AUD Reference Sheet

Frederick Wichert

Last Edited: 20. Juni 2020

Inhaltsverzeichnis

1 Begriffsdefinitonen

Problem Ein Problem im Sinner der Informatik enthält eine Beschreibung der Eingabe, der Ausgabe, aber keinen Übergang.
Eine **Probleminstanz** eine bestimmte Belegung der Eingabevariablen

Algorithmus Ein Algorithmus ist eine endliche Folge von Rechenschritten, die eine Eingabe in eine Ausgabe umwandelt.

Datenstruktur Eine Datenstruktur ist eine Methode, Daten abzuspeichern und zu organisieren, sowie deren Zugriff auf die Daten und die Modifikation der Daten zu erleichtern.

2 Pseudocode

2.1 Aufbau

- Pseudocode folgt keinen festen Regeln und dient der Veranschaulichung.
- Er entspricht Code einer beliebigen Programmierspracheund folgt keiner Sprachspetifischen Syntax, ist aber an solche angelehnt.
- Keywords werden in Caps geschrieben

2.2 Beispiele

Initiliasieiren von Variablen

```
variable = value;
```

Schleifen und Anweisungen

```
WHILE bedingung ...
ENDWHILE

FOR value TO value
IF bedingung THEN vlaue
```

Arrays

- Üblicherweise als `A` bezeichnet
`A[i] = ... ;`

3 Algorithmen

3.1 Definition

Ein Algorithmus ist eine endliche Folge von Rechenschritten, die eine Eingabe in eine Ausgabe umwandelt.

3.2 Anforderungen

→ Spezifizirung der Eingabe und Ausgabe
→ Eindeutigkeit - Jeder Einzelschritt ist klar definiert mit festgelgeter Reihenfolge
→ Endlichkeit - Die Notation hat eine Endliche Länge

3.3 Eigenschaften

→ Determiniertheit - Für gleiche Eingabe wird gleiche Ausgabe berechnet (mögliche, andere Zwischenzustände)
→ Determinismus - Für die gleiche Eingabe ist die Ausführung und Ausgabe identisch
→ Terminierung - Der Algorithmus läuft für jede Eingabe nur endlich lange
→ Korrektheit - Der Algorithmus berechnet stets die spezifizierte Ausgabe, falls dieser terminiert
→ Effzienz - Sparsamkeit bei Ressourcen (Zeit, Speicher, Energie etc.)

4 Effizienz von Algorithmen

- Effizienzfaktoren
  - Rechenzeit (Anzahl der Einzelschritte)
  - Kommunikationsaufwand (z.B. Netzwerk)
  - Speciherplatzbedarf
  - Zugriff auf Speicher (z.B. Festplatte)

- (Konkret) Laufzeit hängt von vielen Faktoren ab
  - Länge der Eingabe
  - Implementierung der Basisoperationen
  - Takt der CPU

5 Asymptotische Komplexität

Eine Abschätzung des (zetilichen) Aufwands eines Algorithmus in Abhängigkeit der Eingabe in form einer Asymptote in Abhängigkeit der Eingabemenge n.

- Beispiel: Summe bis n
  - `summe=summe+i`
  - Ablauf:
    - Variablen deklarieren
    - 1 zur Summe addieren
    - 2 zur Summer addieren
    - ...
    - n zur Sumer addieren
    - Ausgabe der Summe
  - Alle Basisoperationen benötigen konstant viel Aufwand
  - Anzahle der Operationen wächst mit Eingabe n
  - Für liner wachsende Eingabe wächst der Aufwand auch linear
- Beispiel: Laufzeit von InsertionSort
  - Quadratische Funktion: $an^2 + bn + c$
  - Bei großem n überweigt der quadratische Anteil → Terme niedriger Ordnung werden irrelevant
  - s

5.1 Asymptotische Notation

- Wie ist das Verhalten der Laufzeit $T(n)$ für sehr große Eingaben $\in N$
  - bei kleinen Eingaben (z.B. $n = 5$) ist die reale Laufzeit mehr dominiert von den Initialisierungskosten und nicht dem Algorithmus selbst
  - Komplexität ist unabhängig von konstanten Faktoren und Summanden
  - Berücksichtigen nicht Betrachtung:
    - Rechnergeschwindigkeit
    - Aufwände der Initialisierung
  - Komplexitätsmessungen via Funktionsklassen ausreichend
    - Wie verhält sich der Algorithmus für große Problemgrößen
    - Wie verändert sich die Laufzeit, wenn die Problemgröße verdoppelt wird

Komplexitätsfunktion
Das Wachstumsverhalten reicht aus um die wichtigen Punkte eines Algorithmus zu veranschaulichen.

