Quicksort: Effiziente Implementierung

- Array-Implementierung
- „inplace“
- 2-Wegevergleiche

Quicksort: Effiziente Implementierung

Procedure qSort(a : **Array of** Element; $\ell, r : \mathbb{N}$)
    if $\ell \geq r$ **then return**
    k:= pickPivotPos(a, ℓ, r)
    m:= partition(a, ℓ, r, k)
    qSort$(a, \ell, m-1)$
    qSort$(a, m+1, r)$

Quicksort: Effiziente Implementierung

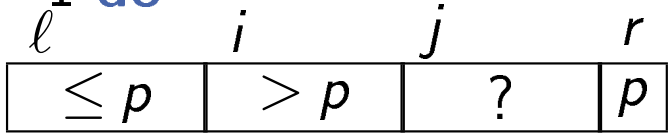
Function partition(a : **Array of** Element; $\ell, r, k : \mathbb{N}$)

$p := a[k]$ // pivot

swap($a[k], a[r]$)

$i := \ell$

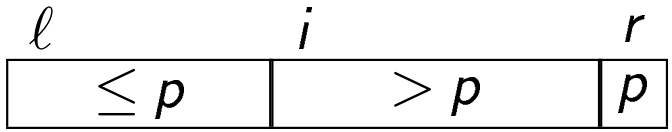
for $j := \ell$ **to** $r - 1$ **do**



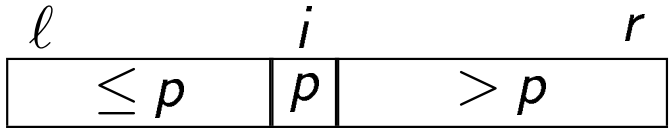
if $a[j] \leq p$ **then**

swap($a[i], a[j]$)

$i{+}{+}$



swap($a[i], a[r]$)



