Algorithmen und Datenstrukturen

Quicksort

Dominik S. Kaaser

19. März 2014

Inhalt

Divide & Conquer

Definition

Divide&Conquer (Teile&Erobere) ist eine auf Rekursion beruhende Algorithmentechnik.

Ein Divide&Conquer-Algorithmus löst ein Problem in drei Schritten:

1. Teile ein Problem in mehrere Unterprobleme.
2. Erobere jedes gegebene Unterproblem durch rekursive Lösung. Probleme sind kleine Unterprobleme, diese werden direkt gelöst.
3. Kombiniere die Lösungen der Teilprobleme zu einer Gesamtlösung.

Divide & Conquer

Definition

Divide&Conquer (Teile&Erobere) ist eine auf Rekursion beruhende Algorithmentechnik.

Ein Divide&Conquer-Algorithmus löst ein Problem in drei Schritten:

- **Teile** ein Problem in mehrere Unterprobleme.
- **Erobere** jedes einzelne Unterproblem durch rekursive Lösung. Ausnahme sind kleine Unterprobleme, diese werden direkt gelöst.
- **Kombiniere** die Lösungen der Teilprobleme zu einer Gesamtlösung.

Divide & Conquer

Definition

Divide&Conquer (Teile&Erobere) ist eine auf Rekursion beruhende Algorithmentechnik.

Ein Divide&Conquer-Algorithmus löst ein Problem in drei Schritten:

- **Teile** ein Problem in mehrere Unterprobleme.
- **Erobere** jedes einzelne Unterproblem durch rekursive Lösung. Ausnahme sind kleine Unterprobleme, diese werden direkt gelöst.
- **Kombiniere** die Lösungen der Teilprobleme zu einer Gesamtlösung.

Divide & Conquer

Definition

Divide&Conquer (Teile&Erobere) ist eine auf Rekursion beruhende Algorithmentechnik.

Ein Divide&Conquer-Algorithmus löst ein Problem in drei Schritten:

- **Teile** ein Problem in mehrere Unterprobleme.
- **Erobere** jedes einzelne Unterproblem durch rekursive Lösung. Ausnahme sind kleine Unterprobleme, diese werden direkt gelöst.
- **Kombiniere** die Lösungen der Teilprobleme zu einer Gesamtlösung.

Divide & Conquer

Definition

Divide&Conquer (Teile&Erobere) ist eine auf Rekursion beruhende Algorithmentechnik.

Ein Divide&Conquer-Algorithmus löst ein Problem in drei Schritten:

- **Teile** ein Problem in mehrere Unterprobleme.
- **Erobere** jedes einzelne Unterproblem durch rekursive Lösung. Ausnahme sind kleine Unterprobleme, diese werden direkt gelöst.
- **Kombiniere** die Lösungen der Teilprobleme zu einer Gesamtlösung.

Average-case Laufzeit

- Betrachte alle Permutationen der n Eingabezahlen.
- Berechne für jede Permutation die Laufzeit des Algorithmus bei dieser Permutation.
- Die Average-case Laufzeit ist dann der Durchschnitt über all diese Laufzeiten.
- Die Average-case Laufzeit ist die *erwartete Laufzeit* einer zufällig und gleichverteilt gewählten Permutation aus der Menge aller Permutationen der n Eingabezahlen.

Average-case Laufzeit

- Betrachte alle Permutationen der n Eingabezahlen.
- Berechne für jede Permutation die Laufzeit des Algorithmus bei dieser Permutation.
- Die Average-case Laufzeit ist dann der Durchschnitt über all diese Laufzeiten.
- Die Average-case Laufzeit ist die *erwartete Laufzeit* einer zufällig und gleichverteilt gewählten Permutation aus der Menge aller Permutationen der n Eingabezahlen.

Average-case Laufzeit

- Betrachte alle Permutationen der n Eingabezahlen.
- Berechne für jede Permutation die Laufzeit des Algorithmus bei dieser Permutation.
- Die Average-case Laufzeit ist dann der Durchschnitt über all diese Laufzeiten.
- Die Average-case Laufzeit ist die *erwartete Laufzeit* einer zufällig und gleichverteilt gewählten Permutation aus der Menge aller Permutationen der n Eingabezahlen.

Average-case Laufzeit

- Betrachte alle Permutationen der n Eingabezahlen.
- Berechne für jede Permutation die Laufzeit des Algorithmus bei dieser Permutation.
- Die Average-case Laufzeit ist dann der Durchschnitt über all diese Laufzeiten.
- Die Average-case Laufzeit ist die *erwartete Laufzeit* einer zufällig und gleichverteilt gewählten Permutation aus der Menge aller Permutationen der n Eingabezahlen.

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 $\Theta\left(n^2\right)$.
- Die durchschnittliche Laufzeit ist jedoch $\Theta\left(n \log n\right)$.
- Eine randomisierte Version von Quicksort besitzt erwartete Laufzeit $\Theta\left(n \log n\right)$.

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 $\Theta\left(n^2\right)$.
- Die durchschnittliche Laufzeit ist jedoch $\Theta\left(n \log n\right)$.
- Eine randomisierte Version von Quicksort besitzt erwartete Laufzeit $\Theta\left(n \log n\right)$.

