# Algorithmen und Datenstrukturen 2023

bergery@student.ethz.ch

Oktober 2023

ETH zürich

## Contents

# 1 Vorwort

Dieses Skript dient dazu eine grobe Übersicht für die Vorlesung "Algorithmen und Datenstrukturen" zu geben. Jegliches aufgeführte Informationen sollte man an der Prüfung können.
Für etliche Fehler in diesem Skript wird keine Verantwortung übernommen.

# 2 Mathematische Grundlagen

## 2.1 Induktion

Die vollständige Induktion ist eine mathematische Beweismethode, nach der eine Aussage für alle natürlichen Zahlen bewiesen wird, die größer oder gleich einem bestimmten Startwert sind. Da es sich um unendlich viele Zahlen handelt, kann eine Herleitung nicht für jede Zahl einzeln erbracht werden. Sie ist ein deduktives Verfahren.
Der Beweis, dass die Aussage $A(n)$ für alle $n \geq n_0 (n_0$ meist 0 oder 1), gilt, wird dabei in zwei Etappen durchgeführt:

1. Im *Induktionsanfang* wird die Gültigkeit der Aussage $A(n_0)$ für eine kleinste Zahl $n_0$ gezeigt.
2. Im *Induktionsanfang* wird für ein beliebiges $n \geq n_0$ die Gültigkeit der Aussage $A(n+1)$ aus der Gültigkeit von $A(n)$ geschlussfolgert.

**Beispiel:**
Aussage: $A(n) := 1 + 2 + 3 + ... + n = \frac{n \cdot (n+1)}{2}$ für $n \geq 1$
*Base Case*: Für $n = 1$ gilt: $1 = \frac{1 \cdot (1+1)}{2} \rightarrow 1 = 1$ ✓
*Induction hypothesis*: "We now assume that it is true for $n = k$, i.e., $1 + 2 + 3 + ... + k = \frac{k \cdot (k+1)}{2}$.
*Induction step*: $k \rightarrow k + 1$
$1 + 2 + 3 + ... + k + (k + 1) \overset{\text{I.H}}{=} \frac{k \cdot (k+1)}{2} + (k + 1)$, where $\frac{k \cdot (k+1)}{2} + (k + 1) = \frac{k^2 + 3k + 2}{2} = \frac{(k+1) \cdot (k+2)}{2}$ □.
By the principle of mathematical induction, we conclude that $1 + 2 + 3 + ... + n = \frac{n \cdot (n+1)}{2}$ is true for all $n \in \mathbb{N}$.

## 2.2 Wichtige Annäherungen

Damit wir schnelle Resultate erhalten können, sind Approximationen ein fundamentaler Bestandteil. Aus diesem Grund folgende Liste:

- $n! \leq n^n$, for $n \geq 1$
- $\left(\frac{n}{2}\right)^{n/2} \leq n!, \ for \ n \geq 1$
- $\sum_{i=0}^{n} \log n = \ \log \prod_{i=1}^{n} n = \log n! \ \approx n \log n \rightarrow \mathcal{O}(n \log n)$
- $\sum_{i=0}^{n} 1 = (n + 1) \rightarrow \mathcal{O}(n)$
- $\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \rightarrow \mathcal{O}(n^2)$
- $\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \rightarrow \mathcal{O}(n^3)$
- $\sum_{i=0}^{n} i^3 = \frac{n^2(n+1)^2}{4} \rightarrow \mathcal{O}(n^4)$
- binomial coefficient: $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
- $\binom{n}{0} = \binom{n}{n} = 1 \qquad \binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1} \qquad \binom{n}{n-k} = \binom{n}{k}$
- $\sum_{i=1}^{n} i^k \leq \sum_{i=1}^{n} n^k$

| Notation | Definition | Mathematische Definition |
|---|---|---|
| $f \in o(g)$ | asymptotisch gegenüber $g$ vernachlässigbar | $\lim_{x\to a} \left\lvert \frac{f(x)}{g(x)} \right\rvert = 0$ |
| $f \in \mathcal{O}(g)$ | asymptotische obere Schranke | $\limsup_{x\to a} \left\lvert \frac{f(x)}{g(x)} \right\rvert < \infty$ |
| $f = \Omega(g)$ | asymptotische untere Schranke, $f$ ist nicht in $o(g)$ | $\limsup_{x\to a} \left\lvert \frac{f(x)}{g(x)} \right\rvert > 0$ |
| $f \in \Theta(g)$ | scharfe Schranke, sowohl $f \in \mathcal{O}(g)$ als auch $f \in \Omega(g)$ | $0 < \liminf_{x\to a} \left\lvert \frac{f(x)}{g(x)} \right\rvert \leq \limsup \left\lvert \frac{f(x)}{g(x)} \right\rvert < \infty$ |
| $f \in \omega(g)$ | asymptotisch dominant, $g \in o(f)$ | $\lim_{x\to a} \frac{f(x)}{g(x)} \right\rvert = \infty$ |

