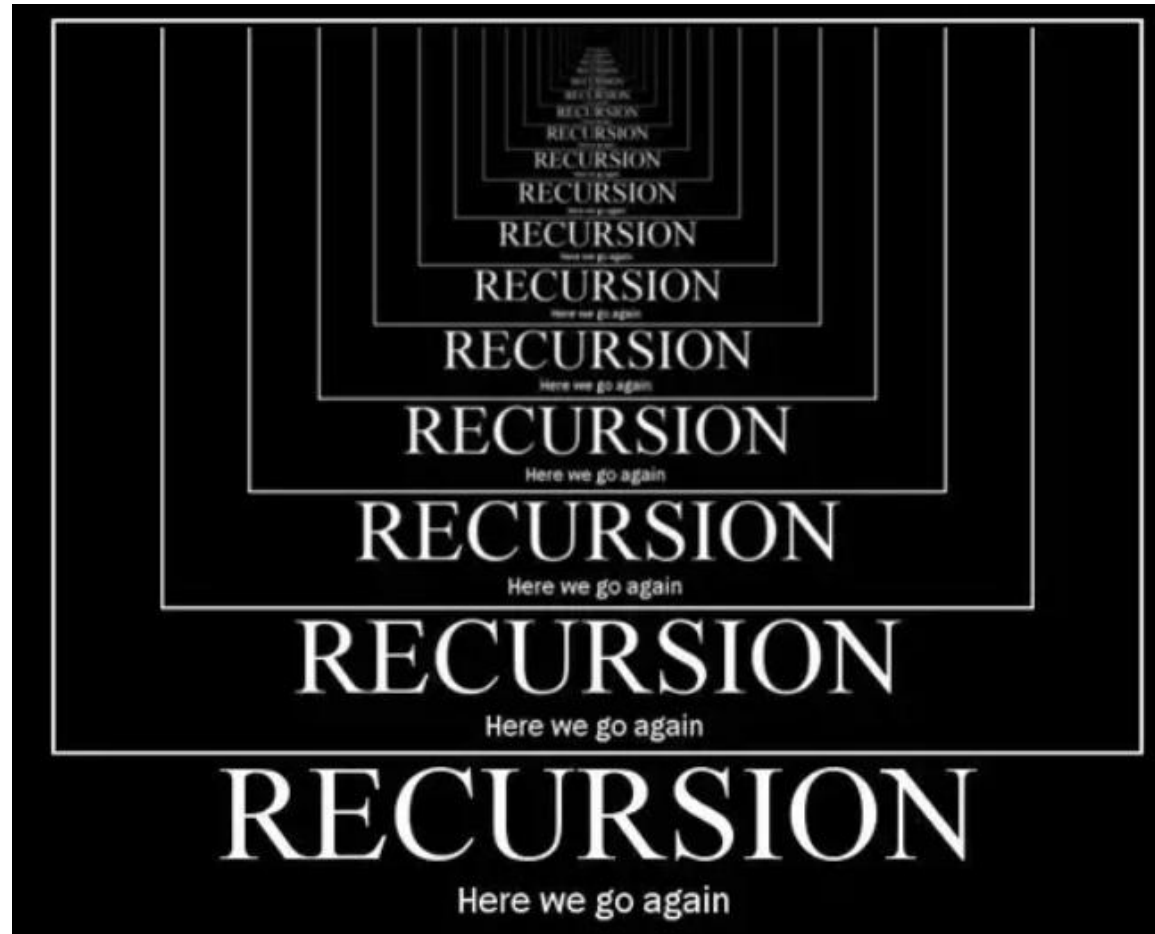
Dr. Philipp Hurni, Kantonsschule Sursee

Algorithmen III

- Rekursion und rekursive Algorithmen
 - Definition
 - Eigenschaften
 - Babushka Übung
 - Weitere Übungen zu Rekursion
- Algorithmen vs. Computational Thinking
 - Begriff "Computational Thinking"
 - Gemeinsamkeiten und Unterschiede