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 $\Theta\left(n^2\right)$.
- Die durchschnittliche Laufzeit ist jedoch $\Theta\left(n \log n\right)$.
- Eine randomisierte Version von Quicksort besitzt erwartete Laufzeit $\Theta\left(n \log n\right)$.

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 $\Theta\left(n^2\right)$.
- Die durchschnittliche Laufzeit ist jedoch $\Theta\left(n \log n\right)$.
- Eine randomisierte Version von Quicksort besitzt erwartete Laufzeit $\Theta\left(n \log n\right)$.

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 $\Theta\left(n^2\right)$.
- Die durchschnittliche Laufzeit ist jedoch $\Theta\left(n \log n\right)$.
- Eine randomisierte Version von Quicksort besitzt erwartete Laufzeit $\Theta\left(n \log n\right)$.

Quicksort

Eingabe

Ein zu sortierendes Teilarray $A[p \dots r]$.

Teilungsschritt

Berechne einen Index q mit $p \leq q \leq r$ und vertausche die Reihenfolge der Elemente in $A[p \dots r]$ so, dass

- die Elemente in $A[p \dots q-1]$ nicht größer sind als $A[q]$ und
- die Elemente in $A[q+1 \dots r]$ nicht kleiner sind als $A[q]$.

Eroberungsschritt

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

Kombinationsschritt

Entfällt, da nach Eroberungsschritt das Array bereits sortiert ist.

Quicksort

Eingabe

Ein zu sortierendes Teilarray $A[p \dots r]$.

Teilungsschritt

Berechne einen Index q mit $p \leq q \leq r$ und vertausche die Reihenfolge der Elemente in $A[p \dots r]$ so, dass

- die Elemente in $A[p \dots q-1]$ nicht größer sind als $A[q]$ und
- die Elemente in $A[q+1 \dots r]$ nicht kleiner sind als $A[q]$.

Eroberungsschritt

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

Kombinationsschritt

Entfällt, da nach Eroberungsschritt das Array bereits sortiert ist.

Quicksort

Eingabe

Ein zu sortierendes Teilarray $A[p \dots r]$.

Teilungsschritt

Berechne einen Index q mit $p \leq q \leq r$ und vertausche die Reihenfolge der Elemente in $A[p \dots r]$ so, dass

- die Elemente in $A[p \dots q-1]$ nicht größer sind als $A[q]$ und
- die Elemente in $A[q+1 \dots r]$ nicht kleiner sind als $A[q]$.

Eroberungsschritt

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

Kombinationsschritt

Entfällt, da nach Eroberungsschritt das Array bereits sortiert ist.

Quicksort

Eingabe

Ein zu sortierendes Teilarray $A[p \dots r]$.

Teilungsschritt

Berechne einen Index q mit $p \leq q \leq r$ und vertausche die Reihenfolge der Elemente in $A[p \dots r]$ so, dass

- die Elemente in $A[p \dots q-1]$ nicht größer sind als $A[q]$ und
- die Elemente in $A[q+1 \dots r]$ nicht kleiner sind als $A[q]$.

Eroberungsschritt

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

Kombinationsschritt

Entfällt, da nach Eroberungsschritt das Array bereits sortiert ist.

Pseudo-Code

```
1 Algorithm quicksort(A, p, r)
2     if p < r then
3         q ← partition(A, p, r)
4         quicksort(A, p, q − 1)
5         quicksort(A, q + 1, r)
```

- Die `partition`-Prozedur ordnet die Elemente *in place*.
- Aufruf, um Array A zu sortieren: `quicksort`$(A, 1, A.\texttt{length})$

Pseudo-Code

```
1 Algorithm quicksort(A, p, r)
2     if p < r then
3         q ← partition(A, p, r)
4         quicksort(A, p, q − 1)
5         quicksort(A, q + 1, r)
```

```
1 Algorithm partition(A, p, r)
2     x ← A[r]
3     i ← p − 1
4     for j ← p to r − 1 do
5         if A[j] ≤ x then
6             i ← i + 1
7             A[i] ↔ A[j]
8     A[i + 1] ↔ A[r]
9     return i + 1
```

- Die `partition`-Prozedur ordnet die Elemente *in place*.
- Aufruf, um Array A zu sortieren: `quicksort`$(A, 1, A.\texttt{length})$

Pseudo-Code

```
1 Algorithm quicksort(A, p, r)
2     if p < r then
3         q ← partition(A, p, r)
4         quicksort(A, p, q − 1)
5         quicksort(A, q + 1, r)
```

```
1 Algorithm partition(A, p, r)
2     x ← A[r]
3     i ← p − 1
4     for j ← p to r − 1 do
5         if A[j] ≤ x then
6             i ← i + 1
7             A[i] ↔ A[j]
8     A[i + 1] ↔ A[r]
9     return i + 1
```

- Die `partition`-Prozedur ordnet die Elemente *in place*.
- Aufruf, um Array A zu sortieren: `quicksort`$(A, 1, A.\texttt{length})$

