Wie "langsam" muss Sortieren sein?

Frage: Gibt es Sortieralgorithmen mit Laufzeit $o(n \cdot \log n)$?

Beschränke Betrachtung auf
Vergleichsbasierte Algoritmen

- Vergleich ob < , = , > ist die einzige erlaubte Operation auf Schlüsseln
  (außer Kopieren oder im Speicher Bewegen)
- Algorithmus muss für jeden Typ von Schlüssel funktionieren, solange < , = , > definiert sind und eine totale Ordnung auf den Schlüsseln darstellen

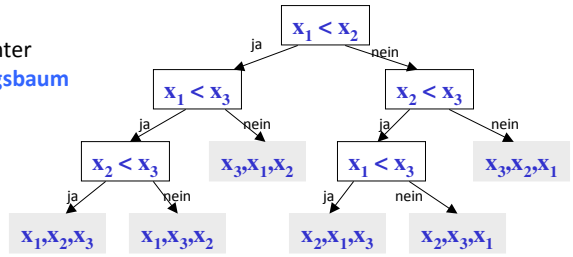
z.B. unzulässig: arithmetische Operationen auf Schlüsseln,
Verwendung von Schlüssel als Index in Feld

Wenn die Eingabegröße fixiert wird, kann jeder vergleichsbasierte Algorithmus als schleifenfreies Programm von **if**-Statements geschrieben werden

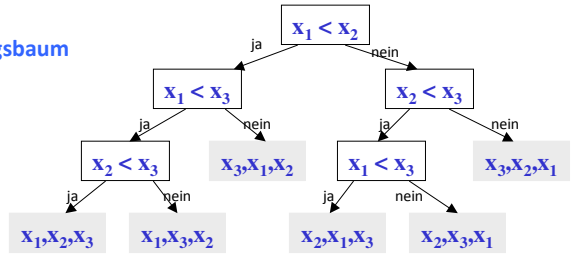
Bsp.: Programm um n=3 Schlüssel x_1,x_2,x_3 zu sortieren

if $x_1<x_2$ **then if** $x_1<x_3$ **then if** $x_2<x_3$ **then** output x_1,x_2,x_3
else output x_1,x_3,x_2
else output x_3,x_1,x_2
else if $x_2<x_3$ **then if** $x_1<x_3$ **then** output x_2,x_1,x_3
else output x_2,x_3,x_1
else output x_3,x_2,x_1

Äquivalenter
Entscheidungsbaum



Entscheidungsbaum



- Programmdurchlauf entspricht Wurzel-Blatt Pfad
- Länge des Pfades entspricht Anzahl der Schlüsselvergleiche bei diesem Programmdurchlauf
- Blatt entspricht der (sortierten) Ausgabepermutation der Eingabe
- Worst-case Laufzeit des Programmes entspricht dem längsten Wurzel-Blatt Pfad im Baum, also der Höhe des Baums.
- Zu zeigen:
  Jeder Entscheidungsbaum fürs Sortieren muss große Höhe haben.

- Programmdurchlauf entspricht Wurzel-Blatt Pfad
- Länge des Pfades entspricht Anzahl der Schlüsselvergleiche bei diesem Programmdurchlauf
- Blatt entspricht der (sortierten) Ausgabepermutation der Eingabe
- Worst-case Laufzeit des Programmes entspricht dem längsten Wurzel-Blatt Pfad im Baum, also der Höhe des Baums.
- Zu zeigen:
  Jeder Entscheidungsbaum fürs Sortieren muss große Höhe haben.

