

Notizen Semesterprüfung

Grundlagen

Algorithmen

- Landau-Symbole: Werden verwendet, um die Laufzeit von Algorithmen zu beschreiben. Besonders die sog. Gross-Oh-Notation ($O(\dots)$).
- Bei der Implementierung von Algorithmen muss man sich entscheiden, ob dies durch eine rekursive oder iterative Funktion tut. Genauso muss man sich entscheiden, ob er eine bestehende Datenstruktur verändert oder eine neue Datenstruktur zurückliefert. Schlussendlich muss man auch entscheiden, ob man eine Datenstruktur durch eine Klasse oder etwa durch eine Liste oder gar durch eine Hash-Tabelle implementiert.
- Eigenschaften von Algorithmen:
 - Determinismus: Das Verfahren ist determiniert, weil zu einer bestimmten Eingabe jedes Mal dasselbe Resultat ausgegeben wird.
 - Determiniertheit: Das Schema ist deterministisch, weil es jedem Teilergebnis einen eindeutigen nächsten Schritt zuordnet.
 - Terminiertheit: Die Anzahl der Schritte ist endlich, das Verfahren liefert nach dem finalen Schritt ein Ergebnis.
 - Finitheit: Der Algorithmus selbst muss eine endliche Beschreibung haben. Zudem muss der von ihm belegte Speicher zu jeder Zeit begrenzt sein und der Algorithmus muss ausführbar sein.

Was sind Algorithmen und weshalb ist es wichtig, die richtigen Algorithmen für das jeweilige Problem zu kennen?

- Ein Ablauf mit endlichen Schritten (terminiert) der mit den gleichen Eingaben immer die gleichen Ausgaben (determiniert) erbringt. Jedem Teilergebnis wird ein eindeutiger nächster Schritt zugeordnet (deterministisch). Finitheit: Der Algorithmus selbst muss eine endliche Beschreibung haben. Zudem muss er von ihm belegte Speicher zu jeder Zeit begrenzt sein und der Algorithmus muss ausführbar sein.
- Nicht jeder Algorithmus geht das gleiche Problem mit der gleichen Effizienz an --> wir müssen für das jeweilige Problem den richtigen Algorithmus auswählen

Was sagt die Größenordnung der Laufzeit aus und welche können wir unterscheiden? O-Notation oder Landau-Notation

- Wir ordnen die Algorithmen damit in ihre Komplexitätsklassen ein - also eigentlich wie lange sie dauern, bei steigender Größe der Eingabe
- Die Angabe ist hierbei jeweils unabhängig der Hardware
- O: Konstante Laufzeit

- $O(\log n)$: Logarithmisch - wächst langsam --> binäre Suche
- $O(n)$: Linear - wächst proportional zu n --> einfaches Durchlaufen einer Liste
- $O(n \log n)$: Log-linear - schneller als linear, aber langsamer als quadratisch
- $O(n^2)$: Quadratisch - Laufzeit steigt stark --> Bubblesort, doppelte Schleifen
- $O(n^n)$: Exponentiell - extrem ineffizient bei grossen n --> Rekursive Algorithmen wie Backtracking
- Es ist immer effizienter einen Algorithmus zu verbessern anstatt der Hardware!

Beispiel Suche in einem sortierten und nicht-sortierten Telefonbuch nach einem Eintrag. Performance?

- Bei unsortiertem Telefonbuch: $O(n)$: wächst linear, also proportional zu n
- Bei sortiertem Telefonbuch: $O(\log n)$: Logarithmisch, also wächst langsam

Logarithmen

Der Ausdruck $\log_{10} 100$ entspricht der Frage: "Wie viele 10er muss man miteinander multiplizieren, um 100 zu erhalten?" Die Antwort lautet $2: 10 \cdot 10 = 100$. Also ist $\log_{10} 100 = 2$.

Logarithmen sind die Umkehrung von Exponentialfunktionen.

Übrigens: Wenn es um die Laufzeit und die Landau-Notation geht, bedeutet \log stets \log_2 , also der Logarithmus zur Basis 2. \

Für eine Liste mit 8 Elementen gilt, $\log 8 == 3$, denn $2^3 = 8$.

Es müssen also höchstens 3 Schritte unternommen werden, um ein Element in einer Liste mit 8 Elementen zu finden. \

Rekursion

- Was sind Rekursionen?
 - Rekursionen sind Funktionen, die sich selbst aufrufen und haben zwei Bestandteile - den Basisfall (Abbruchbedingung) und den Rekursionsfall (sich selbst mit einem vereinfachten Problem aufrufen)
 - Statt das Problem direkt zu lösen, zerlegt eine Rekursion das Problem in kleinere Teilprozesse desselben Typs und löst diese rekursiv
- Wie funktionieren diese?
 - Der Rekursionsfall ruft sich selbst immer wieder mit einem vereinfachten Problem auf, bis die Abbruchbedingung (Basisfall) erreicht ist
 - Die Rekursionsaufrufe werden gestackt, was zu einem Stack-Overflow führen kann - -> und allgemein viel Arbeitsspeicher belegt
- Alternativen zu Rekursionen?
 - Die Iteration --> Schleifen
- Weshalb sind Rekursionen in Algorithmen so wichtig?

- Ein Problem in kleinere Teile zu zerlegen und separat zu lösen, macht einen Algorithmus verständlicher

Arrays, Listen und Iteratoren

- Welche Datenstrukturen kennt Python?
 - list --> Dynamisches Array, veränderbar --> ist geordnet (erscheint immer in der gleichen Reihenfolge wie ich sie eingefügt habe)
 - tuple --> unveränderbare Liste --> wird als Record-Alternative verwendet
 - set --> Ungeordnete Sammlung ohne Duplikate --> ist nicht geordnet
 - dict --> Key-Value-Pairs
- Einfach und doppelt verkettete Listen
 - Einfach: Jeder Knoten zeigt nur auf den nächsten Knoten
 - ermöglicht nur die Navigation vorwärts
 - Doppelt: Jeder Knoten zeigt auf den nächsten und den vorherigen Knoten
 - ermöglicht bidirektionale Navigation
- Wie iteriere ich über solche dynamischen Listen? Was ist O?
 - Der erste Knoten (head) wird ausgewählt, dann wird über die Verknüpfung der nächste Knoten ausgewählt
 - Laufzeitkomplexität
 - Durchlaufen: $O(n)$
 - Einfügen/Löschen am Anfang: $O(1)$
 - Einfügen/Löschen irgendwo nach dem Anfange: $O(n)$ --> da wir zuerst den korrekten Ort finden müssen
- Was ist ein Array im allgemeinen?
 - Unter Array versteht man eine Datenstruktur, die Elemente des gleichen Typs in einem kontinuierlichen Speicherblock speichert
 - Vorteile: Schneller Zugriff über Index: $O(1)$
 - Feste Grösse (in vielen Sprachen)
- Wie unterscheidet sich dazu die Arrays und Listen in Python?
 - In Python-Listen können beliebige Datentypen gespeichert werden
 - Es sind viele Methoden für die Python-list verfügbar
 - Die Python-Liste hat keine vorgegebene feste Grösse
- Implementieren Sie eine doppelt verkettete Liste
 - Siehe PyCharm PVA 01

Stack, Queues und Bags

- Skizzieren Sie, was der Unterschied zwischen Stack, Queues und Bags sind?
 - Stack
 - Stapel --> neues kommt oben drauf
 - LIFO
 - Queue

- Neues kommt hinten angehängt
 - FIFO
- Bag
 - Alles wird ohne Struktur in den Container geschmissen
 - Beim "ziehen" wird ein "zufälliges" Element entnommen
- Was ist der Datentyp Deque?
 - Double Ending Queue
 - Ich kann von beiden Seiten einfügen und auslesen
- Implementieren Sie mit Deque ein Stack und Queue
 - Siehe PyCharm PVA 01
- Für welche Problemstellung ist welche Datenstruktur geeignet?
 - Stack
 - Rückgängig-Funktion, rekursive Algorithmen
 - Queue
 - Warteschlangen, Aufgabenplanung, Datenstromverarbeitung
 - Bag
 - Sammlung von Elementen ohne Reihenfolge (z.B. Inventar in Spielen)

