# 1 Sortieren

## 1.1 Einführung

Das Sortierproblem

- Ausgangspunkt: Folge von Datensätzen $D_1, D_2, \cdots, D_n$

| | | | |
|---|---|---|---|
| $D_1$ | $D_2$ | $\cdots$ | $D_n$ |

- zu sortierende Elemente heißen auch Schlüssel(werte)
- Ziel: Datensätze so anzuordnen, dass die Schlüsselwerte sukzessive ansteigen (oder absteigen)
- Bedingung: Schlüssel(werte) müssen vergleichbar sein

Durchführung:

- Eingabe: Sequenz von Schlüsselwerten $\{a_1, a_2, \ldots, a_n\}$
- Eingabe ist eine **Instanz** des Sortierproblems
- Ausgabe: Permutation $\{a'_1, a'_2, \ldots, a'_n\}$ derselben Folge mit Eigenschaft $a'_1 \leq \ldots \leq a'_n$
- Algorithmus **korrekt**, wenn dieser das Problem für alle Instanzen löst

## 1.2 Arrays

Reihung (Feld) fester Länge von Daten des gleichen Typs

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| A | 12 | 47 | 17 | 98 | 72 | | | | |

ABBILDUNG 1: beispielhafte Darstellung eines Arrays

- $A$: Bezeichnung des Arrays mit dem Namen „$A$“
- $A[i]$: Zugiff auf das $(i+1)$-te Element des Arrays

Beispiel: $A[2] = 17$
$\Longrightarrow$ Arrays erlauben effizienten Zugriff auf Elemente: konstanter Aufwand

## 1.3 Exkurs: Totale Ordnung

Sei $M$ eine nicht leere Menge und $\leq \subseteq M \times M$ eine binäre Relation auf $M$.

Das Paar $(M, \leq)$ heißt genau dann eine totale Relation auf der Menge $M$, wenn folgende Eigenschaften erfüllt sind:

- Reflexivität: $\forall x \in M : x \leq x$
- Transitivität: $\forall x, y, z \in M : x \leq y \land y \leq z \Rightarrow x \leq z$
- Antisymmetrie: $\forall x, y \in M : x \leq y \land y \leq x \Rightarrow x = y$
- Totalität: $\forall x, y \in M : x \leq y \lor y \leq x$

Beispiele:

- $\leq$ Ordnung auf natürlichen Zahlen
- Lexikographische Ordnung $\leq_{lex}$ ist eine totale Ordnung

## 1.4 Vergleichskriterien von Suchalgorithmen

- Berechnungsaufwand: $\mathcal{O}(n)$
- Effizienz: Best Case vs. Average Case vs Worst Case
- Speicherbedarf:
  - in-Place (in situ): zusätzlicher Speicher von der Eingabegröße unabhängig
  - out-of-place: Speichermehrbedarf von Eingabegröße abhängig
- Stabilität: stabile Verfahren verändern die Reihenfolge von äquivalenten Elementen nicht
- Anwendung als Auswahlfaktor:
  - Hauptoperationen beim Sortieren: Vergleichen und Vertauschen
  - Anwendung spielt eine enorme Rolle:
    - Verfahren mit vielen Vertauschungen und wenig Vergleichen, wenn Vergleichen teuer
    - Verfahren mit wenig Vertauschungen und vielen Vergleichen, wenn Umsortieren teuer

## 1.5 Analyse von Algorithmen (I)

- **Schleifeninvariante** (SIV):
  - Sonderform der Invariante
  - Am Anfang/Ende jedes Schleifendurchlaufs und vor/nach jedem Schleifendurchlauf gültig
  - Wird zur Feststellung der Korrektheit von Algorithmen verwendet
  - Eigenschaften:
    - Initialisierung: Invariante ist vor jeder Iteration wahr
    - Fortsetzung: Wenn SIV vor der Schleife wahr ist, dann auch bis Beginn der nächsten Iteration
    - Terminierung: SIV liefert bei Schleifenabbruch, helfende Eigenschaft für Korrektheit
  - Beispiel für Umsetzung: Insertion Sort - SIV
- **Laufzeitanalyse:**
  - Aufstellung der Kosten und Durchführungsanzahl für jede Zeile des Quelltextes
  - Beachte: Bei Schleifen wird auch der Aufruf gezählt, der den Abbruch einleitet
  - Beispiel für Umsetzung: Insertion Sort - Laufzeit
  - Zusätzliche Überprüfung des `Best Case`, `Worst Case` und `Average Case`
- **Effizienz von Algorithmen**:
  - Effizienzfaktoren
    - Rechenzeit (Anzahl der Einzelschritte)
    - Kommunikationsaufwand
    - Speicherplatzbedarf
    - Zugriffe auf Speicher
  - Laufzeit hängt von versch. Faktoren ab
    - Länge der Eingabe
    - Implementierung der Basisoperationen
    - Takt der CPU

## 1.6 Analyse von Algorithmen (II)

- **Komplexität:**
  - Abstrakte Rechenzeit $T(n)$ ist abhängig von den Eingabedaten
  - Übliche Betrachtungsweise der Rechenzeit ist asymptotische Betrachtung
- **Asymptotik:**
  - Annäherung an einer sich ins Unendliche verlaufende Kurve
  - z.B.: $f(x) = \frac{1}{x} + x$ | Asymptote: $g(x) = x$ | ($\frac{1}{x}$ läuft gegen Null)
- **Asymptotische Komplexität:**
  - Abschätzung des zeitlichen Aufwands eines Algorithmus in Abhängigkeit einer Eingabe
  - Beispiel für Umsetzung: Insertion Sort - Laufzeit $\Theta$
- **Asymptotische Notation:**
  - Betrachtung der Laufzeit $T(n)$ für sehr große Eingaben $n \in \mathbb{N}$
  - Komplexität ist unabhängig von konstanten Faktoren und Summanden
  - Nicht berücksichtigt: Rechnergeschwindigkeit / Initialisierungsaufwände
  - Komplexitätsmessung via Funktionsklasse ausreichend
    - Verhalten des Algorithmus für große Problemgrößen
    - Veränderung der Laufzeit bei Verdopplung der Problemgröße
- **Gründe für die Nutzung der theoretischen Betrachtung statt der Messung der Laufzeit**
  - *Vergleichbarkeit*
    - Laufzeit abhängig von konkreter Implementierung und System
    - Theoretische Betrachung ist frei von Abhängigkeiten und Seiteneffekten
    - Theoretische Betrachtung lässt direkte Vergleichbarkeit zu
  - *Aufwand*
    - Wieviele Testreihen?
    - In welcher Umgebung?
    - Messen führt in der Ausführung zu hohem, praktischen Aufwand
  - *Komplexitätsfunktion*
    - Wachstumsverhalten ausreichend
    - Praktische Evaluation mit Zeiten nur für Auswahl von Systemen möglich
    - Theoretischer Vergleich (Funktionsklassen) hat ähnlichen Erkenntnisgewinn

