SAARLAND UNIVERSITY
COMPUTER SCIENCE

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

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

SAARLAND UNIVERSITY
COMPUTER SCIENCE

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. $\{0,\dots,K-1\}$

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

Problem:
Sortiere n Stücke $x_1,\ldots,x_n$ nach $key(x_i)$, 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

349	823	613	328
718	613	718	349
618	718	618	529
823	618	823	613
328	328	328	618
529	349	529	718
613	529	349	823

Auswählen nach Rang (Selektion)

SAARLAND UNIVERSITY COMPUTER SCIENCE

Geg.: Folge X von n Schlüsseln, eine Zahl k mit $1 \leq k \leq n$
Ges.: ein k-kleinster Schlüssel von X, also den Schlüssel x_k für X sortiert als $x_1 \leq x_2 \leq \cdots \leq x_n$

trivial lösbar in Zeit $O(kn)$ (k mal Minimum Entfernen), oder auch in Zeit $O(n \cdot \log n)$ (Sortieren)

Ziel: $O(n)$ Zeit Algorithmus für beliebiges k (z.B. auch $k=n/2$, "*Median* von X")

Vereinfachende Annahme für das Folgende: alle Schlüssel in X sind verschieden, also für sortiertes X gilt $x_1 < x_2 < \cdots < x_n$

Übung: Adaptieren Sie die folgenden Algorithmen, sodass diese Annahme nicht notwendig ist und die asymptotischen Laufzeiten erhalten bleiben.

SAARLAND UNIVERSITY COMPUTER SCIENCE

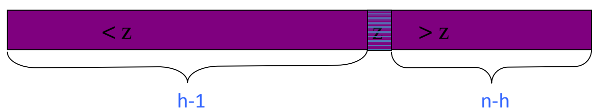
Geg.: Folge X von n Schlüsseln, eine Zahl k mit $1 \leq k \leq n$
Ges.: ein k-kleinster Schlüssel von X, also den Schlüssel x_k für X sortiert als $x_1 \leq x_2 \leq \cdots \leq x_n$

Idee: Dezimiere!

Wähle irgendein $z \in X$ und berechne $X_{<z} = \{x \in X \mid x<z\}$ und $X_{>z} = \{x \in X \mid x>z\}$

(z.B. durch Partitionsfunktion aus der letzten Vorlesung)

Es gilt dann $z = x_h$ mit $h-1 = |X_{<z}|$.



Fall $h=k$: $\Rightarrow$ z ist das gesuchte x_k

Fall $h>k$: $\Rightarrow$ x_k liegt in $X_{<z}$ und ist darin der k-kleinste Schlüssel ($X_{>z}$ ist irrelevant)

Fall $h<k$: $\Rightarrow$ x_k liegt in $X_{>z}$ und ist darin der $(k-h)$-kleinste Schlüssel ($X_{<z}$ ist irrelevant)

Also – x_k wird bei gegebenem z entweder sofort gefunden, oder man kann es rekursiv in $X_{<z}$ oder $X_{>z}$ finden. Welcher Fall für gewähltes z eintritt ist a priori nicht bekannt. Es wäre also günstig, wenn sowohl $X_{<z}$ als auch $X_{>z}$ "wenig" Schlüssel enthalten.

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Sei $½ < \alpha < 1$:
Wir nennen $z \in X$ einen **α-guten Splitter** für X, wenn sowohl $|X_{<z}| \leq \alpha|X|$ als auch $|X_{>z}| \leq \alpha|X|$ gilt.

Algorithmus zum Finden des k-kleinsten Schlüssel in X (bei festgelegtem α)

Select(X , k)
1. If $|X|$ klein (z.B. $|X| \leq 50$) then verwende eine triviale Methode.
2. Finde einen α-guten Splitter $z \in X$ für X
3. Berechne $X_{<z} = \{x \in X | x < z\}$ und $X_{>z} = \{x \in X | x > z\}$ und bestimme $h = |X_{<z}| + 1$.
4. If $h = k$ then return z
   else if $h > k$ then return **Select**($X_{<z}$, k)
   else (* $h < k$ *) return **Select**($X_{>z}$, $k-h$)

Laufzeitanalyse: $T(n)$ Laufzeit von **Select**(X , k), wobei $n = |X|$
$S_\alpha(n)$ (erwartete) Laufzeit um α-guten Splitter zu finden

1. $a \cdot n$ für eine Konstante a
2. $S_\alpha(n)$
3. $c \cdot n$ für eine Konstante c
4. $T(\alpha n)$

$$\begin{array}{ll} T(n) \leq a \cdot n & \text{wenn } n \leq 50 \\ T(n) \leq c \cdot n + S_\alpha(n) + T(\alpha n) & \text{wenn } n > 50 \end{array}$$

Wie findet man einen **α-guten Splitter** für X ?

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Methode 1: Randomisiert

Ziehe ein zufälliges Element z von X und bestimme die Größen von $|X_{<z}|$ und $|X_{>z}|$ und bestimme so, ob z ein α-guter Splitter ist. (Zeit $O(n)$)

Wiederhole dies, bis ein α-guter Splitter gefunden ist.

Die $(1-\alpha)n$ kleinsten Schlüssel in X sind keine α-guten Splitter, weil sonst $X_{>z}$ zu groß
Die $(1-\alpha)n$ größten Schlüssel in X sind keine α-guten Splitter, weil sonst $X_{<z}$ zu groß

Es gibt also $n - 2(1-\alpha)n = (2\alpha - 1)n = \beta n$ viele α-gute Splitter.

Chance, zufällig gezogenes z ein α-guter Splitter, ist β.

Die erwartete Anzahl von Wiederholungen, bis ein α-guter Splitter gefunden wird, ist also $1/\beta$.

Für die erwartete Laufzeit, um einen α-guten Splitter zu finden, gilt

$$S_\alpha(n) = (1/\beta) O(n) \leq b_\alpha \cdot n \text{ für irgendeine Konstante } b_\alpha.$$

Sei $\frac{1}{2} < \alpha < 1$:
Wir nennen $z \in X$ einen **α-guten Splitter** für X, wenn sowohl
$|X_{<z}| \leq \alpha |X|$ als auch $|X_{>z}| \leq \alpha |X|$ gilt.

Laufzeitanalyse: $T(n)$ Laufzeit von **Select**(X , k), wobei $n=|X|$
$S_\alpha(n)$ (erwartete) Laufzeit um α-guten Splitter zu finden

$$T(n) \leq a \cdot n \quad \text{wenn } n \leq 50$$
$$T(n) \leq c \cdot n + S_\alpha(n) + T(\alpha n) \quad \text{wenn } n > 50$$

Methode 1: $S_\alpha(n) \leq b_\alpha \cdot n$

$$T(n) \leq a \cdot n \quad \text{wenn } n \leq 50$$
$$T(n) \leq c \cdot n + b_\alpha \cdot n + T(\alpha n) = C_\alpha \cdot n + T(\alpha n) \quad \text{wenn } n > 50$$

$\Rightarrow$ $T(n) \leq B_\alpha \cdot n \,/\, (1-\alpha) = O(n)$ mit $B_\alpha = \max\{ a , C_\alpha \}$

mit Induktion

Auswahl nach Rang kann in $O(n)$ erwarteter Laufzeit gelöst werden.