

5. Divide and Conquer

Divide-and-Conquer: Allgemeines Prinzip für effiziente Problemlösungsstrategien.

Vorgehensweise:

- **Teile:** Zerlege das Problem in mehrere, meist zwei, kleinere Teilprobleme.
- **Herrsche (Conquer):** Löse jeden der Teilprobleme rekursiv.
- **Zusammenführen (Combine):** Fasse die Lösungen der Teilprobleme zu einer Gesamtlösung zusammen.

Zitat:

Divide et impera.

Veni, vidi, vici.

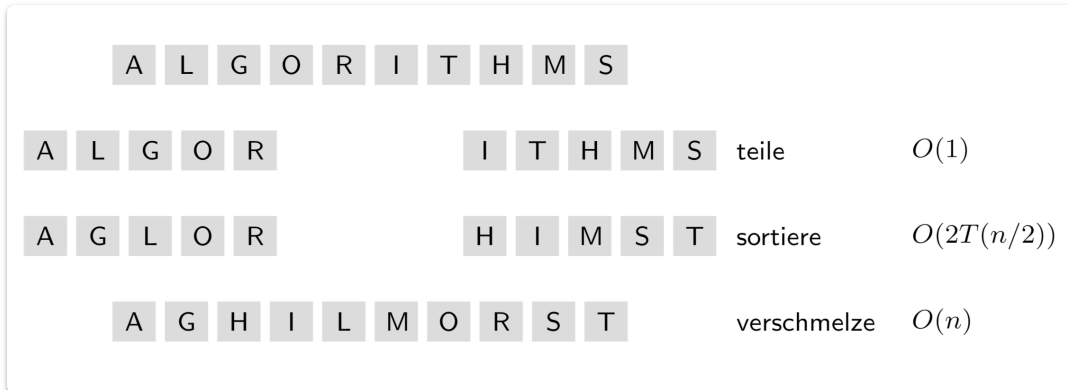
Julius Caesar

Sortieren: Wiederholung

- Primitive Sortiervverfahren:
 - Bubblesort
 - Insertionsort
 - Selectionsort
- Laufzeit: Die Laufzeit dieser Verfahren liegt im Worst- und Average-Case immer in $\Theta(n^2)$.
- Frage: Kann man im Worst- und Average-Case schneller sortieren?
- Antwort: Ja. Mergesort ist ein Beispiel dafür.

Mergesort (Sortieren durch Mischen)

- Mergesort:
 - Teile Array in zwei Hälften.
 - Sortiere jede Hälfte rekursiv.
 - Verschmelze zwei Hälften zu einem sortierten Ganzen.



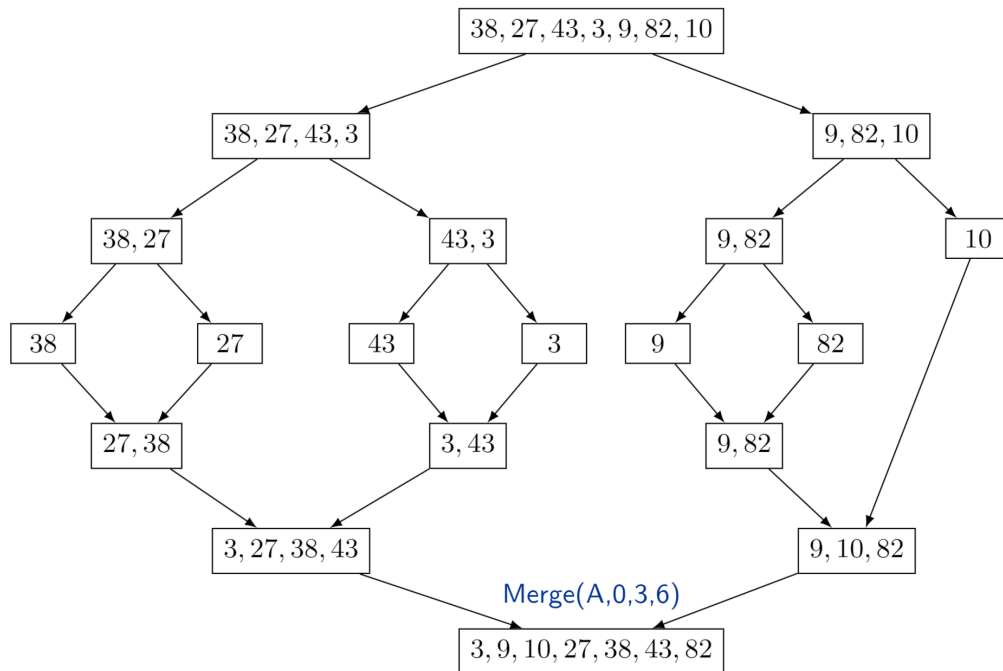
- Pseudocode:
 - Mergesort für ein Array A .
 - Sortiert den Bereich $A[l]$ bis $A[r]$.

```

Mergesort( $A, l, r$ ):
  if  $l < r$ 
     $m \leftarrow \lfloor (l + r) / 2 \rfloor$ 
    Mergesort( $A, l, m$ )
    Mergesort( $A, m + 1, r$ )
    Merge( $A, l, m, r$ )
  