## 1.7 Analyse von Algorithmen (III)

- **$\Theta$-Notation**
  - $\Theta$-Notation beschränkt eine Funktion asymptotisch von oben und unten
  - Funktionen $f, g : \mathbb{N} \to \mathbb{R}_{>0}$ ($\mathbb{N}$: Eingabelänge, $\mathbb{R}$: Zeit)

$$\Theta(g) = \{f : \exists \underbrace{c_1, c_2 \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}}_{\text{Positive Konstanten}}, \forall n \geq n_0, \underbrace{0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)}_{f(n) \text{ wird von } c_1 g(n) \text{ und } c_2 g(n) \text{ für hinreichend große } n \text{ eingeschlossen}}\}$$

Funktion $f$

Für alle $n$ größer gleich $n_0$

  - $\Theta(g)$ enthält alle $f$, die genauso schnell wachsen wie $g$
  - Schreibweise: $f \in \Theta(g)$ (korrekt), manchmal auch $f = \Theta(g)$
  - $g(n)$ ist eine asymptotisch scharfe Schranke von $f(n)$
  - $f(n) = \Theta(g(n))$ gilt, wenn $f(n) = O(g(n))$ und $f(n) = \Omega(g(n))$ erfüllt sind

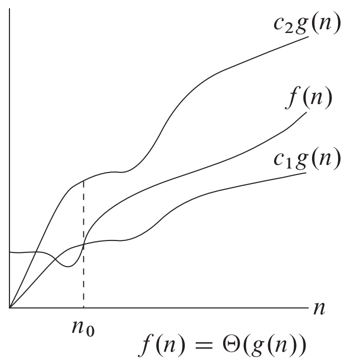



Abbildung 2: Veranschaulichung

- z.B.: $f(n) = \frac{1}{2}n^2 - 3n \mid f(n) \in \Theta(n^2)$?
- Aus $\Theta(n^2)$ folgt, dass $g(n) = n^2$
- Vorgehen:
  - Finden eines $n_0$ und $c_1, c_2$, sodass
  - $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ erfüllt ist
  - Konkret: $c_1 * n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 * n^2$
  - Division durch $n^2$: $c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c2$
  - Ab $n = 7$ positives Ergebnis: $0,0714 \mid n_0 = 7$
  - Deswegen setzen wir $c_1 = \frac{1}{14}$
  - Für $n \to \infty :\ 0,5 \mid c_2 = 0,5$
  - Natürlich auch andere Konstanten möglich

- **$\mathcal{O}$-Notation**
  - $\mathcal{O}$-Notation beschränkt eine Funktion asymptotisch von oben

$$O(g) = \{f : \underbrace{\exists\, c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}}_{\text{Positive Konstanten}}, \forall n \geq n_0, \underbrace{0 \leq f(n) \leq cg(n)}_{f(n) \text{ wird von } cg(n) \text{ für hinreichend große } n \text{ beschränkt}}\}$$

Funktion $f$

Für alle $n$ größer gleich $n_0$

  - $O(g)$ enthält alle $f$, die höchstens so schnell wie $g$ wachsen
  - Schreibweise: $f = \mathcal{O}(g)$
  - $f(n) = \Theta(g) \rightarrow f(n) = \mathcal{O}(g) \mid \Theta(g(n)) \subseteq \mathcal{O}(g(n))$
  - Ist $f$ in der Menge $\Theta(g)$, dann auch in der Menge $\mathcal{O}(g)$

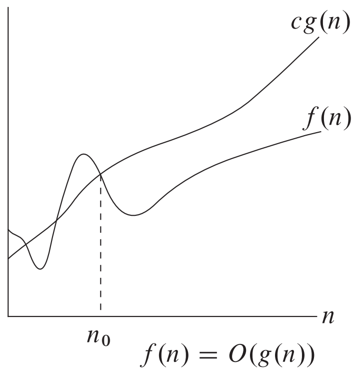



Abbildung 3: Veranschaulichung

    * z.B.: $f(n) = n + 2 \mid f(n) = \mathcal{O}(n)$?
    * Ja $f(n)$ ist Teil von $\mathcal{O}(n)$ für z.B. $c = 2$ und $n_0 = 2$

- **$O$-Notation Rechenregeln**
  - Konstanten:
    * $f(n) = a$ mit $a \in \mathbb{R}$ konstante Funktion $\rightarrow f(n) = O(1)$
    * z.B. $3 \in O(1)$
  - Skalare Multiplikation:
    * $f = O(g)$ und $a \in \mathbb{R} \rightarrow a * f = O(g)$
  - Addition:
    * $f_1 = O(g_1)$ und $f_2 = O(g_2) \rightarrow f_1 + f_2 = O(max\{g_1, g_2\})$
  - Multiplikation:
    * $f_1 = O(g_1)$ und $f_1 = O(g_2) \rightarrow f_1 * f_2 = O(g_1 * g_2)$

- **$\Omega$-Notation**
  - $\Omega$-Notation beschränkt eine Funktion asymptotisch von unten

$$\Omega(g) = \{f : \exists\, \underbrace{c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}}_{\text{Positive Konstanten}}, \forall n \geq n_0, \underbrace{0 \leq cg(n) \leq f(n)}_{f(n) \text{ wird von } cg(n) \text{ für hinreichend große } n \text{ unten beschränkt}}\}$$

