Untere Schranke für allgemeine Sortierverfahren

- Prinzipielle Frage: wie schnell kann ein Algorithmus (im worst case) überhaupt sein?
- Satz:
  Zum Sortieren einer Folge von n Keys mit einem allgemeinen Sortierverfahren sind im Worst-Case ebenso wie im Average-Case mindestens $\Omega(n \log n)$ Vergleichsoperationen zwischen zwei Schlüsseln erforderlich.
- Beweis durch Modellierung von allgemeinen Sortierverfahren als Entscheidungsbäume

Wichtiges Charakteristikum von allgemeinem Sortieren

- Allgemeines Sortieren = Vergleichsbasiertes Sortieren:
  - Nur Vergleich von Elementpaaren wird benutzt, um die Ordnung einer Folge zu erhalten
  - Für alle Algos gilt: pro Vergleich eine konstante Anzahl weitere Operationen (z.B. 2 Elemente kopieren, Schleifenzähler erhöhen, …)

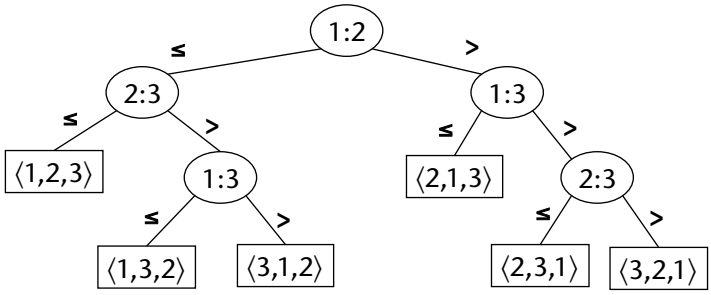
  → Daher: untere Schranke der Vergleichsanzahl = untere Schranke für die Komplexität eines vergleichsbasiertes Sortieralgorithmus'
- Alle bisher behandelten Sortierverfahren sind vergleichsbasiert
- Die bisher beste Worst-Case-Komplexität ist $O(n \log n)$ (Mergesort, Heapsort)
- Voriger Satz besagt: worst-case Komplexität von Merge- und Heapsort ist optimal

Der Entscheidungsbaum (*decision tree*)

- Abstraktion eines Sortierverfahrens durch einen Binärbaum
- Ein Entscheidungsbaum stellt eine Folge von Vergleichen dar
  - von irgendeinem Sortieralgorithmus
  - für Eingaben einer vorgegebenen Größe
  - lässt alles andere (Kontrollfluß und Datenverschiebungen) außer Acht, es werden nur Vergleiche betrachtet
- Interne Knoten bekommen Bezeichnung *i:j* = die Positionen der Elemente im Array, die verglichen werden
- Blätter werden mit Permutationen $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ bezeichnet, die der Algorithmus bestimmt

Beispiel

- Entscheidungsbaum für Insertionsort mit drei Elementen



  - Beinhaltet 3! = 6 Blätter

- Ausführen des Sortieralgorithmus' für bestimmte Eingabe entspricht dem Verfolgen eines Weges von der Wurzel zu einem Blatt
- Entscheidungsbaum bildet alle möglichen Ausführungsabläufe ab
- Bei jedem internen Knoten findet ein Vergleich $a_i \le a_j$ statt.
  - für $a_i \le a_j$, folge dem linken Unterbaum
  - sonst, folge dem rechten Unterbaum
- An einem Blatt ist die Ordnung $a_{\pi(1)} \le a_{\pi(2)} \le \ldots \le a_{\pi(n)}$ festgelegt.
- Ein korrekter Sortieralgorithmus *muß* alle Permutationen erzeugen können
  - M.a.W.: jede der *n!* Permutationen muß bei mindestens einem Blatt des Entscheidungsbaumes vorkommen

Untere Schranke für Worst-Case

- Anzahl der Vergleiche im Worst-Case eines Sortieralgorithmus' = Länge des längsten Weges im Entscheidungsbaum von der Wurzel zu irgendeinem Blatt
  = die Höhe des Entscheidungsbaumes
- Untere Schranke für die Laufzeit = untere Schranke für die Höhe aller Entscheidungsbäume, in denen jede Permutation als erreichbares Blatt vorkommt