```

- Aufruf: Mergesort($A, 0, n - 1$) für ein Array A mit n Elementen.

Beispiel



Merging (Verschmelzen)

- Vorgehen: Verschmelze zwei sortierte Listen zu einer sortierten Gesamtliste.
- Wie kann man effizient verschmelzen?
 - Benutze temporäres Array.
 - Durchlaufe beide Listen vom Anfang an.
 - Führe die Elemente beider Listen im Reißverschlussverfahren zusammen, übernehme dabei jeweils das kleinste Element der beiden Listen.
 - Hat lineare Laufzeit.



- Pseudocode: Merge auf ein Array *A*. Verwendet Hilfsarray *B*.

```

Merge(A, l, m, r):
  i ← l, j ← m + 1, k ← l
  while i ≤ m und j ≤ r
    if A[i] ≤ A[j]
      B[k] ← A[i], i ← i + 1
    else
      B[k] ← A[j], j ← j + 1
    k ← k + 1
  if i > m
    for h ← j bis r
      B[k] ← A[h], k ← k + 1
  else
    for h ← i bis m
      B[k] ← A[h], k ← k + 1
  for h ← l bis r
    A[h] ← B[h]

```

Eine nützliche Rekursionsgleichung

- Definition: $C(n)$ = Anzahl der Schlüsselvergleiche (*comparisons*) in Mergesort bei einer Eingabegröße n .
- Mergesort Rekursion:

$$C(n) \leq \begin{cases} 0 & \text{wenn } n = 1 \\ \underbrace{C(\lceil n/2 \rceil)}_{\text{linke Hälfte}} + \underbrace{C(\lfloor n/2 \rfloor)}_{\text{rechte Hälfte}} + \underbrace{n}_{\text{Verschmelzen}} & \text{sonst} \end{cases}$$

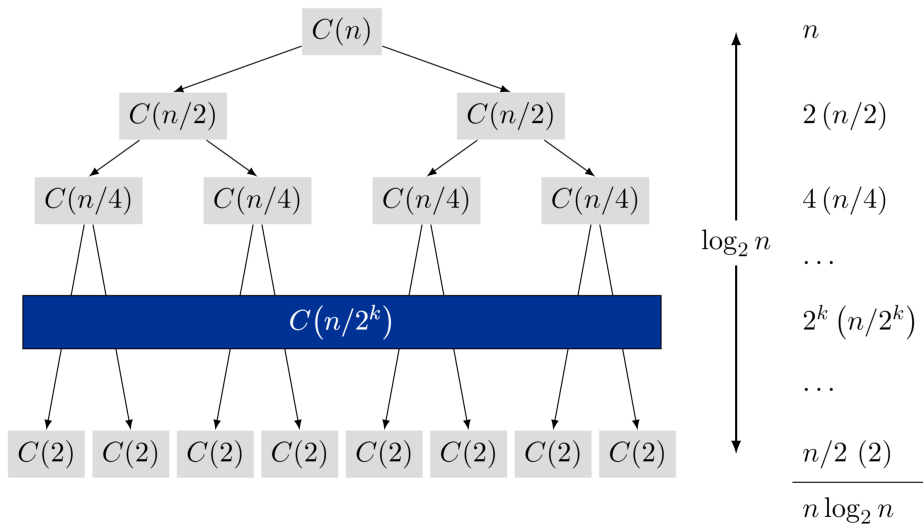
- Lösung: $C(n) = O(n \log_2 n)$.
- Beweis: Wir beschreiben mehrere Wege das $O(n \log_2 n)$ zu beweisen.
- Annahmen:
 - Wir nehmen anfänglich an, dass n eine Zweierpotenz ist.
 - Für ein allgemeines n' mit $\frac{n}{2} < n' < n$ (wobei n eine Zweierpotenz ist) gilt dann:

$$C(n') = O(n \log n)$$

- da $O(\frac{n}{2} \log \frac{n}{2}) = O(n \log n)$ gilt.