Sortieren

Komplexitätsübersicht

Algorithmus	Best Case	Average Case	Worst Case	Speicherbedarf	Stabil
Lineare Suche	$O(1)$	$O(n)$	$O(n)$	$O(1)$	n/a
Binäre Suche	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$	n/a
Selectionsort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	nein
Bubblesort	$O(n)$ (optimiert)	$O(n^2)$	$O(n^2)$	$O(1)$	ja
Insertionsort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	ja
Mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	ja
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	nein
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	nein
Priority Queue (Heap)	$O(1)$ fuer peek	$O(\log n)$ fuer insert	$O(\log n)$ fuer extract	$O(n)$	n/a

Algorithmus	Best Case	Average Case	Worst Case	Speicherbedarf	Stabil
Binomial Heap	$O(1)$ fuer peek	$O(1)$ fuer insert	$O(\log n)$ fuer extract	$O(n)$	n/a

Legende:

- JA = stabil (gleiche Werte behalten ihre Reihenfolge)
- NEIN = nicht stabil
- *optim.* = mit Abbruch, wenn keine Vertauschung mehr nötig ist

Wichtige Hinweise

- **Suche:**
 - Lineare Suche nur bei kleinen oder unsortierten Datenmengen sinnvoll.
 - Lineare Suche nur bei kleinen oder unsortierten Datenmengen sinnvoll.
 - Binäre Suche nur bei sortierten Datenstrukturen.
- **Einfache Sortierverfahren:**
 - Gut für kleine Datenmengen oder Lehrzwecke.
 - Insertionsort ist oft die beste Wahl bei fast sortierten Listen.
- **Effiziente Sortierverfahren:**
 - Mergesort ist stabil, benötigt aber zusätzlichen Speicher.
 - Quicksort ist oft am schnellsten in der Praxis, aber Worst Case beachten.
 - Heapsort ist speichereffizient, aber nicht stabil.
- **Prioritätswarteschlangen:**
 - Heap-Implementierungen sind Standard für effiziente `insert` / `extract` - Operationen.
 - In Python ist `heapq` ein Min-Heap — für Max-Heap Prioritäten invertieren.

Einfluss der Datenstruktur auf Sortieralgorithmen

Array vs. Verkettete Liste

Aspekt	Array (kontigu im Speicher)	Einfach/zweifach verkettete Liste
Zugriff auf Element	$O(1)$ direkter Indexzugriff	$O(n)$ sequentiell
Speicher	Fester Block, evtl. Overhead bei Vergrößerung	Mehr Speicher pro Element (Zeiger)
Vertauschen von Elementen	Einfach durch Indexzugriff	Aufwendig: Zeiger umhängen
Einfügen/Löschen in der Mitte	$O(n)$ (Verschieben nötig)	$O(1)$ wenn Position bekannt

Aspekt	Array (kontigu im Speicher)	Einfach/zweifach verkettete Liste
Cache-Lokalität	Sehr gut (benachbarte Elemente im Speicher)	Schlecht (Elemente verstreut im Speicher)

Folgen für Sortieralgorithmen:

- **Algorithmen mit vielen zufälligen Zugriffen** (z. B. Quicksort, Heapsort) sind auf Arrays deutlich schneller, da direkter Indexzugriff möglich ist.
- **Mergesort** kann auf verketteten Listen effizient sein, da das „Teilen“ und „Zusammenführen“ nur Zeigeroperationen erfordert (kein Kopieren).
- **Insertionsort** kann auf verketteten Listen vorteilhaft sein, wenn viele Einfügungen in der Mitte nötig sind.

7.2 Abhängigkeiten und Trade-offs zwischen den „drei Fragen“

Die „drei Fragen“ beziehen sich typischerweise auf:

1. **Laufzeitkomplexität** (Best/Average/Worst Case)
2. **Speicherbedarf**
3. **Stabilität** (Erhält Reihenfolge gleicher Elemente)

Wichtige Erkenntnis:

- Diese Eigenschaften sind **nicht völlig unabhängig**.
- **Trade-offs:**
 - **Speicher vs. Laufzeit:**
 - Mergesort ist stabil und hat gute Laufzeit $O(n \log n)$, benötigt aber $O(n)$ zusätzlichen Speicher.
 - Heapsort ist speichereffizient ($O(1)$), aber nicht stabil.
 - **Stabilität vs. Speicher:**
 - Stabile Sortierungen benötigen oft zusätzlichen Speicher (Ausnahme: Insertionsort).
 - **Laufzeit vs. Einfachheit:**
 - Einfache Algorithmen wie Insertionsort sind leicht zu implementieren, aber bei großen n ineffizient.
- **Beispiel:** Will man Stabilität **und** In-Place-Sortierung **und** $O(n \log n)$ Laufzeit, muss man oft Kompromisse eingehen oder komplexe Hybridverfahren einsetzen (z. B. Timsort in Python).

**  Merksatz **:

Die Wahl des Sortieralgoritmus hängt nicht nur von der theoretischen Komplexität ab, sondern auch von der Datenstruktur, den Speicherrestriktionen und der Frage, ob Stabilität erforderlich ist. Arrays

begünstigen Algorithmen mit direktem Indexzugriff, verkettete Listen profitieren von Verfahren, die auf sequentiellen Zugriff optimiert sind.

Entscheidungsmatrix – Wahl des Sortieralgorithmus

Datenstruktur	Anforderung(en)	Empfohlener Algorithmus	Begründung
Array	Schnellste Laufzeit , Stabilität egal, Speicher knapp	Quicksort (mit gutem Pivot)	Sehr gute Average-Case-Performance $O(n \log n)$, in-place, nutzt Cache-Lokalität.
Array	Stabilität + $O(n \log n)$ Laufzeit	Mergesort oder Timsort	Stabil, vorhersehbare Laufzeit, Timsort (Python <code>sort</code>) optimiert für reale Daten.
Array	Minimaler Speicher + $O(n \log n)$ Laufzeit	Heapsort	In-place, konstante Speicherkomplexität, aber nicht stabil.
Array	Sehr kleine n oder fast sortiert	Insertionsort	$O(n)$ im Best Case, sehr geringer Overhead.
Verkettete Liste	Stabilität + $O(n \log n)$ Laufzeit	Mergesort (Listen-Variante)	Kein Kopieren nötig, nur Zeigeränderungen, stabil.
Verkettete Liste	Einfache Implementierung	Insertionsort (Listen-Variante)	Einfaches Einfügen durch Zeigeränderung, gut bei kleinen n.
Verkettete Liste	Speicher knapp	Mergesort	Kein zusätzlicher Speicher außer Rekursionstack.
Beliebig	Prioritätswarteschlange benötigt	Heapsort oder Heap-API (<code>heapq</code>)	Heap-Struktur erlaubt effizientes <code>insert / extract</code> .

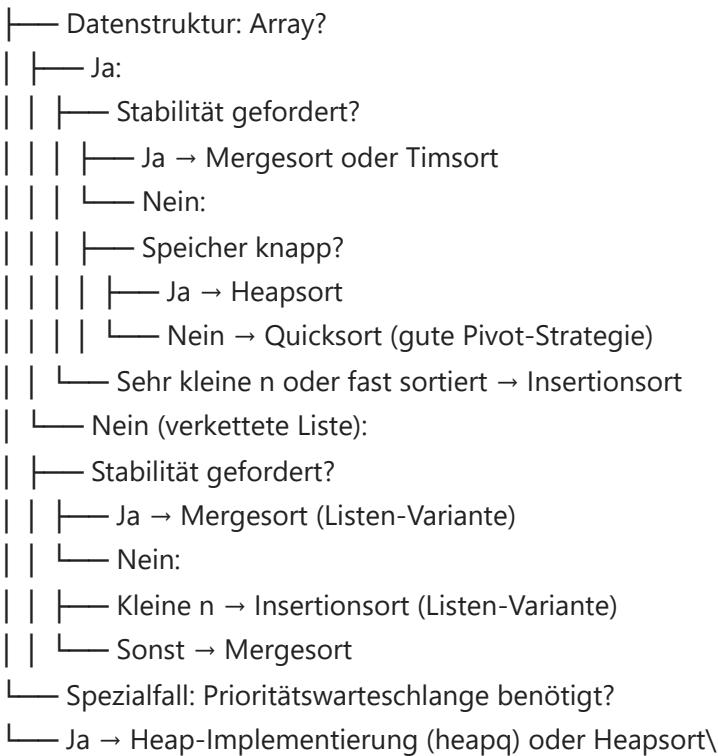
Entscheidungslogik (Merksätze)

- **Array + Stabilität** → Mergesort/Timsort.
- **Array + Speicher sparen** → Heapsort.
- **Array + Geschwindigkeit** → Quicksort (Pivot-Strategie beachten).
- **Liste + Stabilität** → Mergesort.
- **Liste + Einfachheit** → Insertionsort.
- **Fast sortiert** → Insertionsort schlägt oft komplexere Verfahren.