Table 1: Definition der $\mathcal{O}$-Notation

## 2.3 Big-O Notation

Landau-Symbole (engl. Big-O notation) werden verwendet, um das asymptotische Verhalten von Funktionen und Folgen zu beschreiben. In der Informatik werden sie bei der Analyse von Algorithmen verwendet und geben ein Maß für die Anzahl der Elementarschritte oder der Speichereinheiten in Abhängigkeit von der Größe des gegebenen Problems an.
Für Annäherungen wird auch sehr oft die **Regel von de l'Hôpital** angewendet:
Let $f, g : \mathbb{R} \to \mathbb{R}$ be differentiable functions with $f(x) \to \infty, g(x) \to \infty$ for $x \to \infty$.
If $\lim_{x\to\infty} \frac{f'(x)}{g'(x)}$ exists, then $\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$

### 2.3.1 Definition O-Notation

In der Tabelle (**??**) sehen wir jegliche Definition der $\mathcal{O}$-Notation. Wir unterscheiden zwischen, $\Omega$ (Omega) [lower-bound], $\Theta$ (Theta) [tight-bound] und $\mathcal{O}$ [upper-bound]. Damit dies noch mathematisch formuliert ist haben wird folgendes:
Let $f, g : \mathbb{R} \to \mathbb{R}^+$ such that the limit of $\frac{f}{g}$ exists. Then:

$$\lim_{x\to\infty} \frac{f}{g} = \infty \Rightarrow g \in \mathcal{O}(f) \text{and} f \in \Omega(g)$$

$$\lim_{x\to\infty} \frac{f}{g} = C \in \mathbb{R}^+ \setminus \{0\} \Rightarrow f \in \Theta(g) \text{and } g \in \Theta(f)$$

$$\lim_{x\to\infty} \frac{f}{g} = 0 \Rightarrow f \in \mathcal{O}(g) \text{and } g \in \Omega(f)$$

### 2.3.2 Master Theorem

Damit das Rechnen mit solchen Grenzwerten schneller geht, haben wir das Master-Theorem, welches sehr praktische Anwendung mit sich bringt:

$$T(n) \leq aT(n/2) \ + Cn^b := \begin{cases} T(n) \leq \mathcal{O}(n^b), & \text{b} > \log_2(a) \\ T(n) \leq \mathcal{O}(n^b \log(n)) & \text{b} = \log_2(a) \\ T(n) \leq \mathcal{O}(n^{\log_2(a)}) & \text{b} < \log_2(a) \end{cases} \tag{1}$$

### 2.3.3 Overview

Damit wir nun einmal sehen können wie die asympotischen Grenzen sich verhalten, folgende Abbildung: Die graphische Darstellung der Funktionen im Bild (**??**) zeigt uns leider nur sechs Funktionen. Um möglichst viele Funktionen in aufsteigendem, asymptotischen Wachstum zu sehen, folgende Abbildung:

$\mathcal{O}(1) \in \mathcal{O}(\log\log n) \in \mathcal{O}(\log n) \in \mathcal{O}(\sqrt{n}) \in \mathcal{O}(n) \in \mathcal{O}(n \log n) \in \mathcal{O}(n^2) \in \mathcal{O}(2^n) \in \mathcal{O}(2^{n^2}) \in \mathcal{O}(n!)$

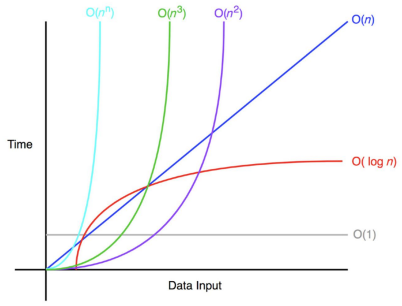



Figure 1: Wachstum verschiedener Funktionen

## 2.4 Erster Algorithmus

### 2.4.1 Maximum Subarray Sum

Für einen ersten Algorithmus schauen wir den `Maximum Subarray Sum` Algorithmus an. Dieser Algorithmus enthält ein Array von $n$ rationalen Zahlen $a_1, ..., a_n$ und gesucht ist ein Teilstück mit maximaler Summe.
**Beispiel**:
Sei ein Array [-2, -5, 6, -2, -3, 1, 5, -6] gegeben, so ist die maximale Teilsumme dieses Array 7, nämlich $6 + (-2) + (-3) + 1 + 5 = 7$.