5.2 Schranken der asymptotischen Komplexität

Best case

- Komplexität im "bestenFall
- Wie schnell ist der Algorithmus, wenn der Input so vorliegt, dass dieser am schnellsten terminiert?
- Beispiel: Vorsortierte Liste für (simplen) Sortieralgorithmus

Average case

- Komplexität im "durchschnittlichenFall
- Wie schnell ist der Algorithmus, wenn ein üblicher Input vorliegt?
- Beispiel: Zufällig verteilte Liste für (simplen) Sortieralgorithmus

Worst case

- Komplexität im ßchlechtestenFall
- Wie schnell ist der Algorithmus, wenn der Input so vorliegt, dass dieser am spätestens terminiert?
- Beispiel: Negativ-sortierte Liste für (simplen) Sortieralgorithmus

5.3 Notationen und Definition

Θ-Notation

$$\text{Funktion en } f, g : \mathbb{N} \to \mathbb{R}_{>0}$$

Dabei ist $\mathbb{N}$ Die Eingabelänge und $\mathbb{R}$ die Zeit

$$\Theta(g) = \{f : \exists c_1, c_2 \in \mathbb{R}_{>0}, n_0, \in \mathbb{N}, \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

Schreibweise: $f \in \Theta(g)$ oder $f = \Theta(g)$

O-Notation Obere asymptotische Schranke

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

Sprechweise: f wächst höchstens so schnell wie g
Sprechewise: $f = O(G)$

Beachte: $f(n) = \Theta(n) \implies f(n) = O(n)$
$\Theta(g(n)) \subseteq O(g(n))$

Rechenregeln

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

Ω-Notation Untere asymptotische Schranke

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

Sprechweise: f wächst mindestens so schnell wie g
Sprechewise: $f = \Omega(G)$

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

6 Analyse von Divide-and-Conquer Algorithmen

$T(n)$ ist Laufzeit eines Problems der Größe n.

Für kleines Problem, d.h. $n \leq c$ mit c ist eine Konstante, dann benötigt die Direkte Lösung konstante Zeit $\Theta(1)$.

Für sonstige n gilt: Das Aufteilen des Problems führt zu a Teilproblemen, von denen jedes die größe $1/b$ der Größe des ursprünglichen Problems hat.

Das Lösen eines Teilproblems der Größe n/b dauert $T(n/b)$ und somit benötigen wir für das Lösen von a solcher Probleme $aT(n/b)$.

$D(n)$ ist die Zeit um das Problem aufzuteilen, $C(n)$ ist die Zeit um die Teillösungen zur Lösung des ursprünglichen Problems zusammenzufügen.

$$T(n) = \left\{ \begin{array}{l} \Theta(1) \\ aT(n/b) + D(n) + C(n) \end{array} \right\}$$

Falls $n \leq c$

Methoden zur Lösung von Rekursionsgleichung

Substitutionsmethode: wir erraten eine Schranke und nutzen methematische Induktion, um die Korrektheit zu beweisen.
Rekursionsbaum-Methode: wandelt die Rekursionsgleichung in einen Baum um, dessen Knoten die Kosten der Rekursion in den versxchiedenen Ebenen darstellt.
Mastermethode: leifert Schranken für Rekursionsgleichung der Form: $T(n) = aT(n/b) + f(n)$

Substitutionsmethode Zur Lösung der Rekursionsgleichung, 2 Schritte:

- Rate die Form der Lösung, z.B. durch
  - Scharfes Hinsehen
  - kurze Eingaben ausprobieren und einsetzen
- Anwendung von vollständiger Induktion, um die Konstante zu finden und zu zeigen, dass das Geratene eien Lösung ist

Rekursionsbaum-Methode Grundidee: Stelle das Ineinander-Einsetzen als Baum dar und analysiere die Kosten

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 die Kosten innerhalb jeder Ebene des Baums
3. Die Gesamtkosten ist die Summe über die Kosten aller Ebenen

Geratenes kann durch Substitutionsmethode überprüft werden.