Pseudo-Code

```
1 Algorithm quicksort(A, p, r)
2     if p < r then
3         q ← partition(A, p, r)
4         quicksort(A, p, q − 1)
5         quicksort(A, q + 1, r)
```

```
1 Algorithm partition(A, p, r)
2     x ← A[r]
3     i ← p − 1
4     for j ← p to r − 1 do
5         if A[j] ≤ x then
6             i ← i + 1
7             A[i] ↔ A[j]
8     A[i + 1] ↔ A[r]
9     return i + 1
```

- Die `partition`-Prozedur ordnet die Elemente *in place*.
- Aufruf, um Array A zu sortieren: `quicksort`$(A, 1, A.\texttt{length})$

Korrektheit

```
1 Algorithm partition(A, p, r)
2     x ← A[r]
3     i ← p − 1
4     for j ← p to r − 1 do
5         if A[j] ≤ x then
6             i ← i + 1
7             A[i] ↔ A[j]
8     A[i + 1] ↔ A[r]
9     return i + 1
```

- Das Element $x = A[r]$ wird als *Pivot*-Element verwendet.
- Der Algorithmus teilt das Array in vier (möglicherweise leere) Teilbereiche.
- Wir betrachten die Schleife in Zeilen 4 bis 7.

Korrektheit

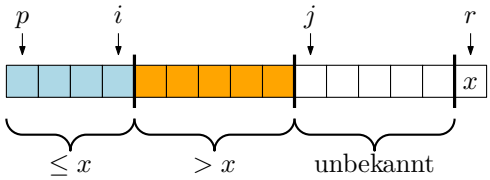
```
1 Algorithm partition(A, p, r)
2     x ← A[r]
3     i ← p − 1
4     for j ← p to r − 1 do
5         if A[j] ≤ x then
6             i ← i + 1
7             A[i] ↔ A[j]
8     A[i + 1] ↔ A[r]
9     return i + 1
```

- Das Element $x = A[r]$ wird als *Pivot*-Element verwendet.
- Der Algorithmus teilt das Array in vier (möglicherweise leere) Teilbereiche.
- Wir betrachten die Schleife in Zeilen 4 bis 7.

Korrektheit

```
1 Algorithm partition(A, p, r)
2     x ← A[r]
3     i ← p − 1
4     for j ← p to r − 1 do
5         if A[j] ≤ x then
6             i ← i + 1
7             A[i] ↔ A[j]
8     A[i + 1] ↔ A[r]
9     return i + 1
```

- Das Element $x = A[r]$ wird als *Pivot*-Element verwendet.
- Der Algorithmus teilt das Array in vier (möglicherweise leere) Teilbereiche.
- Wir betrachten die Schleife in Zeilen 4 bis 7.



Korrektheit

```
1 Algorithm partition(A, p, r)
2     x ← A[r]
3     i ← p − 1
4     for j ← p to r − 1 do
5         if A[j] ≤ x then
6             i ← i + 1
7             A[i] ↔ A[j]
8     A[i + 1] ↔ A[r]
9     return i + 1
```

- Das Element $x = A[r]$ wird als *Pivot*-Element verwendet.
- Der Algorithmus teilt das Array in vier (möglicherweise leere) Teilbereiche.
- Wir betrachten die Schleife in Zeilen 4 bis 7.

Korrektheit

```
1 Algorithm partition(A, p, r)
2     x ← A[r]
3     i ← p − 1
4     for j ← p to r − 1 do
5         if A[j] ≤ x then
6             i ← i + 1
7             A[i] ↔ A[j]
8     A[i + 1] ↔ A[r]
9     return i + 1
```

Schleifeninvariante

Vor Durchlauf der Schleife in Zeilen 4 – 7 mit Index j gilt für jeden Index k:

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 $A[k] = x$.

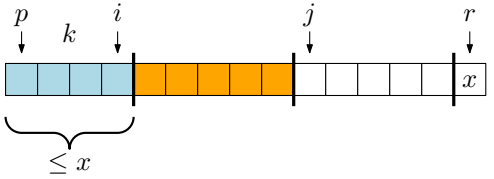
Korrektheit

```
1 Algorithm partition(A, p, r)
2     x ← A[r]
3     i ← p − 1
4     for j ← p to r − 1 do
5         if A[j] ≤ x then
6             i ← i + 1
7             A[i] ↔ A[j]
8     A[i + 1] ↔ A[r]
9     return i + 1
```

Schleifeninvariante

Vor Durchlauf der Schleife in Zeilen $4-7$ mit Index j gilt für jeden Index k:

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 $A[k] = x$.



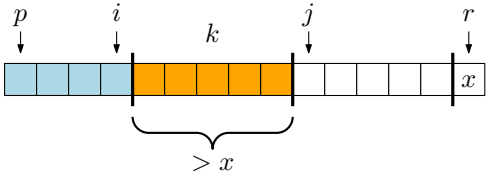
Korrektheit

```
1 Algorithm partition(A, p, r)
2     x ← A[r]
3     i ← p − 1
4     for j ← p to r − 1 do
5         if A[j] ≤ x then
6             i ← i + 1
7             A[i] ↔ A[j]
8     A[i + 1] ↔ A[r]
9     return i + 1
```

Schleifeninvariante

Vor Durchlauf der Schleife in Zeilen $4-7$ mit Index j gilt für jeden Index k:

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 $A[k] = x$.