Beweis durch Rekursionsbaum

$$C(n) \leq \begin{cases} 0 & \text{wenn } n = 1 \\ \underbrace{2C(n/2)}_{\text{Sortieren beider H\u00e4lften}} + \underbrace{n}_{\text{Verschmelzen}} & \text{sonst} \end{cases}$$



Beweis durch Aufl\u00f6sen der Rekursion

Behauptung: Wenn $C(n)$ die Rekursion erf\u00fcllt, dann $C(n) \leq n \log_2 n$.

$$C(n) \leq \begin{cases} 0 & \text{wenn } n = 1 \\ \underbrace{2C(n/2)}_{\text{Sortieren beider H\u00e4lften}} + \underbrace{n}_{\text{Verschmelzen}} & \text{sonst} \end{cases}$$

Beweis: F\u00fcr $n > 1$:

$$\begin{aligned}
 \frac{C(n)}{n} &\leq \frac{2C(n/2)}{n} + 1 = \frac{C(n/2)}{n/2} + 1 \\
 &\leq \frac{2C(n/4)}{n/2} + \frac{n/2}{n/2} + 1 = \frac{C(n/4)}{n/4} + 1 + 1 \\
 &\leq \dots \\
 &\leq \frac{C(n/n)}{n/n} + \underbrace{1 + \dots + 1}_{\log_2 n} \\
 &= \log_2 n
 \end{aligned}$$

Beweis durch Induktion

Behauptung: Wenn $C(n)$ die Rekursion erfüllt, dann $C(n) \leq n \log_2 n$.

$$C(n) \leq \begin{cases} 0 & \text{wenn } n = 1 \\ \underbrace{2C(n/2)}_{\text{Sortieren beider Halfen}} + \underbrace{n}_{\text{Verschmelzen}} & \text{sonst} \end{cases}$$

Beweis: (durch Induktion auf n)

- Induktionsanfang: $n = 1$.
- Induktionsbehauptung: $C(n) \leq n \log_2 n$.
- Ziel: Zeige, dass $C(2n) \leq 2n \log_2 2n$.

$$\begin{aligned} C(2n) &\leq 2C(n) + 2n \\ &\leq 2n \log_2 n + 2n \\ &= 2n(\log_2 n + 1) \\ &= 2n(\log_2(2n/2) + 1) \\ &= 2n(\log_2 2n - \log_2 2 + 1) \\ &= 2n(\log_2 2n - 1 + 1) \\ &= 2n \log_2 2n \end{aligned}$$

Analyse der Mergesort-Rekursion

Behauptung: Wenn $C(n)$ die folgende Rekursion erfüllt, dann $C(n) \leq n \lceil \log_2 n \rceil$.

$$C(n) \leq \begin{cases} 0 & \text{wenn } n = 1 \\ \underbrace{C(\lceil n/2 \rceil)}_{\text{linke Halfte}} + \underbrace{C(\lfloor n/2 \rfloor)}_{\text{rechte Halfte}} + \underbrace{n}_{\text{Verschmelzen}} & \text{sonst} \end{cases}$$

Beweis: (durch Induktion auf n)

- Induktionsanfang: $n = 1$.
- Definiere $n_1 = \lceil n/2 \rceil$, $n_2 = \lfloor n/2 \rfloor$.
- Induktionsschritt: Ist wahr fur $1, 2, \dots, n-1$.

