Algorithmen & Datenstrukturen

by Michelle Honegger

ETH Zürich – HS2025

Contents

1 Induktion, Asymptotisches Wachstum

1.1 Induktion

Die vollständige Induktion ist ein Beweisverfahren für Aussagen über die natürlichen Zahlen.

Vorgehen:

1. Induktionsanfang (IA/BC): Zeige $A(1)$.
2. Induktionsannahme (IH): Dabei wird angenommen, dass $A(k)$ gilt.
3. Induktionsschritt (IS): Zeige $A(k) \Rightarrow A(k+1)$.

Beispiel:

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}, \quad \forall n \in \mathbb{N}.$$

IA/BC: $1 = \frac{1 \cdot 2}{2}$.
IH: Wir nehmen an die Eigenschaft $1 + 2 + ... + k = \frac{k(k+1)}{2}$ gilt für irgendein $k \in \mathbb{N}$.
Englisch: Assume the property holds for some integer $k \in \mathbb{N}$ that is $1+2+...+k = \frac{k(k+1)}{2}$.
IS

$$1 + 2 + \cdots + k + (k+1) = \frac{k(k+1)}{2} + (k+1) = \frac{(k+1)(k+2)}{2}.$$

Normalerweise markieren wir noch die Stelle an welcher wir den Induktionsschritt verwendet haben und beenden unseren Beweis mit folgendem Satz: Hiermit ist es mittels Induktion bewiesen, dass $1 + 2 + ... + n = \frac{n(n+1)}{2}$ für jedes $n \in \mathbb{N}$ gilt. Englisch: Thus the property is proven for for any integer $n \in \mathbb{N}$ by the principle of mathematical induction. Wir benutzen jeweils einen neuen Variablen Namen in unserem Beweis als im Statement. Dies liegt daran, dass wir im Statement mit n von "allen Zahlen" sprechen. Im Beweis selbst fixieren wir eine Zahl k und beweisen es dann für $k + 1$.

1.2 Asymptotisches Wachstum

Zur Analyse von Algorithmen betrachtet man das Wachstumsverhalten von Funktionen $f, g : \mathbb{N} \to \mathbb{R}^+$.

Definition: f wächst asymptotisch schneller als g, wenn

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0.$$

L'Hôpital: Für differenzierbare Funktionen gilt:

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = \lim_{x \to \infty} \frac{f'(x)}{g'(x)}.$$

1.3 Leetcode of the Week

- Problem 509: Fibonacci Number

2 $\mathcal{O}-, \Theta-$ & Ω-Notation

2.1 Asymptotische Notationen

In dieser Übungsstunde wurden die wichtigsten Landau-Notationen zur Analyse von Algorithmen vertieft.

Big-O (O) $f(n) \in O(g(n))$ bedeutet, dass f *höchstens so schnell* wächst wie g, bis auf einen konstanten Faktor:

$$\exists c > 0,\, n_0 \in \mathbb{N} :\ \forall n \geq n_0 :\ f(n) \leq c\, g(n).$$

Big-Omega (Ω) $f(n) \in \Omega(g(n))$ bedeutet, dass f *mindestens so schnell* wächst wie g:

$$\exists c > 0,\, n_0 \in \mathbb{N} :\ \forall n \geq n_0 :\ f(n) \geq c\, g(n).$$

Big-Theta (Θ) $f(n) \in \Theta(g(n))$ gilt genau dann, wenn f sowohl in $O(g(n))$ als auch in $\Omega(g(n))$ liegt. Damit wächst f asymptotisch *gleich schnell* wie g:

$$\exists c_1, c_2 > 0,\, n_0 :\ \forall n \geq n_0 :\ c_1 g(n) \leq f(n) \leq c_2 g(n).$$

Beispiele

$$n^2 + 3n + 1 \in \Theta(n^2)$$
$$5n \log n \in O(n^{1+\varepsilon})\ \forall \varepsilon > 0$$
$$2^n \in \Omega(n^k)\ \forall k \in \mathbb{N}$$

Intuition

- O beschreibt eine *obere Schranke* für das Wachstum.
- Ω beschreibt eine *untere Schranke*.
- Θ bedeutet, dass obere und untere Schranke dieselbe Ordnung haben.

