

Datenstrukturen und Algorithmen

Vorlesung 9: Quicksort (K7)

Joost-Pieter Katoen

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

<https://moves.rwth-aachen.de/teaching/ss-18/dsa1/>

18. Mai 2018



Übersicht

- 1 **Quicksort**
 - Das Divide-and-Conquer Paradigma
 - Partitionierung
 - Quicksort Algorithmus
 - Komplexitätsanalyse
- 2 Weitere Sortierverfahren
 - Bubblesort
 - Countingsort
- 3 Vergleich der Sortieralgorithmen

Übersicht

- 1 **Quicksort**
 - Das Divide-and-Conquer Paradigma
 - Partitionierung
 - Quicksort Algorithmus
 - Komplexitätsanalyse
- 2 Weitere Sortierverfahren
 - Bubblesort
 - Countingsort
- 3 Vergleich der Sortieralgorithmen

Divide-and-Conquer

Teile-und-Beherrsche Algorithmen (divide and conquer) teilen das Problem in mehrere Teilprobleme auf, die dem Ausgangsproblem ähneln, jedoch von kleinerer Größe sind.

Sie lösen die Teilprobleme **rekursiv** und kombinieren diese Lösungen dann, um die Lösung des eigentlichen Problems zu erstellen.

Das Paradigma von Teile-und-Beherrsche umfasst 3 Schritte auf jeder Rekursionsebene:

Teile das Problem in eine Anzahl von Teilproblemen auf.

Beherrsche die Teilprobleme durch rekursives Lösen. Hinreichend kleine Teilprobleme werden direkt gelöst.

Verbinde die Lösungen der Teilprobleme zur Lösung des Ausgangsproblems.

Beispiel: Mergesort (s. Vorlesung 7).

Quicksort – Idee

Mergesort sortiert zunächst rekursiv, danach verteilt er sozusagen die Elemente an die richtigen Stellen.

Bei **Quicksort** werden die Elemente zuerst auf die richtige Seite („Hälfte“) des Arrays gebracht, dann wird jeweils rekursiv sortiert.

Quicksort wurde 1961 von Tony Hoare (Großbritannien) entwickelt.

Partitionierung (I)

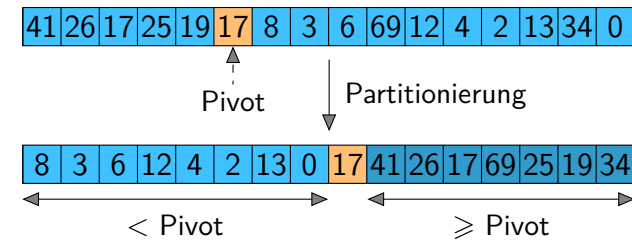
Sobald ein Pivot ausgewählt ist, kann das Array in $O(n)$ partitioniert werden, z. B. folgendermaßen:

- ▶ Arbeite mit drei Bereichen: „< Pivot“, „ \geq Pivot“ und „ungeprüft“.
- ▶ Schiebe die linke Grenze nach rechts, solange das zusätzliche Element < Pivot ist.
- ▶ Schiebe die rechte Grenze nach links, solange das zusätzliche Element \geq Pivot ist.
- ▶ Tausche das links gefundene mit dem rechts gefundenen Element.
- ▶ Fahre fort, bis sich die Grenzen treffen.

(Es gibt auch andere Verfahren.)

Das obige Schema ist ähnlich zu Dijkstra's **Dutch National Flag** Problem.

Quicksort – Strategie



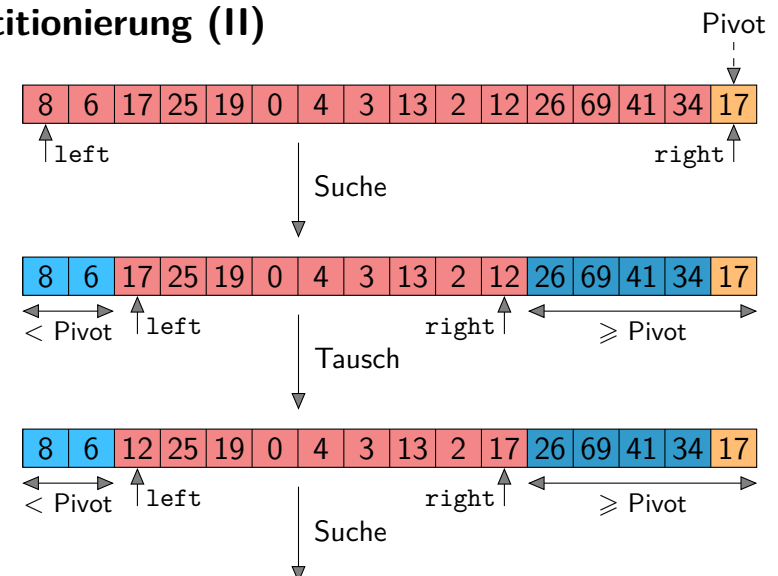
Teile Wähle ein **Pivotelement** aus dem zu sortierenden Array und **partitioniere** das zu sortierende Array in zwei Teile auf:

1. **Kleiner** als das Pivotelement, sowie
2. **mindestens so groß** wie das Pivotelement.