Funktion $f$

Für alle $n$ größer gleich $n_0$

  - $\Omega$-Notation enthält alle $f$, die mindestens so schnell wie $g$ wachsen
  - Schreibweise: $f = \Omega(g)$

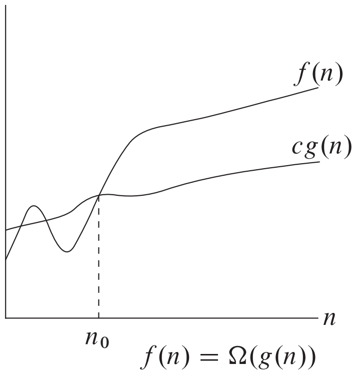



ABBILDUNG 4: Veranschaulichung

- **Komplexitätsklassen**
  - $n$ ist hier die Länge der Eingabe

| Klasse | Bezeichnung | Beispiel |
|---|---|---|
| $\Theta(1)$ | Konstant | Einzeloperation |
| $\Theta(\log n)$ | Logarithmisch | Binäre Suche |
| $\Theta(n)$ | Linear | Sequentielle Suche |
| $\Theta(n \log n)$ | Quasilinear | Sortieren eines Arrays |
| $\Theta(n^2)$ | Quadratisch | Matrixaddition |
| $\Theta(n^3)$ | Kubisch | Matrixmultiplikation |
| $\Theta(n^k)$ | Polynomiell | |
| $\Theta(2^n)$ | Exponentiell | Travelling-Salesman* |
| $\Theta(n!)$ | Faktoriell | Permutationen |

  - Ausführungsdauer, falls eine Operation $n$ genau $1\mu s$ dauert

| Eingabe-größe $n$ | $\log_{10} n$ | $n$ | $n^2$ | $n^3$ | $2^n$ |
|---|---|---|---|---|---|
| 10 | 1µs | 10µs | 100µs | 1ms | ~1ms |
| 100 | 2µs | 100µs | 10ms | 1s | ~4x$10^{16}$y |
| 1000 | 3µs | 1ms | 1s | 16min 40s | ? |
| 10000 | 4µs | 10ms | 1min 40s | ~11,5d | ? |
| 100000 | 5µs | 100ms | 2h 46min 40s | ~31,7y | ? |

Es gilt: $\log(n) < \sqrt{n} < n < n \cdot \log(n) < n^2 < n! < 2^n < n^n$

- **Asymptotische Notationen in Gleichungen**
  - $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$
  - $\Theta(n)$ fungiert hier als Platzhalter für eine beliebige Funktion $f(n)$ aus $\Theta(n)$
  - z.B.: $f(n) = 3n + 1$
- **$o$-Notation**
  - $o$-Notation stellt eine echte obere Schranke dar
  - Ausschlaggebend ist, dass es für alle $c \in \mathbb{R}_{>0}$ gelten muss
  - Außerdem $<$ statt $\leq$
  - z.B.: $2n = o(n^2)$ und $2n^2 \neq o(n^2)$

  $$o(g) = \{f : \underbrace{\forall c \in \mathbb{R}_{>0}}_{\substack{\text{Gilt für \textbf{alle} Konstanten } c > 0. \\ \text{In } \mathcal{O}\text{-Notation gilt es für eine Konstante } c > 0}}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) < cg(n)\}$$
- **$\omega$-Notation**
  - $\omega$-Notation stellt eine echte untere Schranke dar
  - Ausschlaggebend ist, dass es für alle $c \in \mathbb{R}> 0$ gelten muss
  - Außerdem $>$ statt $\geq$
  - z.B.: $\frac{n^2}{2} = \omega(n)$ und $\frac{n^2}{2} \neq \omega(n^2)$

  $$\omega(g) = \{f : \forall c \in \mathbb{R}_{>0}, \exists n_0 \in \mathbb{N} \forall n \geq n_0, 0 \leq cg(n) < f(n)\}$$

## 1.8 **Insertion Sort** (Sortieren durch Einfügen)

- **Idee**
  - Halte die linke Teilfolge sortiert
  - Füge nächsten Schlüsselwert hinzu, indem es an die korrekte Position eingefügt wird
  - Wiederhole den Vorgang bis Teilfolge aus der gesamten Liste besteht
- **Code**

Insertion-Sort(A)

```
FOR j = 1 TO A.length - 1
  key = A[j]
  // Füge A[j] in die sortierte Sequenz A[0...j-1] ein
  i = j - 1
  WHILE i >= 0 and A[i] > key
      A[i + 1] = A[i]
      i = i - 1
  A[i + 1] = key
```

- **Schleifeninvariante von `Insertion Sort`**
  - Zu Beginn jeder Iteration der `for`-Schleife besteht die Teilfolge `A[0...j-1]` aus den Elementen der ursprünglichen Teilfolge `A[0...j-1]` enthaltenen Elementen, allerdings in sortierter Reihenfolge.
- **Korrektheit von `Insertion Sort`**
  - *Initialisierung:*
    - Beginn mit `j=1`, also Teilfeld `A[0...j-1]` besteht nur aus einem Element `A[0]`. Dies ist auch das ursprüngliche Element und Teilfeld ist sortiert.
  - *Fortsetzung:*
    - Zu zeigen ist, dass die Invariante bei jeder Iteration erhalten bleibt. Ausführungsblock der `for`-Schleife sorgt dafür, dass `A[j-1], A[j-2]`,... je um Stelle nach rechts geschoben werden bis `A[j]` korrekt eingefügt wurde. Teilfeld `A[0...j]` besteht aus ursprünglichen Elementen und ist sortiert. Inkrementieren von j erhält die Invariante.
  - *Terminierung:*
    - Abbruchbedingung der `for`-Schleife, wenn `j > A.length - 1`. Jede Iteration erhöht j. Dann bei Abbruch ist `j = n` und einsetzen in Invariante liefert das Teilfeld `A[0...n-1]` welches aus den ursprünglichen Elementen besteht und sortiert ist. Teilfeld ist gesamtes Feld.
  - Algorithmus `Insertion Sort` arbeitet damit korrekt.