Rekursion ~ Selbstähnlichkeit



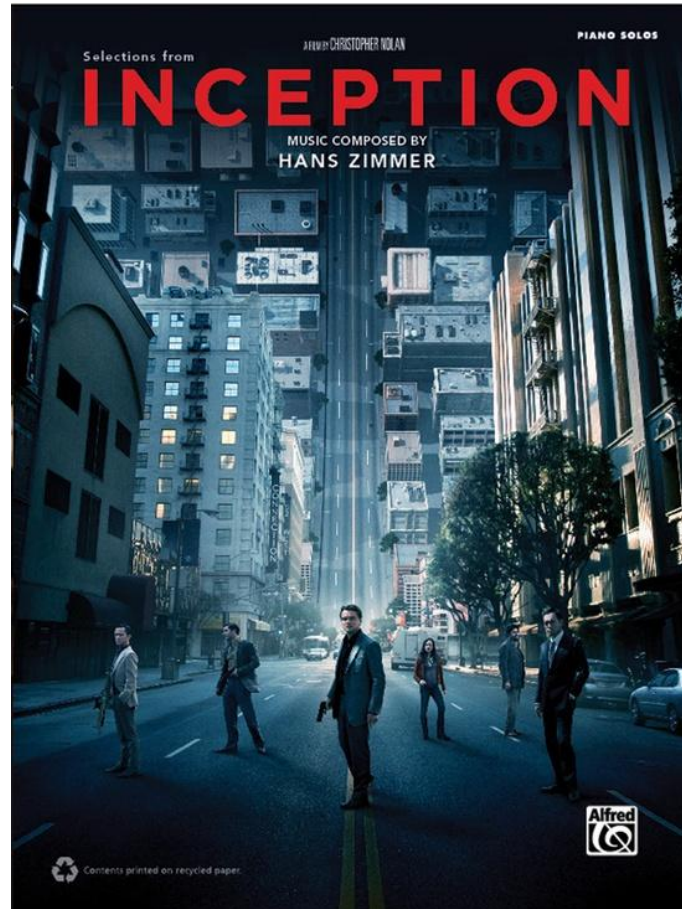
Babushkas sind auch eine Form der Rekursion



Die Natur ist voller Rekursionen



Rekursion in Hollywood



The 5 Levels Of INCEPTION				
LEVEL	WHO DREAMED IT?	WHO GOES THERE?	WHY ARE THEY THERE?	THE KICK
LEVEL 1 REALITY	No one... We think	Cobb, Arthur, Ariadne, Eames, Saito, Yusuf and Robert Fischer Jr.	To drug Fischer Jr. and bring his subconscious into a dream.	There isn't one. The timer counts down and the machine shuts off.
LEVEL 2 VAN CHASE	Yusuf "The Chemist"	Cobb, Arthur, Ariadne, Eames, Saito, Yusuf and Robert Fischer Jr.	Fischer Jr. is kidnapped. They force him to give them random numbers which are used later, and begin planting the idea in his head that his father wants him to break up the company.	Yusuf drives the van off a bridge. That fails. A second Kick occurs when the van hits the water.
LEVEL 3 THE HOTEL	Arthur "The Point Man"	Cobb, Arthur, Ariadne, Eames, Saito and Robert Fischer Jr.	Fischer Jr. is tricked into believing Browning is a traitor. He joins the team for their next mission.	Arthur blows up an elevator, simulating freefall.
LEVEL 4 SNOW FORTRESS	Eames "The Forger"	Cobb, Ariadne, Eames, Saito and Robert Fischer Jr.	Fischer Jr. must be taken to the fort, where the idea they wish to plant will finally take hold.	Eames blows up the supports of the fortress, dropping it and causing freefall.
LEVEL 5 LIMBO	No one It's a shared state	Cobb, Ariadne, Saito, Robert Fischer Jr. and Mal's projection	To get Fischer Jr. and Saito out.	Ariadne and Fischer fall off a building. Cobb and Saito shoot themselves.