2.2 Leetcode of the Week

- Problem 1: Two Sum

3 Ein Problem - Viele Lösungswege

3.1 Maximum Subarray Sum Problem

Gegeben ist ein Array $a_0, a_1, \ldots, a_{n-1}$. Gesucht: Ein zusammenhängendes Teilarray mit maximaler Summe.

Naiver Ansatz ($\mathbf{O}(n^3)$) Betrachte alle möglichen Intervalle $[i, j]$ und summiere deren Inhalt explizit. Es gibt $O(n^2)$ Intervalle, jede Summation kostet $O(n) \rightarrow$ insgesamt $O(n^3)$.

Verbesserter Ansatz ($\mathbf{O}(n^2)$) Statt jedes Mal neu zu summieren, werden Teilsummen genutzt:

$$S_{i,j} = \sum_{k=i}^{j} a_k$$

Mit einem laufenden Summenarray lässt sich jedes Intervall in $O(1)$ berechnen. Damit ergeben sich $O(n^2)$ Intervalle und $O(n^2)$ Gesamtlaufzeit.

Divide & Conquer ($\mathbf{O}(n \log n)$) Das Maximum-Subarray liegt

- vollständig links,
- vollständig rechts,
- oder über der Mitte.

Rekursive Lösung analog zu Mergesort, Gesamtlaufzeit $T(n) = 2T(n/2)+O(n) \rightarrow O(n \log n)$.

Optimale Lösung (Kadane, $\mathbf{O}(n)$) Idee: Iterativ aufbauen.

$$R_j = \begin{cases} R_{j-1} + a_j, & \text{falls } R_{j-1} \geq 0 \\ a_j, & \text{sonst} \end{cases}$$

$$R_M = \max_j R_j$$

```
Algorithm Kadane(a[0..n-1]):
    R = a[0]
    RM = a[0]
    for j = 1 to n-1:
        R = max(R + a[j], a[j])
        RM = max(RM, R)
    return RM
```

Beispiel-Trace

$$a = [5, 8, -4, 1, -12, 4, 13, -2, -4, -1]$$

j	0	1	2	3	4	5	6	7	8	9
a_j	5	8	-4	1	-12	4	13	-2	-4	-1
R_j	5	13	9	10	-2	4	17	15	11	10
R_M	5	13	13	13	13	13	17	17	17	17

4 Suchen und Sortieren I

4.1 Invarianten

Invarianten – Definition Eine **Invariante** ist eine Bedingung, die während der Ausführung eines Algorithmus zu jedem Zeitpunkt gilt. Sie dient häufig dazu, Korrektheit zu beweisen oder den Fortschritt einer Schleife zu beschreiben.

Beispiel: Beim Sortieren ist nach jeder Iteration ein Teil des Arrays bereits sortiert – dieser Teil ist die Invariante.

Falls ihr selbst eine Invariante aufstellen sollt, ist es besonders wichtig, darauf zu achten, dass diese auch nach der 1. und letzten Iteration hält.

4.2 Suchalgorithmen

4.2.1 Lineare Suche (Linear Search)

Idee: Durchlaufe das gesamte Array und vergleiche jedes Element mit dem Suchwert. Das Array muss hierbei nicht sortiert sein.

```
Algorithm LinearSearch(A[0..n-1], x):
    for i = 0 to n-1:
        if A[i] == x:
            return i
    return -1
```

Laufzeit:

- Best Case: $O(1)$ – das gesuchte Element steht an erster Stelle.
- Worst Case: $O(n)$ – das Element steht am Ende oder fehlt.