$$\begin{aligned} C(n) &\leq C(n_1) + C(n_2) + n \\ &\leq n_1 \lceil \log_2 n_1 \rceil + n_2 \lceil \log_2 n_2 \rceil + n \\ &\leq n_1 \lceil \log_2 n_1 \rceil + n_2 \lceil \log_2 n_1 \rceil + n \\ &= n \lceil \log_2 n_1 \rceil + n \\ &\leq n (\lceil \log_2 n \rceil - 1) + n \\ &= n \lceil \log_2 n \rceil \end{aligned}$$

$$\begin{aligned} n_1 &= \lceil n/2 \rceil \\ &\leq \lceil 2^{\lceil \log_2 n \rceil} / 2 \rceil \\ &= 2^{\lceil \log_2 n \rceil} / 2 \\ &\Rightarrow \log_2 n_1 \leq \lceil \log_2 n \rceil - 1 \end{aligned}$$

- Asymptotische untere Schranke fur $C(n)$: Das Verschmelzen fur n Elemente benotigt zumindest $C_{best} = \lfloor \frac{n}{2} \rfloor$ Schlusselvergleiche.
- Daher gilt:

$$C_{best}(n) = C_{worst}(n) = C_{avg}(n) = \Theta(n \log n)$$

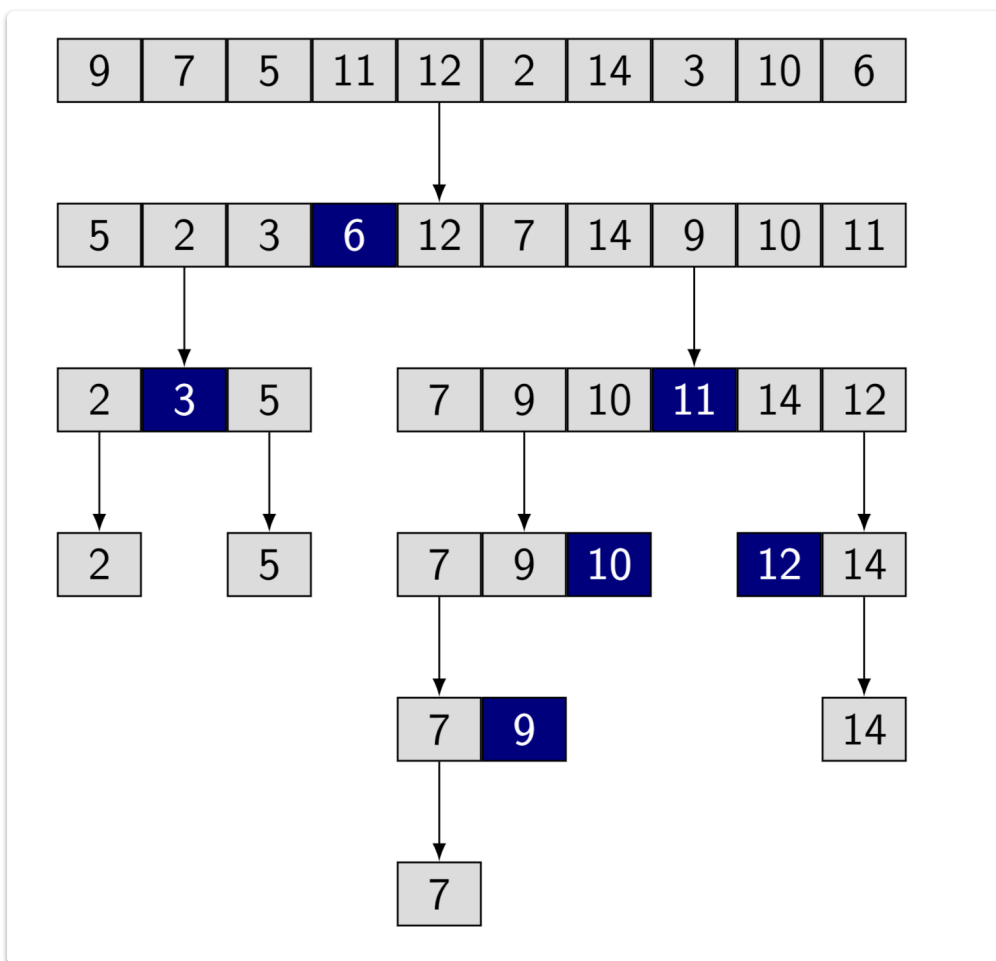
- Allgemein: Die Laufzeit von Mergesort wird durch die Anzahl der notwendigen Vergleiche dominiert.
- Daher gilt:

