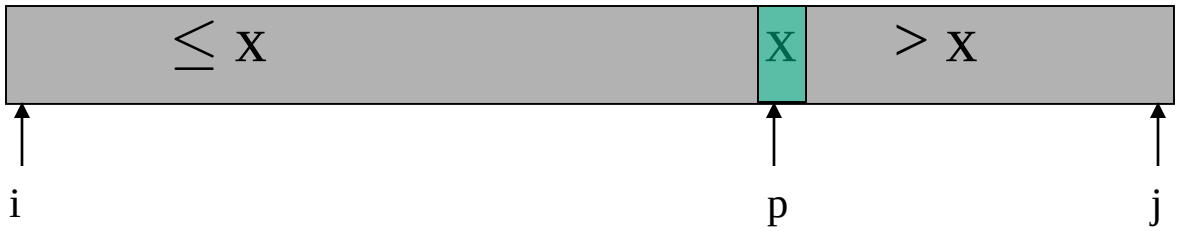
Quicksort

Wollen „schnellen" Sortieralgorithmus, der wenig Zusatzspeicher braucht (Mergesort benötigt ein Hilfsfeld für das „Mergen")

Idee: wähle „Pivotelement" x in Feld und stelle Feld so um:



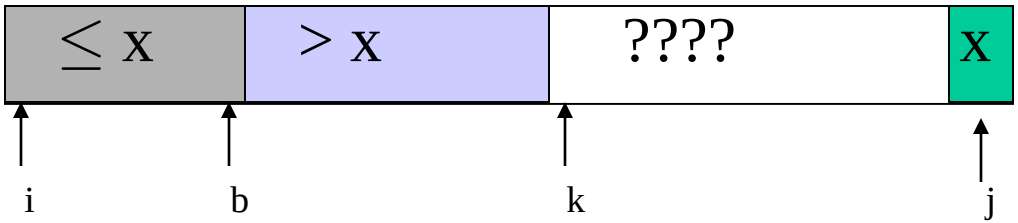
sortiere Teilfeld der „kleinen" Elemente ($\leq$ x) rekursiv

sortiere Teilfeld der „großen" Elemente (> x) rekursiv

```
Quicksort( A , i , j )
        if  i < j then     p = Partition( A , i , j , A[j] )
                           Quicksort( A , i , p-1 )
                           Quicksort( A , p+1 , j )
```

Partitionieren

Invariante:




```
Partition( A , i , j , x )
        b = i-1
        for k = i to j do  swap( A[k] , A[b+1] )
                           if  A[b+1] ≤ x then b++
        return b
```

Überlege Korrektheit des Rückgabewertes im letzten Schritt !

Es wird kein Hilfsfeld verwendet.

Laufzeit ist O(j – i + 1), also linear in der Feldgröße

Laufzeitanalyse von Quicksort

$T(n)$ Zeit, um ein Feld mit n Elementen zu sortieren

$T(n) = O(1)$ wenn $n \leq 2$

$T(n) \leq c \cdot n + T(n_1) + T(n_2)$ wenn $n>2$

dabei gilt $n_1 + n_2 = n-1$

Schlechtester Fall: $n_1=0$, $n_2=n-1$ $\Rightarrow$ $T(n) = \Theta(n^2)$

Bester Fall: $n_1 \approx n_2 \approx n/2$ $\Rightarrow T(n) = \Theta(n \log n)$

„Durchschnittlicher Fall" ?