Mastermethode Allgemeine Form der Rekursionsgleichung:
$T(n) = aT(n/b) + f(n)$ mit $q \geq 1, b > 1$ und $f(n)$ eine asymptotische positive Funktion.

- Problem wird in a Teilprobleme der größe n/b aufgeteilt.
- Lösen jedes der a Teilprobleme benötigt Zeit $T(n/b)$.
- Funktion $f(n)$ umfasst die Kosten, um das Problem in Teilprobleme aufzuteilen und um die Teillösungen zu vereinigen.

Mastertheorem Seien $a \geq 1$ und $b > 1$ Konstanten. Sei $f(n)$ eine positive Funktion und $T(n)$ über den nicht negativen ganzen Zahlen durch die Rekursionsgleichung

$$T(n) = aT(n/b) + f(n)$$

definiert, wobei wir n/b so interpretieren, dass damit entweder $\lfloor n/b \rfloor$ oder $\lceil n/b \rceil$ gemeint ist. Dann besitzt $T(n)$ die folgenden asymptotischen Schranken:

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 $af(n/b) \leq cf(n)$ für eine Konstante $c < 1$ und hinreichend großen n, dann ist $T(n) = \Theta(f(n))$.

Das Mastertheorem verstehen:
In jedem der 3 Fälle wird die Funktion $f(n)$ mit $n^{\log_b a}$ verglichen.
Intuition:

1. Wenn $f(n)$ olynomial 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 Grße haben, dann gilt $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Wenn $f(n)$ polinimial grßer ist als $n^{\log_b a}$ und $af(n/b) \leq cf(n)$ erfüllt, dann ist $T(n) = \Theta(f(n))$.

Nicht abedeckte Fälle sind:

1. Wenn $f(n)$ kleiner als $n^{\log_b a}$, aber nicht polinimial klainer.
2. Wenn $f(n)$ größer als $n^{\log_b a}$, aber nicht polinimial größer.
3. Regularitätsbedingung $af(n/b) \leq cf(n)$ nicht erfüllt.

7 Datenstrukturen: Theorie

7.1 Besipiele

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	

Begriffsunterscheidung
Abstrakter Datentyp ("was")
Beschreibt Stack mit Operationen
Datenstrukturen ("wie")
Beschreibt die Stack-Operationen als Array oder Verkettete Liste

8 Datenstruktur: Stacks

Operationen:

- new(S) - erzeugt neuen (leeren) Stack
- isEmpty(S) - gibt an, ob Stack S leer ist
- pop(S) - gibt oberstes Element vom Stack S zurück und löscht es, falls vorhanden
- push(S,k) - schreibt k als neues oberstes Element auf S
- $\longrightarrow$ LIFO - last in, first out

Intuitive erwartete Bedingugen sind erfüllt, auch ohne algebraische Spezifikiation

Darstellung als Array

```
new(S)

1  S.A[]=ALLOCATE(MAX);
2  S.top=-1;
```

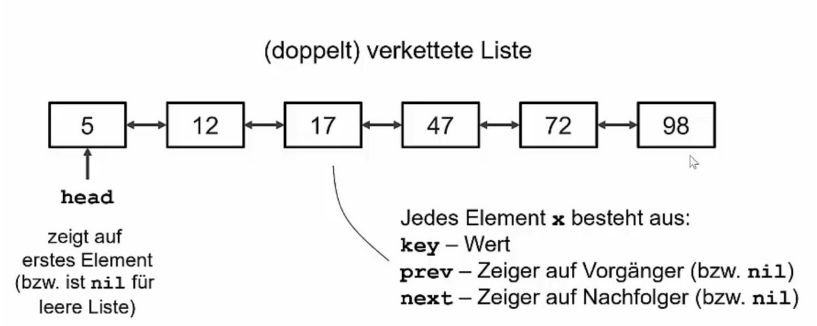
```
isEmpty(S)

1  IF S.top<0 THEN
2     return true
3  ELSE
4     return false;
```

```
pop(S)

1  IF isEmpty(S) THEN
2     error `underflow`
3  ELSE
4     S.top=S.top-1;
5     return S.A[S.top+1];
```

```
push(S,k)

1  IF S.top==MAX-1 THEN
2     error `overflow`
3  ELSE
4     S.top=S.top+1;
5     S.A[S.top]=k;
```