Island – Lake – Island – Lake – Island



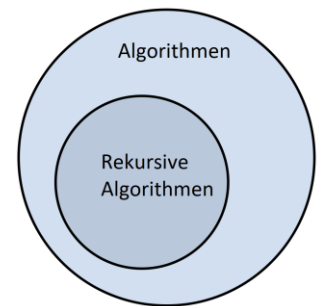
Rekursion «Definition»

[Rekursion Wikipedia](#)

- Prinzipiell unendlicher Prozess, welcher sich selbst oder Teile von sich selbst enthält
- Problemlösungsstrategie, mit welcher komplexe Sachverhalte elegant erfasst werden können
- Probleme kann man "iterativ" oder "rekursiv" lösen

Was ist ein rekursiver Algorithmus?

- Ein rekursiver Algorithmus ist ein Ablauf bzw. eine Schrittfolge, mit der ein Problem eindeutig, in endlich vielen Schritten gelöst wird
- Ein rekursiver Algorithmus ruft sich selbst (oder Teile von sich selbst) auf. Er terminiert immer mit einer Abbruchbedingung!
- Rekursive Algorithmen sind eine Unterklasse der Algorithmen – man kann teilweise Dinge iterativ oder Rekursiv lösen
- Es ist nicht "besser" etwas rekursiv zu tun – es ist oft eher eine "Geschmackssache"



Einfachstes Beispiel – Die Fakultätsfunktion

Iterative Formel

$$\begin{aligned} n! &= n \cdot (n-1)! = n \cdot (n-1) \cdot (n-2)! = \dots \\ &= n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1 \\ &= \prod_{i=1}^n i \end{aligned}$$

Rekursive Formel

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

Beispiel: $4! = 4 * 3 * 2 * 1$

Iterativer Ansatz

```
def factorial_iterative(n):  
    val = 1  
    for i in range(1, n+1):  
        val = val * i  
    return val
```

```
print("Iterative:", factorial_iterative(5))  
print("Recursive:", factorial_recursive(5))
```

Rekursiver Ansatz

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



Abbruchbedingung



Rekursiver Selbstaufruf

Erstes Übungsbeispiel: die Summe einer Liste berechnen

Iterativ

```
zahlen = [1,5,6,2,3,8,6,3,2,4]
```

```
def sum_iter(liste):  
    summe = 0  
    for zahl in liste:  
        summe += zahl  
    return summe
```

```
ergebnis = sum_iter(zahlen)  
print("Summe der Zahlen ist:", ergebnis)
```

Rekursiv

```
zahlen = [1,5,6,2,3,8,6,3,2,4]
```

```
def sum_rek(summe, liste):
```

Funktionsdefinition

Was muss hier rein?
(2 Fälle)

Aufruf der rekursiven
Funktion

```
ergebnis = sum_rek(0, zahlen)  
print("Die Summe der Zahlen ist:", ergebnis)
```

Siehe: Summe_Rekursiv

Erstes Übungsbeispiel: die Summe einer Liste berechnen

Iterativ

```
zahlen = [1,5,6,2,3,8,6,3,2,4]
```

```
def sum_iter(liste):  
    summe = 0  
    for zahl in liste:  
        summe += zahl  
    return summe
```

```
ergebnis = sum_iter(zahlen)  
print("Summe der Zahlen ist:", ergebnis)
```

Rekursiv

```
zahlen = [1,5,6,2,3,8,6,3,2,4]
```

```
def sum_rek(summe, liste):  
    if len(liste) == 0:           Abbruchbedingung  
        return summe  
    else:  
        erstes = liste.pop(0)  
        summe = erstes + sum_rek(summe, liste)  
        return summe
```

```
ergebnis = sum_rek(0, zahlen)  
print("Die Summe der Zahlen ist:", ergebnis)
```

Erstes Übungsbeispiel: die Summe einer Liste berechnen

```
sum_rek( 0 , [1, 5, 6, 2, 3, 8, 6, 3, 2, 4] )  
return 1 + sum_rek( 0 , [5, 6, 2, 3, 8, 6, 3, 2, 4] )
```

