6. Divide & Conquer – Quicksort

- Quicksort ist wie Merge-Sort ein auf dem Divide&Conquer-Prinzip beruhender Sortieralgorithmus.
- Von Quicksort existieren unterschiedliche Varianten, von denen einige in der Praxis besonders effizient sind.
- Die worst-case Laufzeit von Quicksort ist $\Theta(n^2)$.
- Die durchschnittliche Laufzeit ist jedoch $\Theta(n\log(n))$.
- Eine randomisierte Version von Quicksort besitzt erwartete Laufzeit $\Theta(n\log(n))$.

Quicksort - Idee

Eingabe: Ein zu sortierendes Teilarray $A[p \ldots r]$.

Teilungsschritt: Berechne einen Index $q, p \leq q \leq r$ und vertausche die Reihenfolge der Elemente in $A[p \ldots r]$, so dass die Elemente in $A[p \ldots q-1]$ nicht größer und die Elemente in $A[q+1 \ldots r]$ nicht kleiner sind als $A[q]$.

Eroberungsschritt: Sortiere rekursiv die beiden Teilarrays $A[p \ldots q-1]$ und $A[q+1 \ldots r]$.

Kombinationsschritt: Entfällt, da nach Eroberungsschritt das Array $A[p \ldots r]$ bereits sortiert ist.

Quicksort - Pseudocode

Quicksort(A,p,r)

1. **if** $p < r$
2. **then** $q \leftarrow$ Partition(A,p,r)
3. Quicksort$(A,p,q\text{-}1)$
4. Quicksort$(A,q+1,r)$

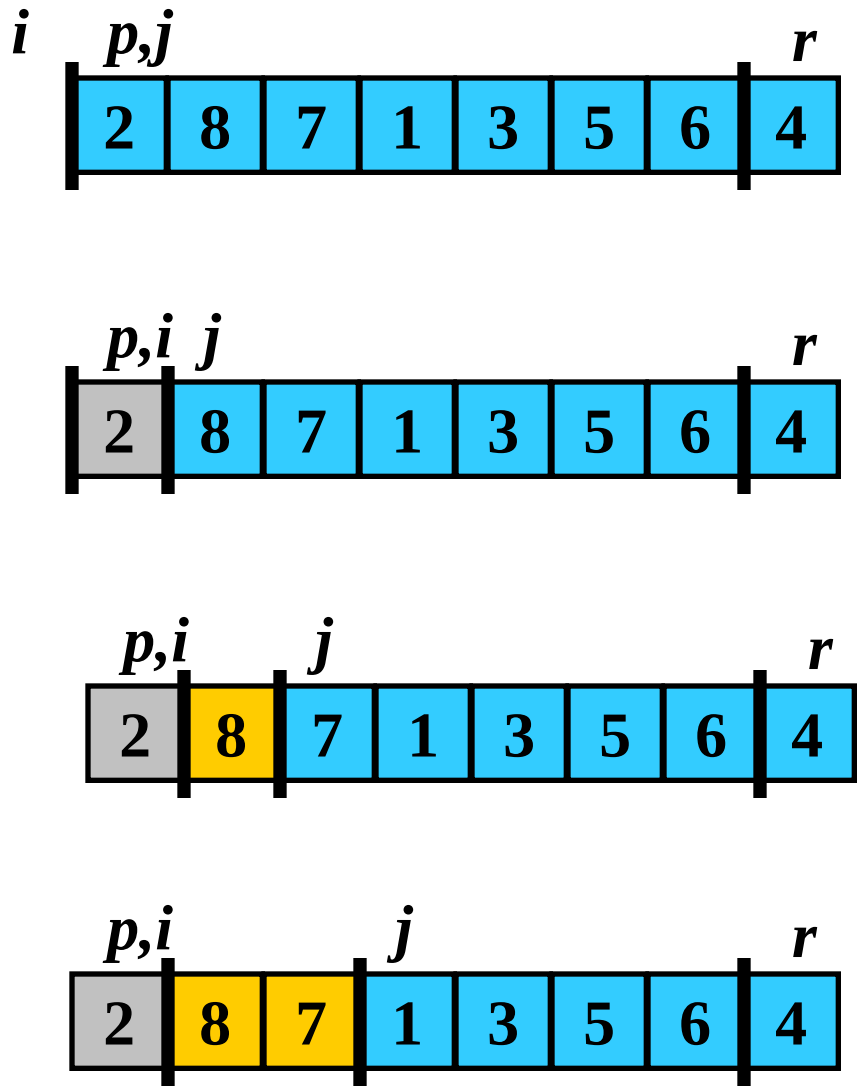
Aufruf, um Array A zu sortieren: Quicksort$(A,1,length[A])$

