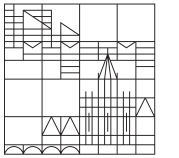
Universität Konstanz



Kontaktstudium IMP

Algorithmik
Rekursive Sortierverfahren

Barbara Pampel

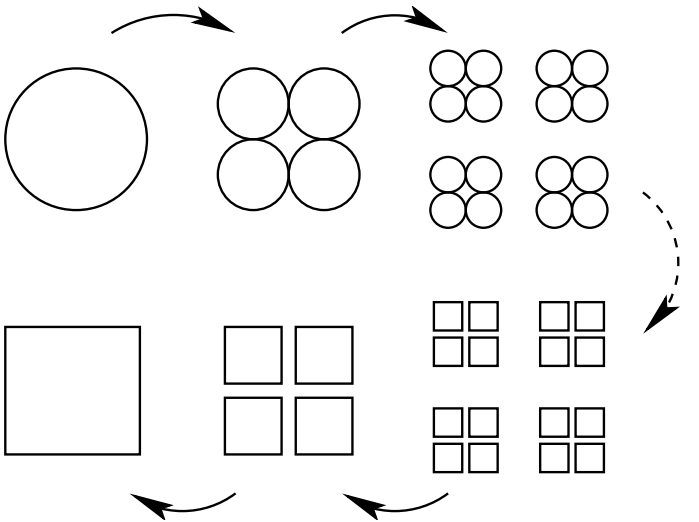
Universität Konstanz, 2018/2019

Inhalt

Inhalt

Divide and Conquer

- Latein: *Divide et impera*
- Deutsch: *teile und herrsche*
- zerlege ein Problem so lange in Teilprobleme, bis man es lösen (*beherrschen*) kann
- impliziert eigentlich noch eine Rekombination



Mergesort

- „Sortieren durch Mischen”
- Listen mit einem Element sind trivialerweise sortiert
- Zwei sortierte Listen zu einer großen sortierten Liste zu verschmelzen ist einfach
  - kleinstes Element der Gesamtliste ist immer am Anfang einer der beiden Listen
  - Entfernen des kleinsten Elements aus dieser Liste und am Ende in die Gesamtliste einfügen
- Algorithmisches Vorgehen
  - wiederholtes Zerteilen der Liste, bis viele Listen mit nur noch einem Element übrig bleiben
  - wiederholtes Zusammenfügen von bereits sortierten Teillisten, bis nur noch eine Liste übrig bleibt

Mergesort-Algorithmus

Divide-and-Conquer-Verfahren

- Zerteilen
  - Aufteilen in zwei Teillisten (mit maximal halber Größe),

- Zusammensetzen
  - Mischen der beiden Teillisten in eine große sortierte Liste

Mergesort-Algorithmus

Divide-and-Conquer-Verfahren

- Zerteilen
  - Aufteilen in zwei Teillisten (mit maximal halber Größe),
  - Sortieren der beiden Teillisten:

- Zusammensetzen
  - Mischen der beiden Teillisten in eine große sortierte Liste

Mergesort-Algorithmus

Divide-and-Conquer-Verfahren

- Zerteilen
  - Aufteilen in zwei Teillisten (mit maximal halber Größe),
  - Sortieren der beiden Teillisten:
    Rekursiver Aufruf
    - Aufruf der Methode in der Methode selbst
    - als Ausgabe wird die sortierte Teilliste angenommen und in den weiteren Schritten verwendet
    - braucht beherrschbaren Basisfall
- Zusammensetzen
  - Mischen der beiden Teillisten in eine große sortierte Liste

Mergesort

Funktion mergesort(Liste L der Länge n)

Falls $n > 1$

ja

- $m := \lfloor \frac{n}{2} \rfloor$ ist die Mitte der Liste L
- $P := \{l_i \in L \mid 1 \leq i \leq m\}$ ist die linke Teilliste
- $Q := \{l_i \in L \mid m < i \leq n\}$ ist die rechte Teilliste
- mergesort(P)
- mergesort(Q)
- $i := 1, j := 1, k := 1$
- Solange $k \leq n$ und P und Q nicht leer
  - Falls $s(p_i) \leq s(q_j)$
    - ja:
      - $l_k := p_i$
      - $k := k + 1, i := i + 1$
    - nein:
      - $l_k := q_j$
      - $k := k + 1, j := j + 1$
- hänge (Rest von) P bzw. Q an L an