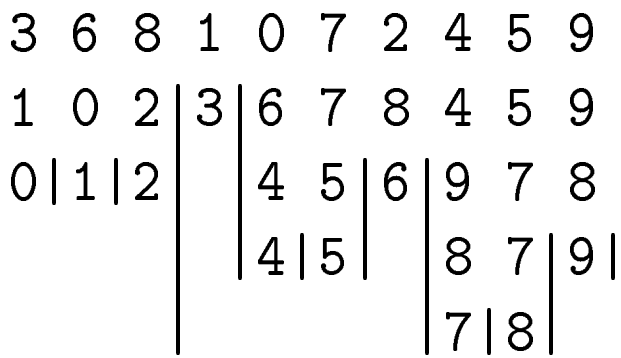
return i

Beispiel: Partitionierung, $k = 1$

p, $\overline{i}$, $\underline{j}$

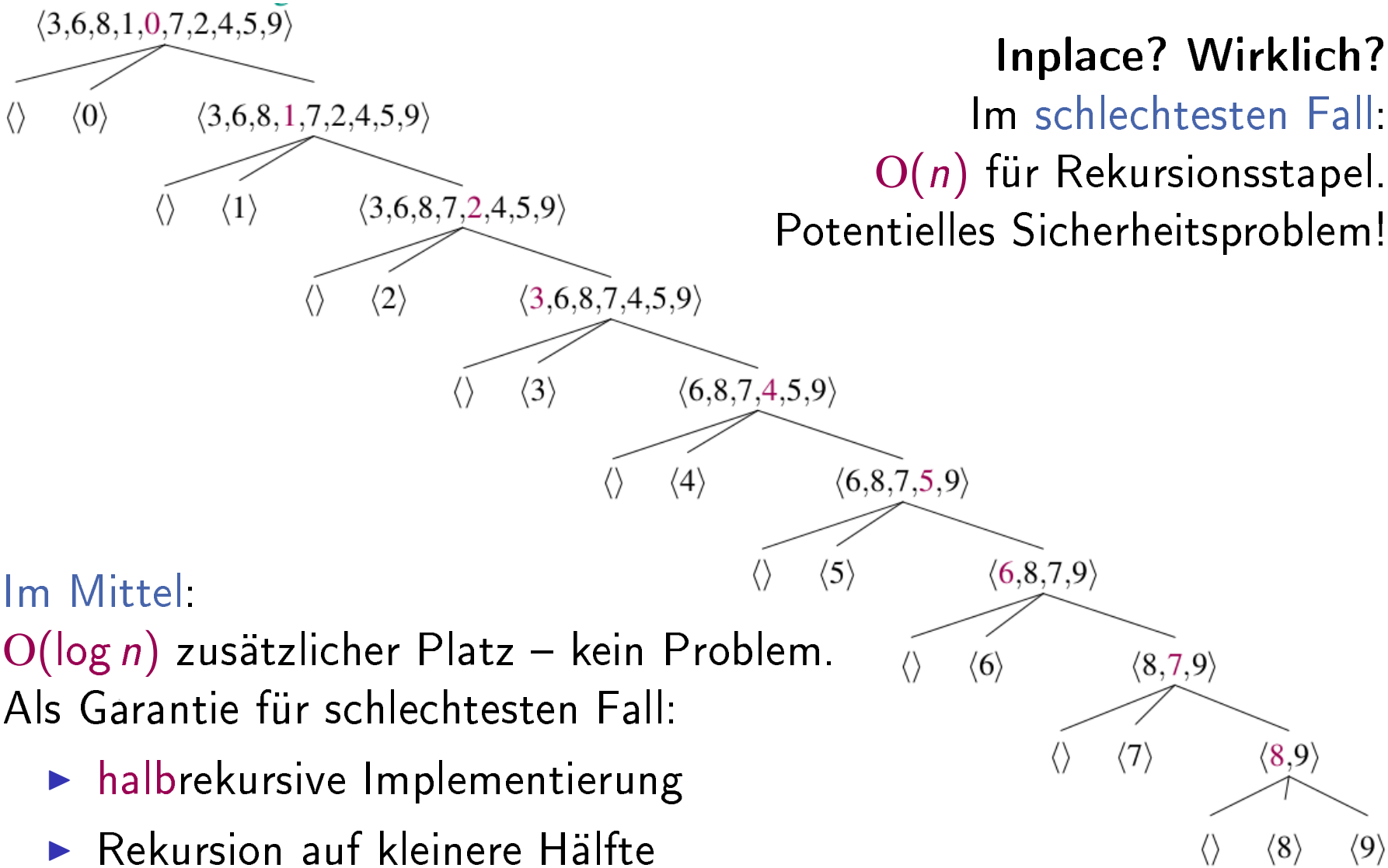
3	6	8	1	0	7	2	4	5	9
$\overline{9}$	6	8	1	0	7	2	4	5	3
$\overline{9}$	$\underline{6}$	8	1	0	7	2	4	5	3
$\overline{9}$	6	$\underline{8}$	1	0	7	2	4	5	3
$\overline{9}$	6	8	$\underline{1}$	0	7	2	4	5	3
1	$\overline{6}$	8	9	$\underline{0}$	7	2	4	5	3
1	0	$\overline{8}$	9	6	$\underline{7}$	2	4	5	3
1	0	$\overline{8}$	9	6	7	$\underline{2}$	4	5	3
1	0	2	$\overline{9}$	6	7	8	$\underline{4}$	5	3
1	0	2	$\overline{9}$	6	7	8	4	$\underline{5}$	3
1	0	2	$\overline{9}$	6	7	8	4	5	3
1	0	2	$\overline{3}$	6	7	8	4	5	9

Beispiel: Rekursion



Größerer Basisfall

Procedure qSort(a : **Array of** Element; $\ell, r : \mathbb{N}$)
    if $r - \ell + 1 \leq n_0$ **then**
        insertionSort($a[\ell..r]$)
    else
        k:= pickPivotPos(a, ℓ, r)
        m:= partition(a, ℓ, r, k)
        qSort($a, \ell, m-1$)
        qSort($a, m+1, r$)



Inplace? Wirklich?

Im schlechtesten Fall:
$O(n)$ für Rekursionsstapel.
Potentielles Sicherheitsproblem!