Partition - Pseudocode

Partition(A,p,r)

1. $x \leftarrow A[r]$
2. $i \leftarrow p$-1
3. **for** $j \leftarrow p$ **to** r-1
4. **do if** $A[j] \leq x$
5. **then** $i \leftarrow i+1$
6. $A[i] \leftrightarrow A[j]$
7. $A[i+1] \leftrightarrow A[r]$
8. **return** $i+1$

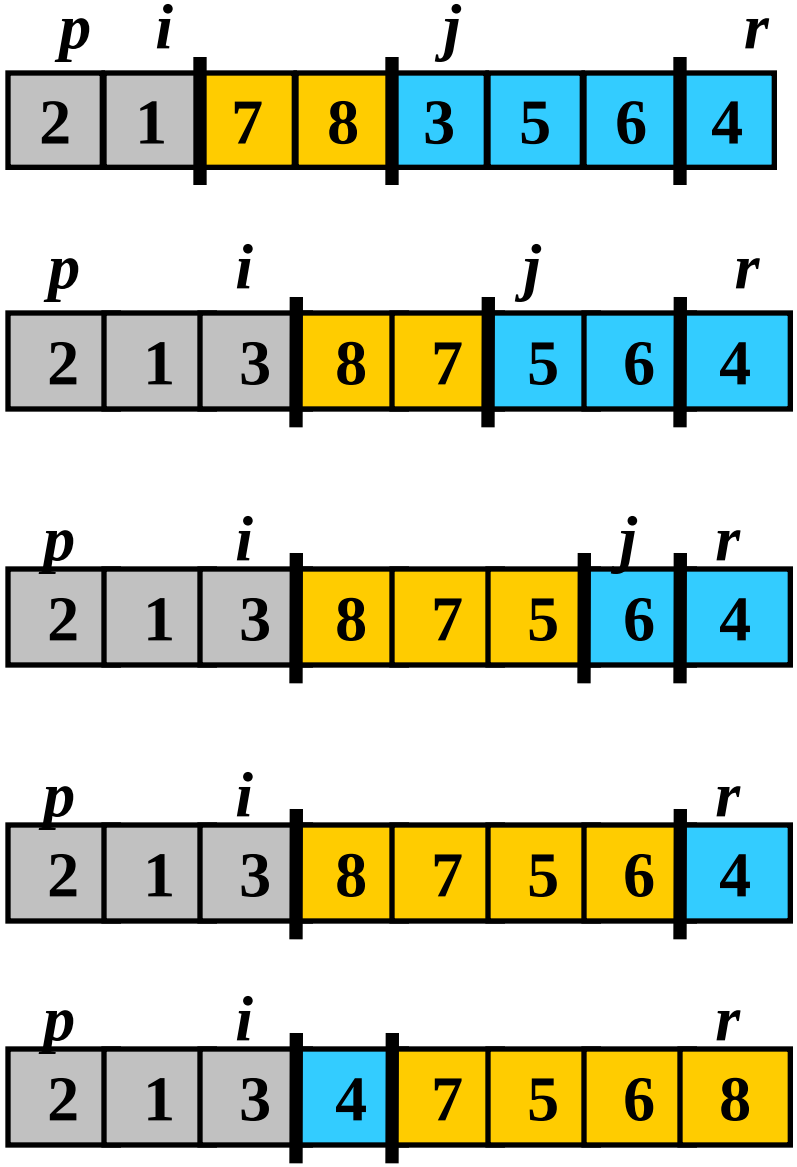
Illustration von Partition (1)



Partition(A,p,r)