Erstes Übungsbeispiel: die Summe einer Liste berechnen

```
sum_rek( 0 , [1, 5, 6, 2, 3, 8, 6, 3, 2, 4] )  
return 1 + sum_rek( 0 , [5, 6, 2, 3, 8, 6, 3, 2, 4] )  
    sum_rek( 0 , [5, 6, 2, 3, 8, 6, 3, 2, 4] )  
    return 5 + sum_rek( 0 , [6, 2, 3, 8, 6, 3, 2, 4] )  
        sum_rek( 0 , [6, 2, 3, 8, 6, 3, 2, 4] )  
        return 6 + sum_rek( 0 , [2, 3, 8, 6, 3, 2, 4] )  
            sum_rek( 0 , [2, 3, 8, 6, 3, 2, 4] )  
            return 3 + sum_rek( 0 , [3, 8, 6, 3, 2, 4] )  
                sum_rek( 0 , [3, 8, 6, 3, 2, 4] )  
                return 3 + sum_rek( 0 , [8, 6, 3, 2, 4] )  
                    sum_rek( 0 , [8, 6, 3, 2, 4] )  
                    return 8 + sum_rek( 0 , [6, 3, 2, 4] )  
                        sum_rek( 0 , [6, 3, 2, 4] )  
                        return 6 + sum_rek( 0 , [3, 2, 4] )  
                            sum_rek( 0 , [3, 2, 4] )  
                            return 3 + sum_rek( 0 , [2, 4] )  
                                sum_rek( 0 , [2, 4] )  
                                return 2 + sum_rek( 0 , [4] )  
                                    sum_rek( 0 , [4] )  
                                    return 4 + sum_rek( 0 , [] )  
                                        sum_rek( 0 , [] )  
                                        return 0
```

Abbruchbedingung!



Erstes Übungsbeispiel: die Summe einer Liste berechnen

```
sum_rek( 0 , [1, 5, 6, 2, 3, 8, 6, 3, 2, 4] ) = 40
return 1 + 5 + 6 + 2 + 3 + 6 + 3 + 2 + 4 + 0
    sum_rek( 0 , [5, 6, 2, 3, 8, 6, 3, 2, 4] )
    return 5 + 6 + 2 + 3 + 6 + 3 + 2 + 4 + 0
        sum_rek( 0 , [6, 2, 3, 8, 6, 3, 2, 4] )
        return 6 + 2 + 3 + 6 + 3 + 2 + 4 + 0
            sum_rek( 0 , [2, 3, 8, 6, 3, 2, 4] )
            return 2 + 3 + 6 + 3 + 2 + 4 + 0
                sum_rek( 0 , [3, 8, 6, 3, 2, 4] )
                return 3 + 6 + 3 + 2 + 4 + 0
                    sum_rek( 0 , [8, 6, 3, 2, 4] )
                    return 8 + 6 + 3 + 2 + 4 + 0
                        sum_rek( 0 , [6, 3, 2, 4] )
                        return 6 + 3 + 2 + 4 + 0
                            sum_rek( 0 , [3, 2, 4] )
                            return 3 + 2 + 4 + 0
                                sum_rek( 0 , [2, 4] )
                                return 2 + 4 + 0
                                    sum_rek( 0 , [4] )
                                    return 4 + 0
                                        sum_rek( 0 , [] )
                                        return 0
```

Abbruchbedingung!