 **Tipp:** Wenn in einer Aufgabe nicht explizit gesagt wird, welche Datenstruktur vorliegt, **immer** annehmen, dass es ein Array ist. Falls Stabilität gefordert ist → Mergesort/Timsort nennen. Falls Speicher knapp ist → Heapsort nennen. Falls Geschwindigkeit im Vordergrund steht → Quicksort nennen, aber Worst Case erwähnen.

牢固树立 – Wahl des Sortieralgorithmus

Start



Selectionsort

- **Prinzip:**
 1. Finde das kleinste Element im unsortierten Teil.
 2. Tausche es mit dem ersten Element des unsortierten Teils.
 3. Wiederhole für den Rest.
- **Komplexität:**
 - $O(n^2)$ Zeit.
 - $O(1)$ Speicher.
- **Stabilität:** Nicht stabil.

```
In [ ]: def selection_sort(lst):  
    n = len(lst)  
    for i in range(n):  
        min_index = i  
        for j in range(i + 1, n):  
            if lst[j] < lst[min_index]:  
                min_index = j  
        lst[i], lst[min_index] = lst[min_index], lst[i]
```

Bubblesort

- **Prinzip:**
 - Vergleicht benachbarte Elemente und tauscht sie, falls nötig.

- Größtes Element „blubbert“ nach oben.
- **Komplexität:**
 - $O(n^2)$ Zeit.
 - $O(1)$ Speicher.
- **Stabilität:** Stabil.

```
In [ ]: def bubble_sort(lst):
    n = len(lst)
    for i in range(n):
        for j in range(0, n - i - 1):
            if lst[j] > lst[j + 1]:
                lst[j], lst[j + 1] = lst[j + 1], lst[j]
```

Insertionsort

- **Prinzip:**
 - Baut sortierte Teilliste auf, indem Elemente an der richtigen Stelle eingefügt werden.
- **Komplexität:**
 - $O(n^2)$ Zeit.
 - $O(1)$ Speicher.
- **Stabilität:** Stabil.
- **Vorteil:** Sehr effizient für kleine oder fast sortierte Listen.

```
In [ ]: def insertion_sort(lst):
    for i in range(1, len(lst)):
        key = lst[i]
        j = i - 1
        while j >= 0 and key < lst[j]:
            lst[j + 1] = lst[j]
            j -= 1
        lst[j + 1] = key
```

Mergesort

- **Prinzip:**
 1. Teile die Liste in zwei Hälften.
 2. Sortiere jede Hälfte rekursiv.
 3. Führe die beiden sortierten Hälften zusammen (Merge).
- **Komplexität:**
 - $O(n \log n)$ Zeit.
 - $O(n)$ Speicher.
- **Stabilität:** Stabil.

```
In [ ]: def merge_sort(lst):
    if len(lst) > 1:
        mid = len(lst) // 2
        left = lst[:mid]
```

```

        right = lst[mid:]

        merge_sort(left)
        merge_sort(right)

        i = j = k = 0
        while i < len(left) and j < len(right):
            if left[i] <= right[j]:
                lst[k] = left[i]
                i += 1
            else:
                lst[k] = right[j]
                j += 1
            k += 1

        while i < len(left):
            lst[k] = left[i]
            i += 1
            k += 1
        while j < len(right):
            lst[k] = right[j]
            j += 1
            k += 1
    
```

Quicksort

- **Prinzip:**
 1. Wähle ein **Pivot**-Element.
 2. Teile die Liste in zwei Teile: kleiner als Pivot, größer als Pivot.
 3. Sortiere beide Teile rekursiv.
- **Komplexität:**
 - Durchschnitt: **O(n log n)**.
 - Worst Case: **O(n²)** (schlechtes Pivot).
- **Speicher:** **O(log n)** (rekursiver Stack).
- **Stabilität:** Nicht stabil.

```
In [ ]: def quicksort(lst):
    if len(lst) <= 1:
        return lst
    pivot = lst[len(lst) // 2]
    left = [x for x in lst if x < pivot]
    middle = [x for x in lst if x == pivot]
    right = [x for x in lst if x > pivot]
    return quicksort(left) + middle + quicksort(right)
```

Heapsort

- **Prinzip:**
 1. Baue einen **Max-Heap** aus der Liste.
 2. Tausche das größte Element (Wurzel) mit dem letzten Element.

3. Reduziere Heap-Größe und heapify erneut.

- **Komplexität:**
 - **O(n log n)** Zeit.
 - **O(1)** Speicher.
- **Stabilität:** Nicht stabil.

```
In [ ]: def heapify(lst, n, i):  
    largest = i  
    l = 2 * i + 1  
    r = 2 * i + 2  
  
    if l < n and lst[l] > lst[largest]:  
        largest = l  
    if r < n and lst[r] > lst[largest]:  
        largest = r  
    if largest != i:  
        lst[i], lst[largest] = lst[largest], lst[i]  
        heapify(lst, n, largest)  
  
def heapsort(lst):  
    n = len(lst)  
    for i in range(n // 2 - 1, -1, -1):  
        heapify(lst, n, i)  
    for i in range(n - 1, 0, -1):  
        lst[i], lst[0] = lst[0], lst[i]  
        heapify(lst, i, 0)
```

Priority Queue

- **Definition:** Datenstruktur, bei der jedes Element eine Priorität hat.
- **Operationen:**
 - `insert(item, priority)`
 - `extract_max()` oder `extract_min()`
- **Implementierungen:**
 - **Liste** (unsortiert oder sortiert) → ineffizient.
 - **Heap** (Binary Heap, Binomial Heap, Fibonacci Heap) → effizient.

Priority Queue mit Heap

Der folgende Code könnte mit negativem Vorzeichen auch als Max-Heap verwendet werden.

```
In [ ]: import heapq  
  
# Min-Heap  
pq = []  
heapq.heappush(pq, (2, "Code schreiben"))  
heapq.heappush(pq, (1, "Kaffee trinken"))  
heapq.heappush(pq, (3, "Meeting"))
```

```

while pq:
    priority, task = heapq.heappop(pq)
    print(priority, task)

```

Binary-Sort

- **Prinzip:**
 - Kombiniert Binärsuche mit Insertionsort.
 - Findet die Einfügeposition mit Binärsuche, dann fügt es das Element ein.
- **Komplexität:**
 - $O(n^2)$ Zeit (Insertionsort).
 - $O(1)$ Speicher.
- **Stabilität:** Stabil.

```

In [ ]: def binary_search(lst, val, start, end):
    while start <= end:
        mid = (start + end) // 2
        if lst[mid] < val:
            start = mid + 1
        else:
            end = mid - 1
    return start

```

```

In [1]: from __future__ import annotations

from collections import deque
from dataclasses import dataclass, field
from typing import Generic, Iterator, List, TypeVar

T = TypeVar("T")

@dataclass
class Node(Generic[T]):
    value: T
    children: List["Node[T]"] = field(default_factory=list)

    def add(self, child: "Node[T] | T") -> "Node[T]":
        """Bequemes Anhängen; akzeptiert Node oder Wert."""
        if isinstance(child, Node):
            self.children.append(child)
        else:
            self.children.append(Node(child))
        return self


class Tree(Generic[T]):
    def __init__(self, root: Node[T] | None):
        self.root = root

    # -----
    # 1) Rekursive Traversierungen (DFS)