1. $x \leftarrow A[r]$
2. $i \leftarrow p$-1
3. **for** $j \leftarrow p$ **to** r-1
4. **do if** $A[j] \leq x$
5. **then** $i \leftarrow i+1$
6. $A[i] \leftrightarrow A[j]$
7. $A[i+1] \leftrightarrow A[r]$
8. **return** $i+1$

Illustration von Partition (2)



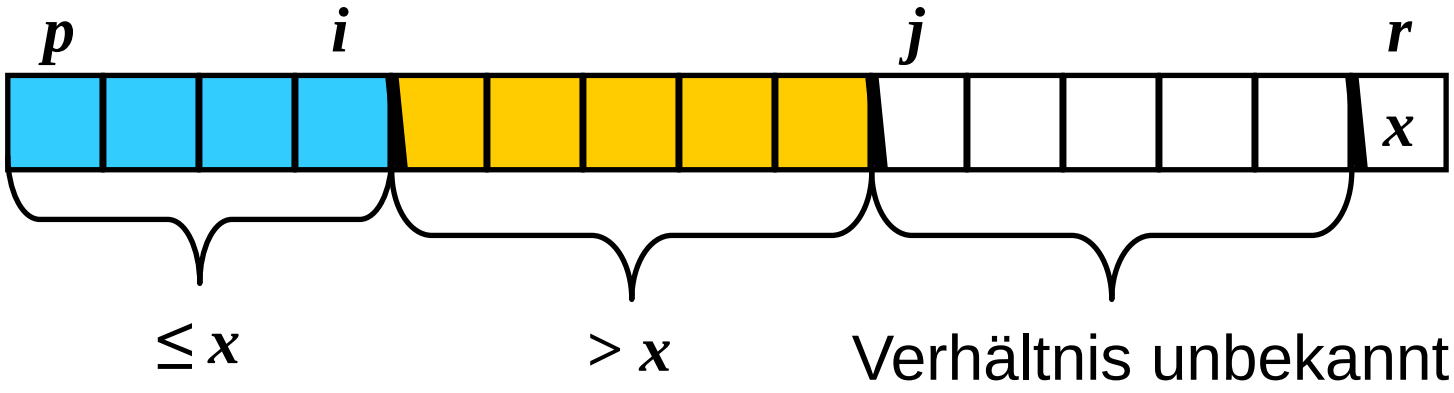
Partition(A,p,r)

1. $x \leftarrow A[r]$
2. $i \leftarrow p$-1
3. **for** $j \leftarrow p$ **to** r-1
4. **do if** $A[j] \leq x$
5. **then** $i \leftarrow i+1$
6. $A[i] \leftrightarrow A[j]$
7. $A[i+1] \leftrightarrow A[r]$
8. **return** $i+1$

Korrektheit von Partition - Invariante

Invariante I(i,j): Für alle $k \in \{p,\ldots,r\}$ gilt:

1. Falls $p \leq k \leq i$, dann ist $A[k] \leq x$
2. Falls $i+1 \leq k \leq j-1$, dann ist $A[k] > x$
3. Falls $k = r$, dann ist $A[k] = x$



Korrektheit von Partition (1)