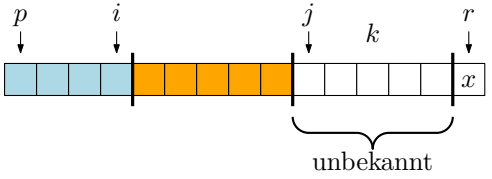
Korrektheit

```
1 Algorithm partition(A, p, r)
2     x ← A[r]
3     i ← p − 1
4     for j ← p to r − 1 do
5         if A[j] ≤ x then
6             i ← i + 1
7             A[i] ↔ A[j]
8     A[i + 1] ↔ A[r]
9     return i + 1
```

Schleifeninvariante

Vor Durchlauf der Schleife in Zeilen $4-7$ mit Index j gilt für jeden Index k:

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 $A[k] = x$.



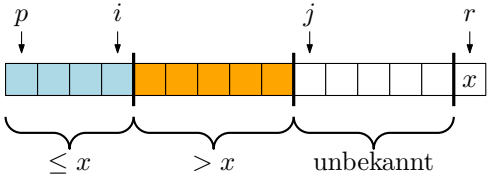
Korrektheit

```
1 Algorithm partition(A, p, r)
2     x ← A[r]
3     i ← p − 1
4     for j ← p to r − 1 do
5         if A[j] ≤ x then
6             i ← i + 1
7             A[i] ↔ A[j]
8     A[i + 1] ↔ A[r]
9     return i + 1
```

Schleifeninvariante

Vor Durchlauf der Schleife in Zeilen $4-7$ mit Index j gilt für jeden Index k:

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

Initialisierung

Vor dem ersten Schleifendurchlauf gilt $i = p - 1$ und $j = p$. Daher gibt es in diesem Fall keine Indizes zwischen p und i (1. Bedingung) bzw. zwischen $i + 1$ und $j - 1$ (2. Bedingung). Die erste Zeile sorgt dafür, dass die dritte Bedingung ebenfalls erfüllt ist.

Korrektheit

Initialisierung

Vor dem ersten Schleifendurchlauf gilt $i = p - 1$ und $j = p$. Daher gibt es in diesem Fall keine Indizes zwischen p und i (1. Bedingung) bzw. zwischen $i + 1$ und $j - 1$ (2. Bedingung). Die erste Zeile sorgt dafür, dass die dritte Bedingung ebenfalls erfüllt ist.

Erhaltung

Wir unterscheiden zwei Fälle:

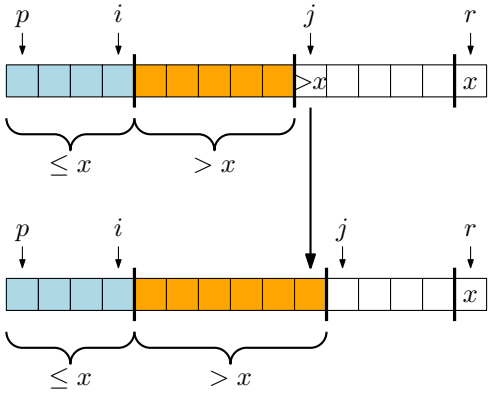
1. $A[j] > x$
2. $A[j] \leq x$

Fall 1: $A[j] > x$

Erhaltung, Fall 1

$$A[j] > x$$

```
4 for j ← p to r − 1 do
5     if A[j] ≤ x then
6         i ← i + 1
7         A[i] ↔ A[j]
```



Korrektheit

Initialisierung

Vor dem ersten Schleifendurchlauf gilt $i = p - 1$ und $j = p$. Daher gibt es in diesem Fall keine Indizes zwischen p und i (1. Bedingung) bzw. zwischen $i + 1$ und $j - 1$ (2. Bedingung). Die erste Zeile sorgt dafür, dass die dritte Bedingung ebenfalls erfüllt ist.

Erhaltung

Wir unterscheiden zwei Fälle:

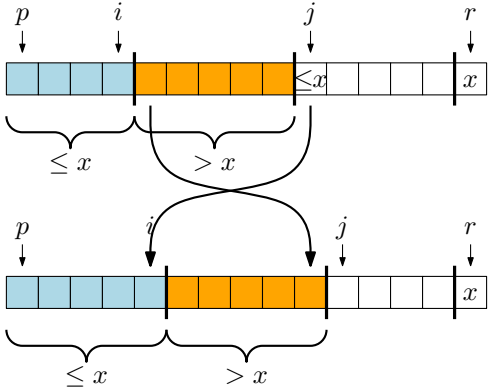
1. $A[j] > x$
2. $A[j] \leq x$

Fall 2: $A[j] \leq x$

Erhaltung, Fall 2

$$A[j] \leq x$$

```
4 for j ← p to r − 1 do
5     if A[j] ≤ x then
6         i ← i + 1
7         A[i] ↔ A[j]
```



Korrektheit

Initialisierung

Vor dem ersten Schleifendurchlauf gilt $i = p - 1$ und $j = p$. Daher gibt es in diesem Fall keine Indizes zwischen p und i (1. Bedingung) bzw. zwischen $i + 1$ und $j - 1$ (2. Bedingung). Die erste Zeile sorgt dafür, dass die dritte Bedingung ebenfalls erfüllt ist.