- **Laufzeitanalyse von `Insertion Sort`**

| Zeile | Kosten | Anzahl |
|---|---|---|
| 1 | $c_1$ | $n$ |
| 2 | $c_2$ | $n-1$ |
| 3 | $0$ | $n-1$ |
| 4 | $c_4$ | $n-1$ |
| 5 | $c_5$ | $\sum_{j=1}^{n-1} t_j$ |
| 6 | $c_6$ | $\sum_{j=1}^{n-1} (t_j - 1)$ |
| 7 | $c_7$ | $\sum_{j=1}^{n-1} (t_j - 1)$ |
| 8 | $c_8$ | $n-1$ |

  * Festlegung der Laufzeit für jede Zeile
  * Jede Zeile besitzt gewissen Kosten $c_i$
  * Jede Zeile wird $x$ mal durchgeführt
  * $Laufzeit = Anzahl * Kosten$ jeder Zeile
  * Schleifen: Abbruchüberprüfung zählt auch
  * $t_j$: Anzahl der Abfragen der `While`-Schleife
  * Laufzeit: $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=1}^{n-1} t_j + c_6 \sum_{j=1}^{n-1} (t_j - 1) + c_7 \sum_{j=1}^{n-1} (t_j - 1) + c_8(n-1)$

  – *Warum $n$ in Zeile 1?*
    * Die Überprüfung der Fortführungsbedingung beinhaltet auch die letze Überprüfung
    * Quasi die Überprüfung, durch die die Schleife abbricht
  – *Warum $\sum_{j=1}^{n-1}$ in Zeile 5?*
    * Aufsummierung aller einzelnen $t_j$ über die Anzahl der Schleifendurchläufe
    * Diese ist allerdings $n-1$ und nicht $n$, da die Abbruchüberprüfung dort auch enthalten ist
  – *Warum $t_j - 1$ in Zeile 6?*
    * Selbes Argument wie oben, bei $t_j$ ist die Abbruchüberprüfung enthalten
    * Deswegen wird die `while`-Schleife nur $t_j - 1$-mal ausgeführt
  – *`Best Case`*
    * zu sortierendes Feld ist bereits sortiert
    * $t_j$ wird dadurch zu 1, da die `While`-Schleife immer nur einmal prüft (Abbruch)
    * Die zwei Zeilen innerhalb der `While`-Schleife werden nie ausgeführt
    * Durch Umformen ergibt sich, dass die Laufzeit eine lineare Funktion in $n$ ist
  – *`Worst Case`*
    * zu sortierendes Feld ist umgekehrt sortiert
    * $t_j$ wird dadurch zu $j+1$, da die `While`-Schleife immer die gesamte Länge prüft
    * Durch Umformen ergibt sich, dass die Laufzeit eine quadratische Funktion in $n$ ist ($n^2$)
  – *`Average Case`*
    * im Mittel gut gemischt
    * $t_j$ wird dadurch zu $j/2$
    * Die Laufzeit bleibt aber eine quadratische Funktion in $n$ ($n^2$)

- **Asymptotische Laufzeitbetrachtung $\Theta$**
  – $T(n)$ lässt sich als quadratische Funktion $an^2 + bn + c$ betrachten
  – Terme niedriger Ordnung sind für große $n$ irrelevant
  – Deswegen Vereinfachung zu $n^2$ und damit $\Theta(n^2)$

## 1.9 Bubble Sort

- **Idee**
  - Vergleiche Paare von benachbarten Schlüsselwerten
  - Tausche das Paar, falls rechter Schlüsselwert kleiner als linker
- **Code**

BubbleSort(A)

```
FOR i = 0 TO A.length - 2
    FOR j = A.length - 1 DOWNTO i + 1
        IF A[j] < A[j-1]
            SWAP(A[j], A[j-1])
```

- **Analyse von `Bubble Sort`**
  - *Anzahl der Vergleiche:*
    - Es werden stets alle Elemente der Teilfolge miteinander verglichen
    - Unabhängig von der Vorsortierung sind `Worst` und `Best Case` identisch
  - *Anzahl der Vertauschungen:*
    - `Best Case`: 0 Vertauschungen
    - `Worst Case`: $\frac{n^2-n}{2}$ Vertauschungen
  - *Komplexität:*
    - `Best Case`: $\Theta(n)$
    - `Average Case`: $\Theta(n^2)$
    - `Worst Case`: $\Theta(n^2)$

## 1.10 Selection Sort

- **Idee**
  - Sortieren durch direktes Auswählen
  - `MinSort`: "wähle kleines Element in Array und tausche es nach vorne"
  - `MaxSort`: "wähle größtes Element in Array und tausche es nach vorne"
- **Code - MinSort**

Selection-Sort(A)

```
FOR i = 0 TO A.length - 2
    k = i
    FOR j = i + 1 TO A.length - 1
        IF A[j] < A[k]
            k = j
    SWAP(A[i], A[k])
```

## 1.11 Divide-And-Conquer Prinzip

- Anderer Ansatz im Gegensatz zu z.B. `InsertionSort` (inkrementelle Herangehensweise)
- Laufzeit ist im schlechtesten Fall immer noch besser als `InsertionSort`
- Prinzip: Zerlege das Problem und löse es direkt oder zerlege es weiter
- *Divide:*
  - Teile das Problem in mehrere Teilprobleme auf
  - Teilprobleme sind Instanzen des gleichen Problems
- *Conquer:*
  - Beherrsche die Teilprobleme rekursiv
  - Falls Teilprobleme klein genug, löse sie auf direktem Weg
- *Combine:*
  - Vereine die Lösungen der Teilprobleme zu Lösung des ursprünglichen Problems

## 1.12 Merge Sort

- **Idee**
  - *Divide:* Teile die Folge aus $n$ Elementen in zwei Teilfolgen von je $\frac{n}{2}$ Elemente auf
  - *Conquer:* Sortiere die zwei Teilfolgen rekursiv mithilfe von `MergeSort`
  - *Combine:* Vereinige die zwei sortierten Teilfolgen, um die sortierte Lösung zu erzeugen
- **Code**