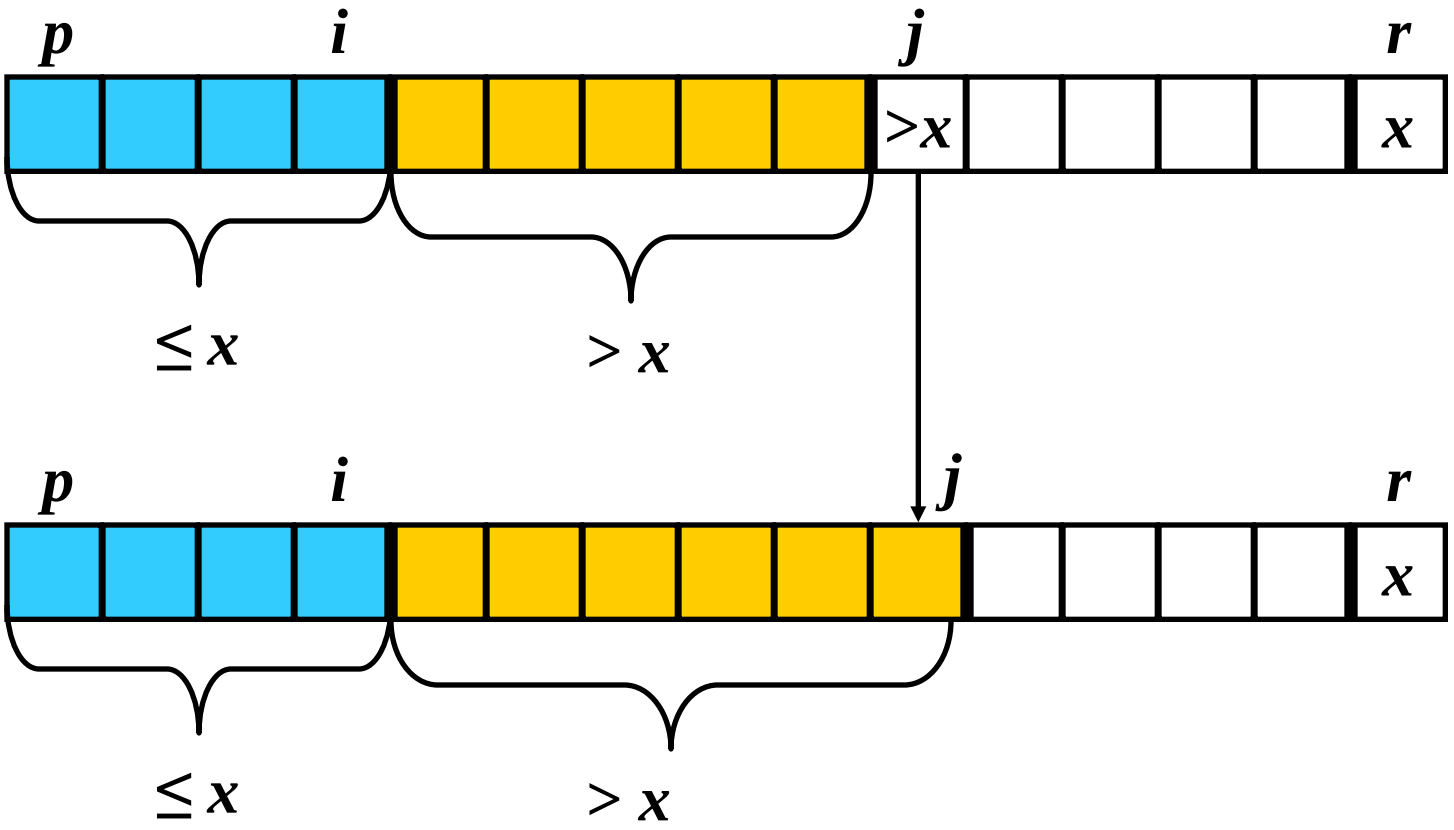
Initialisierung: Vor dem ersten Schleifendurchlauf gilt I(p-1,p), denn in diesem Fall sind die ersten beiden Bedingungen der Invariante leere Aussagen. Die 3. Bedingung gilt aufgrund von Zeile 1 des Quicksort Algorithmus. Also gilt auch I(i,j) am Anfang des ersten Schleifendurchlaufs.

Erhaltung: Wir unterscheiden zwei Fälle

1. $A[j] > x$
2. $A[j] \leq x$

1.Fall: Damit ist dann die 2. Bedingung auch für $k=j$ wahr, d.h. I(i,j+1) gilt.