Erhaltung

Wir unterscheiden zwei Fälle:

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

Terminierung

Es gilt $j = r$ und alle Elemente des Arrays wurden mit x verglichen. Zeile 8 stellt nun sicher, dass x zwischen die Elemente kleiner gleich x und die Elemente größer als x plaziert wird. Damit genügt die von Partition berechnet Aufteilung immer den Anforderungen von Quicksort.

Laufzeitanalyse

```
1 Algorithm partition(A, p, r)
2     x ← A[r]
3     i ← p − 1
4     for j ← p to r − 1 do
5         if A[j] ≤ x then
6             i ← i + 1
7             A[i] ↔ A[j]
8     A[i + 1] ↔ A[r]
9     return i + 1
```

- Pro Zeile konstante Zeit.
- Die Schleife in Zeilen 4 – 7 wird $n = (r - p)$ mal durchlaufen.

Laufzeitanalyse

```
1 Algorithm partition(A, p, r)
2     x ← A[r]
3     i ← p − 1
4     for j ← p to r − 1 do
5         if A[j] ≤ x then
6             i ← i + 1
7             A[i] ↔ A[j]
8     A[i + 1] ↔ A[r]
9     return i + 1
```

- Pro Zeile konstante Zeit.
- Die Schleife in Zeilen 4 – 7 wird $n = (r - p)$ mal durchlaufen.

Laufzeitanalyse

```
1 Algorithm partition(A, p, r)
2     x ← A[r]
3     i ← p − 1
4     for j ← p to r − 1 do
5         if A[j] ≤ x then
6             i ← i + 1
7             A[i] ↔ A[j]
8     A[i + 1] ↔ A[r]
9     return i + 1
```

- Pro Zeile konstante Zeit.
- Die Schleife in Zeilen 4 – 7 wird $n = (r - p)$ mal durchlaufen.

Laufzeitanalyse

```
1 Algorithm partition(A, p, r)
2     x ← A[r]
3     i ← p − 1
4     for j ← p to r − 1 do
5         if A[j] ≤ x then
6             i ← i + 1
7             A[i] ↔ A[j]
8     A[i + 1] ↔ A[r]
9     return i + 1
```

- Pro Zeile konstante Zeit.
- Die Schleife in Zeilen 4 – 7 wird $n = (r - p)$ mal durchlaufen.

Satz

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

Laufzeiten

Satz

Es gibt ein $c > 0$, sodass für alle n und alle Eingaben der Größe n Quicksort mindestens Laufzeit $c \cdot n \cdot \log n$ besitzt.

Laufzeiten

Satz

Es gibt ein $c > 0$, sodass für alle n und alle Eingaben der Größe n Quicksort mindestens Laufzeit $c \cdot n \cdot \log n$ besitzt.

Satz

Quicksort besitzt worst-case Laufzeit $\Theta\left(n^2\right)$.

Laufzeiten

Satz

Es gibt ein $c > 0$, sodass für alle n und alle Eingaben der Größe n Quicksort mindestens Laufzeit $c \cdot n \cdot \log n$ besitzt.

Satz

Quicksort besitzt worst-case Laufzeit $\Theta\left(n^2\right)$.

Satz

Quicksort besitzt average-case Laufzeit $\mathrm{O}\left(n \log n\right)$.

Balanzierte Partitionierung

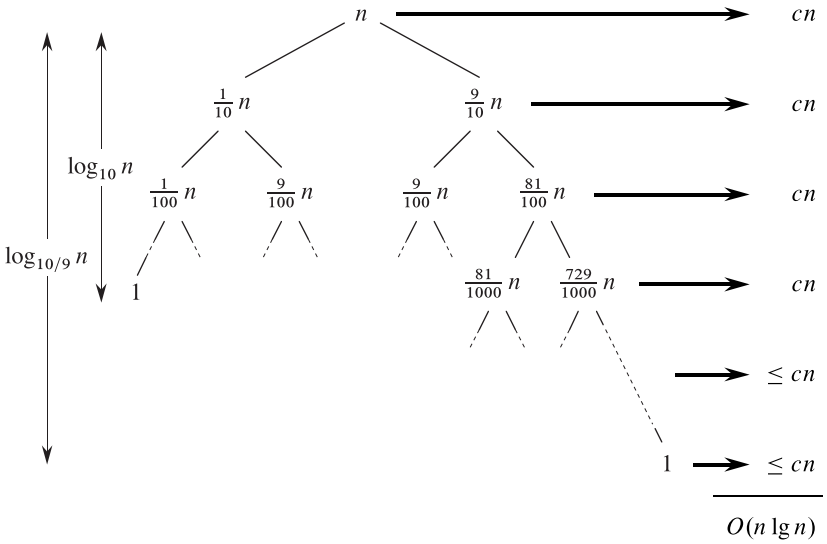


Image source: Cormen et al., Introduction to Algorithms, 3rd Ed.