$$T_{best}(n) = T_{worst}(n) = T_{avg}(n) = \Theta(C(n)) = \Theta(n \log n)$$

Quicksort

- Quicksort: Benutzt auch das Divide-and-Conquer-Prinzip, aber auf eine andere Art und Weise.
- Teile: Wähle „Pivotelement“ x aus Folge A , z.B. das an der letzten Stelle stehende Element. Teile A ohne x so in zwei Teilfolgen A_1 und A_2 , dass gilt:
 - A_1 enthält nur Elemente $\leq x$.
 - A_2 enthält nur Elemente $\geq x$.
- Herrsche:
 - Rekursiver Aufruf für A_1 .
 - Rekursiver Aufruf für A_2 .
- Kombiniere: Bilde A durch Hintereinanderfügen von A_1, x, A_2 .

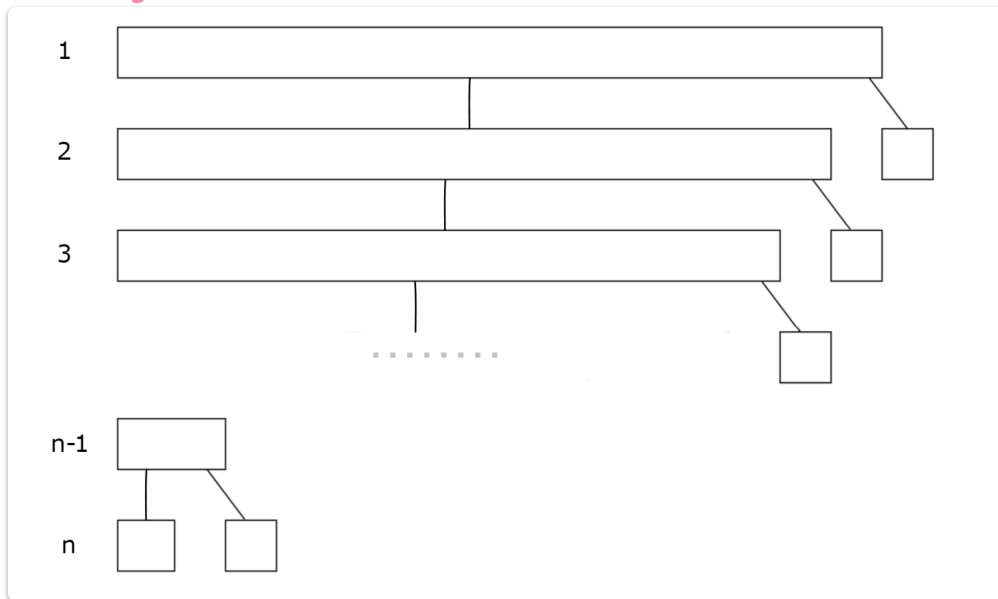
Beispiel



Analyse

- Best-Case:
 - Die beiden Teilfolgen haben *immer (ungefähr) die gleiche Länge*.
 - Die Höhe des Aufrufbaumes ist $\Theta(\log n)$.
 - Auf jeder Aufrufebeine werden $\Theta(n)$ Vergleiche durchgeführt.
 - Die Anzahl der Vergleiche und die Laufzeit liegen in $\Theta(n \log n)$.

- Worst-Case: Jede (Teil-)Folge wird *immer beim letzten (oder immer beim ersten) Element geteilt*.



- Die Anzahl der Vergleiche ist $\Theta(n^2)$.

Worst-Case:

- Mögliches Szenario:
 - *Aufsteigend sortierte Folge* und es wird *immer das hinterste Element als Pivotelement gewählt*.
 - Alle restlichen Elemente sind kleiner als das Pivotelement und daher wird die Rekursion *nur für die linke Seite* (alle restlichen Elemente außer dem Pivotelement) weitergeführt. Die rechte Seite ist leer (*Terminierung der Rekursion*).
 - Die Größe der Teilfolge in der nächsten Rekursionsstufe verringert sich *immer nur um 1!*
 - $\Theta(n)$ rekursive Aufrufe.
 - Die Laufzeit liegt in $\Theta(n^2)$.
 - Der zum Array zusätzlich benötigte Speicherplatz ist durch die $\Theta(n)$ rekursiven Aufrufe ebenfalls $\Theta(n)$.
- „Vermeiden“ des Worst-Case in der Praxis:
 - Pivotelement *immer zufällig wählen*.
 - *Randomisierter Quicksort*.
 - Es ist *nicht mehr* die sortierte Folge das Worst-Case-Szenario.
 - Betrachte *jeweils das erste, letzte und mittlere Element* (an der Position $\lfloor \frac{n}{2} \rfloor$) und nimm den *Median als Pivotelement*.