```

```

# -----
# def preorder_recursive(self) -> List[T]:
#     out: List[T] = []

#     def rec(n: Node[T] | None) -> None:
#         if n is None:
#             return
#         out.append(n.value)
#         for c in n.children:
#             rec(c)

#     rec(self.root)
#     return out

def postorder_recursive(self) -> List[T]:
    out: List[T] = []

    def rec(n: Node[T] | None) -> None:
        if n is None:
            return
        for c in n.children:
            rec(c)
        out.append(n.value)

    rec(self.root)
    return out

# -----
# 2) Nicht-rekursiv: Pre-Order als Iterator (Stack)
# -----
def iter_preorder(self) -> Iterator[T]:
    """Pre-Order ohne Rekursion (depth-first)."""
    if self.root is None:
        return
    stack: List[Node[T]] = [self.root]
    while stack:
        node = stack.pop()
        yield node.value
        # wichtig: Kinder rückwärts pushen
        for c in reversed(node.children):
            stack.append(c)

# -----
# 3) Breitensuche (Level-Order) als Iterator (Queue)
# -----
def iter_bfs(self) -> Iterator[T]:
    if self.root is None:
        return
    q: deque[Node[T]] = deque([self.root])
    while q:
        node = q.popleft()
        yield node.value
        for c in node.children:
            q.append(c)

```

```

# -----
# Demo
# -----
if __name__ == "__main__":
    #          A
    #          /   |   \
    #          B   C   D
    #          /   \   \
    #          E   F   G
    A = Node("A")
    B, C, D = Node("B"), Node("C"), Node("D")
    E, F, G = Node("E"), Node("F"), Node("G")
    A.add(B).add(C).add(D)
    C.add(E).add(F)
    D.add(G)

    tree = Tree(A)

    print("Pre-Order (rekursiv): ", tree.preorder_recursive())
    print("Post-Order (rekursiv):", tree.postorder_recursive())
    print("Pre-Order (iterativ): ", list(tree.iter_preorder()))
    print("BFS / Level-Order:    ", list(tree.iter_bfs()))

```

Pre-Order (rekursiv): ['A', 'B', 'C', 'E', 'F', 'D', 'G']
 Post-Order (rekursiv): ['B', 'E', 'F', 'C', 'G', 'D', 'A']
 Pre-Order (iterativ): ['A', 'B', 'C', 'E', 'F', 'D', 'G']
 BFS / Level-Order: ['A', 'B', 'C', 'D', 'E', 'F', 'G']

Suchen

Binomial Tree



Besser bei Werner - Heap-Algorithmen nachschlagen. Dort ist auch der Binomial Heap (Eine Menge von Heaps in absteigender Ordnung) erklärt.

Übung Binary Tree

In dieser Übung mussten wir einen Binary Tree in Python implementieren. mit Einfügen, Suchen und Löschen. Teil war auch folgendes

- Nachteile von Binary Tree: kann in eine Seite verfallen
- Andere Möglichkeiten zur Implementierung: Dictionary, Tupel, Listen
 - Ist Speichereffizienter aber mühsamer zu traversieren

Die Herausforderung der Suche

Die Suche ist eine der zentralen Aufgaben in der Informatik. Ob in Datenbanken, Dateisystemen oder Suchmaschinen — überall geht es darum, Informationen effizient wiederzufinden. Bei kleinen Datenmengen ist das trivial. Doch moderne Systeme arbeiten

mit Terabytes oder sogar Petabytes an Daten. Ein Beispiel: Die British Library besitzt etwa 75 TB an digitalisierbaren Daten. Google hingegen speichert über 1 Million TB — also mehr als 1000 Petabyte. Bei solchen Größenordnungen ist die Wahl des Suchalgorithmus entscheidend.

Ein einfacher linearer Suchalgorithmus mit Laufzeit $O(n)O(n)$ wird bei solchen Datenmengen schnell unbrauchbar. Selbst ein schneller Rechner, der ein Byte in 50 ns durchsuchen kann, würde für 100 TB über 1,5 Stunden brauchen. Deshalb sind effizientere Verfahren wie binäre Suchbäume, Hashing oder Bloomfilter notwendig.

Binäre Suchbäume: Struktur und Funktionsweise

Ein binärer Suchbaum (BST) ist eine Baumstruktur, bei der jeder Knoten maximal zwei Kinder hat — ein linkes und ein rechtes. Jeder Knoten besitzt einen Schlüsselwert, über den man ihn identifizieren kann. Die zentrale Eigenschaft lautet:

- Alle Schlüssel im linken Teilbaum sind kleiner oder gleich dem Schlüssel des Knotens.
- Alle Schlüssel im rechten Teilbaum sind größer oder gleich dem Schlüssel des Knotens.

Diese Ordnung ermöglicht eine Suche mit durchschnittlicher Laufzeit $O(\log n)$, sofern der Baum balanciert ist. Ist der Baum jedoch unbalanciert — etwa durch ungünstige Einfügereihenfolge — kann die Höhe bis zu n betragen, was wieder zu $O(n)$ führt.



Implementierung in Python

Die Klasse `BTree` repräsentiert einen binären Suchbaum. Jeder Knoten speichert:

- `key` : den Schlüssel
- `val` : den zugehörigen Wert
- `ltree` : linken Teilbaum
- `rtree` : rechten Teilbaum

Wichtige Methoden:

- `search(key)` : rekursive Suche nach einem Schlüssel
- `insert(key, val)` : rekursives Einfügen eines neuen Knotens
- `deleteND(key)` : nicht-destruktives Löschen eines Knotens
- `minEl()` / `maxEl()` : finden des kleinsten bzw. größten Elements
- `__str__()`, `__len__()`, `height()` : Ausgabe, Knotenzahl und Höhe

Balancierte Suchbäume

AVL-Bäume

AVL-Bäume sind binäre Suchbäume, bei denen die Balance gewahrt bleibt: Die Höhen der linken und rechten Teilbäume eines Knotens dürfen sich um höchstens 1 unterscheiden.
Jeder Knoten speichert:

- `height` : Höhe des Teilbaums
- `balance` : Differenz der Höhen (rechts minus links)

Beim Einfügen oder Löschen wird der Baum durch Rotationen automatisch balanciert.
Dadurch bleibt die Laufzeit garantiert bei $O(\log n)$, auch im Worst Case.

- Wie funktionieren AVL-Bäume?
- Was ist eine einfache und eine doppelte Rotation?
- Weshalb unterscheiden wir links und rechts?
- Implementierung von AVL-Baum mit Insert-, Delete- und Such-Funktion in Python

=> Unbedingt noch anschauen und entsprechende Notizen anfertigen + Code mit Kommentaren generieren!

Gute Seite: <https://techvidvan.com/tutorials/avl-tree-in-data-structure/> Im Repo von Werner ist ein entsprechender Source-Code vorhanden.

Rot-Schwarz-Bäume

Rot-Schwarz-Bäume sind ebenfalls balancierte BSTs, bei denen jeder Knoten eine Farbe (rot oder schwarz) besitzt. Durch strenge Farbregeln wird die Balance gewahrt. Sie sind etwas weniger strikt als AVL-Bäume, benötigen aber weniger Rotationen und sind daher oft schneller in der Praxis.

- Was ist der genaue Unterschied zu binären Bäumen und den AVL-Bäumen?
- Wie funktionieren Rot-Schwarz-Bäume?
- Welche Konstellationen können wir bei der Wiederherstellung der Konsistenz unterscheiden?
- Implementierung eines Rot-Schwarz-Baums mit Insert- und Such-Funktion

Einige Aussagen:

- Ein Node ist entweder Rot oder Schwarz
- Root und Leaves (NIL) sind Schwarz
- Wenn ein Node Rot ist, dann sind seine Kinder Schwarz
- Alle Pfade von einer Node zu ihren NILs beinhalten die gleiche Anzahl an schwarzen Nodes
- Der längste Pfad (root zu entferntestem NIL) ist niemals länger als die doppelte Länge des kürzesten Pfades (root zum am nächsten NIL)
 - Kürzester Pfad: alles schwarze Nodes
 - Längster Pfad: abwechselnd rot und schwarz

- Es gibt 4 Varianten der Regelverletzung und es ist klar, was bei welcher Variante getan werden muss => Siehe Bild unten.