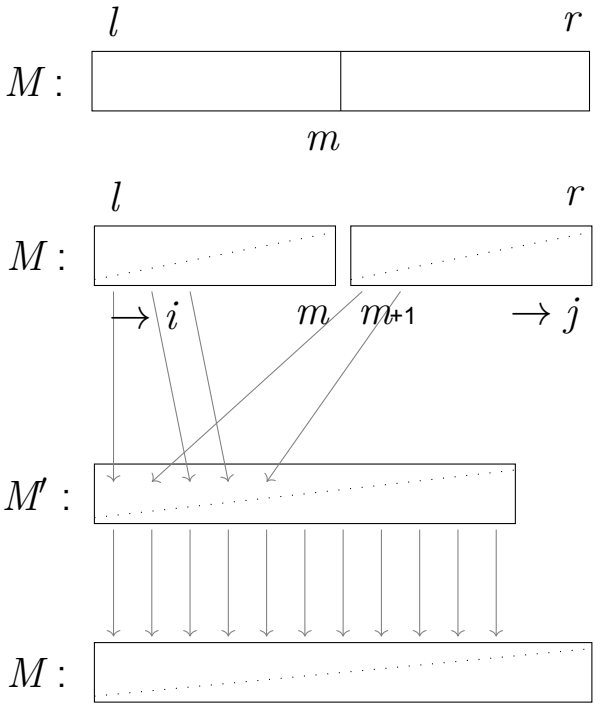
nein

- $\varnothing$

Beispiel Mergesort

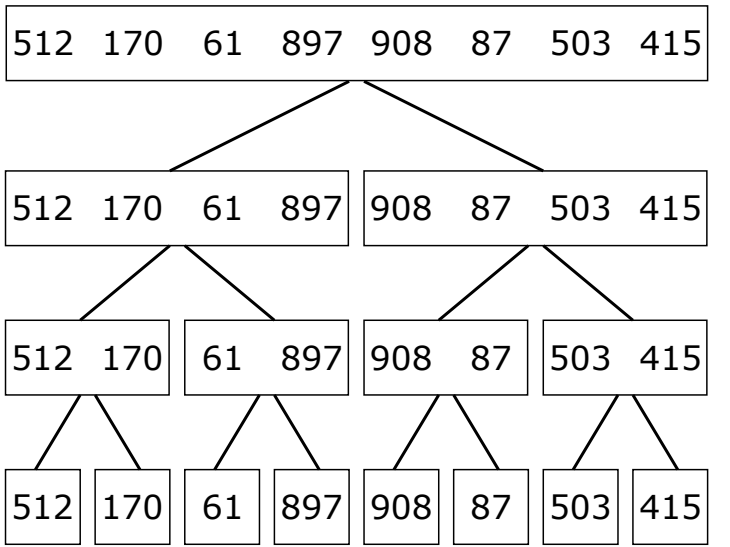
42	92	79	96	66	4	85	mergesort(*(42,92,79,96,66,4,85)*)
42	92	79					mergesort(*(42,92,79)*)
42							mergesort(*(42)*)
42							
	92	79					mergesort(*(92, 79)*)
	92						mergesort(*(92)*)
	92						
		79					mergesort(*(79)*)
		79					
	79	92					
42	79	92					
			96	66	4	85	mergesort(*(96,66,4,85)*)
			96	66			mergesort(*(96,66)*)
			96				mergesort(*(96)*)
			96				
				66			mergesort(*(66)*)
				66			
			66	96			
					4	85	mergesort(*(4,85)*)
					4		mergesort(*(4)*)
					4		
						85	mergesort(*(85)*)
						85	
					4	85	
			4	66	85	96	
4	42	66	79	85	92	96	

Mergesort auf Arrays - Grafisch



Aufwandsabschätzung

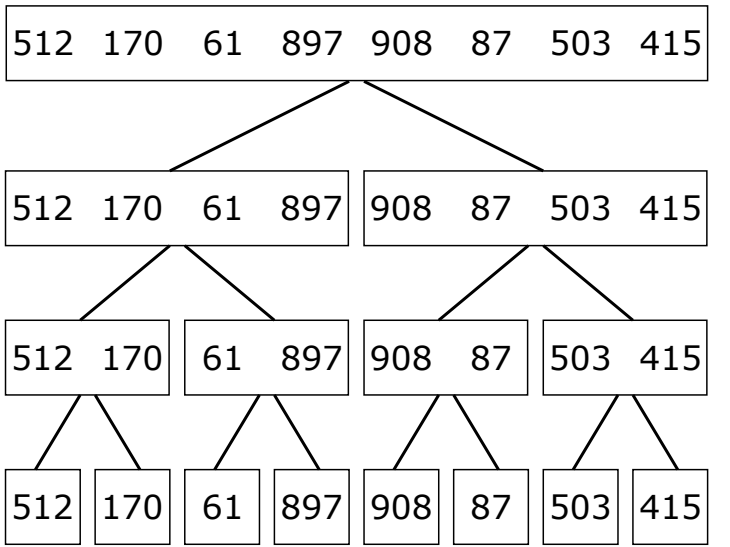
- Es entsteht durch die rekursiven Aufrufe eine virtuelle Baumstruktur