Babushka – "öffnen"



- Ich habe die Babushka Olga...
- In ihrem Bauch ist eine weitere Babushka – Tatjana...
- In ihrem Bauch ist eine weitere Babushka – Svetlana...
- Schreibe ein Programm, welches die Namen aller Babushka's und "Unter-Babushkas" ausdrückt. Die Einnestung kann beliebig tief sein – und zwar so:

Im Bauch von Olga ist Tatjana
Im Bauch von Tatjana ist Svetlana
Im Bauch von Svetlana ist ... usw

Versuche das a) iterativ zu tun und b) rekursiv

```
babushka = ["Olga", ["Tatjana", ["Svetlana", ["Natalya", ["Katerina", ["Alina", ["Nadia"]]]]]]]]
```

Lösungen: Babushka – "öffnen"



```
# iterativ
babushka = ["Olga", ["Tatjana", ["Svetlana", ["Natalya", ["Katerina", ["Alina", ["Nadia"]]]]]]]]
while(len(babushka) > 1):
    name = babushka[0]
    babushka = babushka[1]
    print("im Bauch von", name, "ist", babushka[0])
```

```
# rekursiv
def openBabushka(babushka):
    if (len(babushka) > 1):
        name = babushka[0]
        babushka = babushka[1]
        print("im Bauch von", name, "ist", babushka[0])
        openBabushka(babushka)
    else:
        print("im Bauch von", babushka[0], "ist niemand mehr")
```

Keine Schlaufe, dafür Funktion!

Abbruchbedingung

```
babushka = ["Olga", ["Tatjana", ["Svetlana", ["Natalya", ["Katerina", ["Alina", ["Nadia"]]]]]]]]
openBabushka(babushka)
```

Übungen zu Rekursion A

Schreibe die folgende Funktion rekursiv:

```
def summe_iterativ(liste):  
    gesamt_summe = 0  
  
    for element in liste:  
        gesamt_summe = gesamt_summe + element  
  
    return gesamt_summe  
  
# Beispiel  
zahlen = [5, 3, 2, 8]  
ergebnis = summe_iterativ(zahlen)
```

Lösung zu Rekursion A

```
def summe_rekursiv(liste):  
    # Wenn die Liste leer ist (Länge 0), gib 0 zurück.  
    if not liste:  
        return 0  
  
    # REKURSIVER SCHRITT  
    erstes_element = liste[0]  
    liste.pop(0) # Entferne das erste Element  
    return erstes_element + summe_rekursiv(liste)  
  
# Beispiel  
zahlen = [5, 3, 2, 8]  
ergebnis = summe_rekursiv(zahlen)
```

Übungen zu Rekursion B

- Schreibe eine Funktion `countVowels(word)`, welche die Anzahl Vokale in einem String zählt.

`countVowels("Hallihallo")` sollte 4 ergeben (a,i,a,o)

- Zuerst Iterativ
- Dann Rekursiv

Lösung zu Rekursion B

```
def countVowels(word):  
    counter = 0  
    for i in range(len(word)):  
        character = word[i]  
        if character in ["a", "e", "i", "o", "u"]:  
            counter = counter + 1  
    return counter  
  
print(countVowels("banana"))
```

```
def countVowels(word):  
    # Wenn das Wort leer ist, gibt die Rekursion 0 zurück  
    # Dies beendet die Kette der Aufrufe.  
    if len(word) == 0:  
        return 0  
  
    # Der erste Buchstabe des verbleibenden Strings  
    character = word[0]  
  
    # Der Rest des Strings (ohne das erste Zeichen)  
    rest_of_word = word[1:len(word)]  
  
    if character in ["a", "e", "i", "o", "u"]:  
        # Vokal gefunden: Zähle 1 + rekursives Ergebnis des Rests  
        return 1 + countVowels(rest_of_word)  
    else:  
        # Konsonant gefunden: Zähle 0 + rekursives Ergebnis des Rests  
        return 0 + countVowels(rest_of_word)  
  
print(countVowels("banana"))
```