Beherrsche: Sortiere die Teile rekursiv und setze dann das Pivotelement zwischen die sortierten Teile.

Verbinde: Da die Teilfelder in-place sortiert werden ist keine Arbeit nötig, um sie zu verbinden.

Partitionierung (II)



Partitionierung – Algorithmus

```

1 int partition(int E[], int left, int right) {
2   // Wähle einfaches Pivotelement
3   int ppos = right, pivot = E[ppos];
4   while (true) {
5     // Bilineare Suche
6     while (left < right && E[left] < pivot) left++;
7     while (left < right && E[right] >= pivot) right--;
8     if (left >= right) {
9       break;
10    }
11    swap(E[left], E[right]);
12  }
13  swap(E[left], E[ppos]);
14  return left; // gib neue Pivotposition als Splitpunkt zurück
15 }

```

Quicksort – Platzbedarf

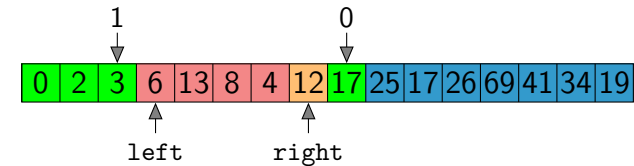
Auf den ersten Blick sieht Quicksort nach einem **in-place** Sortieralgorithmus aus. – **Nicht ganz:**

- ▶ Die rekursiven Aufrufe benötigen Speicherplatz für alle `left` und `right` Parameter.
- ▶ Im Worst-Case wird durch die Partitionierung nur ein Element abgespalten.
- ▶ Dann wird für diese n Elemente $\Theta(n)$ Platz auf dem Stack benötigt.
- ▶ Man kann den Platzbedarf aber auf $\Theta(\log(n))$ reduzieren (siehe Aufgabe 7-4 im Buch).
- ▶ Hauptidee: sortiere nur das größte Teilarray rekursiv, die kleineren iterativ.

Theorem

Die Platzkomplexität von Quicksort ist in $\Theta(\log(n))$.

Quicksort – Algorithmus und Animation



```

1 void quickSort(int E[], int left, int right) {
2   if (left < right) {
3     int i = partition(E, left, right);
4     // i ist Position des Split-punktes (Pivot)
5     quickSort(E, left, i - 1); // sortiere den linken Teil
6     quickSort(E, i + 1, right); // sortiere den rechten Teil
7   }
8 }

```

Quicksort – Worst-Case Analyse

- ▶ Im Worst-Case wird als Pivot **immer** das kleinste oder größte Element im Array genommen.
 - ▶ Dadurch ist das Partitionieren **maximal unbalanciert**:
 - ▶ Ein Teil ist leer, der andere enthält alle verbleibenden Elemente.
 - ▶ Das passiert etwa, wenn das Array bereits auf- oder absteigend sortiert ist.
- ⇒ Der Rekursionsbaum für Quicksort enthält $n-1$ Ebenen.
- ▶ Man erhält: $W(n) = \sum_{i=1}^n (i-1) = \frac{n \cdot (n-1)}{2} \in \Theta(n^2)$
 - ▶ Das ist aber genauso schlecht wie Insertionsort, Bubblesort, usw.
 - ▶ Wenn das Array bereits aufsteigend sortiert ist, braucht Insertionsort nur $O(n)$.

Theorem

Die Worst-Case Laufzeit von Quicksort ist in $\Theta(n^2)$.

Quicksort – Best-Case Analyse

- ▶ Divide-and-conquer funktioniert besonders gut, wenn die Aufteilung so **gleichmäßig wie möglich** geschieht.
- ▶ Wenn wir also das Array mit n Elementen in zwei der Größe $n/2$ teilen, so erhalten wir $\log_2(n)$ Ebenen im Rekursionsbaum.
- ▶ Die Partitionierung hat eine lineare Zeitkomplexität, d. h. jede Ebene braucht $O(n)$ Zeit.
- ▶ Man erhält: $T(n) = 2 \cdot T(n/2) + c \cdot n$ für $n > 1$ mit $T(1) = 1$.
- ▶ Anwendung des Mastertheorems liefert: $T(n) \in \Theta(n \cdot \log(n))$.

