Quicksort

Während bei Mergesort die Aufteilungsphase trivial ist und die eigentliche Arbeit nach den rekursiven Aufrufen geschieht, ist bei Quicksort die anfängliche Zerlegungsphase aufwendig, das anschließende Zusammensetzen jedoch trivial.
Quicksort arbeitet auch nach dem Divide and conquer Prinzip.

Beim Zerlegen geht man nach folgender **Methode** vor:

Um eine Folge F zu sortieren, wählen wir ein beliebiges Element P und benutzen es als Angelpunkt, genannt Pivotelement, für eine Aufteilung der Folge F ohne P in zwei Teilfolgen F1 und F2. F1 besteht nur aus Elementen von F, die kleiner oder gleich P sind, F2 nur aus Elementen von F, die größer oder gleich P sind.
Das Sortierproblem kann man nun dadurch lösen, dass man F1 und F2 rekursiv auf die selbe Weise sortiert und die Ergebnisse dann in offensichtlicher Weise zusammensetzt.

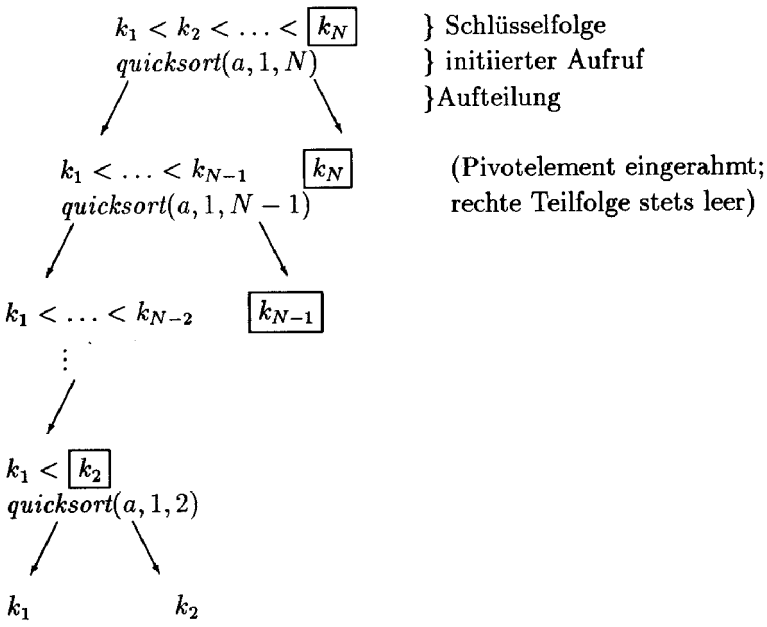
Die Laufzeit von Quicksort hängt stark von der Wahl des Pivotelements ab. Doch dazu später mehr.

Beispiel für die Aufteilungsphase:
Als Pivotelement P nehmen wir den Schlüssel am rechten Ende des zu sortierenden Feldes. Die Aufteilung des Feldes kann man nun folgendermaßen erreichen. Man Setzt einen Positionszeiger i auf das linke Ende des Feldes und einen Positionszeiger j an das Element links neben dem Pivotelement, also das zweite von rechts. Man wandert nun mit dem Positionszeiger i vom linken Ende des aufzuteilenden Feldes nach rechts über alle Elemente hinweg, die kleiner P sind, bis man auf ein Element trifft das größer oder gleich P trifft. Symmetrisch dazu wandert man mit dem Zeiger j vom rechten Enden des Feldes nach links über alle Elemente die größer P sind, bis man auf ein Element trifft dass kleiner oder gleich P ist. Dann vertauscht man F[i] und F[j].
Dies wiederholt man solange bis die Positionszeiger i und j übereinander hinweggelaufen sind. Die Position, bei der die Zeiger übereinander laufen ist auch zugleich die Position des Pivotelements.

Array-Position	3	4	5	6	7	8	9	10	...	
Schlüssel	...	5	7	3	1	6	[4]	...		4 ist Pivot-Element
	$\uparrow_i$						$\uparrow_j$			
		$\uparrow_i$			$\uparrow_j$					1. Halt der Zeiger i, j
	...	1	7	3	5	6	[4]	...		
			$\uparrow_i$	$\uparrow_j$						2. Halt der Zeiger i, j
	...	1	3	7	5	6	[4]	...		
			$\uparrow_j$	$\uparrow_i$						letzter Halt der Zeiger i, j
	...	1	3	[4]	5	6	7	...		
			$\uparrow_j$	$\uparrow_i$						