Überlegung zum Average-case

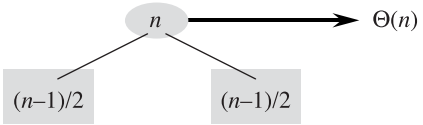
- Der Großteil der Partitionierungen (ca. 80%) ist besser balanziert als $9 : 1$.
- Annahme: Gute und schlechte Aufteilungen wechseln sich im Baum ab.
- Der Aufwand für eine schlechte und anschließend eine gute Partitionierung ist (asymptotisch) nicht größer als für eine gute Partitionierung.
- Wenn also immer gute und schlechte Partitionierungen sich abwechseln, ist die gesamte Laufzeit trotzdem $O(n \log n)$!

Überlegung zum Average-case

- Der Großteil der Partitionierungen (ca. 80%) ist besser balanziert als $9:1$.
- Annahme: Gute und schlechte Aufteilungen wechseln sich im Baum ab.
- Der Aufwand für eine schlechte und anschließend eine gute Partitionierung ist (asymptotisch) nicht größer als für eine gute Partitionierung.
- Wenn also immer gute und schlechte Partitionierungen sich abwechseln, ist die gesamte Laufzeit trotzdem $O(n \log n)$!

Überlegung zum Average-case

- Der Großteil der Partitionierungen (ca. 80%) ist besser balanziert als $9:1$.
- Annahme: Gute und schlechte Aufteilungen wechseln sich im Baum ab.

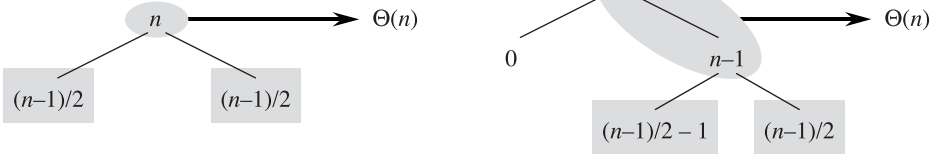


- Der Aufwand für eine schlechte und anschließend eine gute Partitionierung ist (asymptotisch) nicht größer als für eine gute Partitionierung.
- Wenn also immer gute und schlechte Partitionierungen sich abwechseln, ist die gesamte Laufzeit trotzdem $\mathrm{O}(n \log n)$!

Image source: Cormen et al., Introduction to Algorithms, 3rd Ed.

Überlegung zum Average-case

- Der Großteil der Partitionierungen (ca. 80%) ist besser balanziert als $9:1$.
- Annahme: Gute und schlechte Aufteilungen wechseln sich im Baum ab.

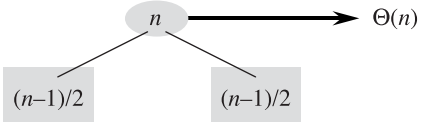


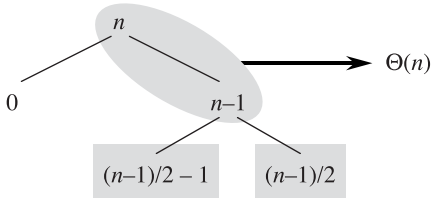
- Der Aufwand für eine schlechte und anschließend eine gute Partitionierung ist (asymptotisch) nicht größer als für eine gute Partitionierung.
- Wenn also immer gute und schlechte Partitionierungen sich abwechseln, ist die gesamte Laufzeit trotzdem $\mathrm{O}(n \log n)$!

Image source: Cormen et al., Introduction to Algorithms, 3rd Ed.

Überlegung zum Average-case

- Der Großteil der Partitionierungen (ca. 80%) ist besser balanziert als $9:1$.
- Annahme: Gute und schlechte Aufteilungen wechseln sich im Baum ab.



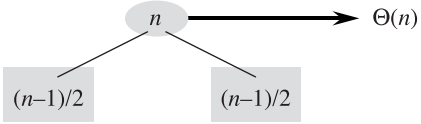


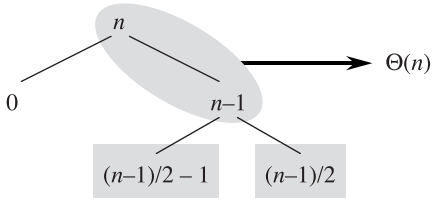
- Der Aufwand für eine schlechte und anschließend eine gute Partitionierung ist (asymptotisch) nicht größer als für eine gute Partitionierung.
- Wenn also immer gute und schlechte Partitionierungen sich abwechseln, ist die gesamte Laufzeit trotzdem $\mathrm{O}(n \log n)$!

Image source: Cormen et al., Introduction to Algorithms, 3rd Ed.

Überlegung zum Average-case

- Der Großteil der Partitionierungen (ca. 80%) ist besser balanziert als $9:1$.
- Annahme: Gute und schlechte Aufteilungen wechseln sich im Baum ab.