Beispiele:

$$\text{Best Case: } [\underline{5}, 3, 2, 7, 9], \quad x = 5$$
$$\text{Worst Case: } [5, 3, 2, 7, \underline{9}], \quad x = 9$$

4.2.2 Binäre Suche (Binary Search)

Voraussetzung: Das Array ist *sortiert*. **Idee:** Vergleiche mit dem mittleren Element und teile das Problem jeweils in zwei Hälften.

```
Algorithm BinarySearch(A[0..n-1], x):
    low = 0
    high = n - 1
    while low <= high:
        mid = (low + high) // 2
        if A[mid] == x:
            return mid
        else if A[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Laufzeit:

- Best Case: $O(1)$ – Treffer in der Mitte.
- Worst Case: $O(\log n)$ – bei jedem Schritt wird das Suchintervall halbiert.

Beispiele:

$$\text{Best Case: } [1, 3, \underline{5}, 7, 9], \quad x = 5$$
$$\text{Worst Case: } [1, 3, 5, 7, \underline{9}], \quad x = 9$$

4.3 Sortieralgorithmen

4.3.1 Bubble Sort

Idee: Wiederholt benachbarte Elemente vergleichen und vertauschen, bis alles sortiert ist.

```
Algorithm BubbleSort(A[0..n-1]):
    for i = 0 to n-2:
        for j = 0 to n-2-i:
            if A[j] > A[j+1]:
                swap(A[j], A[j+1])
```

Laufzeit:

- Best Case: $O(n)$ – wenn das Array schon sortiert ist.
- Worst Case: $O(n^2)$ – wenn das Array absteigend sortiert ist.

Beispiele:

$$\text{Best Case: } [1, 2, 3, 4, 5]$$
$$\text{Worst Case: } [5, 4, 3, 2, 1]$$

4.3.2 Selection Sort

Idee: Finde das kleinste Element und platziere es an der aktuellen Position.

```
Algorithm SelectionSort(A[0..n-1]):
    for i = 0 to n-2:
        minIndex = i
        for j = i+1 to n-1:
            if A[j] < A[minIndex]:
                minIndex = j
        swap(A[i], A[minIndex])
```

Laufzeit:

- Best Case: $O(n^2)$ – keine Abkürzungen möglich.
- Worst Case: $O(n^2)$ – immer gleich viele Vergleiche.

Beispiele:

$$\text{Best/Worst Case: } [5, 4, 3, 2, 1]$$

4.3.3 Insertion Sort

Idee: Baue das sortierte Array schrittweise auf, indem jedes neue Element an die richtige Position eingefügt wird.

```
Algorithm InsertionSort(A[0..n-1]):
    for i = 1 to n-1:
        key = A[i]
        j = i - 1
        while j >= 0 and A[j] > key:
            A[j+1] = A[j]
            j = j - 1
        A[j+1] = key
```

Laufzeit:

- Best Case: $O(n)$ – Array ist bereits sortiert.
- Worst Case: $O(n^2)$ – Array ist absteigend sortiert.

Beispiele:

$$\text{Best Case: } [1, 2, 3, 4, 5]$$

$$\text{Worst Case: } [5, 4, 3, 2, 1]$$

4.3.4 Merge Sort

Idee: Teile das Array rekursiv in zwei Hälften, sortiere diese und führe sie zusammen.

```
Algorithm MergeSort(A):
    if length(A) <= 1:
        return A
    mid = length(A) // 2
    L = MergeSort(A[0..mid-1])
    R = MergeSort(A[mid..])
    return Merge(L, R)

Function Merge(L, R):
    result = []
    while L and R not empty:
        if L[0] <= R[0]:
            append L[0] to result; remove L[0]
        else:
            append R[0] to result; remove R[0]
    append remaining elements of L and R
    return result
```

Laufzeit:

- Best Case: $O(n \log n)$ – gilt immer.
- Worst Case: $O(n \log n)$ – unabhängig von der Eingabe.

Beispiele:

Best/Worst Case: $[5, 4, 3, 2, 1]$

Zusammenfassung der Laufzeiten:

Algorithmus	**Best Case**	**Worst Case**
Linear Search	$O(1)$	$O(n)$
Binary Search	$O(1)$	$O(\log n)$
Bubble Sort	$O(n)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$

5 Sortieren II & Datenstrukturen I

5.1 Sortieren II

5.1.1 Quicksort

Idee: Wähle ein Pivotelement, teile das Array in zwei Teilarrays (alle kleineren Elemente links, alle größeren rechts) und sortiere die Teilarrays rekursiv.