Bei der Darstellung als Array kann die maximale Größe erreicht werden. Danach wird beim Hinzufügen das ganze Array in ein grösSeres kopiert. Da dies in $\Theta(n^2)$ liegt wird ein Faktor festgelegt. So wird beispielweise bei erreichen der maximalen Größe die Größe verdoppelt, haben wir weniger als 1/Faktor*Faktor, also weniger als ein Viertel der Elemente belegt so halbieren wir unser Array.

9 Datenstruktur: Verkettete Listen



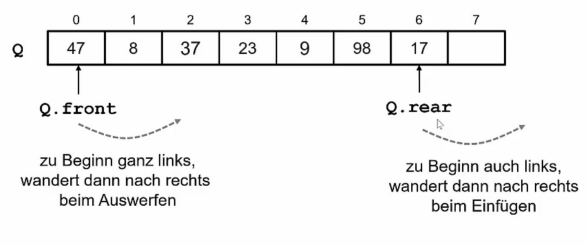
10 Datenstruktur: Queues

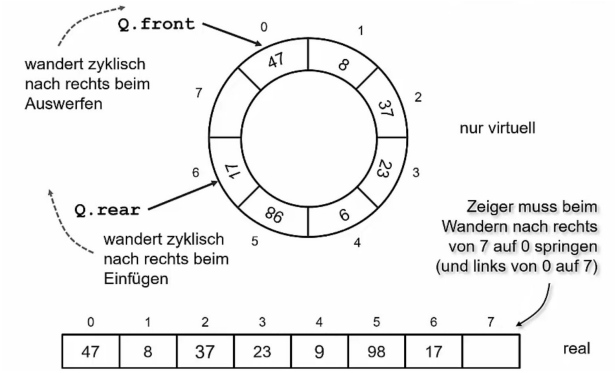
Operationen:

- new(Q) - erzeugt neue (leere) Queue
- isEmpty(Q) - gibt an, ob Queue leer ist
- dequeue(Q) - gibt vorderstes Element der Queue zurück und löscht es
- enqueue(Q,k) - schreibt k als neuees hinterstes Element auf Q
- $\longrightarrow$ FIFO - first in, first out

Darstellung als Array
Bei der Darstellung als Array stößt man nun auf Probleme. Es werden nun zwei Zeiger, und nicht mehr wie bei Stacks nur einer, benötigt. Da die Schlange aber nun durch das Array durchwandert bleibt sehr viel Arrrayplatz ungenutzt, bzw. man bräuchte ein unendlich großes Array. Daher gibt es zwei Darstellungsmöglichkeiten:

Als Array: Als zyklisches Array:





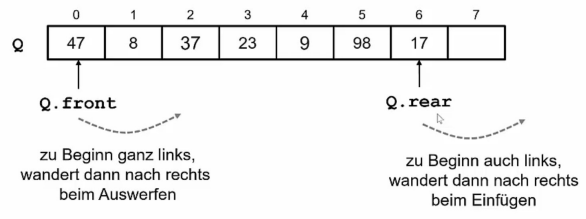
11 Datenstruktur: Binärbüume

Operationen:

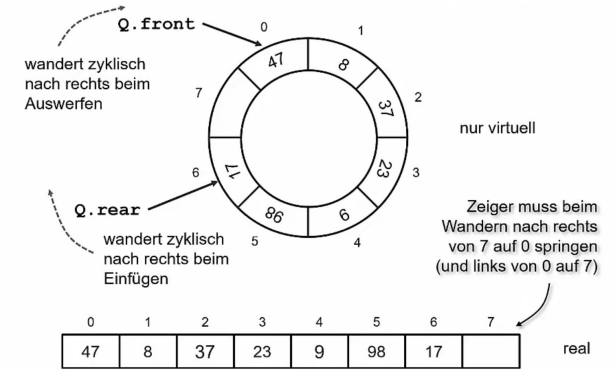
- new(Q)