Übungen zu Rekursion C

Annahmen

- Die Liste ist sortiert
- Das Suchelement kommt vor

Schreiben Sie die Funktion nun rekursiv!

```
def binaere_suche_iterativ(liste, zielwert):
    start_index = 0
    end_index = len(liste) - 1

    # Die Schleife muss laufen, solange es einen Suchbereich gibt.
    while start_index <= end_index:
        mitte_index = (start_index + end_index) // 2
        mitte_wert = liste[mitte_index]
        # 1. TREFFER
        if mitte_wert == zielwert:
            # Wir sind fertig - wir haben das Element gefunden
            return mitte_index
        # 2. LINKER PFAD
        elif zielwert < mitte_wert:
            end_index = mitte_index - 1 # Suchbereich auf die linke Hälfte
        # 3. RECHTER PFAD
        else: # zielwert > mitte_wert
            start_index = mitte_index + 1 # Suchbereich auf die rechte Hälfte
    # HINWEIS: wir nehmen an, dass der Wert existiert,
    return -1
```

```
zahlen = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]
suchelement = 38
index = binaere_suche_iterativ(zahlen, suchelement)
print("Index für ", suchelement, "ist", index)
```

Lösungen zu Rekursion C

```
def binaere_suche_rekursiv(liste, zielwert, start_index, end_index):
    if start_index > end_index:
        return -1

    # Mitte des aktuellen Bereichs berechnen
    mitte_index = (start_index + end_index) // 2
    mitte_wert = liste[mitte_index]

    # Treffer
    if mitte_wert == zielwert:
        return mitte_index

    # REKURSIVER SCHRITT
    # 1. Wert ist kleiner: Suche in der LINKEN Hälfte
    elif zielwert < mitte_wert:
        # Rekursiver Aufruf mit neuem End-Index (mitte_index - 1)
        neuer_end_index = mitte_index - 1
        return binaere_suche_rekursiv(liste, zielwert, start_index, neuer_end_index)

    # 2. Wert ist grösser: Suche in der RECHTEN Hälfte
    else: # zielwert > mitte_wert
        # Rekursiver Aufruf mit neuem Start-Index (mitte_index + 1)
        neuer_start_index = mitte_index + 1
        return binaere_suche_rekursiv(liste, zielwert, neuer_start_index, end_index)

zahlen = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]
suchelement = 38
index = binaere_suche_rekursiv(zahlen, suchelement, 0, len(zahlen) - 1)
print("Index für ", suchelement, "ist", index)
```