MERGE-SORT(A,p,r)

```
IF p < r
  q = ⌊(p+r)/2⌋ // Teilen in 2 Teilfolgen
  MERGE-SORT(A,p,q) // Sortieren der beiden Teilfolgen
  MERGE-SORT(A,q+1,r)
  MERGE(A,p,q,r) // Vereinigung der beiden sortierten Teilfolgen
```

MERGE(A,p,q,r)

```
n1 = q - p + 1
n2 = r - q
Let L[0...n1] and R[0...n2] be new arrays
FOR i = 0 TO n1 - 1 // Auffüllen der neu erstellten Arrays
  L[i] = A[p + i]
FOR j = 0 TO n2 - 1
  R[j] = A[q + j + 1]
L[n1] = ∞ // Einfügen des Sentinel-Wertes
R[n2] = ∞
i = 0
j = 0
FOR k = p TO r  // Eintragweiser Vergleich der Elemente
  IF L[i] ≤ R[j]
      A[k] = L[i] // Sortiertes Zurückschreiben in Original-Array
      i = i + 1
  ELSE
      A[k] = R[j]
      j = j + 1
```

(Teilarrays werden nicht parallel bearbeitet)

- **Korrektheit von MergeSort**
  - *Schleifeninvariante*

    Zu Beginn jeder Iteration der `for`-Schleife (Letztes `for` in Methode `MERGE`) enthält das Teilfeld `A[p...k-1]` die `k-p` kleinsten Elemente aus `L[0...`$n_1$`]` und `R[0...`$n_2$`]` in sortierter Reihenfolge. Weiter sind `L[i]` und `R[i]` die kleinsten Elemente ihrer Arrays, die noch nicht zurück kopiert wurden.
  - *Initialisierung*

    Vor der ersten Iteration gilt `k=p`. Daher ist `A[p...k-1]` leer und enthält 0 kleinste Elemente von `L` und `R`. Wegen `i=j=0` sind `L[i]` und `R[i]` die kleinsten Elemente ihrer Arrays, die noch nicht zurück kopiert wurden.

- *Fortsetzung*

  Müssen zeigen, dass Schleifeninvariante erhalten bleibt. Dafür nehmen wir an, dass `L[i]` $\leq$ `R[j]`. Dann ist `L[i]` kleinstes Element, welches noch nicht zurück kopiert wurde. Da Array `A[p...k-1]` die `k-p` kleinsten Elemente enthält, wird der Array `A[p...k]` die `k-p+1` kleinsten Elemente enthalten, nachdem der Wert nach der Durchführung von `A[k]=L[i]` kopiert wurde. Die Erhöhung der Variablen `k` und `i` stellt die Schleifeninvariante für die nächste Iteration wieder her. Wenn `L[i]>R[j]` dann analoges Argument in der `ELSE`-Anweisung.

- *Terminierung*

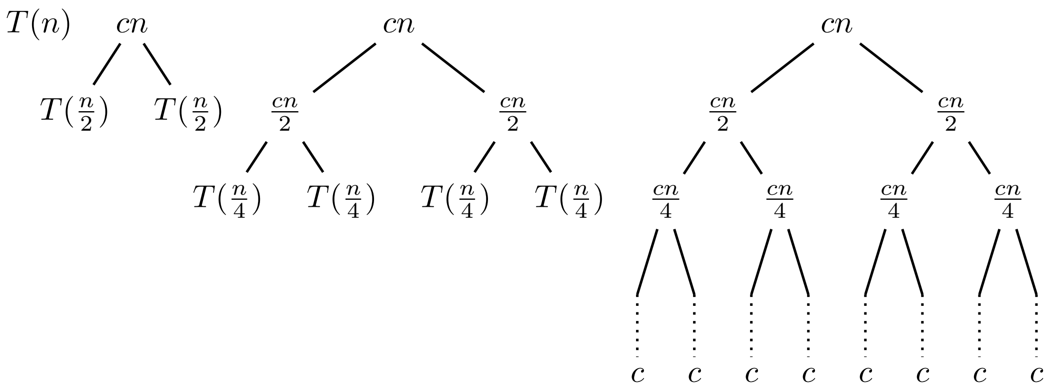
  Beim Abbruch gilt `k=r+1`. Durch die Schleifeninvariante enthält `A[p...r]` die kleinste Elemente von `L[0...`$n_1$`]` und `R[0...`$n_2$`]` in sortierter Reihenfolge. Alle Elemente außer der Sentinels wurden komplett zurück kopiert. `MergeSort` ist außerdem ein stabiler Algorithmus.

- **Analyse von MergeSort**
  - Ziel: Bestimme Rekursionsgleichung für Laufzeit $T(n)$ von $n$ Zahlen im schlechtesten Fall
  - Divide: Berechnung der Mitte des Feldes: Konstante Zeit $\Theta(1)$
  - Conquer: Rekursives Lösen von zwei Teilproblemen der Größe $\frac{n}{2}$: Laufzeit von $2\,T(\frac{n}{2})$
  - Combine: `MERGE` auf einem Teilfeld der Länge $n$: Lineare Zeit $\Theta(n)$

$$T(n) = \begin{Bmatrix} \Theta(1) & \text{falls } n = 1 \\ 2\,T(\frac{n}{2}) + \Theta(n) & \text{falls } n > 1 \end{Bmatrix}$$

  - Lösen der Rekursionsgleichung mithilfe eines Rekursionsbaums

$$T(n) = \begin{cases} c & \text{falls } n = 1, \\ 2T(\frac{n}{2}) + cn & \text{falls} n > 1 \end{cases}$$




    - ⋆ Verwenden der Konstante $c$ statt $\Theta(1)$
    - ⋆ $cn$ stellt den Aufwand an der ersten Ebene dar
    - ⋆ Der addierte Aufwand jeder Stufe (aller Knoten) ist auch $cn$
    - ⋆ Die Azahl der Ebenen lässt sich mithilfe von $lg(n) + 1$ bestimmen (2-er Logarithmus)
    - ⋆ Damit ergibt sich für die Laufzeit: $cn \cdot lg(n) + cn$
    - ⋆ Für $\lim_{n\to\infty}$ wird diese zu $n \cdot lg(n)$
    - ⋆ Laufzeit beträgt damit $\Theta(n \cdot lg(n))$
    - ⋆ Laufzeit von `MergeSort` ist in jedem Fall gleich