Darstellung als Array
Bei der Darstellung als A

Als Array:



Als zyklisches Array:



12 Sortieralgorithmen: Theorie

12.1 Warum möchte man Sortieren

- Sachen Packen
- Zeitmanagement
- etc

12.2 Bedingungen

- Ausgangspunkt als Folge von Datensätzen
- Zu sortierenden elemente heißen Schlüsselelemente (-werte)
- Ziel: Schlüsselwerte nach einem Ordnungsschlüssel zu sortieren

Bedingug → Schlüsselwerte müssen vergleichbar sein

12.3 Vergleichskriterien

- Berechnungsaufwand: O(n)
- Effizienz: Best Case vs. Average Case vs. Worst Case
- Speicherbedarf: → in-place: zustzlicher Speicher von der Eingabegröße unabhängig
→ out-of-place: Speichermerhbedarf von eingabegröße abhängig
- Stabilität: Erahltenbleiben des sekundären Sortierschlüssels

▷ Sortierverfahren abhängig von Anwendung, Unterschiedliche Zahl von Vertauschungen vs Vergleichungen

12.4 Vergleichsverfahren

Korrektheit mittels Schleifeninvariante

- Eine Schleifeninvariante beschreibt was bei dem Durchlauf einer Schleife zu Beginn immer gegeben ist
  Damit kann die Korrektheit eines Algorithmus bewiesen werden
  ▷ Vergleichbar mit einem Indutkionsbeweis
- Eine Schleifeninvariante muss 3 Eigenschaften erfüllen:
  - Initialisierung: Invariante ist vor jeder Iteration wahr
  - Fortsetzung: Wenn die Invariante vor der Schleife wahr ist dann bleibt sie auch bis zum Beginn der nächsten Iteration wahr
- Bei dem Insertionsort:
  - Zu Beginn ist die Leinke Teilmenge immer sortiert

13 Entwurfsprinzipien des Sortieren

▷ IsertionSort: inkrementelle Herangehenweise

▷ Divide-and-Conquer Ansatz (Teile-undBeherrsche):

- Ansatz um Sortieralgorithmus zu entwerfen
- Laufzeit: im schlechtesten Fall immer noch besser als InsertionSort
- Gibt Methoden mit welchen wir die Laufzeit einfach bestimmen können

13.1 Devide-and-Conquer

Prinzip: Zerlege das Problem und löse es direkt *oder* durch weitere Zerlegung.

Divide: Teile das Problem in mehrere Teilprobleme auf, die kleinere Instanzen des gleichen Problems darstellen.

Conquer: Beherrsche die Teilprobleme rekursiv. Wenn die Teilprobleme klein genug sind, dann löse die Teilprobleme auf direktem Weg.

Combine: Vereinige die Lösung der Teilprobleme zur Lösung des ursprünglichen Problems.

Beispiel: Sortieralgorithmus: MergeSort

14 Sortieralgorithmus: InsertionSort

Grundidee

- Sortieren durch Einfügen:
- Die Linke Teilmenge ist Sortiert, und passend wird immer ein neues Element eingefügt

Eigenschaften

- Stabieler Algorithmus

Code

```
INSERTION-SORT(A)
1  FOR j = 1 TO A.length −1
2     key = A[j]
3     // Füge A[j] in die sortierte Sequenz A[0 .. j − 1]
4     i = j − 1
5     WHILE i ≥ 0 and A[i] > key
6           A[i + 1] = A[i]
7           i = i − 1
8     A[i + 1] = key
```

Beweis der Korrektheit

Um die Korrekthiet des Algorithmuses zu bestimmen nutzen wir die Schleifeninvariante

- Initialisierung: Beginn mit j=1, also mit dem Teilfeld bis j-1, besteht aus einem element und ist Sortiert
- Fortsetzung: Zu zeigen ist, dass die Invariante bei jeder Iteration erhalten bleibt. Die for-Schleife verschiebt jedes Element der sortierten Menge um eins nach rechts bis das zu sortierende Element eingefügt werden kann. Inkrementieren von j erhält die Schleifeninvariante
- Terminierung: Abbruhcbedingung for-Schleife wenn j>A.length-1. Jede Itearation erhöht j. Dann bei Abbruch j=n und einsetzen in Invariante liefert das Teilfeld A[0...n-1] welches aus den ursprünglichen Elementen besteht → Teilfeld = gesamtes Feld