Average-Case:

- Komplizierter Beweis zeigt, dass die Anzahl der Vergleiche und die Laufzeit dafür auch in $\Theta(n \log n)$ liegen.
- Beispiel für Average-Case: Jede (Teil-)Liste wird nahe der Mitte geteilt.

- Die Anzahl der Vergleiche ist $\Theta(n \log n)$.

Speicherplatzkomplexität

- Speicherplatzkomplexität: Ist ein Maß für das Anwachsen des Speicherbedarfs eines Algorithmus in Abhängigkeit von der Eingabegröße.
- Beispiele für Speicherplatzkomplexität:
 - Quicksort: Worst-Case liegt in $\Theta(n)$, Best-/Average-Case liegt in $\Theta(\log n)$.
 - Mergesort: Best-/Average-/Worst-Case liegt in $\Theta(n)$.
- Praxis: In der Praxis wird daher Quicksort sehr oft Mergesort vorgezogen.

Vergleich von Sortierverfahren

Tabelle: Laufzeit und Vergleiche.

Sortierverfahren	Best-Case	Average-Case	Worst-Case
Insertionsort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Selectionsort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Quicksort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$

Tabelle: Zusätzlicher Speicher.

Sortierverfahren	Best-Case	Average-Case	Worst-Case
Insertionsort	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Selectionsort	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Mergesort	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Quicksort	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$

Einsatz von Sortierverfahren

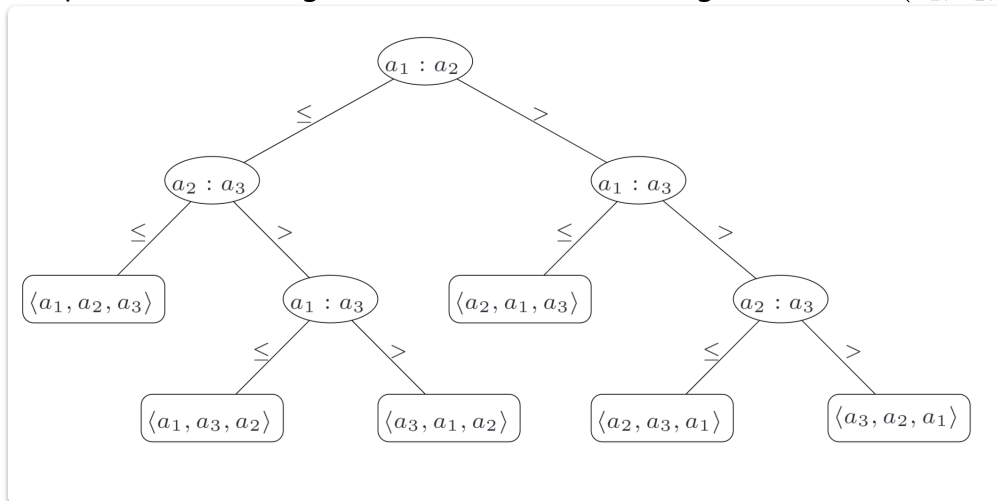
- Quicksort:
 - Wird sehr oft in allgemeinen Sortiersituationen bevorzugt.
- Mergesort:
 - Mergesort wird hauptsächlich für das Sortieren von Listen verwendet.
 - Wird auch für das Sortieren von Dateien auf externen Speichermedien eingesetzt.
 - Dabei wird aber eine iterative Version von Mergesort (*Bottom-up-Mergesort*) verwendet, bei der nur $\log n$ -mal eine Datei sequentiell durchgegangen wird.

Eine untere Schranke

- Ausgangslage: Wir haben verschiedene Sortieralgorithmen kennengelernt. Die Worst-Case Laufzeit liegt im Bereich $O(n \log n)$ bis $O(n^2)$.
- Frage: Geht es besser als in $O(n \log n)$ Zeit, z.B. $O(n)$?
- Antwort: Wir zeigen für das allgemeine Sortierproblem unter der Annahme, dass n verschiedene Schlüssel sortiert werden müssen, dass $O(n \log n)$ optimal ist.