Es ist erkennbar, dass bei jeder Variante die Buchstaben etwas anders platziert sind. Aber schlussendlich muss einfach in die Lösung rechts migriert werden.

=> Video ist in der Präsi von PVA 03 zu finden.

B-Bäume

B-Bäume sind eine Verallgemeinerung von Binärbümen. Jeder Knoten kann mehr als zwei Kinder und mehrere Schlüssel besitzen. Sie sind besonders für Datenbanken geeignet, da sie große Knoten verwenden, die viele Daten auf einmal laden. Das reduziert die Anzahl der teuren Festplattenzugriffe. Die Suchzeit wird dadurch physisch optimiert.



Knoten können auch mehrere Schlüssel haben.



Hashing und Hashtabellen

Hashing ist eine Technik, bei der ein Schlüssel durch eine Hashfunktion auf eine Speicheradresse abgebildet wird. Die Suche erfolgt direkt über diese Adresse — im Idealfall in O(1). Probleme entstehen durch Kollisionen, wenn mehrere Schlüssel denselben Hashwert erzeugen. Diese werden durch Verkettung oder offene Adressierung gelöst.

- Hashing bietet uns die unglaubliche Performance von O(1)!
- Wie funktioniert dies und welche Rolle hat dabei die Hashfunktion?
 - Werte werden gehasht und der Hash besagt, an welchem Index der Wert eingefügt werden muss
 - Dadurch ist es möglich beim Suchen einfach den Suchwert zu hashen und man hat direkt den index, an dem man das Resultat finden wird
 - Die Hashfunktion liefert den Hash und muss mit Kollisionen umgehen können
 - Sie muss dafür sorgen, dass möglichst wenig Kollisionen auftreten
 - Gleichmäßige Verteilung der hash-Werte (keys)
 - Einfach zu berechnen
 - Auflösen von Kollisionen
- Welche Bedeutung hat der Grad an Kollisionen und welche Ansätze haben wir, damit umzugehen und den Grad zu verbessern?
 - Der Grad besagt, wie wahrscheinlich eine Kollision für die gewählte Hash-Funktion ist
 - Möglichkeiten

- Verkette Liste
 - Wenn am berechneten index (Hash) schon ein Wert ist, wird einfach eine Verkettung vorgenommen (linked List?)
- Linear probing
- Plus 3 rehash
- Quadratic probing
- Double hashing
 - First hashing: Finde den index
 - Second hash: Finde den offset zum berechneten Index
 - Probing: Finde anhand des Offsets den nächsten freien Slot --> wird wiederholt, bis ein freier Slot gefunden ist oder die Tabelle komplett durchsucht wurde
- Was bedeutet dies für die Performance?
- Welche Hashfunktionen können wir unterscheiden?

=> Passendes Video: https://www.youtube.com/watch?v=g_eWEoC_vOo&t=147 => Das Repo von Werner zeigt hierfür verschiedene Hashfunktionen => Hashing ist perfekt fürs Caching geeignet!

Bloomfilter

Bloomfilter sind probabilistische Datenstrukturen, die mit mehreren Hashfunktionen arbeiten. Sie beantworten die Frage: „Ist ein Element möglicherweise enthalten?“ — mit hoher Geschwindigkeit und geringem Speicherbedarf.

Eigenschaften:

- **Vorteil:** extrem schnell, konstante Laufzeit, platzsparend
- **Nachteil:** falsch-positive Ergebnisse möglich (aber keine falsch-negativen)
- Welchen Zweck haben Bloomfilter?
 - Sie beantworten die Frage: ist ein Element im Set vorhanden?
 - Dafür antwortet der Bloomfilter mit "sicher nicht" oder "wahrscheinlich ja"
 - Da es ein möglicherweise Ja gibt, nennt man das "Probabilistic", dafür verwendet er viel weniger Speicherplatz als beispielsweise ein Hash-Table
 - Wir können keine Items vom Bloom-Filter entfernen, er vergisst nie
- Wie funktionieren diese?
 - Es wird eine bestimmte Anzahl Buckets (bspw. eine Liste mit n Elementen) erstellt und alle Buckets beinhaltet 1 Bit mit Wert 0
 - Wird ein Wert hinzugefügt, wird dieser mehrfach gehasht (gibt Indexe zurück) und die entsprechenden Bits (Buckets/Indexe) werden mit 1 belegt
 - Beim Suchen eines Wertes wird erneut die Hash-Funktion aufgerufen und geprüft, ob in den erzeugten Indexen eine 1 zu finden ist. Wenn ja, dann ist der Wert möglicherweise vorhanden

- Aber natürlich können die 1 auch durch andere Werte gesetzt worden sein, daher "möglicherweise ja"
- Implementierung eines Bloomfilters in Python
 - Siehe im eigenen Modul-Repo unter Bloomfilter.py

=> Gutes Video: <https://www.youtube.com/watch?v=V3pxngeLqw&t=7s>

Huffman-Codierung: Anwendung von Binärbäumen

Die Huffman-Codierung ist ein Komprimierungsverfahren, das Binärbäume verwendet, um Zeichen effizient zu codieren. Häufige Zeichen erhalten kurze Codes, seltene Zeichen lange. Die Codes sind nicht gleich lang — das ist der Unterschied zu ISO-8859-1 oder UTF-8.

Der Huffman-Baum wird so konstruiert, dass jeder Buchstabe ein Blatt ist. Der Pfad von der Wurzel zum Blatt ergibt den Code:

- Links = 0
- Rechts = 1

Beispiel:

- t = 1
- i = 00
- l = 01

Das Wort „tilt“ wird so zu `10000101`. Die Decodierung erfolgt durch Traversieren des Baums — nicht durch Zerlegen in 8-Bit-Blöcke.

Wichtige Eigenschaften:

- Keine Zyklen → keine Endlosschleifen
- Jeder Code eindeutig → keine Überlappung
- Nur Blattknoten enthalten Zeichen

Skip-Listen, Tries und Patricia-Tries

Skip-Listen

Skip-Listen sind verkettete Listen mit mehreren Ebenen. Sie erlauben eine Suche in $O(\log n)$, indem sie „Sprungverbindungen“ nutzen — ähnlich wie ein Expresszug, der nicht an jeder Station hält.

- Wie funktionieren Skip-Listen?
 - Es wird eine verkettete Liste generiert, die mit zusätzlichen Autobahnen (Skips) versehen werden
 - Beim Einfügen wird eine "zufällige" Zahl entschieden, die anzeigt wie viele Autobahnen (Skips) erstellt werden sollen

- Die Autobahnen werden dann vom vorherigen zum nächsten generiert
- Je mehr Autobahnen erstellt werden, desto grösser können auch die Sprünge entstehen
- Welche Vorteile haben diese bei der Suche?
 - Jede implementierte Autobahn (jeder Skip) erhöht die Wahrscheinlichkeit, dass ich mein Ziel rasch finde
- Implementierung einer Skip-Liste in Python und demonstrieren, wie das Einfügen schrittweise funktioniert

Tries

Tries sind Bäume, bei denen jeder Pfad ein Wort repräsentiert. Die Suche hängt nicht von der Anzahl der Wörter ab, sondern nur von der Wortlänge. Ideal für Textsuche.