Computational Thinking

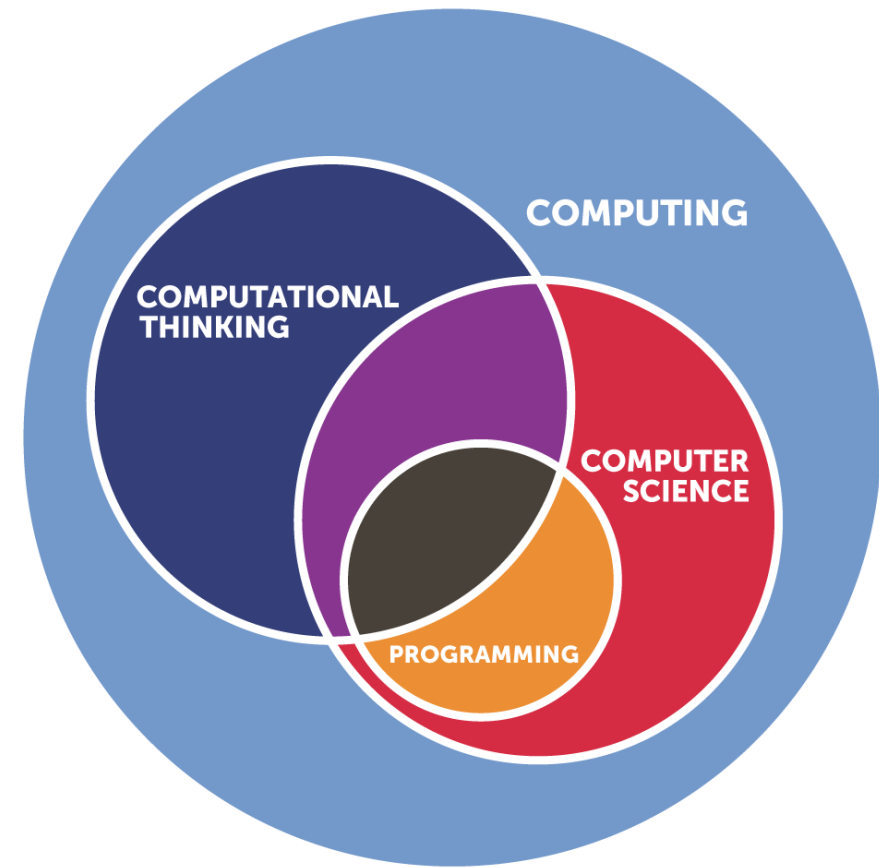
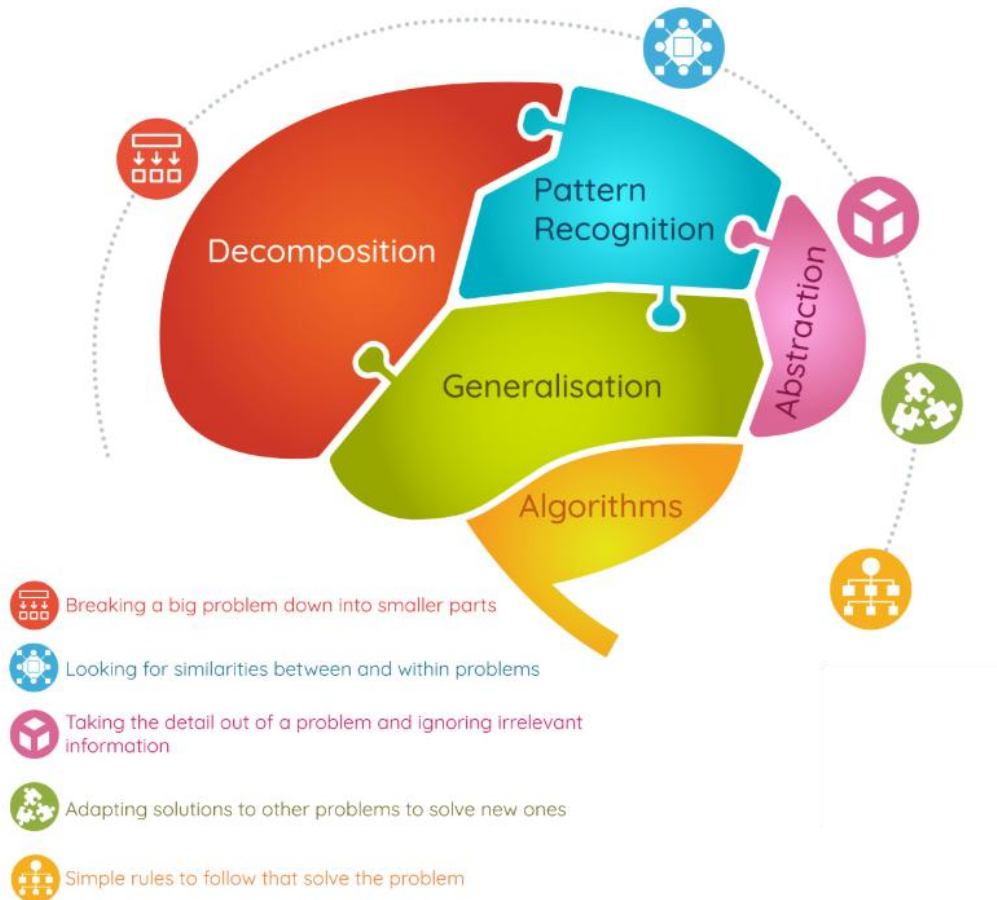
A decorative graphic consisting of several horizontal bars. A thick green bar spans the width of the slide. Below it, on the right side, are several thinner bars in varying shades of green and white, creating a stepped, architectural effect.