15 Sortieralgorithmus: BubbleSort

Grundidee

▷ Vergleiche Paare von benachbarten Schlüsselwerten
▷ Tausche das Paar, falls rechter Schlüsselwert kleiner ist als linker

Code

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

Analyse
Anzahl der Vergleiche

- es werden stets alle Elemente er Teilfolge verglichen
- unabhängig von der Sortierung

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: $\Theta(n^2)$

16 Sortieralgorithmus: SelectionSort

Grundidee
▷ Sortieren durch direktes Auswählen

- MinSort: "wähle kleinstes Element in Array und tausche es nach vorne
- MaxSort: "wähle größetes Element in Array und tausche es nach hinten

Code

```
SELECTION-SORT(A)
1  FOR i = 0 TO A.length −2
2  k = i
3     FOR j = i + 1 TO A.length −1
4        IF A[j] < A[k]
5           k = j
6  SWAP(A[i], A[k])
```

17 Sortieralgorithmus: MergeSort

Grundidee
▷ Baut auf dem Devide-and-Conquer Prinziep auf

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

Code

Prinziep des MergeSorts

```
MERGE-SORT(A, p, r)
1  IF p < r
2        q = ⌊(p + r)/2⌋
3        MERGE-SORT(A, p, q)
4        MERGE-SORT(A, q + 1, r)
5        MERGE(A, p, q, r)
```

Beim Halbieren wird immer abgerundet, so kann jede Teilfolge sortiert werden.

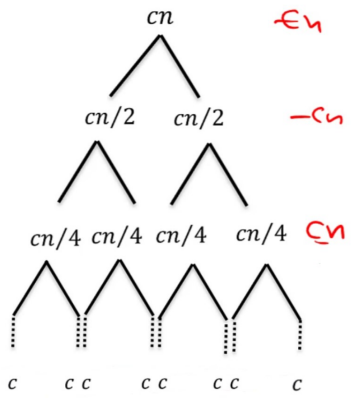
Prinziep des gesamtes Codes

```
MERGE(A, p, q, r)          A[p...r]   q: geteilt
1  n1 = q − p + 1
2  n2 = r − q
3  let L[0..n1] and R[0..n2] be new arrays
4  FOR i = 0 TO n1 − 1
5       L[i] = A[p + i]
6  FOR j = 0 TO n2 − 1
7       R[j] = A[q + j + 1]
8  L[n1] = ∞
9  R[n2] = ∞
10 i = 0
11 j = 0
12 FOR k = p TO r
13      IF L[i] ≤ R[j]
14           A[k] = L[i]
15           i = i + 1
16      ELSE A[k] = R[j]
17           j = j + 1
```

Analyse
Ziel: Bestimme Rekursionsgleichung für Laufzeit $T(n)$ von n Zahlen im schlechtesten Fall

- **Divide:** Es wird einfach die Mitte des Feldes berechnet. Dies benötigt konstante Zeit, also $\Theta(1)$
- **Conquer:** Wir lösen zwei Teilprobleme der Größe $\frac{n}{2}$ rekursiv. Dies trägt mit $2T(\frac{n}{2})$ uir Laufzeit bei
- **Combine:** Die Prozedur `MERGE` auf einem Teilfeld der Länge n lineare Zeit, also $\Theta(n)$

$$T(n) = \left\{ \begin{array}{ll} c & \text{falls } n = 1 \\ 2T(\frac{n}{2}) & \text{falls } n > 1 \end{array} \right\}$$



Daraus ergibt sich eine Komplexität von $\Theta(n \log n)$

Beweis der Korrektheit
Schleifeninvariante: Zu Beginn jeder Iteration der for-Schleife (Zeile 12-17) 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 $R[j]$ 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 $R[j]$ die kleinsten Elemente ihrer Arrays, die noch nicht zurück kopiert wurde.
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 kopiret 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 Zeile 14 kopiert wurde. Nach Erhöung der Variablen k und i stellt die Schleifeninvariante für die nächste Iteration wieder her.
Wenn $L[i] > R[j]$ dann analoges Argument mit Zeilen 16-17. **Terminierung:** Beim Abbruch gilt $k = r + 1$. Durch die Schleifeninvariante enthält $A[p...r]$ die kleinsten Elemente von $L[0...n_1]$ und $R[0...n_2]$ in sortierter Reihenfolge. Alle Elemente außer der Sentinels wurden korrekt zurück kopiert.