Im Mittel:
$O(\log n)$ zusätzlicher Platz – kein Problem.
Als Garantie für schlechtesten Fall:

- halbrekursive Implementierung
- Rekursion auf kleinere Hälfte

Halbrekursive Implementierung

Procedure qSort(a : **Array of** Element; $\ell, r : \mathbb{N}$)
  while $r - \ell + 1 > n_0$ **do**
    k:= pickPivotPos(a, ℓ, r)
    m:= partition(a, ℓ, r, k)
    if $m < (\ell + r)/2$ **then** qSort$(a, \ell, m-1)$; ℓ:= $m+1$
    else qSort$(a, m+1, r)$; r:= $m-1$
  insertionSort$(a[\ell..r])$

Halbrekursive Implementierung

Procedure qSort(a : **Array of** Element; $\ell, r : \mathbb{N}$)
  while $r - \ell + 1 > n_0$ **do**
    k:= pickPivotPos(a, ℓ, r)
    m:= partition(a, ℓ, r, k)
    if $m < (\ell + r)/2$ **then** qSort$(a, \ell, m-1)$; ℓ:= $m+1$
    else qSort$(a, m+1, r)$; r:= $m-1$
  insertionSort$(a[\ell..r])$

Satz: Rekursionstiefe $\leq \left\lceil \log \frac{n}{n_0} \right\rceil$

Beweisidee: Induktion. Teilproblemgröße halbiert sich (mindestens) mit jedem rekursiven Aufruf

Quadratische Komplexität bei gleichen Elementen?

- Variante aus dem Buch verwenden
- oder doch Drei-Wege-Partitionierung

Procedure qSortTernary(a : **Array of** Element; $\ell, r : \mathbb{N}$)

if $\ell \geq r$ **then return**

p:= $key(a[\text{pickPivotPos}(a, \ell, r)])$

(m, m'):= partitionTernary(a, ℓ, r, p)

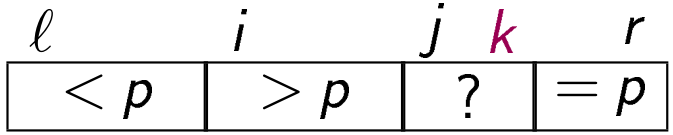
qSortTernary$(a, \ell, m-1)$

qSortTernary$(a, m'+1, r)$

Function partitionTernary(a : **Array of** Element; $\ell, r : \mathbb{N}; p : Key$)

$i := \ell, \quad j := \ell, \quad k := r$

invariant



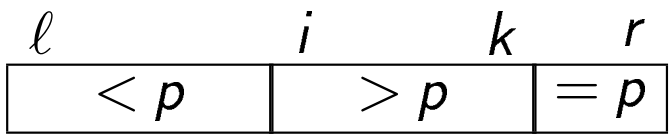
while $(j \leq k)$

if $a[j] = p$ **then** swap$(a[j], a[k])$, $k--$;

else if $a[j] < p$ **then** swap$(a[j], a[i])$, $i++$, $j++$;

else $j++$;

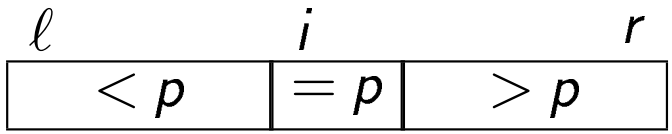
assert



if $r - k \leq k - i + 1$ **then** swap$(a[i..i + (r - k) - 1], a[k + 1..r])$

else swap$(a[i..k], a[r - (k - i)..r])$

assert



return $(i, i + (r - k) - 1)$

Vergleich Quicksort $\leftrightarrow$ Mergesort

Pro Mergesort

- $O(n \log n)$ Zeit (deterministisch)

  qsort: $\exists$ det. Varianten
- $n \log n + O(n)$ Elementvergleiche ($\approx$ untere Schranke)