Erhaltung – 1.Fall

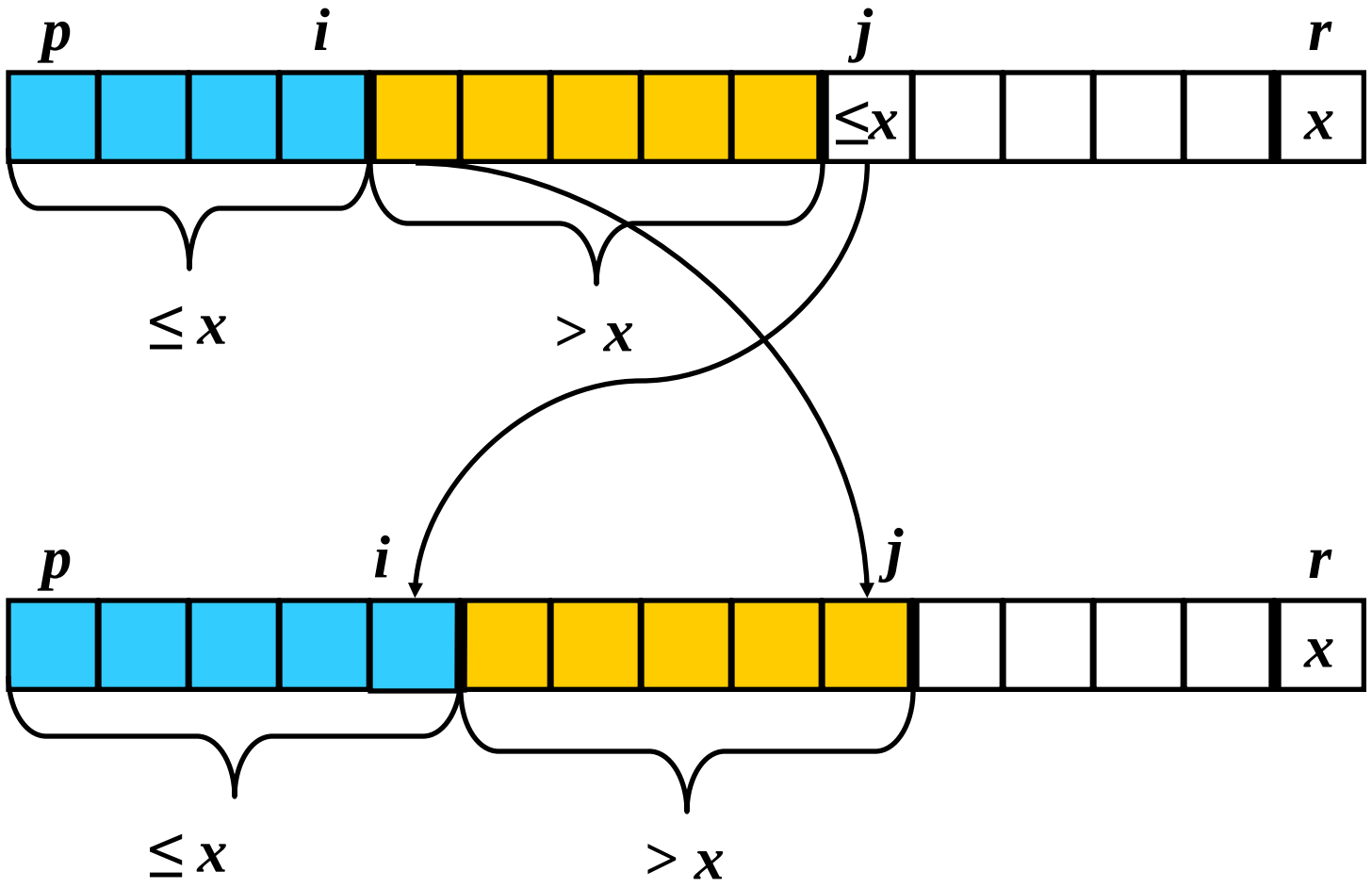


Korrektheit von Partition (2)

2.Fall ($A[j] \leq x$): In diesem Fall wird $A[i+1]$ (welches $> x$ ist) mit $A[j]$ (welches $\leq x$ ist) vertauscht. Da i auf $i+1$ gesetzt wird, gilt damit $I(i,j+1)$ nach Abschluss des then-Falls.

In jedem Fall gilt also am Ende der Schleife $I(i,j+1)$.

Erhaltung – 2.Fall



Korrektheit von Partition (3)

Terminierung: Am Ende der Schleife gilt $I(i,r)$, d.h.

1. für alle $p \leq k \leq i$ ist $A[k] \leq x$,
2. für alle $i+1 \leq k \leq r-1$ ist $A[k] > x$ und
3. für $k=r$ ist $A[k]=x$.

Nach der Vertauschung von $A[i+1]$ und $A[r]$ gilt daher:

1. Für alle $p \leq k \leq i$ ist $A[k] \leq x$,
2. für $k=i+1$ ist $A[k]=x$ und
3. für alle $i+2 \leq k \leq r$ ist $A[k] > x$.