```
Algorithm QuickSort(A, low, high):
    if low < high:
        p = Partition(A, low, high)
        QuickSort(A, low, p - 1)
        QuickSort(A, p + 1, high)

Function Partition(A, low, high):
    pivot = A[high]
    i = low - 1
    for j = low to high - 1:
        if A[j] <= pivot:
            i = i + 1
            swap(A[i], A[j])
    swap(A[i + 1], A[high])
    return i + 1
```

Laufzeit:

- Best Case: $O(n \log n)$ – gleichmäßige Teilung.
- Worst Case: $O(n^2)$ – z. B. bereits sortiertes Array ohne zufällige Pivotwahl.

Beispiele:

Best Case: $[4, 2, 6, 3, 1, 5]$

Worst Case: $[1, 2, 3, 4, 5, 6]$

5.1.2 Heapsort

Idee: Nutze eine *Heap*-Struktur, um wiederholt das größte Element effizient zu extrahieren und ans Ende des Arrays zu setzen.

```
Algorithm HeapSort(A[0..n-1]):
    BuildMaxHeap(A)
    for i = n-1 downto 1:
        swap(A[0], A[i])
        heapSize = heapSize - 1
        MaxHeapify(A, 0)

Procedure BuildMaxHeap(A):
```

```
    for i = floor(n/2) downto 0:
        MaxHeapify(A, i)

Procedure MaxHeapify(A, i):
    l = 2*i + 1
    r = 2*i + 2
    largest = i
    if l < heapSize and A[l] > A[largest]:
        largest = l
    if r < heapSize and A[r] > A[largest]:
        largest = r
    if largest != i:
        swap(A[i], A[largest])
        MaxHeapify(A, largest)
```

Laufzeit:

- Best Case: $O(n \log n)$
- Worst Case: $O(n \log n)$

Beispiele:

Best/Worst Case: $[5, 1, 4, 2, 8, 6]$

5.2 Datenstrukturen

5.2.1 Heap

Ein **Heap** ist eine vollständige Binärstruktur, in der für jeden Knoten gilt:

$$A[\text{parent}(i)] \geq A[i] \quad \text{(Max-Heap)} \quad \text{bzw.} \quad A[\text{parent}(i)] \leq A[i] \quad \text{(Min-Heap)}$$

- `insert(x)` – fügt ein Element hinzu ($O(\log n)$)
- `extract_max()` – entfernt das größte Element ($O(\log n)$)
- `heapify()` – stellt Heap-Eigenschaft wieder her ($O(\log n)$)

5.2.2 Array

Ein **Array** ist eine feste, indizierte Sammlung von Elementen gleicher Größe.

- Zugriff auf $A[i]$ in ($O(1)$)
- Einfügen/Löschen in der Mitte: ($O(n)$)

Verkettete Listen (Linked Lists) Eine **singly linked list** besteht aus Knoten mit einem Wert und einem Verweis auf den Nachfolger. Eine **doubly linked list** hat zusätzlich einen Verweis auf den Vorgänger.

- Zugriff auf Element ($O(n)$)
- Einfügen/Löschen an bekannter Position ($O(1)$)

5.2.3 Stack

Eine **Stack** (Stapel“) arbeitet nach dem LIFO-Prinzip (Last In, First Out).

- `push(x)` – Element oben hinzufügen ($O(1)$)
- `pop()` – oberstes Element entfernen ($O(1)$)

5.2.4 Queue

Eine **Queue** (Warteschlange“) arbeitet nach dem FIFO-Prinzip (First In, First Out).

- `add(x)` – am Ende hinzufügen ($O(1)$)
- `poll()` – vom Anfang entfernen ($O(1)$)

5.2.5 Priority Queue

Eine **Priority Queue** speichert Elemente mit Prioritäten. Das Entfernen erfolgt stets nach der höchsten (oder niedrigsten) Priorität. **Typische Implementierung:** mittels Heap.

- `insert(x)` ($O(\log n)$)
- `extract_max()` ($O(\log n)$)
- `peek()` ($O(1)$)

6 Datenstrukturen II & Dynamische Programmierung I

6.1 Baumstrukturen

6.1.1 Binary Search Tree (BST)

Eigenschaft: Für jeden Knoten gilt

$$\text{left.key} \leq \text{root.key} \leq \text{right.key}.$$

Ein Inorder-Traversal liefert die Schlüssel in aufsteigender Reihenfolge.

Operationen:

- `search(x)` – $O(h)$
- `insert(x)` – $O(h)$
- `delete(x)` – $O(h)$

wobei h die Baumhöhe ist ($h \in O(n)$ im Worst Case, $O(\log n)$ im Durchschnitt).