- Satz: Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche im Worst-Case.
- Beweis:
  - Es reicht, die Höhe eines Entscheidungsbaumes zu bestimmen.
  - h = Höhe, l = Anzahl der Blätter im Entscheidungsbaum
  - Im Entscheidungsbaum für n Elemente gilt: $l \geq n!$
  - Im Binärbaum mit der Höhe h gilt: $l \leq 2^h$
  - Also: $n! \leq l \leq 2^h \Rightarrow n! \leq 2^h \Rightarrow h \geq \log(n!)$
  - Stirling-Approximation für $n!$ liefert: $n! > \left(\frac{n}{e}\right)^n$
  - Somit : $h \geq \log(n!) \geq \log\left(\left(\frac{n}{e}\right)^n\right)$

$$= n\log(n) - n\log(e) \Rightarrow h \in \Omega(n \log n)$$

Untere Schranke für Average-Case

- Satz:

  Jedes vergleichsbasierte Sortierverfahren benötigt $\Omega(n \log n)$ Vergleiche im Mittel.

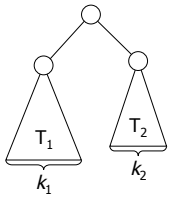
- Wir beweisen zunächst …
- Lemma:

  Die mittlere Tiefe eines Blattes eines Binärbaumes mit k Blättern ist mindestens $\log_2(k)$.

Beweis des Lemmas

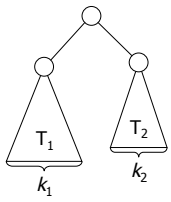
- Beweis durch Widerspruch
  - Annahme: Lemma ist falsch
  - Sei T der kleinste Binärbaum, der Lemma verletzt; T habe k Blätter
- $k \geq 2$ muss gelten (Lemma gilt ja für $k = 1$)
- T hat linken Teilbaum T_1 mit k_1 Blättern und rechten Teilbaum T_2 mit k_2 Blättern
  - es gilt $k_1 + k_2 = k$
  - bezeichne mit $\overline{d}(T)$ die mittlere Tiefe von Baum T
  - da $k_1, k_2 < k$ sind, gilt das Lemma für T_1, T_2:

$$\overline{d}(T_1) \geq \log(k_1)$$
$$\overline{d}(T_2) \geq \log(k_2)$$



- Für jedes Blatt von T gilt: Tiefe dieses Blattes, bezogen auf die Wurzel von T = Tiefe + 1, bezogen auf die Wurzel von T_1 bzw. T_2

$$k\overline{d}(T) = \sum_{\substack{\text{Blätter} \\ \text{in } T}} \text{Tiefe von } b$$



- Also:
  Summe aller Blattiefen in T = $k_1\left(\overline{d}(T_1)+1\right) + k_2\left(\overline{d}(T_2)+1\right)$

$$\begin{aligned}\Rightarrow \overline{d}(T) &= \frac{1}{k}\left(k_1(\overline{d}(T_1)+1) + k_2(\overline{d}(T_2)+1)\right) \\ &\geq \frac{k_1}{k}\left(\log(k_1)+1\right)) + \frac{k_2}{k}\left(\log(k_2)+1\right)) \\ &= \frac{1}{k}\left(k_1\log(2k_1) + k_2\log(2k_2)\right) =: f(k_1, k_2)\end{aligned}$$

- Funktion $f(k_1,k_2)$ nimmt, unter der Nebenbedingung $k_1 + k_2 = k$, das Minimum bei $k_1 = k_2 = k/2$ an
- Also

$$\overline{d}(T) \geq \frac{1}{k}\left(\frac{k}{2}\log(k) + \frac{k}{2}\log(k)\right) = \log(k)$$

- Widerspruch zur Annahme!

Beweis des Satzes

- Mittlere Laufzeit eines Sortierverfahrens = mittlere Tiefe eines Blattes im Entscheidungsbaum
- Entscheidungsbaum hat $k \geq N!$ viele Blätter

also

$$\overline{d} \geq \log N! \geq \log\left(\frac{N}{2}\right)^{\frac{N}{2}} = \frac{N}{2}\log\left(\frac{N}{2}\right)$$