Die Ausbalanciertheit der beiden Hälften der Zerlegung in jeder Rekursionsstufe erzeugt also einen **asymptotisch schnelleren** Algorithmus.

Fazit: Wenn man eine Aufgabe zerlegt, ist es am besten, sie in gleich große Teile zu teilen.

Theorem

Die Best-Case Laufzeit von Quicksort ist in $\Theta(n \cdot \log(n))$.

Quicksort – Average-Case-Analyse (I)

- ▶ Annahmen:
 1. das Pivotelement kann in $O(1)$ Zeit gewählt werden
 2. alle Elemente im zu sortierenden Array E sind unterschiedlich
 3. alle möglichen Permutationen haben die gleiche Wahrscheinlichkeit
- ▶ Elemente in den Teilarrays sind noch nicht verglichen worden
 ⇒ Permutationen in Teilarrays haben die gleiche Wahrscheinlichkeit
- ▶ Partitionierung eines Arrays mit $n-1$ Elemente fordert $n-1$ Vergleiche
- ▶ Wir erhalten damit für $n > 1$ folgende Rekursionsgleichung:

$$A(n) = n-1 + \sum_{i=0}^{n-1} \Pr\{\text{Pivot endet an Stelle } i\} \cdot (A(i) + A(n-i-1))$$

wobei $A(0) = A(1) = 0$.

Quicksort – Balancierte Zerlegung

- ▶ Die mittlere Laufzeit von Quicksort ist viel näher an der des besten Falls als an der des schlechtesten Falls.
- ▶ Schlüssel zum Verständnis: wie schlägt sich die Balanciertheit der Zerlegung in der Rekursionsgleichung nieder?
- ▶ Betrachte z. B. eine Zerlegung im Verhältnis 9:1. Dann erhält man für $n > 1$:

$$T(n) \leq T(9n/10) + T(n/10) + c \cdot n$$

- ▶ Rekursionsbaumanalyse liefert: $T(n) \in \mathcal{O}(n \cdot \log(n))$.
- ▶ Diese 9:1 “unbalancierte” Zerlegung liefert asymptotisch die gleiche Zeit wie bei einer Aufteilung zu gleichen Teilen!
- ▶ Eine Aufteilung im Verhältnis 99:1 liefert auch: $T(n) \in \mathcal{O}(n \cdot \log(n))$.

Quicksort – Average-Case-Analyse (II)

$$\begin{aligned} A(n) &= n-1 + \sum_{i=0}^{n-1} \frac{1}{n} \cdot (A(i) + A(n-i-1)) \\ &\quad \left| \sum_{i=0}^{n-1} A(n-i-1) = A(n-1) + A(n-2) + \dots + A(0) \right. \\ &= n-1 + \sum_{i=1}^{n-1} \frac{2}{n} \cdot A(i). \end{aligned}$$

Intermezzo: wir wollen $\sum_i A(i)$ loswerden; folgender Trick hilft:

$$\begin{aligned} n \cdot A(n) - (n-1) \cdot A(n-1) &= 2 \cdot A(n-1) + 2 \cdot (n-1) \\ &\quad \left| \text{teile durch } n \cdot (n+1) \text{ und setze } A'(n) = A(n)/(n+1) \right. \end{aligned}$$

$$\begin{aligned} A'(n) &= A'(n-1) + \frac{2 \cdot (n-1)}{n \cdot (n+1)} \quad \text{mit } A'(0) = 1 \quad \left| \text{Umformen} \right. \\ &= \sum_{i=1}^n \frac{2 \cdot (i-1)}{i \cdot (i+1)} \end{aligned}$$

Quicksort – Average-Case-Analysis (III)

$$\begin{aligned}
 A'(n) &= \sum_{i=1}^n \frac{2 \cdot (i-1)}{i \cdot (i+1)} && | \text{ calculus} \\
 &= 2 \cdot \sum_{i=1}^n \frac{1}{i+1} - 2 \cdot \sum_{i=1}^n \frac{1}{i \cdot (i+1)} && | \text{ calculus} \\
 &= 2 \cdot \sum_{i=1}^n \frac{1}{i} - 2 + \frac{2}{n+1} - 2 \cdot \sum_{i=1}^n \frac{1}{i \cdot (i+1)} && | \text{ harmonische Reihe} \\
 &\leq 2 \cdot \ln(n) - \frac{2n}{n+1} && | A'(n) = A(n)/(n+1) \\
 A(n) &\in \mathcal{O}(n \cdot \log(n))
 \end{aligned}$$

Da die Best-Case Laufzeit in $\Omega(n \cdot \log(n))$ liegt, folgt folgender Satz:

Theorem

Die mittlere Laufzeit von Quicksort ist in $\Theta(n \cdot \log(n))$.