```
Algorithm BST-Search(root, x):
    if root == NIL or root.key == x:
        return root
    if x < root.key:
        return BST-Search(root.left, x)
    else:
        return BST-Search(root.right, x)
```

Best Case: balancierter Baum → $O(\log n)$ **Worst Case:** degenerierter Baum → $O(n)$

6.1.2 2-3 Trees

Idee: Balancierter Suchbaum, in dem jeder innere Knoten 2 oder 3 Kinder hat. Alle Blätter liegen auf derselben Ebene.

Eigenschaften:

- Schlüssel in jedem Knoten sortiert (2-Knoten: 1 Schlüssel, 3-Knoten: 2 Schlüssel)
- Immer perfekt balanciert → Höhe $O(\log n)$

Operationen:

- `search(x)`, `insert(x)`, `delete(x)` – alle $O(\log n)$

Einfügen und Entfernen im 2-3-Baum

- **Einfügen:** Finde das passende Blatt. Ist es ein 2-Knoten, füge den Schlüssel ein. Ist es ein 3-Knoten, entsteht ein 4-Knoten → splitte, ziehe neuen Schlüssel, oder Schlüssel in gleichem Bucket nach oben. (Links: grösseren, Mitte: egal, Rechts: Kleineren)
- **Löschen:** Entferne den Schlüssel. Falls ein Knoten unterläuft, versuche Umverteilung oder Fusion (merge) mit Geschwister.

TODO: Add figures

6.2 Einführung in Dynamische Programmierung

Grundidee: Zerlege ein Problem in überlappende Teilprobleme, speichere Zwischenresultate (Memoization oder Tabulation), um wiederholte Berechnungen zu vermeiden.

6.2.1 Fibonacci-Zahlen

Rekursion:

$$F(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ F(n-1) + F(n-2), & n > 1 \end{cases}$$

Base Cases: $F(0) = 0,\ F(1) = 1$
Laufzeit:

- Rekursive naive Variante: $O(2^n)$
- DP (iterativ oder Memoization): $O(n)$

```
Algorithm Fibonacci(n):
    F[0] = 0
    F[1] = 1
    for i = 2 to n:
        F[i] = F[i-1] + F[i-2]
    return F[n]
```

6.2.2 Maximum Subarray Sum (MSS, Kadane revisited)

Rekursion:

$$R_j = \max(a_j, a_j + R_{j-1}), \quad R_M = \max_j R_j$$

Base Case: $R_0 = a_0$
Laufzeit: $O(n)$

```
Algorithm MSS(A[0..n-1]):
    R = A[0]; RM = A[0]
    for j = 1 to n-1:
        R = max(A[j], R + A[j])
        RM = max(RM, R)
    return RM
```

6.2.3 Jump Game (minimale Sprunganzahl)

Rekursion:

$$dp[k] = \max\{\, i + A[i] \mid dp[k-2] < i \leq dp[k-1] \}$$

Base Cases: $dp[0] = 0,\ dp[1] = A[0]$ **Laufzeit:** $O(n)$

```
Algorithm MinJumps(A[0..n-1]):
    dp[0] = 0
    dp[1] = A[0]
    k = 1
    while dp[k] < n-1:
        k = k + 1
        dp[k] = - infinity
        for i = dp[k-2] + 1 to dp[k-1]:
            dp[k] = max(dp[k], i + A[i])
    return k
```

6.2.4 Longest Common Subsequence (LCS)

Rekursion:

$$L(i,j) = \begin{cases} 0, & i = 0 \text{ oder } j = 0 \\ L(i-1, j-1) + 1, & X_i = Y_j \\ \max(L(i-1,j), L(i,j-1)), & X_i \neq Y_j \end{cases}$$

Base Cases: $L(i,0) = L(0,j) = 0$
Laufzeit: $O(mn)$