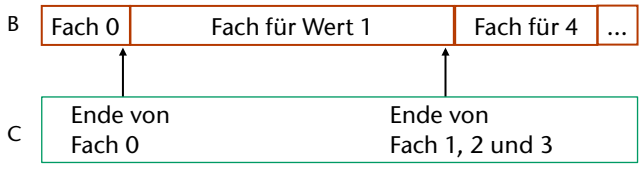
$$\overline{d} \in \Omega(N\log(N))$$

Lineare Sortierverfahren

- Bisherige Sortieralgorithmen basieren auf den Operationen
  - Vergleich zweier Elemente
  - Vertauschen der Elemente
- Führt bestenfalls zum Aufwand N·log(N) (schneller geht es nicht)
- Distributionsort: Klasse von Sortierverfahren, die zusätzliche Operationen (neben Vergleichen) verwenden, z.B. arithmetische Operationen, Zählen, Zahldarstellung als Ziffernfolge, …
- Allgemeines Schema (ganz grob):
  - Verteilung der Daten auf Fächer (*distribute*)
  - Einsammeln der Daten aus den Fächern, wobei Ordnung innerhalb der Fächer erhalten bleiben muß(!) (*gather*)

Counting-Sort

- Vorbedingung: Keys kommen aus einem diskreten Bereich
  - Annahme hier: Keys sind natürliche Zahlen
- Zunächst simple Idee: reserviere für jeden mögl. Wert ein Fach
  - Problem: jedes Fach müsste potentiell Platz für alle Datensätze bieten
- Trick: verwende nur ein Ausgaberarray B und mache die Fächer genau so groß, wie sie benötigt werden. Dazu muß man sich in einem zweiten Array C die Fächergrenzen merken
- Beispiel:



Der Algorithmus

- Die Algorithmusidee:
  1. Bestimme $k = \max_i \{a_i\}$
  2. Für alle i, 0 ≤ i ≤ k, bestimme Anzahl C_i der a_j mit $a_j \le i$:

$$C_i := |\{a_j \in A \mid a_j \le i\}|, \quad C_{-1} = 0$$

$$C_i - C_{i-1} = |\{a_j \in A \mid a_j = i\}|$$

  3. Erzeuge Ausgabe-Array *B*, genauso groß wie *A*
  4. Kopiere a_j mit a_j = i in die Felder:

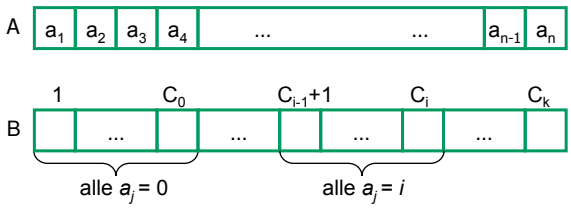


Illustration von Countingsort

```
C = [0] * (k+1)
for j in range( 0, len(A) ):
    C[ A[j] ] += 1
# C[i] = # elements a_j = i
for i in range( 1, k+1 ):
    C[i] += C[i-1]
# C[i] = # elements a_j <= i
for j in range( len(A), 0 ):
    C[A[j]] -= 1
    B[ C[A[j]] ] = A[j]
```

	0	1	2	3	4	5	6	7
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	2	4	7	7	8

	0	1	2	3	4	5
C	2	0	2	3	0	1

Illustration von Countingsort

```
C = [0] * (k+1)
for j in range( 0, len(A) ):
    C[ A[j] ] += 1
# C[i] = # elements a_j = i
for i in range( 1, k+1 ):
    C[i] += C[i-1]
# C[i] = # elements a_j <= i
for j = len(A)-1, …, 0:
    C[A[j]] -= 1
    B[ C[A[j]] ] = A[j]
```

	0	1	2	3	4	5	6	7
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	0	2	2	4	7	7

	0	1	2	3	4	5	6	7
B	0	0	2	2	3	3	3	5

Analyse

```
C = [0] * (k+1)                          } O(k)
for j in range( 0, len(A) ):             } O(n)
    C[ A[j] ] += 1
# C[i] = # elements a_j = i
for i in range( 1, k+1 ):                } O(k)
    C[i] += C[i-1]
# C[i] = # elements a_j <= i
for j = len(A)-1,…, 0:                   } O(n)
    C[A[j]] -= 1
    B[ C[A[j]] ] = A[j]
```

- Satz: Counting-Sort besitzt Laufzeit $O(n+k)$
- Korrolar: Gilt $k \in O(n)$, so besitzt Counting-Sort Laufzeit $O(n)$