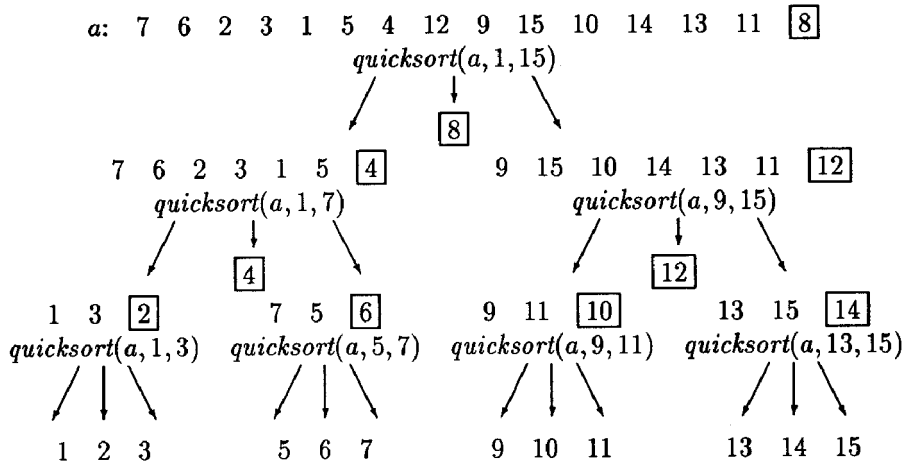
Analyse:

Die Laufzeit von Quicksort hängt wie bereits oben gesagt stark von der Wahl des Pivotelements und der damit verbundnen Zahl von rekursiven Aufrufen ab.
Wählt man z.B. als Pivotelement immer das größte (kleinste) Element in der Folge, so ist beim Aufteilungsschritt immer eine Teilfolge leer und die andere enthält jeweils ein Element weniger als die Ausgangsfolge.
Dieser Sachverhalt ist in der folgenden Graphik dargestellt.



Damit ist klar, dass Quicksort maximal n+(n-1)+(n-2)+...+1 = $O(n^2)$ Schlüsselvergleiche und auch $O(n^2)$ viele Bewegungen der Elemente erfordert, da ja im worstcase alle Elemente je einmal den Platz wechseln. Quicksort.

Quicksort benötigt im schlechtesten Fall also quadratische Laufzeit.
Im günstigsten Fall haben die durch die Aufteilung entstehenden Teilfolgen etwa die gleiche Länge, damit hat auch der Baum der initiierten rekursiven Aufrufe die minimale Höhe, nämlich ungefähr log n.



Zur Aufteilung aller Folgen auf einem Niveau werden O(n) Schlüsselvergleiche durchgeführt.
Da der Baum im günstigsten Fall die Höhe log n hat folgt unmittlebar:
Bestcase → O(n log n)
AverageCase → O(n log n)

> **Warum:** Man geht davon aus, dass die Elemente im allgemeinen rein zufällig angeordnet sind, so dass auch alle Teilfolgen wiederum zufällig angeordnet sind. Daher entsteht wiederum ein Baum mit log n Tiefe.

FAZIT

- BestCase/AverageCase : O (n log n)
- WorstCase $O(n^2)$
    - Um eine bessere Aufteilung der Felder zu erreichen, bedient man sich einer der oder einer Kombination der folgenden Strategien:
        - Randomisierten Quicksort: Alle Elemente werden zufällig neu angeordnet. Oder: Man wählt das Pivotelement zufällig aus.
        - 3-Median-Strategie: Man wählt ein Element vom rechten, eins vom linken Rand und eines Aus der Mitte und bildet den Median dieser Elemente. Dieses Element ist dann das Pivotelement.

- Wendet man die Strategien zur Verbesserung von Quicksort an ist zwar nicht ausgeschlossen, dass ein schlechter Fall, also $O(n^2)$-Eintritt, der Erwartungswert für die Laufzeit von Quicksort, liegt aber bei O(n log n)

- Quicksort kann auch iterativ programmiert werden. → Laufzeiten nur wenig höher, aber die gleiche Schranken.
- In der Regel der schnellste Algorithmus → wird am häufigsten eingesetzt.
- Quicksort ist ein In-Place Algorithmus. → Benötigt nur minimalen Speicher zur Zwischenspeicherung.
- Wie kann man Quicksort verbessern ?
    1. Siehe Strategien von oben
    2. Iterativen Algorithmus mittels eines Stacks wählen, da Rekursion idR. mehr Speicher benötigt.