Bei Rückgabe von $i+1$ wird daher die Zahlenfolge korrekt durch die rekursiven Aufrufe sortiert, wie wir noch sehen werden.

Laufzeit von Partition

Partition(A,p,r)

1. $x \leftarrow A[r]$
2. $i \leftarrow p$-1
3. **for** $j \leftarrow p$ **to** r-1
4. **do if** $A[j] \leq x$
5. **then** $i \leftarrow i+1$
6. $A[i] \leftrightarrow A[j]$
7. $A[i+1] \leftrightarrow A[r]$
8. **return** $i+1$

- Pro Zeile konstante Zeit.
- Schleife Zeilen 3-6 wird n=r-p-mal durchlaufen.

Satz 6.1: Partition hat Laufzeit $\Theta(n)$ bei Eingabe eines Teilarrays mit n Elementen.

Korrektheit von Quicksort

Quicksort(A,p,r)
1. If p<r then
2. q ← Partition(A,p,r)
3. Quicksort(A,p,q-1)
4. Quicksort(A,q+1,r)

Behauptung: Quicksort sortiert A[p,…,r]

Potenzialfunktion: $\phi(A,p,r) = r-p$

1. Bei jedem rek. Aufruf sinkt ϕ um mind. 1 auf Wert ≥ 0
2. Durch den Basisfall ist ϕ beschränkt durch 0 (r=p).
3. Da ϕ ganzzahlig ist, kann ϕ nur endlich viele Werte annehmen.

Korrektheit von Quicksort

Quicksort(A,p,r)
1. If p<r then
2. q ← Partition(A,p,r)
3. Quicksort(A,p,q-1)
4. Quicksort(A,q+1,r)

Initialisierung: für $p \geq r$ ist $A[p,\ldots,r]$ trivialerweise sortiert

Erhaltung (p<r): Wir nehmen an, dass
1. am Ende von Partition alle Werte in $A[p,\ldots,q-1] \leq A[q]$ und alle Werte in $A[q+1,\ldots,r] \geq A[q]$ sind (s. Folie 12),
2. Quicksort(A,p,q-1) das Feld $A[p,\ldots,q-1]$ und Qicksort(A,q+1,r) das Feld $A[q+1,\ldots,r]$ sortiert.

Dann folgt daraus, dass Quicksort(A,p,r) $A[p,\ldots,r]$ sortiert.

Laufzeit von Quicksort

Satz 6.2: Es gibt ein $c>0$, so dass für alle n und alle Eingaben der Größe n Quicksort mindestens Laufzeit $cn\log(n)$ besitzt.

Satz 6.3: Quicksort besitzt worst-case Laufzeit $\Theta(n^2)$.

Satz 6.4: Quicksort besitzt *average-case* Laufzeit $O(n\log(n))$.

Average-case Laufzeit: Betrachten alle Permutationen der n Eingabezahlen. Berechnen für jede Permutation Laufzeit von Quicksort bei dieser Permutation. Average-case Laufzeit ist dann der Durchschnitt über all diese Laufzeiten.

Laufzeit von Quicksort (2)

- Sei $Q_E(n)$ die erwartete Laufzeit von Quicksort für eine zufällige Permutation der Länge n, wobei alle Permutationen gleichwahrscheinlich sind.
- Die Zahl $A[n]$ ist die i kleinste Zahl mit Wahrscheinlichkeit $1/n$ für alle $i \in \{1, \ldots, n\}$.
- $Q_E(n) \leq \frac{1}{n} \sum_{i=1}^{n} (Q_E(i-1) + Q_E(n-i)) + cn$
- $\sum_{i=1}^{n} Q_E(i-1) = \sum_{k=0}^{n-1} Q_E(k) = \sum_{i=1}^{n} Q_E(n-i)$

Laufzeit von Quicksort (3)

Wir nehmen vereinfacht Gleichheit an (worst case).