- Der Aufwand für eine schlechte und anschließend eine gute Partitionierung ist (asymptotisch) nicht größer als für eine gute Partitionierung.
- Wenn also immer gute und schlechte Partitionierungen sich abwechseln, ist die gesamte Laufzeit trotzdem $\mathrm{O}(n \log n)$!

Image source: Cormen et al., Introduction to Algorithms, 3rd Ed.

Randomisierung

- Schlechte Eingaben für Quicksort können durch Randomisierung vermieden werden, d.h. der Algorithmus wirft gelegentlich eine Münze, um sein weiteres Vorgehen zu bestimmen.
- Worst-case Laufzeit bei ungünstigen Münzwürfen immer noch $\Theta\left(n^2\right)$.
- Es gibt keine schlechten Eingaben, bei denen Quicksort bei allen Münzwürfen Laufzeit $\Theta\left(n^2\right)$ besitzt.
- Analyse der erwarteten Laufzeit, wobei Erwartungswert über Münzwürfe genommen wird. Die erwartete Laufzeit ist $\Theta\left(n \log n\right)$.

Randomisierung

```
1 Algorithm randomized_quicksort(A, p, r)
2     if p < r then
3         q ← randomized_partition(A, p, r)
4         randomized_quicksort(A, p, q − 1)
5         randomized_quicksort(A, q + 1, r)
```

- Hierbei ist $\texttt{random}(p, r)$ eine Funktion, die zufällig einen Wert aus $[p \dots r]$ wählt. Dabei gilt für alle $i \in [p \dots r]$

$$\Pr[\texttt{random}(p, r) = i] = \frac{1}{r - p + 1} .$$

Randomisierung

```
1 Algorithm randomized_quicksort(A, p, r)
2     if p < r then
3         q ← randomized_partition(A, p, r)
4         randomized_quicksort(A, p, q − 1)
5         randomized_quicksort(A, q + 1, r)
```

```
1 Algorithm randomized_partition(A, p, r)
2     i ← random(p, r)
3     A[r] ↔ A[i]
4     return partition(A, p, r)
```

- Hierbei ist $\texttt{random}(p, r)$ eine Funktion, die zufällig einen Wert aus $[p \dots r]$ wählt. Dabei gilt für alle $i \in [p \dots r]$

$$\Pr[\texttt{random}(p, r) = i] = \frac{1}{r - p + 1} .$$

Randomisierung

```
1 Algorithm randomized_quicksort(A, p, r)
2     if p < r then
3         q ← randomized_partition(A, p, r)
4         randomized_quicksort(A, p, q − 1)
5         randomized_quicksort(A, q + 1, r)
```

```
1 Algorithm randomized_partition(A, p, r)
2     i ← random(p, r)
3     A[r] ↔ A[i]
4     return partition(A, p, r)
```

- Hierbei ist `random`(p, r) eine Funktion, die zufällig einen Wert aus $[p \dots r]$ wählt. Dabei gilt für alle $i \in [p \dots r]$

$$\Pr[\texttt{random}(p, r) = i] = \frac{1}{r - p + 1} \ .$$

Randomisierung

- Schlechte Eingaben für Quicksort können durch Randomisierung vermieden werden, d.h. der Algorithmus wirft gelegentlich eine Münze, um sein weiteres Vorgehen zu bestimmen.
- Worst-case Laufzeit bei ungünstigen Münzwürfen immer noch $\Theta\left(n^2\right)$.
- Es gibt keine schlechten Eingaben, bei denen Quicksort bei allen Münzwürfen Laufzeit $\Theta\left(n^2\right)$ besitzt.
- Analyse der erwarteten Laufzeit, wobei Erwartungswert über Münzwürfe genommen wird. Die erwartete Laufzeit ist $\Theta\left(n \log n\right)$.

Randomisierung

- Schlechte Eingaben für Quicksort können durch Randomisierung vermieden werden, d.h. der Algorithmus wirft gelegentlich eine Münze, um sein weiteres Vorgehen zu bestimmen.
- Worst-case Laufzeit bei ungünstigen Münzwürfen immer noch $\Theta\left(n^2\right)$.
- Es gibt keine schlechten Eingaben, bei denen Quicksort bei allen Münzwürfen Laufzeit $\Theta\left(n^2\right)$ besitzt.
- Analyse der erwarteten Laufzeit, wobei Erwartungswert über Münzwürfe genommen wird. Die erwartete Laufzeit ist $\Theta\left(n \log n\right)$.

Randomisierung

- Schlechte Eingaben für Quicksort können durch Randomisierung vermieden werden, d.h. der Algorithmus wirft gelegentlich eine Münze, um sein weiteres Vorgehen zu bestimmen.
- Worst-case Laufzeit bei ungünstigen Münzwürfen immer noch $\Theta\left(n^2\right)$.
- Es gibt keine schlechten Eingaben, bei denen Quicksort bei allen Münzwürfen Laufzeit $\Theta\left(n^2\right)$ besitzt.
- Analyse der erwarteten Laufzeit, wobei Erwartungswert über Münzwürfe genommen wird. Die erwartete Laufzeit ist $\Theta\left(n \log n\right)$.