## 1.13 Quicksort

- **Idee**
  - *Pivotelement:*

    Wahl eines Pivotelement `x` aus dem Array (=Mittelsäule bei Sortierung)
  - *Divide:*

    Zerlege den Array `A[p...r]` in zwei Teilarrays `A[p...q-1]` und `A[q+1...r]`, sodass jedes Element von `A[p...q-1]` kleiner oder gleich `A[q]` ist, welches wiederum kleiner oder gleich jedem Element von `A[q+1...r]` ist. Berechnen Sie den Index `q` als Teil vom `Partition` Algorithmus.
  - *Conquer:*

    Sortieren beider Teilarrays `A[p...q-1]` und `A[q+1...r]` durch rekursiven Aufruf von Quicksort.
  - *Combine:*

    Da die Teilarrays bereits sortiert sind, ist keine weitere Arbeit nötig um diese zu vereinigen. `A[p...r]` ist nun sortiert.
- **Code**

QUICKSORT(A,p,r)

```
IF p < r     // Überprüfung, ob Teilarray leer ist
  q = PARTITION(A,p,r)
  QUICKSORT(A,p,q-1)
  QUICKSORT(A,q+1,r)
```

PARTITION(A,p,r)

```
x = A[r]     // Wahl des Pivotelements
i = p - 1    // Index i setzen
FOR j = p TO r - 1 // Auffüllen des Teilarrays mit Elementen
  IF A[j] ≤ x
      i = i + 1
      SWAP(A[i], A[j])
SWAP(A[i+1], A[r]) // Tausch des Pivotelements
RETURN i + 1 // Neuer Index des Pivotelements
```

(Teilarrays werden nicht parallel bearbeitet)

- **Korrektheit von Quicksort**
  - *Schleifeninvariante:*

    Zu Beginn jeder Iteration der `for`-Schleife gilt für den Arrayindex $k$ folgendes:

    1. Ist $p \leq k \leq i$, so gilt `A[k]` $\leq x$
    2. Ist $i + 1 \leq k \leq j - 1$, so gilt `A[k]` $> x$
    3. Ist $k = r$, so gilt `A[k]` $= x$
  - *Initialisierung:*

    Vor der ersten Iteration gilt $i = p - 1$ und $j = p$. Da es keine Werte zwischen $p$ und $i$ gibt und es auch keine Werte zwischen $i + 1$ und $j - 1$ gibt, sind die ersten beiden Eigenschaften trivial erfüllt. Die Zuweisung in `x = A[r]` sorgt für die Erfüllung der dritten Eigenschaft.
  - *Fortsetzung:*

    Zwei mögliche Fälle durch `IF A[j]` $\leq$ `x`. Wenn `A[j] > x`, dann inkrementiert die Schleife nur den Index $j$. Dann gilt Bedingung 2 für `A[j-1]` und alle anderen Einträge bleiben unverändert. Wenn `A[j]` $\leq$ `x`, dann wird Index $i$ inkrementiert und die Einträge `A[i]` und `A[j]` getauscht und schließlich der Index $j$ erhöht. Wegen des Vertauschens gilt `A[i]` $\leq$ `x` und Bedingung 1 ist erfüllt. Analog gilt `A[j-1] > x`, da das Element welches mit `A[j-1]` vertauscht wurde wegen der Invariante gerade größer als $x$ ist.
  - *Terminierung:*

    Bei der Terminierung gilt, dass $j = r$. Daher gilt, dass jeder Eintrag des Arrays zu einer der drei durch die Invariante beschriebenen Mengen gehört.
- **Performanz von Quicksort**
  - *Abhängig von der **Balanciertheit** der Teilarrays*
    - Definition Balanciert: ungefähr gleiche Anzahl an Elementen
    - Teilarrays balanciert: Laufzeit asymptotisch so schnell wie `MergeSort`
    - Teilarrays unbalanciert: Laufzeit kann so langsam wie `InsertionSort` laufen
  - *Zerlegung im **schlechtesten Fall***
    - Partition zerlegt Problem in ein Teilproblem mit $n - 1$ Elementen und eins mit $0$ Elementen
    - Unbalancierte Zerlegung mit Kosten $\Theta(n)$ zieht sich durch gesamte Rekursion
    - Aufruf auf Feld der Größe 0: $T() = \Theta(1)$
    - Laufzeit (rekursiv):
      - $T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$
      - Insgesamt folgt: $T(n) = \Theta(n^2)$
  - *Zerlegung im **besten Fall***
    - Problem wird so balanciert wie möglich zerlegt
    - Zwei Teilprobleme mit maximaler Größe von $\frac{n}{2}$
    - Zerlegung kostet $\Theta(n)$
    - Laufzeit (rekursiv):
      - $T(n) \leq 2T(\frac{n}{2}) + \Theta(n)$
      - Laufzeit beträgt: $O(n\ lg(n))$
    - Solange die Aufteilung konstant bleibt, bleibt die Laufzeit $O(n\ lg(n))$

## 1.14 Laufzeitanalyse von rekursiven Algorithmen

- **Analyse von Divide-And-Conquer Algorithmen**
  - $T(n)$ ist Laufzeit eines Problems der Größe $n$
  - Für kleines Problem benötigt die direkte Lösung eine konstante Zeit $\Theta(1)$
  - Für sonstige $n$ gilt:
    - ⋆ Aufteilen eines Problems führt zu $a$ Teilproblemen
    - ⋆ Jedes dieser Teilprobleme hat die Größe $\frac{1}{b}$ der Größe des ursprünglichen Problems
    - ⋆ Lösen eines Teilproblems der Größe $\frac{n}{b}$: $T(\frac{n}{b})$
    - ⋆ Lösen $a$ solcher Probleme: $a\,T(\frac{n}{b})$
    - ⋆ $D(n)$: Zeit um das Problem aufzuteilen (Divide)
    - ⋆ $C(n)$: Zeit um Teillösungen zur Gesamtlösung zusammenzufügen (Combine)

$$T(n) = \left\{ \begin{array}{ll} \Theta(1) & \text{falls } n \leq c \\ a\,T(\frac{n}{b}) + D(n) + C(n) & \text{sonst} \end{array} \right\}$$

- **Subsitutionsmethode**
  - Idee: Erraten einer Schranke und Nutzen von Induktion zum Beweis der Korrektheit
  - Ablauf:
    1. Rate die Form der Lösung (Scharfes Hinsehen oder kurze Eingaben ausprobieren/einsetzen)
    2. Anwendung von vollständiger Induktion zum Finden der Konstanten und Beweis der Lösung

- **Beispiel**
  - *Betrachten von* `MergeSort`*:*
    - $T(1) \leq c$
    - $T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + cn$
  - *Ziel:*

    Obere Abschätzung $T(n) \leq g(n)$ mit $g(n)$ ist eine Funktion, die durch eine geschlossene Formel dargestellt werden kann.