**Algorithm 1** Maximum Subarray Sum $(a_1, ..., a_n)$

```
1 S_0 ← 0
2 Für i ← 1, ..., n
3     S_i ← S_{i-1} + a_i
4 maxS ← 0
5 Für i ← 1, ..., n
6     Für j ← i, ..., n
7         S ← S_j − S_{i-1}
8         Merke maximales S
```

Dieser naive Algorithmus führt insgesamt $\Theta(n^3)$ viele Additionen aus, was sehr schlecht ist für unseren Algorithmus.

**Algorithm 2** MSS divide and conquer $(a_1, ..., a_n)$

```
1 Wenn n = 1 ist, dann gib max{a_1, 0} zurück.
2 Wenn n > 1 ist:
3     Teile die Eingabe in A_1 = ⟨a_1, ..., a_{n/2}⟩ und A_2 = ⟨a_{n/2+1}, ..., a_n⟩ auf.
4     Berechne rekursiv den Wert W_1 einer besten Lösung für das Array A_1.
5     Berechne rekursiv den Wert W_2 einer besten Lösung für das Array A_2.
6     Berechne grösste Suffixsumme S in A_1.
7     Berechne grösste Präfixsumme P in A_2.
8     Setze W_3 ← S + P.
9     Gib max{W_1, W_2, W_3} zurück.
```

Wir haben schon eine Verbesserung des Algorithmus von $\Theta(n^3)$ vielen Additionen zu $\Theta(n \log n)$ vielen Additionen. In einem letzten Schritt können wir diesen sogar noch einmal verbessern nämlich bis zu $\Theta(n)$ viele Additionen.

**Algorithm 3** MSS-Induktiv $(a_1, ..., a_n)$

```
1 randmax ← 0
2 maxS ← 0
3 Für i ← 1, ..., n:
4     randmax ← randmax + a_i
5     Wenn randmax > maxS:
6         maxS ← randmax
7     Wenn randmax < 0:
8         randmax ← 0
9 Gib maxS zurück.
```

# 3 Suchen und Sortieren

## 3.1 Suchen

In diesem Abschnitt schauen wir wie schnell wir in abstrakten Daten Strukturen suchen, einfügen und löschen können. Die meist gebrauchten Datenstrukturen sind somit in folgender Tabelle aufgeführt mit ihrer korrespondierender Laufzeit:

| Data structure | Search | Insert | Delete |
|---|---|---|---|
| Unsorted Array | $O(n)$ | $O(1)$ | $O(n)$ |
| Sorted Array | $O(\log n)$ | $O(n)$ | $O(n)$ |
| Unsorted List | $O(n)$ | $O(1)$ | $O(n)$ |
| Sorted List | $O(n)$ | $O(n)$ | $O(n)$ |
| Unbalanced Tree | $O(n)$ | $O(n)$ | $O(n)$ |
| AVL tree | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |

Wir stellen fest, dass es im Allgemeinen einen Kompromiss gibt zwischen einfachen Datenstrukturen, die ein einfaches Einfügen und Löschen ermöglichen, aber die Einträge nicht vorverarbeiten, um spätere Suchvorgänge zu erleichtern (ungeordnetes Array), und komplexeren Datenstrukturen mit höheren Einfüge- und Löschkosten, die effizienter abgefragt werden können.

### 3.1.1 Binary Search

Binary Search ist der standart Suchalgorithmus für **sortierte** Arrays, da es einen effizienten, logarithmische Laufzeitkomplexität hat.

**Algorithm 4** Binary search

```
function FindIndex(A, e)                    ▷ Search item e in sorted array A
    l, r ← 0, A.length − 1
    while r > l do
        m ← ⌊(l+r)/2⌋
        if A[m] = e then
            return m
        else if A[m] > e then
            r ← m − 1
        else
            l ← m + 1
    return "not found"
```

Obwohl bei der binären Suche die Kosten für die Suche in geordneten Arrays logarithmisch werden, sind Einfügung und Löschung immer noch linear: im schlimmsten Fall, d.h. wenn das einzufügende oder zu löschende Element das erste Element des Arrays ist, müssen wir alle Elemente um einen Schritt nach rechts/links verschieben.

### 3.1.2 Heaps

Noch effizienter als geordnete Arrays sind daher baumartige Strukturen, bei denen die Kosten für das Einfügen und Löschen ebenfalls logarithmisch sein können. Heaps sind eine spezielle Klasse von

baumbasierten Strukturen, die eine effiziente (zeitkonstante) Methode zur Extraktion des kleinsten oder größten Elements bieten.