Entscheidungsbaum

- Entscheidungsbaum:
 - Die Knoten entsprechen einem Vergleich von Elementen a_i und a_j .
 - Die Blätter entsprechen den Permutationen.
- Beispiel: Entscheidungsbaum des Insertionsort Algorithmus für (a_1, a_2, a_3) .



- Anzahl der Schlüsselvergleiche: Die Anzahl der Schlüsselvergleiche im Worst-Case C_{worst} entspricht genau der Anzahl der Knoten auf dem längsten Pfad von der Wurzel bis zu einem Blatt minus 1.
- Frage: Wie lautet die untere Schranke für die Höhe eines Entscheidungsbaums?
- Satz: Jeder Entscheidungsbaum für die Sortierung von n paarweise verschiedenen Schlüsseln hat die Höhe $\Omega(n \log_2 n)$.

Beweis

Beweis:

- Betrachte einen Entscheidungsbaum der Höhe h , der n unterschiedliche Schlüssel sortiert.
- Der Baum hat mindestens $n!$ Blätter.
- $n! \leq 2^h$, das impliziert $h \geq \log_2(n!)$.
- Hilfsrechnung:

$$n! \geq n \cdot (n-1) \cdot (n-2) \cdots \lceil \frac{n}{2} \rceil \geq (\lceil \frac{n}{2} \rceil)^{\lceil \frac{n}{2} \rceil} \geq \frac{n}{2}^{\frac{n}{2}}$$
- $h \geq \log_2(n!) \geq \log_2 \left(\frac{n}{2}^{\frac{n}{2}} \right) \geq \frac{n}{2} \log_2 \frac{n}{2} = \frac{n}{2} (\log_2 n - 1) = \Omega(n \log_2 n)$

Untere Schranke für allgemeines Sortierproblem

- Satz: Jedes allgemeine Sortierverfahren benötigt zum Sortieren von n paarweise verschiedenen Schlüsseln mindestens $\Omega(n \log n)$ Laufzeit im Worst-Case.
- Folgerung: Mergesort ist ein asymptotisch *zeitoptimaler* Sortieralgorithmus. Es geht (asymptotisch) nicht besser!

Lineare Sortiervverfahren

- Beobachtung: Das Ergebnis für die untere Schranke basiert auf der Annahme, dass man keine Eigenschaften der zu sortierenden Elemente ausnutzt, sondern lediglich den Vergleichsoperator.
- Praxis: Tatsächlich sind aber meist Wörter über ein bestimmtes Alphabet (d.h. einer bestimmten Definitionsmenge) gegeben, z.B.:
 - Wörter über $\{a, b, \dots, z, A, B, \dots, Z\}$.
 - Dezimalzahlen.
 - Ganzzahlige Werte aus einem kleinen Bereich.
- Lineare Sortiervverfahren: Diese Information über die Art der Werte und ihren Wertebereich kann man zusätzlich ausnutzen und dadurch im Idealfall Verfahren erhalten, die auch im Worst-Case in linearer Zeit sortieren.

Beispiel: Countsort

Eingabe: n Zahlen im Bereich 0 bis z im Array A , Hilfsarray Counts , $z < n$

```

for  $j \leftarrow 0$  bis  $z$ 
     $\text{Counts}[j] \leftarrow 0$ 
for  $i \leftarrow 0$  bis  $n - 1$ 
     $\text{Counts}[A[i]] \leftarrow \text{Counts}[A[i]] + 1$ 
 $i \leftarrow 0$ 
for  $j \leftarrow 0$  bis  $z$ 
    for  $k \leftarrow 0$  bis  $\text{Counts}[j] - 1$ 
         $A[i] \leftarrow j$ 
         $i \leftarrow i + 1$ 
  
```

Komplexität: Erste Schleife in $\Theta(z)$, zweite Schleife in $\Theta(n)$, dritte (mit vierter) Schleife kann nur maximal n Zahlen bearbeiten und ist daher auch in $\Theta(n)$. Damit läuft der Algorithmus in $\Theta(z + n + n) = \Theta(n)$ (da $z = O(n)$).

Beispiele zu Divide and Conquer:

Gute Beispiele für dieses Prinzip findet man in den Slides ab Seite 54.

Da hätte man ein mal [Inversionen zählen](#) und dann noch [Closest Pair of Points](#)