Randomisierung

- Schlechte Eingaben für Quicksort können durch Randomisierung vermieden werden, d.h. der Algorithmus wirft gelegentlich eine Münze, um sein weiteres Vorgehen zu bestimmen.
- Worst-case Laufzeit bei ungünstigen Münzwürfen immer noch $\Theta\left(n^2\right)$.
- Es gibt keine schlechten Eingaben, bei denen Quicksort bei allen Münzwürfen Laufzeit $\Theta\left(n^2\right)$ besitzt.
- Analyse der erwarteten Laufzeit, wobei Erwartungswert über Münzwürfe genommen wird. Die erwartete Laufzeit ist $\Theta\left(n \log n\right)$.

Satz

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

Randomisierung

- Schlechte Eingaben für Quicksort können durch Randomisierung vermieden werden, d.h. der Algorithmus wirft gelegentlich eine Münze, um sein weiteres Vorgehen zu bestimmen.
- Worst-case Laufzeit bei ungünstigen Münzwürfen immer noch $\Theta\left(n^2\right)$.
- Es gibt keine schlechten Eingaben, bei denen Quicksort bei allen Münzwürfen Laufzeit $\Theta\left(n^2\right)$ besitzt.
- Analyse der erwarteten Laufzeit, wobei Erwartungswert über Münzwürfe genommen wird. Die erwartete Laufzeit ist $\Theta\left(n \log n\right)$.

Satz

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

Lemma

Sei X die Anzahl der Vergleichsoperationen, die während der gesamten Ausführung von Quicksort durchgeführt werden. Dann ist die Laufzeit von Quicksort $\mathrm{O}\left(n + X\right)$.

Median Quicksort

```
1 Algorithm median_quicksort(A, p, r)
2     if p < r then
3         q ← median_partition(A, p, r)
4         median_quicksort(A, p, q − 1)
5         median_quicksort(A, q + 1, r)
```

Median Quicksort

```
1 Algorithm median_quicksort(A, p, r)
2     if p < r then
3         q ← median_partition(A, p, r)
4         median_quicksort(A, p, q − 1)
5         median_quicksort(A, q + 1, r)
```

```
1 Algorithm median_partition(A, p, r)
2     i ← median(A, p, ⌊(p+r)/2⌋, r)
3     A[r] ↔ A[i]
4     return partition(A, p, r)
```

Median Quicksort

```
1 Algorithm median_quicksort(A, p, r)
2     if p < r then
3         q ← median_partition(A, p, r)
4         median_quicksort(A, p, q − 1)
5         median_quicksort(A, q + 1, r)
```

```
1 Algorithm median_partition(A, p, r)
2     i ← median(A, p, ⌊(p+r)/2⌋, r)
3     A[r] ↔ A[i]
4     return partition(A, p, r)
```

```
1 Algorithm median(A, i, j, k)
2     return median of A[i], A[j], A[k]
```

Median Quicksort

- Verbesserung der Güte von Aufteilungen, indem nicht ein festes Element zur Aufteilung benutzt wird, sondern z.B. das mittlere von drei Elementen.
- Können etwa drei zufällige Elemente wählen oder $A[p], A[q], A[r]$ mit $q = \lfloor \frac{p+r}{2} \rfloor$.
- Beide Varianten in der Praxis erfolgreich.
- Analyse der randomisierten Variante ergibt $\Theta(n \log n)$ erwartete Laufzeit.

Median Quicksort

- Verbesserung der Güte von Aufteilungen, indem nicht ein festes Element zur Aufteilung benutzt wird, sondern z.B. das mittlere von drei Elementen.
- Können etwa drei zufällige Elemente wählen oder $A[p], A[q], A[r]$ mit $q = \lfloor \frac{p+r}{2} \rfloor$.
- Beide Varianten in der Praxis erfolgreich.
- Analyse der randomisierten Variante ergibt $\Theta(n \log n)$ erwartete Laufzeit.

Median Quicksort

- Verbesserung der Güte von Aufteilungen, indem nicht ein festes Element zur Aufteilung benutzt wird, sondern z.B. das mittlere von drei Elementen.
- Können etwa drei zufällige Elemente wählen oder $A[p], A[q], A[r]$ mit $q = \lfloor \frac{p+r}{2} \rfloor$.
- Beide Varianten in der Praxis erfolgreich.
- Analyse der randomisierten Variante ergibt $\Theta(n \log n)$ erwartete Laufzeit.

Median Quicksort

- Verbesserung der Güte von Aufteilungen, indem nicht ein festes Element zur Aufteilung benutzt wird, sondern z.B. das mittlere von drei Elementen.
- Können etwa drei zufällige Elemente wählen oder $A[p], A[q], A[r]$ mit $q = \lfloor \frac{p+r}{2} \rfloor$.
- Beide Varianten in der Praxis erfolgreich.
- Analyse der randomisierten Variante ergibt $\Theta(n \log n)$ erwartete Laufzeit.

Fragen?

Quicksort
- Wiederholung
- Einleitung
- Verfahren
- Korrektheit
- Laufzeit
- Varianten