Bubblesort - Idee

Idee

- ▶ Gehe durch die Liste und vertausche alle nebeneinander stehenden Paare, die in falscher Ordnung stehen.
- ▶ Die Liste rechts von der letzten Tauschposition ist bereits sortiert.
- ▶ Wenn keine Änderung, dann fertig.
- ▶ Sonst gehe wieder durch die Liste...

Beim Sortieren steigen/sinken die Elemente zu ihrem Platz in der Ordnung wie Blasen im Wasser auf (daher der Name).

Übersicht

- 1 Quicksort
 - Das Divide-and-Conquer Paradigma
 - Partitionierung
 - Quicksort Algorithmus
 - Komplexitätsanalyse
- 2 Weitere Sortierverfahren
 - Bubblesort
 - Countingsort
- 3 Vergleich der Sortieralgorithmen

Bubblesort

```

1 void bubbleSort(int E[], int n) {
2   while (n > 1) {
3     int j = 1;
4     for (int i = 0; i < n - 1; i++) {
5       if (E[i] > E[i+1]){
6         swap(E[i], E[i+1]);
7         j = i + 1;
8       }
9     }
10    n = j;
11  }
12 }

```

3 8 17 17 19 25 26 41

Countingsort

Countingsort: Idee

- ▶ **Annahme:** Die Schlüssel aller Elemente des Arrays E sind aus $\{0, \dots, k\}$ für ein bekanntes k .
- ▶ **Idee:** Zähle wie oft die einzelnen Schlüssel auftreten und berechne daraus die Positionen der Arrayelemente nach dem Sortieren.
- ▶ Benötigt über Vergleich/Kopieren hinaus weitere **Berechnungen**.
- ▶ Benötigt zusätzlichen **Speicher**.

Countingsort - Komplexitätsanalyse

Komplexität von Countingsort

- ▶ Laufzeit in $\mathcal{O}(n + \text{number of keys})$
- ▶ Speicherbedarf in $\mathcal{O}(n + \text{number of keys})$
- ▶ **Stabil**

Countingsort

```

1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E	1	3	2	4	1	3	3	4	2	0	2
index	0	1	3	6	9						
sorted	0	1	1	2	2	2	3	3	3	4	4

Übersicht

- 1 Quicksort
 - Das Divide-and-Conquer Paradigma
 - Partitionierung
 - Quicksort Algorithmus
 - Komplexitätsanalyse
- 2 Weitere Sortierverfahren
 - Bubblesort
 - Countingsort
- 3 Vergleich der Sortieralgorithmen

Komplexität von Sortieralgorithmen

	<i>Worst-Case</i>	<i>Average-Case</i>	<i>Platzbedarf</i>	<i>Stabil</i>
Insertionsort	$\Theta(n^2)$	$\Theta(n^2)$	in-place	J
Bubblesort	$\Theta(n^2)$	$\Theta(n^2)$	in-place	J
Quicksort	$\Theta(n^2)$	$\Theta(n \cdot \log(n))$	$\Theta(\log(n))$	N*
Mergesort	$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	$\Theta(n)$	J
Heapsort	$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	in-place	N
Countingsort [†]	$\Theta(n+k)$	$\Theta(n+k)$	$\Theta(n+k)$	J

* es gibt Varianten die stabil sind.

† benötigt endlichen Wertebereich (k ist Anzahl versch. Werte)

Nächste Vorlesung

Nächste Vorlesung

Montag 28. Mai, 08:30 (Hörsaal H01). Bis dann!

Einige Bemerkungen

- ▶ Insertion Sort ist einfach und ziemlich effizient für **kleinere** Arrays und **fast sortierte** Eingaben. Wird häufig benutzt als Unterprogramm für andere Sortieralgorithmen für kleinere Instanzen.
- ▶ Mergesort ist ein sehr effizientes Sortierverfahren und wird u. a. benutzt in Perl, Python und Java. Ist leicht anpassbar für Listen und ist stabil.
- ▶ Heapsort ist ein sehr effizientes Sortierverfahren, was jedoch schwieriger auf Listen anzupassen ist und $\mathcal{O}(n \cdot \log(n))$ braucht für fast sortierte Eingaben.
- ▶ Quicksort ist typischerweise ein effizientes Verfahren. Die Wahl des Pivots ist wichtig. Nicht stabil, und nicht so effizient auf fast sortierten Eingaben.
- ▶ Einige Variationen:
 - ▶ **Introsort**: setzt Quicksort ein bis zu einer gewissen Rekursionstiefe und benutzt dann Heapsort.
 - ▶ **Smoothsort**: (komplizierte) Variation von Heapsort die fast $\mathcal{O}(n)$ braucht für fast sortierten Eingaben.