18 Sortieralgorithmus: QuickSort

Grundidee

- Wähle ein Pivotelement x des Arrays
- Zerlege den Array in zwei Teilarrays
  - Erster Teilarray: Enthält alle Elemente $y \leq x$
  - Zweiter Teilarray: Enthält alle Elemente $y > x$
- Sortiere rekursiv auf den Teillisten mit Quicksort
- Einelementige Listen sin schon sortiert

Dies lässt sich auf das Prinziep von Divide-and-Conquer zurückführen:

- **Divide:** Zerlege das Array $A[p...r]$ in zwei Teilarrays $A[p...q-1]$ und $A[q+1...r]$, sodass jedes Element aus $A[p...q-1]$ kleiner oder gleich $A[q]$ ist, welches wiederum kleiner oder gleich jedem Element von $A[q+1...r]$ ist. Berechnen 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.

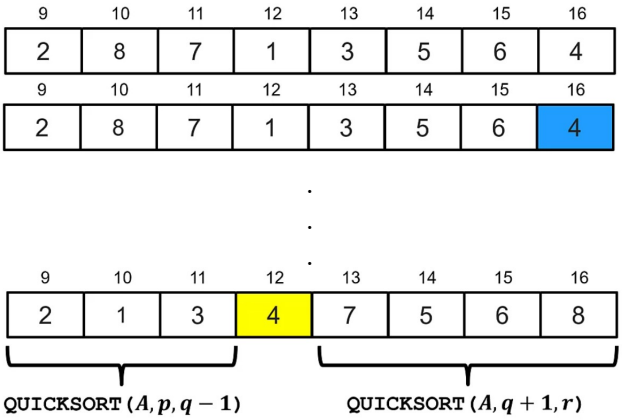
Code

```
QUICKSORT(A, p, r)
1  IF p < r
2       q = PARTITION(A, p, r)
3       QUICKSORT(A, p, q − 1)
4       QUICKSORT(A, q + 1, r)
```

Im Folgenden gucken wir uns den Algorithmus genauer an:

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  FOR j = p TO r − 1
4       IF A[j] ≤ x
5            i = i + 1
6            SWAP(A[i], A[j])
7  SWAP(A[i + 1], A[r])
8  RETURN i + 1
```

Zeile 1: Pivotelement wählen
Zeile 2: Index i setzen
Zeile 3-6: Teilarrays mit Element füllen
Zeile 7: Pivotelement tauschen
Zeile 8: Neuer Index des Pivotelements



Korrektheit von Quicksort

- **Schleifeninvariante:** Zu Beginn jeder Iteration der for-Schleife (Zeile3-6) gilt für den Arrayindex k folgendes:
  - Ist $p \leq k \leq i$, so gilt $A[k] \leq x$.
  - Ist $i+1 \leq k \leq j-1$, so gilt $A[k] > x$.
  - 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 j gibt und es auch keine Werte zwischen $i + 1$ und $j - 1$ gibt, sind die ersten beiden Eigenschaften trivial erfüllt.
  Die Zuweisung in Zeile 1 sorgt für die Erfüllung der dritten Eigenschaft.

- **Fortsetzung:** Zwei mögliche Fälle durch Zeile 4.
  Wenn $A[j] > x$, dann inkrementiert die Schleife ur 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 erfült. Analog gilt $A[j-1] > x$, da das Element welches mit $Aj - 1]$ 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 Menge gehört.

19 Exkurs

19.1 Totale Ordnung

Eine binäre Relation, $\leq$, auf der Menge M bildet eine Total Ordnung genau dann wenn M
- Reflexiv, $x \leq x$ für alle x,
- Transitiv, $x \leq y$ und $y \leq z$ impliziert $x \leq z$,
- Antisymmetrisch ist, $x \leq y$ und $y \leq x$ impliziert $x = y$