- Was sind Tries und welchen Zweck haben sie?
- Wie unterscheidet sich der Patricia-Tries zum einfachen Trie?
- Wie funktionieren Tries?
- Implementieren eines einfachen Trie in Python

Vergleich der Suchtechniken

Technik	Laufzeit	Besonderheit
Lineare Suche	$O(n)$	Einfach, aber ineffizient bei großen Daten
Binärer Suchbaum	$O(h)$	Strukturabhängig, kann entarten
AVL / Rot-Schwarz	$O(\log n)$	Immer balanciert
B-Baum	$O(\log n)$	Optimiert für Festplattenzugriffe
Hashing	$O(1)$	Extrem schnell, Kollisionen möglich
Bloomfilter	$O(1)$	Speicherarm, falsch-positive möglich
Trie	$O(\text{Wortlänge})$	Ideal für Textsuche

Graphen

Was ist ein Graph?

Ein **Graph** $G = (V, E)$ besteht aus

- **Knoten (Vertices)** V
- **Kanten (Edges)** $E \subseteq V \times V$

Anwendungsbeispiele:

- Straßennetz → Knoten = Städte, Kanten = Verbindungen

- Internet → Knoten = Webseiten, Kanten = Links
- Software-Module, Ablaufpläne, soziale Netzwerke, Hierarchien

Wichtig: Die **räumliche Anordnung** spielt keine Rolle – nur, welche Knoten verbunden sind.

Repräsentation von Graphen

Zwei Hauptformen: **1. Adjazenzmatrix**

- Matrix $A = (a_{ij})$, mit $a_{ij} = 1$ falls $(i,j) \in E$, sonst 0
- Vorteil: schneller Zugriff $O(1)$ auf Kantenprüfung
- Nachteil: hoher Speicherverbrauch bei dünn besetzten Graphen

2. Adjazenzliste

- Jeder Knoten speichert seine Nachbarn (z. B. als dict)
- Vorteil: effizient beim Durchlaufen von Nachbarn
- Nachteil: Kantenprüfung langsamer $O(\deg(i))$

```
In [ ]: class Graph:
    def __init__(self, n):
        self.numNodes = n
        self.vertices = [{ } for _ in range(n+1)]

    def addEdge(self, i, j, weight=None):
        self.vertices[i][j] = weight

    def isEdge(self, i, j):
        return j in self.vertices[i]

    def G(self, i):
        return self.vertices[i].keys()

    def V(self):
        return range(1, self.numNodes + 1)
```

Breiten- und Tiefensuche

Graphen können systematisch **durchlaufen** werden, um Strukturen zu analysieren.

Breitensuche (BFS – Breadth First Search)

Idee: Zuerst Startknoten, dann alle Nachbarn, dann Nachbarn der Nachbarn usw. Verwendet eine **Queue** (FIFO).

Wichtige Konzepte:

- $d[i]$: Abstand des Knotens i vom Startknoten
- $\text{pred}[i]$: Vorgänger von i im Suchbaum

Algorithmus (BFS):

```
In [ ]: def bfs(s, graph):
    q = Queue()
    d = [-1 for _ in range(graph.numNodes+1)]
    pred = [None for _ in range(graph.numNodes+1)]
    d[s] = 0
    q.enqueue(s)

    while not q.isEmpty():
        v = q.dequeue()
        for u in graph.G(v):
            if d[u] == -1:
                d[u] = d[v] + 1
                pred[u] = v
                q.enqueue(u)
    return d, pred
```

→ Ergebnis:

- $d[i]$ = kürzeste Entfernung (in Kanten) von s nach i
- pred beschreibt den BFS-Spannbaum

Anwendung:

- Finden von Zusammenhangskomponenten
- Bestimmung kürzester Wege (ungewichtete Graphen)

Tiefensuche (DFS – Depth First Search)

Idee: So tief wie möglich in einen Pfad folgen, dann zurücksetzen (Backtracking). Verwendet **Stack** (LIFO).

Algorithmus (DFS iterativ):

```
In [ ]: def dfs(s, graph):
    pred = [None for _ in range(graph.numNodes+1)]
    st = Stack()
    v = s

    while True:
        unvisited = [u for u in graph.G(v) if pred[u] is None and u != s]
        if unvisited:
            u = unvisited[0]
            st.push(v)
            pred[u] = v
            v = u
        elif not st.isEmpty():
            v = st.pop()
        else:
            break
    return pred
```

Eigenschaften:

- Entdeckt Zyklen, Strukturen, Reihenfolgen
- Grundlage für viele weitere Algorithmen
- Lässt sich auch **rekursiv** formulieren

Topologische Sortierung

Ziel: Ordne Knoten eines **DAG** (Directed Acyclic Graph) so, dass alle Kanten von links nach rechts zeigen.

Algorithmus (über modifizierte DFS):

```
In [ ]: def topSort(s, graph):  
    topLst = []  
    st = Stack()  
    v = s  
    while True:  
        unvisited = [u for u in graph.G(v) if pred[u] is None]  
        if unvisited:  
            u = unvisited[0]  
            st.push(v)  
            pred[u] = v  
            v = u  
        elif not st.isEmpty():  
            topLst.append(v)  
            v = st.pop()  
        else:  
            topLst.append(v)  
            break  
    topLst.reverse()  
    return topLst
```

Beispielanwendung: Aufgabenabhängigkeiten, Projektplanung, Anziehreihenfolge.

Kürzeste Wege

Dijkstra-Algorithmus

Finde kürzeste Wege **von einem Startknoten** zu allen anderen.

Prinzip: Greedy

- Wähle den Knoten mit minimalem bekannten Abstand
- Aktualisiere Nachbarn, falls kürzere Pfade gefunden werden

```
In [ ]: def dijkstra(u, graph):  
    n = graph.numNodes  
    l = {u: 0}; W = set(graph.V())  
    F, k = [], {}
```

```

while W:
    lv, v = min([(l[x], x) for x in l if x in W])
    W.remove(v)
    if v != u:
        F.append(k[v])
    for neighbor in graph.G(v):
        if neighbor in W and (neighbor not in l or l[v] + graph.w(v, neighbor) < l[neighbor]):
            l[neighbor] = l[v] + graph.w(v, neighbor)
            k[neighbor] = (v, neighbor)
return l, F

```

Eigenschaften:

- Laufzeit $O(V^2)$ (oder $O(E \log V)$ mit Heaps)
- Funktioniert nur bei **positiven Kantengewichten**

Warshall-Algorithmus

Berechnet **kürzeste Wege zwischen allen Knotenpaaren** → All-Pairs Shortest Paths

Formel: $W_k[i,j] = \min(W_{k-1}[i,j], W_{k-1}[i,k] + W_{k-1}[k,j])$

```
In [ ]: def warshall(graph):
    n = graph.numNodes + 1
    W = [[graph.w(i,j) for j in graph.V()] for i in graph.V()]
    for k in range(n):
        for i in range(n):
            for j in range(n):
                W[i][j] = min(W[i][j], W[i][k] + W[k][j])
    return W
```

Laufzeit: $O(V^3)$ **Verwendung:** Analyse komplexer Netzwerke, transitive Hülle.

Minimale Spannbäume (MST)

Ziel: Finde Untermenge der Kanten, die alle Knoten verbindet – mit minimaler Gesamtkosten.

Klassische Algorithmen:

- **Prim:** Wachsender Baum, wähle jeweils billigste neue Kante zum bestehenden Teilbaum.
- **Kruskal:** Sortiere Kanten nach Gewicht, füge sie hinzu, solange kein Zyklus entsteht.

Anwendung: Netzwerk-Design, Infrastruktur, Clustering.

Maximaler Fluss

Ziel: Maximiere Fluss von Quelle s zu Senke t bei Kapazitätsgrenzen auf Kanten.

Grundprinzip (Ford–Fulkerson):

1. Starte mit 0-Fluss
2. Suche augmentierende Pfade im Restgraphen
3. Erhöhe Fluss entlang dieser Pfade
4. Wiederhole, bis kein Pfad mehr existiert

Ergebnis: Maximal möglicher Fluss = Summe aller ausfließenden Flüsse aus s.

Anwendungen: Netzwerke, Produktionsplanung, Transport, Matching.

Strings und komplexe Probleme

Tries – effiziente Struktur für Zeichenketten

Das Wort Trie kommt von retrieval. Ein Trie ist eine baumartige Struktur, die speziell dafür entwickelt wurde, Strings bzw. Schlüssel aus Zeichenfolgen effizient zu speichern und nachzuschlagen.