    Wir "raten": $T(n) \leq 4cn\ lg(n)$ und nehmen dies für alle $n' < n$ an und zeigen es für $n$.
  - *Induktion:*
    - $lg$ steht hier für $\log_2$
    - $n = 1$: $T(1) \leq c$
    - $n = 2$: $T(2) \leq T(1) + T(1) + 2c$

      $\leq 4c \leq 8c$

      $T(2) = 4c * 2\ lg(2) = 8c$
  - *Hilfsbehauptungen:*
    - (1): $\lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil = n$
    - (2): $\lfloor \frac{n}{2} \rfloor \leq \frac{n}{2} \leq \frac{2}{3}n$
    - (3): $\log_c(\frac{a}{b}) = \log_c(a) - \log_c(b)$
    - (4): $\log_c(a * b) = \log_c(a) + \log_c(b)$
  - *Induktionsschritt:*
    - Annahme: $n > 2$ und sei Behauptung wahr für alle $n' < n$.

    $$\begin{aligned}
    \text{T(n)} \quad & \leq T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) + cn \\
    & \leq 4c \lfloor \tfrac{n}{2} \rfloor\ lg(\lfloor \tfrac{n}{2} \rfloor) + 4c \lceil \tfrac{n}{2} \rceil\ lg(\lceil \tfrac{n}{2} \rceil) + cn \\
    \text{(HB)} \quad & \leq 4c \cdot lg(\tfrac{2}{3}n) \cdot (\lfloor \tfrac{n}{2} \rfloor + \lceil \tfrac{n}{2} \rceil + cn \\
    & \leq 4c \cdot lg(\tfrac{2}{3}n) \cdot n + cn \\
    \text{(HB)} \quad & \leq 4cn \cdot (lg(\tfrac{2}{3}) + lg(n)) + cn \\
    & = 4cn \cdot lg(n) + 4cn \cdot lg(\tfrac{2}{3}) \\
    & = 4cn \cdot lg(n) + cn(1 + 4 \cdot (lg(2) - lg(3))) \\
    & \leq 4cn \cdot lg(n) \\
    & \Rightarrow \Theta(n\ lg(n))
    \end{aligned}$$

- **Rekursionsbaum**
  - Idee: Stellen das Ineinander-Einsetzen als Baum dar und Analyse der Kosten
  - Ablauf:
    1. Jeder Knoten stellt die Kosten eines Teilproblems dar
       - Die Wurzel stellt die zu analysierenden Kosten $T(n)$ dar
       - Die Blätter stellen die Kosten der Basisfälle dar (z.B. $T(0)$)
    2. Berechnen der Kosten innerhalb jeder Ebene des Baums
    3. Die Gesamtkosten sind die Summe über die Kosten aller Ebenen
  - Rekursionsbaum ist nützlich um Lösung für Subsitutionsmethode zu erraten

- **Beispiel:** $T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + \Theta(n^2)$
  - *Vorüberlegungen:*
    - $\Rightarrow T(n) = 3T(\frac{n}{4}) + cn^2 \; (c > 0)$
    - Je Abstieg verringert sich die Größe des Problems um den Faktor 4.
    - Erreichen der Randbedingung ist vonnöten, die Frage ist wann dies geschieht.
    - Größe Teilproblem bei Level $i$: $\frac{n}{4^i}$
    - Erreichen Teilproblem der Größe 1, wenn $\frac{n}{4^i} = 1$, d.h. wenn $i = log_4(n)$
      $\Rightarrow$ Baum hat also $\log_4 n + 1$ Ebenen
  - *Kosten pro Ebene:*
    - Jede Ebene hat 3-mal soviele Knoten wie darüber liegende
    - Anzahl der Knoten in Tiefe $i$ ist $3^i$
    - Kosten $c(\frac{n}{4^i})^2$ , $i = 0, \cdots, \log_4 n - 1$
    - Anzahl · Kosten = $3^i \cdot c(\frac{n}{4^i})^2 = (\frac{3}{16})^i \cdot cn^2$
  - *Unterste Ebene:*
    - $3^{\log_4(n)} = n\log_4(3)$ Knoten
    - Jeder Knoten trägt $T(1)$ Kosten bei
    - Kosten unten: $n^{\log_4(3)} \cdot T(1) = \Theta(n^{\log_4(3)})$
  - *Addiere alle Kosten aller Ebenen:*
    - $T(n) = cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2 cn^2 + ... + (\frac{3}{16})^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4(3)})$
      $= \sum_{i=0}^{\log_4 n - 1} (\frac{3}{16})^i cn^2 + \Theta(n^{\log_4^3})$
      $= \frac{(\frac{3}{16}^{\log_4 n}) - 1}{\frac{3}{16} - 1} \cdot cn^2 + \Theta(n^{\log_4 3})$ (Verwendung der geometrischen Reihe)
    - Verwendung einer unendlichen fallenden geometrischen Reihe als obere Schranke:
      $T(n) = \sum_{i=0}^{1} (\frac{3}{16})^i \cdot cn^2 + \Theta(n^{\log_4 3})$
      $< \sum_{i=0}^{\infty} (\frac{3}{16})^i \cdot cn^2 + \Theta(n^{\log_4 3})$
      $= \frac{1}{1 - \frac{3}{16}} \cdot cn^2 + \Theta(n^{\log_4 3})$
      $= \frac{16}{13} \cdot cn^2 + Theta(n^{\log_4 3}) = O(n^2)$
  - *Jetzt **Subsitutionsmethode:***
    - Zu zeigen: $\exists d > 0 : T(n) \leq dn^2$
    - Induktionsanfang:
      $T(n) = 3 \cdot T(\lfloor \frac{1}{4} \rfloor) + c \cdot 1^2$
      $= 3 \cdot T(0) + c = c$
    - Induktionsschritt:
      $T(n) \leq 3 \cdot T(\lfloor \frac{n}{4} \rfloor) + cn^2$
      $\leq 3 \cdot d(\lfloor \frac{n}{4} \rfloor)^2 + cn^2$
      $\leq 3d(\frac{n}{4})^2 + cn^2$
      $= \frac{3}{16}dn^2 + cn^2$
      $\leq dn^2$, falls $d \geq \frac{16}{13}c$