Computational Thinking



<https://www.youtube.com/watch?v=bXArJT45t8&t=501s>

Computational Thinking



Computational Thinking

- Lest den Artikel von **Jeanette Wing** (auf Teams)
- Bespricht zu zweit oder zu dritt die folgenden Fragen
 - Wo ist der Unterschied zwischen Algorithmik und Computational Thinking?
 - Wie könnten die Prinzipien des Computational Thinking in anderen Fächern angewendet werden?
 - Wie kann das Prinzip von Computational Thinking in Ihrem Leben wiedergefunden werden?
- Zeit: 20min

Computational Thinking

- **Computational Thinking** ist ein breit gefächelter **Ansatz zur Problemlösung**, bei dem man die Methoden der Informatik nutzt, um komplexe Probleme zu analysieren und zu formulieren. Es ist eine Denkweise, die aus vier Hauptschritten besteht:
 - **Zerlegung (Decomposition)**: Ein grosses, komplexes Problem wird in kleinere, überschaubare Teilprobleme zerlegt.
 - **Mustererkennung (Pattern Recognition)**: Ähnlichkeiten und wiederkehrende Muster in den Teilproblemen werden identifiziert, um die Lösungsfindung zu vereinfachen.
 - **Abstraktion (Abstraction)**: Man konzentriert sich auf die wesentlichen Aspekte der Probleme und ignoriert irrelevante Details.
 - **Algorithmen (Algorithms)**: Eine Schritt-für-Schritt-Anleitung zur Lösung des Problems wird entwickelt.

Computational Thinking vs Algorithmen

- Algorithmen sind das **Endprodukt** oder ein Teilbereich des Computational Thinking. Sie sind eine präzise, geordnete Abfolge von Anweisungen, die von einem Computer oder einer Person ausgeführt werden können, um ein Problem zu lösen.
- Computational Thinking hingegen ist der **gesamte Denkprozess**, der zur Entwicklung dieser Algorithmen führt.
- Während Algorithmen die **konkreten "Werkzeuge"** sind, um eine Aufgabe zu erledigen, ist Computational Thinking die **"Werkzeugkiste"** und die Strategie, wie man die richtigen Werkzeuge auswählt und anwendet.

Weitere Übungsaufgaben zu Rekursion



String Umkehren

Schreibe eine rekursive Funktion, die einen gegebenen String umkehrt.

Eingabe: "hallo"

Ausgabe: "ollah"

String Umkehren

```
def umkehren(s):  
    if len(s)==0:  
        return ""  
    else:  
        return s[len(s)-1] + umkehren(s[0:len(s)-1])
```

```
s = "hallo"  
print(umkehren(s))
```

Grösstes Element in einer Liste

Schreibe eine rekursive Funktion, die das grösste Element in einer Liste von Integern findet.

Eingabe: [8, 3, 15, 2, 10]

Ausgabe: 15

Grösstes Element in einer Liste

```
def groesstes(liste):  
    # 1. Abbruchbedingung: Die Liste hat nur ein Element  
    if len(liste) == 1:  
        return liste[0]  
  
    # 2. Rekursions-Fall: Vergleiche das erste Element mit dem Maximum des Rests  
    max_des_restes = groesstes(liste[1:len(liste)-1])  
    erstes_element = liste[0]  
    if erstes_element > max_des_restes:  
        return erstes_element  
    else:  
        return max_des_restes  
  
eingabe_liste = [8, 3, 15, 2, 10]  
ergebnis = groesstes(eingabe_liste)  
print(ergebnis)
```

N-tes Element suchen

Schreibe eine rekursive Funktion, die das Element an der n-ten Position in einer Liste zurückgibt (z.B. der 3. Index).

Eingabe: Liste ["A", "B", "C", "D"], Index 2

Ausgabe: "C"

N-tes Element suchen

```
def nteselement(liste, index):  
    if index == 0:  
        return liste[0]  
    else:  
        return nteselement(liste[1:len(liste)], index-1)
```

```
eingabe_liste = ["A", "B", "C", "D"]  
ergebnis = nteselement(eingabe_liste,2)
```

```
print(ergebnis)
```