```
Algorithm LCS(X[1..m], Y[1..n]):
    for i = 0..m: L[i,0] = 0
    for j = 0..n: L[0,j] = 0
    for i = 1..m:
        for j = 1..n:
            if X[i] == Y[j]:
                L[i,j] = L[i-1,j-1] + 1
            else:
                L[i,j] = max(L[i-1,j], L[i,j-1])
    return L[m,n]
```

6.2.5 Edit Distance

Rekursion:

$$D(i,j) = \begin{cases} i, & j = 0 \\ j, & i = 0 \\ \min \begin{cases} D(i-1,j)+1, & \text{(delete)} \\ D(i,j-1)+1, & \text{(insert)} \\ D(i-1,j-1)+[X_i \neq Y_j] \end{cases}, & \text{sonst} \end{cases}$$

Base Cases: $D(0,j) = j$, $D(i,0) = i$
Laufzeit: $O(mn)$

```
Algorithm EditDistance(X[1..m], Y[1..n]):
    for i = 0..m: D[i,0] = i
    for j = 0..n: D[0,j] = j
    for i = 1..m:
        for j = 1..n:
            cost = 0 if X[i] == Y[j] else 1
            D[i,j] = min(
                D[i-1,j] + 1,      # delete
                D[i,j-1] + 1,      # insert
                D[i-1,j-1] + cost  # replace
            )
    return D[m,n]
```

7 Dynamische Programmierung II

7.1 Dynamische Programmierung II

7.1.1 Subset Sum

Problem: Gegeben ist eine Menge positiver Zahlen $S = \{s_1, s_2, \ldots, s_n\}$ und eine Zielsumme T. Gesucht ist, ob es eine Teilmenge $S' \subseteq S$ gibt, deren Summe genau T ergibt.

Rekursion:

$$dp[i][t] = \begin{cases} \text{True}, & t = 0 \\ \text{False}, & i = 0,\ t > 0 \\ dp[i-1][t] \vee dp[i-1][t-s_i], & t \geq s_i \\ dp[i-1][t], & t < s_i \end{cases}$$

Base Cases:

$$dp[0][0] = \text{True}, \quad dp[0][t > 0] = \text{False}$$

Laufzeit: $O(nT)$, wobei T die Zielsumme ist.

```
Algorithm SubsetSum(S[1..n], T):
    for t = 0..T: dp[0][t] = False
    dp[0][0] = True
    for i = 1..n:
        for t = 0..T:
            dp[i][t] = dp[i-1][t]
            if t >= S[i]:
                dp[i][t] = dp[i][t] or dp[i-1][t - S[i]]
    return dp[n][T]
```

Beispiel: $S = [3, 5, 7, 9]$, $T = 12 \Rightarrow$ True (z. B. $5 + 7 = 12$)

7.1.2 Knapsack

Problem: Gegeben sind n Gegenstände mit Werten v_i und Gewichten w_i, sowie eine Kapazität W. Gesucht ist die maximale Summe der Werte, die ohne Überschreiten von W aufgenommen werden kann.

Rekursion:

$$dp[i][w] = \begin{cases} 0, & i = 0 \text{ oder } w = 0 \\ dp[i-1][w], & w_i > w \\ \max(dp[i-1][w],\ dp[i-1][w-w_i] + v_i), & \text{sonst} \end{cases}$$

Base Cases:

$$dp[0][w] = 0, \quad dp[i][0] = 0$$

Laufzeit: $O(nW)$