- $Q_E(n) = \frac{2}{n} \sum_{k=0}^{n-1} Q_E(k) + cn$
- $nQ_E(n) = 2 \sum_{k=0}^{n-1} Q_E(k) + cn^2$
- $(n-1)Q_E(n-1) = 2 \sum_{k=0}^{n-2} Q_E(k) + c(n-1)^2$
- $nQ_E(n) - (n-1)Q_E(n-1) = 2Q_E(n-1) + c(2n-1)$
- $nQ_E(n) = (n+1)Q_E(n-1) + c(2n-1)$

Laufzeit von Quicksort (4)

- $\frac{Q_E(n)}{n+1} \leq \frac{Q_E(n-1)}{n} + c\frac{2n-1}{n(n+1)} \leq \frac{Q_E(n-1)}{n} + \frac{2c}{n}$
- $\frac{Q_E(n)}{n+1} \leq \frac{Q_E(n-2)}{n-1} + \frac{2c}{n} + \frac{2c}{n-1} \leq \cdots \leq \frac{Q_E(1)}{2} + 2c\left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\right)$
- Wir wissen $\Sigma_{i=2}^{n}\, 1/i \leq \ln(n)$. Also gilt

$$\frac{Q_E(n)}{n+1} \leq \frac{Q_E(1)}{2} + 2c\ln(n) \leq 2c(\ln(n)+1)$$

Randomisiertes Quicksort (1)

- Schlechte Eingaben für Quicksort können vermieden werden durch *Randomisierung,* d.h. der Algorithmus wirft gelegentlich eine Münze, um sein weiteres Vorgehen zu bestimmen.

- Worst-case Laufzeit bei ungünstigen Münzwürfen immer noch $\Theta(n^2)$.

- Es gibt keine schlechten Eingaben. Dies sind Eingaben, bei denen Quicksort bei allen Münzwürfen Laufzeit $\Theta(n^2)$ besitzt.

- Laufzeit ist in diesem Modell erwartete Laufzeit, wobei Erwartungswert über Münzwürfe genommen wird. Erwartete Laufzeit ist $\Theta(n\log(n))$.

Randomisiertes Quicksort (2)

Randomized-Partition(A,p,r)

1. $i \leftarrow \text{Random}(p,r)$
2. $A[r] \leftrightarrow A[i]$
3. **return** Partition(A,p,r)

Hierbei ist Random eine Funktion, die zufällig einen Wert aus $[p \ldots r]$ wählt. Dabei gilt für alle $i \in [p \ldots r]$:

$$\Pr(\text{Random}(p,r) = i) = \frac{1}{r-p+1}.$$

Randomisiertes Quicksort (3)

Randomized-Quicksort(A,p,r)

1. **if** $p < r$
2. **then** $q \leftarrow$ Randomized-Partition(A,p,r)
3. Randomized-Quicksort$(A,p,q\text{-}1)$
4. Randomized-Quicksort$(A,q+1,r)$

Satz 6.5: Die erwartete Laufzeit von Randomized-Quicksort ist $\Theta(n\log(n))$. Dabei ist der Erwartungswert über die Zufallsexperimente in Randomized-Partition genommen.

Median-Quicksort (1)

- Verbesserung der Güte von Aufteilungen, indem nicht ein festes Element zur Aufteilung benutzt wird, sondern z.B. das mittlere von drei Elementen Zur Aufteilung benutzt wird.

- Können etwa drei zufällige Elemente wählen oder $A[p], A[q], A[r]$ mit $q := \lfloor (p+r)/2 \rfloor$.

- Beide Varianten in der Praxis erfolgreich. Aber nur zufällige Variante kann gut analysiert werden: $\Theta(n\log(n))$ erwartete Laufzeit.

Median-Quicksort (2)

Median (A, i, j, k)

1. **if** $(A[i] \leq A[j]) \wedge (A[k] \leq A[i])$
2. **then return** i
3. **else if** $(A[i] \leq A[j]) \wedge (A[k] \geq A[j])$
4. **then return** j
5. **else return** k

Median - Partition(A,p,r)

1. $i \leftarrow \text{Median}(p, \lfloor (p+r)/2 \rfloor, r)$
2. $A[r] \leftrightarrow A[i]$
3. **return** Partition(A,p,r)

Median-Quicksort (3)

Median - Quicksort(A,p,r)

1. **if** $p < r$
2. **then** $q \leftarrow$ Median - Partition(A,p,r)
3. Median - Quicksort$(A,p,q\text{-}1)$
4. Median - Quicksort$(A,q+1,r)$

Changelog

29.04.16: Folien 12, 15