Tries (auch Präfixbäume) sind Baumstrukturen, bei denen jede Kante ein Zeichen repräsentiert. Damit lassen sich Strings extrem effizient vergleichen oder suchen.

Nutzen:

- perfekt für große Mengen ähnlicher Wörter (Autocomplete, Dictionaries)
- ideal für Prefix-Queries
- Suchzeit $\approx O(m)$ für Musterlänge m (lässt sich vom Gesamttext n!)

Aufbau:

- Wurzel ohne Zeichen
- jede Ebene steht für das nächste Zeichen
- Pfad von der Wurzel bis zu einem Knoten = Präfix
- „Ende-eines-Worts“-Markierungen steuern Treffererkennung

Vorteil gegenüber Hashing:

- kann Teilstrings strukturell erkennen
- erlaubt geordnete Stringoperationen
- eignet sich für *multiple patterns auf einmal*

Nachteile:

- braucht oft mehr Speicher
- eher komplexere Implementierung

Tries werden eingesetzt, wenn:

- viele **Strings mit gemeinsamen Präfixen** existieren

- schnelle **Such-, Einfüge- und Löschoperationen** benötigt werden
- Präfixabfragen wichtig sind

Typische Anwendungsfälle:

- Auto vervollständigung (Search Bars)
- Rechtschreibprüfung / Wörterbücher
- IP-Routing
- Speicherung von Symboltabellen
- DNA-Sequenzen

Kerneigenschaften

- Jeder Pfad vom Root zu einem Knoten entspricht einem **Präfix**
- Ein vollständiges Wort wird durch ein **Ende-Markierung** (z. B. `isEndOfWord`) gekennzeichnet
- Die Laufzeit für Suche/Einfügen ist **$O(m)$** , wobei m die Länge des Wortes ist (nicht die Anzahl gespeicherter Wörter!)



Wie unterscheidet sich der Patricia Trie vom einfachen Trie?

Ein **Patricia Trie** (auch *Radix Tree* oder *Compressed Trie*) ist eine **optimierte Variante** des normalen Tries.

Hauptunterschiede

Einfacher Trie	Patricia Trie
Jeder Buchstabe = ein Knoten	Mehrere Buchstaben pro Knoten
Viele Knoten mit nur einem Kind	Ein-Kind-Pfade werden zusammengefasst
Höherer Speicherverbrauch	Deutlich speichereffizienter
Einfach zu implementieren	Komplexere Logik

Idee hinter dem Patricia Trie

- **Ketten von Knoten mit genau einem Kind werden komprimiert**
- Statt einzelner Zeichen speichert ein Knoten **ganze Teilstrings**
- Die Struktur bleibt logisch gleich, ist aber **flacher**

Vorteil

- Weniger Knoten → weniger Speicher
- Schnellere Traversierung in der Praxis



Wie funktionieren Tries?

Grundprinzip

- Der Trie startet mit einem **Root-Knoten** (ohne Zeichen)
- Jeder Knoten repräsentiert **ein Zeichen**
- Kinder eines Knotens stehen für **mögliche nächste Zeichen**
- Ein Wort ist gespeichert, wenn:
 - der Pfad aller Zeichen existiert
 - der letzte Knoten als **Wortende markiert** ist

Beispiel (Wörter: cat , car , dog)



- Gemeinsame Präfixe (ca) werden **nur einmal gespeichert**
- Dadurch ist der Trie besonders effizient bei ähnlichen Wörtern

Operationen

- **Suchen:** Zeichen für Zeichen dem Baum folgen
- **Einfügen:** Fehlende Knoten anlegen
- **Löschen:** Wort-Ende entfernen, evtl. unnötige Knoten löschen
- **Prefix-Suche:** Traversierung bis zum Präfix, danach alle Unterbäume

String-Matching – Muster im Text finden

Ausgehend aus Kapitel 7 (Praktische Algorithmitk mit Python):

Der primitive Algorithmus

Vergleicht an jeder Position und läuft in **O(n·m)** für Textlänge n und Musterlänge m.

```
In [ ]: def match(M, T):  
    return [i for i in range(len(T)-len(M))  
           if all(T[i+j] == M[j] for j in range(len(M)))]
```

Bequem, aber langsam.

Endliche Automaten

Idee: Erkenne Präfixe des Musters im Text, um unnötige Vergleiche zu vermeiden. Man baut einen deterministischen endlichen Automaten (DEA) mit Zuständen 0..m.

Vorteil:

- Durchlauf ist linear: **O(n)**

Nachteil:

- Konstruktion ist aufwändiger (aber machbar)

Wie funktioniert der primitive Algorithmus für String-Matching?

Der **primitive (naive) String-Matching-Algorithmus** vergleicht ein **Pattern** Zeichen für Zeichen mit allen möglichen Positionen im Text.

Vorgehen

- Das Pattern wird an jeder möglichen Startposition im Text angesetzt
- Zeichen werden **von links nach rechts** verglichen
- Bei einem Fehler → Abbruch und **Verschiebung um eine Position**
- Bei vollständiger Übereinstimmung → Treffer gefunden

Beispiel

Text: ABABAC Pattern: ABA

Vergleiche:

- Position 0: ABA = ABA → Treffer
- Position 1: BAB ≠ ABA
- Position 2: ABA = ABA → Treffer

Laufzeit

- **Worst Case:** $O(n \cdot m)$

mit

- n = Länge des Textes
- m = Länge des Patterns

Was ist sein Nachteil und wie helfen endliche Automaten?

Nachteil des primitiven Algorithmus

- Viele **redundante Vergleiche**
- Bereits verglichene Zeichen werden **nicht wiederverwendet**
- Besonders ineffizient bei:
 - langen Texten
 - sich wiederholenden Mustern

👉 Beispiel: Pattern AAAA , Text AAAAAAAAAA → sehr viele unnötige Vergleiche

Wie helfen endliche Automaten?

Ein **endlicher Automat**

- merkt sich den **aktuellen Matching-Fortschritt**
- verarbeitet den Text **in einem Durchlauf**

- benötigt **keine Rücksprünge im Text**
- ➡ Jeder Textbuchstabe wird **genau einmal** gelesen

Laufzeit nach Vorverarbeitung: $O(n)$

Was ist ein endlicher Automat und wie lässt er sich darstellen?

Definition

Ein **endlicher Automat (Finite Automaton)** ist ein mathematisches Modell mit:

- einer **endlichen Anzahl von Zuständen**
- **Übergängen** zwischen Zuständen
- einem **Startzustand**
- optional **akzeptierenden Zuständen**

Formal: $(Q, \Sigma, \delta, q_0, F)$

Symbol	Bedeutung
Q	Menge der Zustände
Σ	Alphabet
δ	Übergangsfunktion
q_0	Startzustand
F	akzeptierende Zustände

Wird F erreicht, dann wird die Eingabe vom Automaten akzeptiert. F spiegelt also nicht einfach alle möglichen Endzustände wider, sondern nur die, welche wir akzeptieren wollen.

Darstellung

- **Zustandsdiagramm (Graph)** Kreise = Zustände Pfeile = Übergänge Akzeptierte Endzustände = doppelter Kreis Startzustand = Pfeil, der aus dem "nirgendwo" kommt
- **Übergangstabelle**
- Diagramm



👉 In Prüfungen meist: **Zustandsdiagramm**

Knuth–Morris–Pratt (KMP)

Das Arbeitstier in vielen Standardbibliotheken. Es nutzt eine **Verschiebetabelle P**, die für jede Position im Muster die Länge des größten passenden Präfixes angibt.

Laufzeit:

- Musteranalyse: $O(m)$
- Suche: $O(n)$
- total also: **$O(n+m)$**

Wie funktioniert der KMP-Algorithmus?

Zusammenspiel zwischen **LPS** (Longest Proper Prefix, which is also a Suffix) und dem Vermeiden unnötiger Vergleiche

Der **Knuth-Morris-Pratt-Algorithmus (KMP)** ist ein String-Matching-Algorithmus, der ein Pattern effizient in einem Text sucht, **ohne im Text zurückzuspringen**.