```
Algorithm Knapsack(v[1..n], w[1..n], W):
    for i = 0..n: dp[i][0] = 0
    for w = 0..W: dp[0][w] = 0
    for i = 1..n:
        for cap = 1..W:
            if w[i] <= cap:
                dp[i][cap] = max(
                    dp[i-1][cap],
                    dp[i-1][cap - w[i]] + v[i]
                )
            else:
                dp[i][cap] = dp[i-1][cap]
    return dp[n][W]
```

Beispiel:

$$v = [3, 4, 5, 6],\ w = [2, 3, 4, 5],\ W = 5 \Rightarrow \text{Optimaler Wert} = 7$$

7.1.3 Longest Ascending Subsequence (LAS)

Problem: Gegeben ist eine Sequenz $A = [a_1, a_2, \ldots, a_n]$. Gesucht ist die Länge der längsten streng aufsteigenden Teilfolge (nicht notwendigerweise zusammenhängend).

Idee: Statt für jedes Element alle vorherigen zu prüfen ($O(n^2)$), speichern wir für jede mögliche Länge l der Teilfolge das **kleinste mögliche Endelement** einer aufsteigenden Teilfolge dieser Länge. Dieses Hilfsarray nennen wir *tails*.

Rekursive Sichtweise:

$$\text{Für jedes } a_i: \quad \text{Finde kleinstes } j \text{ mit } tails[j] \geq a_i, \text{ und setze } tails[j] = a_i.$$

Falls kein solches j existiert, erweitere die Teilfolge: $tails.append(a_i)$.

Base Case:

$$tails = [a_1] \quad \text{(Beginn mit erstem Element)}$$

Laufzeit: $O(n \log n)$ durch binäre Suche in *tails*.

```
Algorithm LAS(A[1..n]):
    tails = []          # tails[l] = kleinstes Endelement einer Folge der Länge l+1
    for x in A:
        pos = binary_search_first_ge(tails, x)
        if pos == len(tails):
            tails.append(x)
        else:
            tails[pos] = x
    return len(tails)
```

Hilfsfunktion:

```
binary_search_first_ge(tails, x):
    lo = 0; hi = len(tails)
    while lo < hi:
        mid = (lo + hi) // 2
        if tails[mid] < x:
            lo = mid + 1
        else:
            hi = mid
    return lo
```

Beispiel:

$$A = [3, 10, 2, 1, 20, 4, 6] \quad \Rightarrow \quad tails = [1, 4, 6, 20] \quad \Rightarrow \quad \text{Länge(LAS)} = 4$$

Intuition: - $tails[l]$ hält das kleinste mögliche Endelement einer Teilfolge der Länge $l + 1$. - Wenn a_i kleiner ist als ein existierendes $tails[j]$, wird es dort ersetzt, um die Chancen auf längere Teilfolgen zu verbessern. - Die Länge von $tails$ am Ende ist die Länge der längsten aufsteigenden Teilfolge.

8 Rechenregeln

8.1 Logarithmus-Regeln

Für $x, y > 0$ und $a, b > 0$ $(a, b \neq 1)$ gelten:

$$\begin{aligned}
\log(xy) &= \log x + \log y \\
\log\left(\frac{x}{y}\right) &= \log x - \log y \\
\log(x^k) &= k \log x \\
\log_a x &= \frac{\log_b x}{\log_b a} \quad \text{(Basiswechsel)} \\
\log 1 &= 0, \quad \log a = 1 \text{ für Basis } a
\end{aligned}$$

8.2 Summenformeln

Für $n \in \mathbb{N}$:

1. $\displaystyle\sum_{i=1}^{n} i \;=\; \frac{n(n+1)}{2}$

2. $\displaystyle\sum_{j=1}^{n}\sum_{k=j}^{n} 1 \;=\; \sum_{j=1}^{n}(n-j+1) \;=\; \frac{n(n+1)}{2}$

3. $\displaystyle\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n} 1 \;=\; n^3$

4. $\displaystyle\sum_{i=1}^{n} i^3 \;=\; \frac{n^2(n+1)^2}{4}$

5. $\displaystyle\sum_{i=1}^{n}\sum_{k=1}^{i} 1 \;=\; \sum_{i=1}^{n} i \;=\; \frac{n(n+1)}{2}$

6. $\displaystyle\sum_{i=1}^{n} i \log i \;\leq\; \sum_{i=1}^{n} n \log n \;=\; n^2 \log n$