- **Mastertheorem**
  - *Idee:*

    Seien $a \geq 1$ und $b > 1$ Konstanten. Sei $f(n)$ eine positive Funktion und $T(n)$ über den nichtnegativen ganzen Zahlen über die Rekursionsgleichung $T(n) = a\ T(\frac{n}{b}) + f(n)$ defininiert, wobei wir $\frac{n}{b}$ so interpretieren, dass damit entweder $\lfloor \frac{n}{b} \rfloor$ oder $\lceil \frac{n}{b} \rceil$ gemeint ist. Dann besitzt $T(n)$ die folgenden asymptotischen Schranken ($a$ und $b$ werden aus $f(n)$ gelesen):

    1. Gilt $f(n) = O(n^{\log_b(a-\epsilon)})$ für eine Konstante $\epsilon > 0$, dann $T(n) = \Theta(n^{\log_b(a)})$
    2. Gilt $f(n) = \Theta(n^{\log_b(a)})$, dann gilt $T(n) = \Theta(n^{\log_b(a)} lg(n))$
    3. Gilt $f(n) = \Omega(n^{\log_b(a+\epsilon)})$ für eine Konstante $\epsilon > 0$ und $a\ f(\frac{n}{b}) \leq c\ f(n)$ für eine Konstante $c < 1$ und hinreichend großen $n$, dann ist $T(n) = \Theta(f(n))$
  - *Erklärung:*
    - In jedem der 3 Fälle wird die Funktion $f(n)$ mit $n^{\log_b(a)}$ verglichen
      1. Wenn $f(n)$ polynomial kleiner ist als $n^{\log_b(a)}$, dann $T(n) = \Theta(n^{\log_b(a)})$
      2. Wenn $f(n)$ und $n^{\log_b(a)}$ die gleiche Größe haben, gilt $T(n) = \Theta(n^{\log_b(a)} lg(n))$
      3. Wenn $f(n)$ polynomial größer als $n^{\log_b(a)}$ und $a\ f(\frac{n}{b}) \leq c\ f(n)$ erfüllt, dann $T(n) = \Theta(f(n))$
    - (polynomial größer/kleiner: um Faktor $n^\epsilon$ asymptotisch größer/kleiner)
  - *Nicht abgedeckte Fälle:*
    - Wenn einer dieser Fälle eintritt, kann das Mastertheorem nicht angewendet werden
      1. Wenn $f(n)$ kleiner ist als $n^{\log_b(a)}$, aber nicht polynomial kleiner
      2. Wenn $f(n)$ größer ist als $n^{\log_b(a)}$, aber nicht polynomial größer
      3. Regularitätsbedingung $a\ f(\frac{n}{b}) \leq c\ f(n)$ wird nicht erfüllt
      4. $a$ oder $b$ sind nicht konstant (z.B. $a = 2^n$)

- *Beispiel:*
  - $T(n) = 9T(\frac{n}{3}) + n$
    - $a = 9, b = 3, f(n) = n$
    - $\log_b(a) = \log_3(9) = 2$
    - $$\begin{aligned} f(n) = n &= O(n^{\log_b(a-\epsilon)}) \\ &= O(n^{2-\epsilon}) \end{aligned}$$
    - Ist diese Gleichung für ein $\epsilon > 0$ erfüllt? $\Rightarrow \epsilon = 1$
    - **1. Fall** $\Rightarrow T(n) = \Theta(n^2)$
  - $T(n) = T(\frac{2n}{3}) + 1$
    - $a = 1, b = \frac{3}{2}, f(n) = 1$
    - $\log_{\frac{3}{2}} 1 = 0$
    - $$\begin{aligned} f(n) = 1 &= O(n^{\log_b(a)}) \\ &= O(n^0) \\ &= O(1) \end{aligned}$$
    - **2.Fall** $\Rightarrow T(n) = \Theta(1 * lg(n)) = \Theta(lg(n))$
  - $T(n) = 3(T\frac{n}{4}) + n\ lg(n)$
    - $a = 3, b = 4, f(n) = n\ lg(n)$
    - $n^{\log_b(a)} = n^{\log_4(3)} \leq n^{0.793}$
    - $\epsilon = 0.1$ im Folgenden
    - $f(n) = n\ lg(n) \geq n \geq n^{0.793+0.1} \geq n^{0.793}$
    - **3.Fall** $\Rightarrow f(n) = \Omega(n^{\log_b(a+0.1)})$
    - $af(\frac{n}{b}) = 3f(\frac{n}{4}) = 3(\frac{n}{4})\ lg(\frac{n}{4}) \leq \frac{3}{4}n\ lg(n)$
    - Damit ist auch die Randbedingung erfüllt und $T(n) = \Theta(n\ lg(n))$

---

| **Grundlegende Datenstrukturen** | **Fortgeschrittene Datenstrukturen** | **Randomisierte Datenstrukturen** |
|---|---|---|
| Stacks | Rot-Schwarz-Bäume | Skip Lists |
| Verkettete Listen | AVL-Bäume | Hash Tables |
| Queues | Splay-Bäume | Bloom-Filter |
| Bäume | Heaps | |
| Binäre Suchbäume | B-Bäume | |

TABELLE 1: Übersicht Datenstrukturen