Erwartete Laufzeit Analyse von Quicksort

Annahme:
Das gewählte Pivotelement ist mit gleicher Wahrscheinlichkeit 1/n das k-kleinste Element im Feld (k=1,...,n).
D.h. mit W'-keit 1/n gilt n_1=k-1 für k=1,...,n

T(n) ... erwartete Laufzeit von Quicksort

$$T(n) \leq c \cdot n + \sum_{0 \leq k < n} \left((1/n)(T(k) + T(n-1-k)) \right) \text{ wenn } n>2$$

$$= c \cdot n + (2/n) \sum_{0 \leq k < n} T(k)$$

$\Rightarrow$ $T(n) = O(n \log n)$

$$T(n) \leq c \cdot n + \sum_{0 \leq k < n} \left((1/n)(T(k) + T(n-1-k)) \right)$$

$$= c \cdot n + (2/n) \sum_{0 \leq k < n} T(k)$$

$$n \cdot T(n) = c \cdot n^2 + 2 \sum_{0 \leq k < n} T(k)$$

$$n \cdot T(n) - (n-1) \cdot T(n-1) =$$

$$= (c \cdot n^2 + 2 \sum_{0 \leq k < n} T(k)) - (c \cdot (n-1)^2 + 2 \sum_{0 \leq k < n-1} T(k))$$

$$= c \cdot (2n-1) + 2T(n-1)$$

$$\leq 2cn + 2T(n-1)$$

$$n \cdot T(n) - (n+1) \cdot T(n-1) \leq 2cn \quad \Rightarrow \quad T(n)/(n+1) - T(n-1)/n \leq 2c/(n+1)$$

$$\Rightarrow \quad T(n)/(n+1) \leq 2c \sum_{0 < k \leq n} 1/(k+1) \leq 2c(1 + \ln(n+1))$$

$$\Rightarrow \quad T(n) \leq 2c(n+1)(1 + \ln(n+1)) = O(n \cdot \log n)$$

Beachte: $\sum_{1 \leq k \leq n} 1/k \leq 1 + \int_1^n 1/x \, dx = 1 + \log n$

Erwartete Laufzeit Analyse von Quicksort

Annahme:
Das gewählte Pivotelement ist mit gleicher Wahrscheinlichkeit $1/n$ das k-kleinste Element im Feld ($k=1,...,n$).
D.h. mit W'-keit $1/n$ gilt $n_1=k-1$ für $k=1,...,n$

Wie kann man diese Annahme rechtfertigen?

Methode 1: Man nimmt einfach an, Eingaben kommen in "zufälliger" Reihenfolge (ohne wirkliche Rechtfertigung)

Methode 2: Man erzwingt die zufällige Reihenfolge aus Methode 1, indem man vor dem Sortieren das Feld zufällig permutiert. Man verlässt sich also auf die Zufälligkeit des Zufallszahlengenerators. (Randomisierter Algorithmus)

Erwartete Laufzeit Analyse von Quicksort

Annahme:
Das gewählte Pivotelement ist mit gleicher Wahrscheinlichkeit $1/n$ das k-kleinste Element im Feld ($k=1,...,n$).
D.h. mit W'-keit $1/n$ gilt $n_1=k-1$ für $k=1,...,n$

Wie kann man diese Annahme rechtfertigen?

Methode 3: Man erzwingt die Richtigkeit der Annahme, indem man ein zufälliges Element des Feldes als Pivot wählt.

(Zufallszahlengenerator, randomisierter Algorithmus)

Einschub: Münzwurfproblem

(geometrische Verteilung)

Münze mit Unfairness p:

Münzwurf ergibt
Wappen mit Wahrscheinlichkeit p und
Zahl mit Wahrscheinlichkeit 1-p

Wie oft erwarte ich die Münze zu werfen, bis *Wappen* erscheint?

Mit W'keit p erscheint Wappen beim ersten Wurf und das Experiment ist fertig.
Mit W'keit 1-p erscheint Wappen nicht, und ich erwarte zu dem einen schon getanen Wurf soviele Würfe, wie ich am Beginn des Experiments erwartete.

Also für E_p, die erwartete Anzal von Würfen gilt:

$E_p = p \cdot 1 + (1-p) \cdot (1+E_p)$ daher $E_p = 1/p$

Auswählen nach Rang (Selektion)

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.

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.