B Entscheidungsbaum, um n Schlüssel zu sortieren

$\#\text{Blätter}(B) \geq n!$ (mindestens ein Blatt für jede der $n!$ Eingabepermutationen)

$\#\text{Blätter}(B) \leq 2^{\text{Höhe}(B)}$

$$\begin{aligned}\text{Höhe}(B) &\geq \log_2(\text{Blätter}(B)) \\ &\geq \log_2 n! \\ &> \log_2 (n/e)^n \\ &= n\cdot\log_2 n - n\cdot\log_2 e \\ &> n\cdot\log_2 n - 1.5n\end{aligned}$$

$n! > (n/e)^n$ da $e^n = \sum_{i\geq 0} n^i/i! > n^n/n!$

Satz: Für jeden Entscheidungsbaum B zum Sortieren von n Schlüsseln gilt

$$\text{Höhe}(B) > n \cdot \log_2 n - 1.5n.$$

Korollar: Für jeden vergleichsbasierten Algorithmus zum Sortieren von n Schlüsseln gibt es eine Eingabe, für die der Algorithmus mehr als $n \cdot \log_2 n - 1.5n$ Vergleiche durchführt.

Korollar: Jeder vergleichsbasierte Algorithmus zum Sortieren von n Schlüsseln hat im schlechtesten Fall Laufzeit

$$\Omega(n \cdot \log n).$$

Korollar: Jeder vergleichsbasierte Algorithmus zum Sortieren von n Schlüsseln hat im schlechtesten Fall Laufzeit

$$\Omega(n \cdot \log n).$$

Will man schneller als in $\Theta(n \cdot \log n)$. sortieren, muss man anderes machen, als Schlüssel zu vergleichen. Man kann sich auf spezielle Schlüsseltypen konzentrieren und deren Eigenschaften ausnutzen.

Beispiel:
Die Schlüssel sind ganze Zahlen aus einem kleinen Bereich, z.B. $\{0,\ldots,K-1\}$

Problem:
Sortiere n Stücke $x_1,\ldots,x_n$ nach Schlüssel $key(x_i)$, wobei $key(x_i) \in \{0,\ldots,K-1\}$.

Problem:
Sortiere n Stücke $x_1,\ldots,x_n$ nach Schlüssel $key(x_i)$, wobei $key(x_i)\in\{0,\ldots,K-1\}$.

$x_1,\ldots,x_n$ ist gegeben durch Eingabefeld $X[1..n]$

CountingSort
Idee: Bestimme für jedes $h\in\{0,\ldots,K-1\}$ den Wert $C[h]$, der besagt für wie viele Stücke x gilt $key(x)\leq h$.

Die Stücke x mit $key(x)=h$ gehören dann im Ausgabefeld $B[1..n]$ auf die Stellen $C[h-1]+1$ bis $C[h]$. $(C[-1]=0)$

```
CountingSort(X,n,K)
        for (h=0;h<K;h++) C[h]=0;
        for (i=1;i≤n;i++) C[ key(X[i]) ]++;
        for (h=1;h<K;h++) C[h]+=C[h-1];

        for (i=n;i≥1;i--) B[ C[ key(X[i]) ] ] = X[i]
                          C[ key(X[i]) ]--;
        return B[1..n];
```

Verwendet zusätzliche Felder C[0..k-1] fürs Zählen und B[1..n] für die Ausgabe.

Laufzeit: $O(K+n)$

Zusätzlicher Platzbedarf: $K+n$

CountingSort hat Laufzeit und Platzbedarf $\Theta(K+n)$. Unpraktikabel, wenn K sehr groß.

RadixSort:
Idee: Sei $K=B^d$. Betrachte jedes $h\in\{0,\ldots,K-1\}$ geschrieben als d-stellige Zahl zur Basis B. Sortiere $X[]$ wiederholt nach den Stellen in dieser Darstellung, und zwar nach aufsteigender Signifikanz der Stellen. Jede dieser Sortierungen muss ***stabil*** sein, d.h. die relative Ordnung zweier Stücke mit gleichem Schlüssel darf nicht geändert werden.

Für die jeweiligen Sortierungen kann CountingSort verwendet werden, denn diese Methode ist ***stabil***. Damit erzielt man

Laufzeit: $O(d\cdot(B+n))$

Zusätzlicher Platzbedarf: $B+n$

Bsp: B=10, d=3, n=7