Genauer gesagt sind Min- (bzw. Max-) Heaps baumbasierte Datenstrukturen, die die folgende Heap-Invariante erfüllen: Wenn $A$ das Elternteil von $B$ ist, dann ist der Wert von Knoten $A$ kleiner (bzw. größer) als der Wert von Knoten $B$. Wir betrachten hier binäre Min-Heaps, was bedeutet, dass ein Elternteil einen kleineren Wert hat als seine (höchstens zwei) Kinder. Für binäre Min-Haufen gilt zusätzlich die folgende Forminvariante: Der betrachtete Haufen ist immer ein vollständiger binärer Baum, d. h. alle Schichten des Baums sind von oben nach unten und von links nach rechts gefüllt.

Figure 2: Beispiel eines Min-Heaps

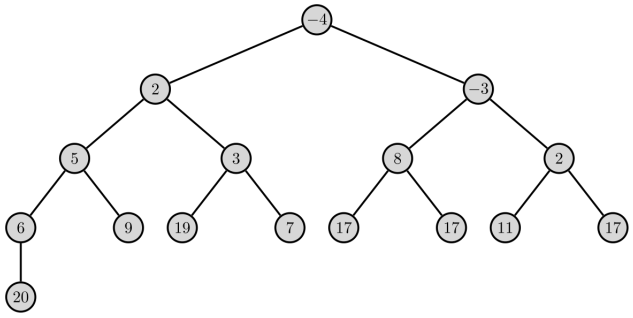
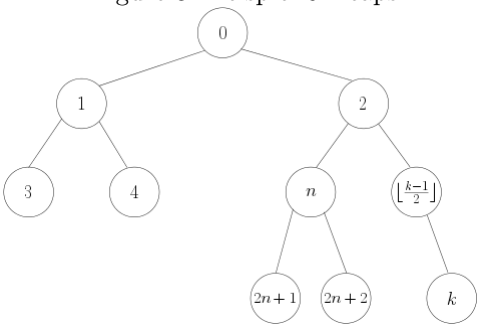



Figure 3: Beispiel 0-Heaps




### Implementation

Die gebräuchlichste Implementierung ist ein Array (mit fester oder dynamischer Größe). Geht man von einem 0-basierten Array aus (vgl. Figure **??**), so sind die *children* des Knotens $n$, $2n+1$ und $2n+2$ und die *parent* des Knotens $k$ ist der Knoten $\lfloor \frac{k-1}{2} \rfloor$.

### Common operations

Here are the most common operations that a min-heap must support:

- **Basic operations**
  - Find min,
  - Delete min,
  - Insert,
  - Find min and delete it (pop);
- **Initialization**
  - Create empty heap,
  - Heapify (transform array into heap);

- **Inspection**
  - Return size,
  - Test if empty;
- **Other**
  - Increase/decrease,
  - Delete,
  - Restore heap invariant,
  - Merge/union.

### Preserving the heap invariant

Die Heap Invariante sagt uns eigentlich aus:

> "Das Label jedes Knoten $n$ ist kleiner oder gleich den Labels der Kinder von $n$."

Bei der Aktualisierung eines Heaps ist es wichtig, die oben beschriebene Invariante beizubehalten. Nach einer Löschung oder einer Einfügung kann es vorkommen, dass ein (einzelnes) Element kleiner wird als eines seiner Kinder; wir können dann die Heap-Invariante in logarithmischer Zeit wiederherstellen, indem wir dieses Element nach unten durchsickern lassen.

**Algorithm 5** Restore heap invariant by percolating element down

```
function PercolateDown(H, i)                          ▷ Percolate element i in H
    e ← H[i]
    if 2i + 2 = H.length then
        if e > H[2i + 1] then
            Swap H[i] and H[2i + 1]
    else if 2i + 2 < H.length then
        l, r ← H[2i + 1], H[2i + 2]
        if l < r then
            if l < e then
                Swap H[i] and H[2i + 1]
                PercolateDown(H, 2i + 1)
        else
            if r < e then
                Swap H[i] and H[2i + 2]
                PercolateDown(H, 2i + 2)
```

### Costs

In der folgenden Tabelle sind die Kosten von Standard-Heap-Operationen für drei Arten von Heaps zusammengefasst: binäre Heaps, binomische Heaps und Fibonacci-Heaps.

| Operation | Binary | Binomial | Fibonacci |
|---|---|---|---|
| search min | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ |
| delete min | $\Theta(\log n)$ | $\Theta(\log n)$ | $O(\log n)$ |
| insert | $\Theta(\log n)$ | $\Theta(1)$ | $\Theta(1)$ |
| increase key | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(1)$ |
| union | $\Theta(m \log n)$ | $O(\log n)$ | $\Theta(1)$ |

### Zusammenfassung- Youtube

Kurze Zusammenfassung anschauen