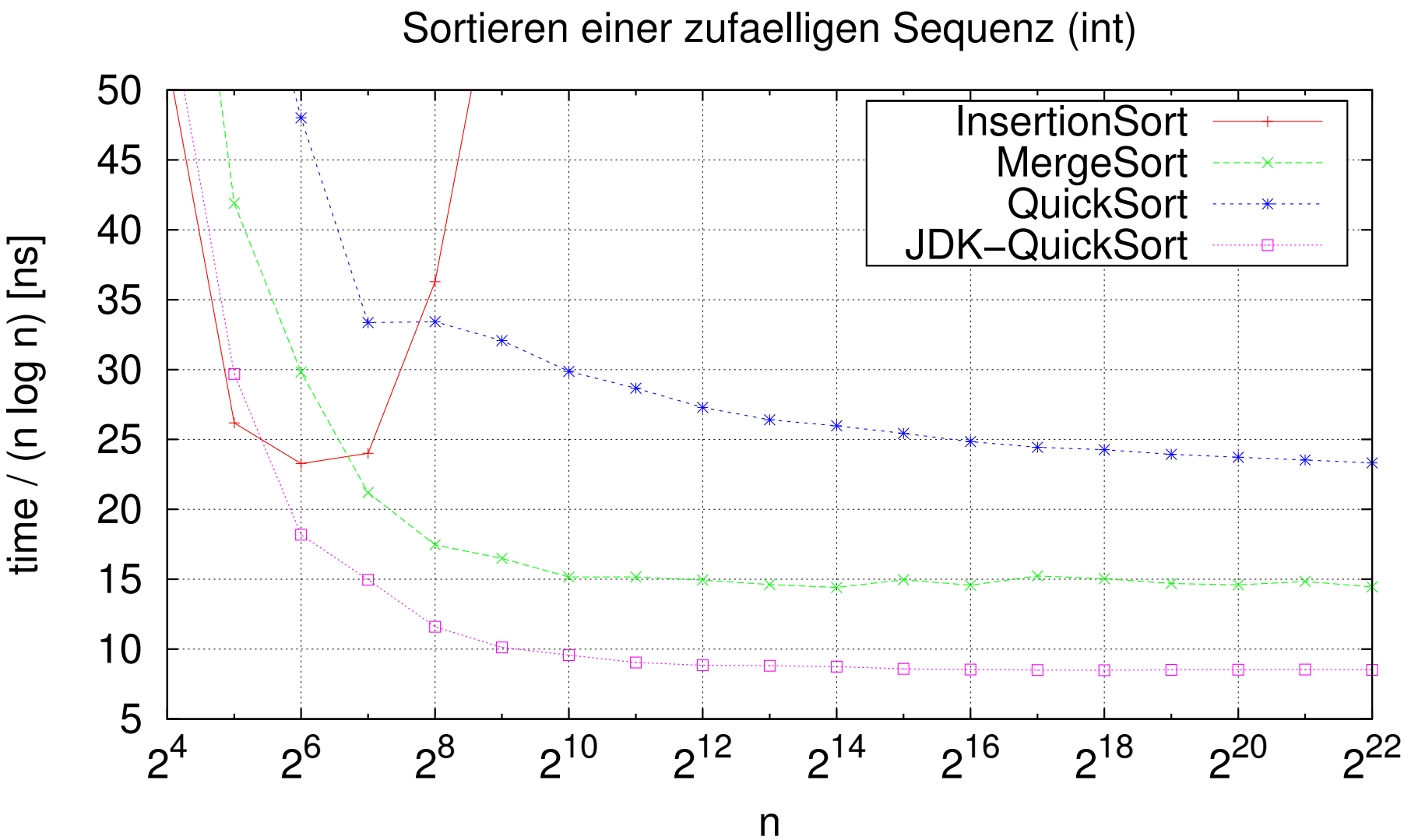
  qsort: möglich bei sorgfältiger Pivotwahl
- Stabil (gleiche Elemente behalten Reihenfolge bei)

  qsort: leicht bei Aufgabe der inplace-Eigenschaft

Pro Quicksort

- inplace
- Etwas schneller?

Benchmark


Sortieren einer zufaelligen Sequenz (int)
time / (n log n) [ns]
50
45
40
35
30
25
20
15
10
5
2^4
2^6
2^8
2^{10}
2^{12}
2^{14}
2^{16}
2^{18}
2^{20}
2^{22}
n
InsertionSort
MergeSort
QuickSort
JDK–QuickSort