Grundidee

- Vor dem Suchen wird das Pattern **vorverarbeitet**
- Dabei wird das **LPS-Array** (Longest Proper Prefix which is also Suffix) berechnet
- Das LPS speichert, **wie viel des bereits gematchten Präfixes wiederverwendet werden kann**, wenn ein Vergleich fehlschlägt

Was ist das LPS-Array?

Für jede Position i im Pattern:

$LPS[i]$ = Länge des **längsten echten Präfixes** von $\text{pattern}[0..i]$, das gleichzeitig ein **Suffix** von $\text{pattern}[0..i]$ ist.

👉 „echt“ = nicht das ganze Wort selbst

Warum vermeidet KMP unnötige Vergleiche?

- Beim Mismatch:
 - **wird das Pattern nicht neu am Text ausgerichtet**
 - der Vergleich setzt im Pattern an $LPS[j-1]$ fort
 - Bereits geprüfte Zeichen werden **nicht erneut verglichen**
- Das spart massiv Vergleiche bei überlappenden Mustern

Wie verbessert sich die Größenordnung O?

Algorithmus	Laufzeit
Primitiv / Naiv	$O(n \cdot m)$
KMP	$O(n + m)$

- n = Länge des Textes
- m = Länge des Patterns

Warum $O(n + m)$?

- LPS-Berechnung: **$O(m)$**

- Durchlauf durch den Text: **O(n)**
- Kein Zurückspringen im Text

Skizzieren Sie den Algorithmus an einem Beispiel

Beispiel

Text: ABABABAC

Pattern: ABABAC

LPS-Array für das Pattern

Index: 0 1 2 3 4 5

Pattern: A B A B A C

LPS: 0 0 1 2 3 0

Ablauf (vereinfacht)

Text: A B A B A B A C

Pattern: A B A B A C

↑ ↑ ↑ ↑ ↑ X

- Mismatch bei C vs B
- Statt Neustart:
 - springe im Pattern zu LPS[4] = 3
- Bereits gematchtes „ABA“ wird weiterverwendet

→ Treffer ohne erneute Vergleiche im Text

Kernidee: Wenn bei einem Mismatch Zeichen übereinstimmten, nutze diese Information und springe im Muster zurück – aber *nicht* im Text.

Zuerst wird eine Longest-Prefix-Suffix Table generiert. Der Such-String wird in einzelne Zeichen zerteilt und basierend auf den Zeichen wird eine neue Table generiert.

Suchstring: A B A B C A B A B Generierter Table: 0 0 1 2 0 1 2 3 4

Also: Wenn ein Zeichen noch keine identischen Vorgänger hat (z.B. A und vorher auch bereits A), dann wird eine 0 gekennzeichnet. An der dritten Position ist zu erkennen, dass A jetzt eine 1 erhält. Nämlich, weil es vorher bereits ein A gegeben hat. Und an der 4 Position hat es eine 2, weil es vorher bereits A B hat.

Mit diesem LPS wird dann auf dem Fliesstext gesucht. Dabei werden zwei Variablen (i für LPS und j für den Fliesstext) jeweils erhöht und verglichen. Wird ein Treffer erzielt, wird die Variable für LPS um die Zahl im LPS verringert. Und dann wird wiederholt. Hier der Code dazu:

```
In [ ]: n = len(text)
m = len(pattern)
lps = LPS(pattern) # muss separat programmiert werden
```

```

i = j = 0

while i < n:
    if pattern[j] == text[i]:
        i += 1
        j += 1
    if j == m:
        print("Pattern found at Index: ", j)
        j = lps[j-1]
    else:
        if i < n and pattern[j] != text[i]:
            if j != 0:
                j = lps[j-1]
            else:
                i += 1
        else:
            i += 1

```

Boyer–Moore

Vergleicht **von rechts nach links** und nutzt zwei Heuristiken:

- **Bad Character:** verschiebt das Muster basierend auf dem nicht passenden Zeichen
- **Good Suffix:** nutzt die Struktur der bereits passenden Endung

Vorteile:

- extrem schnell im Durchschnitt
- oft sublineare Laufzeiten

Nachteil:

- Worst-Case immer noch $O(n \cdot m)$, aber selten relevant
- komplexer als KMP

Rabin–Karp

Verwendet Hashes. Gut für:

- *Multiple Pattern Search*
- schnelles „Candidate Filtering“

Laufzeit:

- average-case $O(n + m)$
- worst-case $O(n \cdot m)$, wenn viele Hash-Kollisionen

Travelling-Salesman-Problem (TSP)

(Referenz: Kapitel 8, *Praktische Algorithmik mit Python*)

Problemdefinition

Gegeben ein Graph – finde die **kürzeste Rundtour**, die jede Stadt genau einmal besucht.

Charakteristik

- **NP-vollständig** → keine bekannten polynomialen Algorithmen
- kleine Instanzen: exakt lösbar
- große Instanzen: Heuristiken / Approximationen nötig

Exact Methods

Brute Force

Alle Permutationen testen — fröhliche Factorial-Explosion. Laufzeit: **O(n!)**

Dynamische Programmierung (Held–Karp)

Optimal, aber besser strukturiert:

- nutzt das Optimalitätsprinzip
- speichert Teillösungen $T(i, S)$ für Startknoten i und Knotensets S
- reduziert die Laufzeit stark:

Laufzeit: $O(n^2 \cdot 2^n) \Rightarrow$ gut für 20–25 Knoten, darüber schnell unbrauchbar.

Das ist das klassische Beispiel, um Dynamische Programmierung an einem harten Problem zu erklären.

Greedy-Algorithmen

(in Verbindung mit [2] Kapitel 10)

Philosophie

Eine Entscheidung nach der anderen — jeweils die lokal beste Option. Hohe Geschwindigkeit, niedrige Garantie.

Greedy ist ideal, wenn:

- Problem „mathematisch schön“ ist, z. B. Dijkstra, Kruskal
- du *Approximationslösungen* brauchst
- NP-vollständige Probleme pragmatisch gelöst werden sollen
- Wir nur eine angenäherte Lösung benötigen und nicht die optimale Lösung

Greedy in TSP

Greedy ist keine optimale Lösung für TSP (außer in Spezialfällen), aber liefert gute Näherungen.

Beispiele:

Nearest Neighbor

Immer nächstgelegene Stadt wählen. Probleme: lokale Verzerrungen, Endphase oft unvorteilhaft.

Nearest/Farthest/Random Insertion

Aufbau einer Tour durch Einfügen neuer Knoten. Überraschung: **Random Insertion schlägt oft beide anderen Methoden.**

Tourverschmelzung

Stichtouren erzeugen, optimal verschmelzen → gute, stabile Lösungen.

Dynamische Programmierung

(basiert auf [2] Kapitel 11 und TSP-DP)

Wann anwendbar?

Wenn:

- das Problem eine *Optimierungsaufgabe* enthält
 - wenn wir ein Minimum oder ein Maximal finden wollen
 - Die längste gemeinsame Teilfolge von zwei Strings finden
- es in **überlappende Teilprobleme** zerlegbar ist
- eine Art „Gitter“, Tabelle oder rekursive Struktur existiert
- der Wert größerer Probleme aus kleineren ableitbar ist

Kerngedanke

Statt wieder und wieder dieselben Teilprobleme zu lösen, speicherst du alle Zwischenresultate in einer Tabelle.

Das spart Zeit — oft dramatisch.

Typische Struktur

- Tabelle (1D, 2D oder 3D)
- Zellen entsprechen Teilproblemen
- Übergangsformel: kombiniere vorherige Lösungen

Beispiele, die du kennen solltest

- TSP (Held–Karp): klassischer DP-Benchmark
- Rucksackproblem
- Edit-Distance / Levenshtein
- Fibonacci (klassisches Einstiegsexperiment)

Abgrenzung zu Greedy

Greedy entscheidet lokal, DP stellt systematisch alle Optionen zusammen.

Wenn Greedy versagt → DP ist oft korrekt, aber teurer.