- Gesamtaufwand $\approx 2n \cdot log_2(n)$

Aufwandsabschätzung

- Es entsteht durch die rekursiven Aufrufe eine virtuelle Baumstruktur
- Anzahl der „Ebenen” ist $\lceil log_2(n) \rceil$
- In jeder Ebene wird gibt es n Vergleiche und Zuweisungen



- Gesamtaufwand $\approx 2n \cdot log_2(n)$

Weitere Überlegungen

Korrektheit

Weitere Überlegungen

Korrektheit

- Merge-Schritt sortiert korrekt
- Reihenfolge wird außerhalb der Merge-Schritte nicht verändert
- Nach und nach alle Elemente in einem Merge erfasst
  $\Rightarrow$ terminiert und sortiert komplett

Speicher

Weitere Überlegungen

Korrektheit
- Merge-Schritt sortiert korrekt
- Reihenfolge wird außerhalb der Merge-Schritte nicht verändert
- Nach und nach alle Elemente in einem Merge erfasst
  $\Rightarrow$ terminiert und sortiert komplett

Speicher
- Selbst bei Mergesort auf Arrays benötigt der Algorithmus zusätzlichen Speicherplatz beim Sortieren

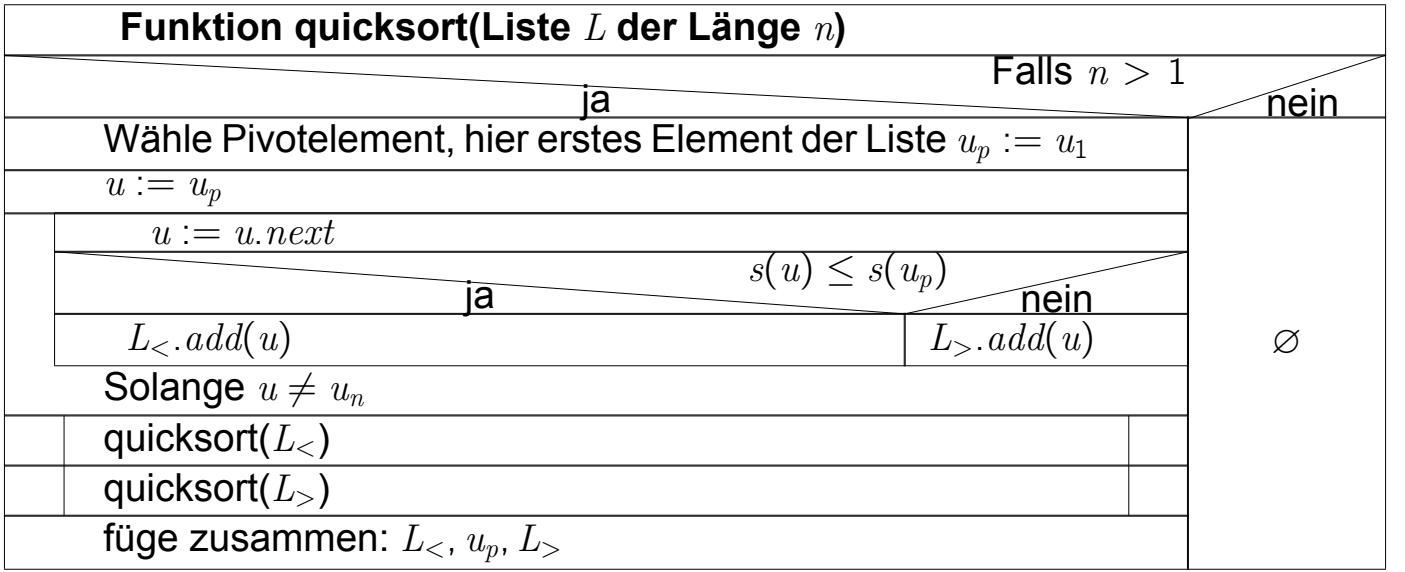
Stabilität
- Mergesort ist ein stabiles Verfahren
  - es werden keine „gleichen” Elemente vertauscht

QuickSort

QuickSort

- Ebenfalls Divide-and-Conquer-Verfahren
- Teilen der Elementmenge nicht unbedingt gleichmäßig
- Auswahl eines beliebigen Elements der Liste, sog. *Pivotelement* u_p
- Bilden neuer (Teil-)Listen:
  - alle Elemente links vom Pivot kleiner oder gleich u_p sind
  - alle Elemente rechts vom Pivot größer u_p sind
- Anordnen:
  - Pivotelement ist bereits an seiner endgültigen Position
  - rekursives Sortieren der linken und rechten (Teil-)Liste

QuickSort auf Liste(n)



Beispiel QuickSort auf Liste(n)

512 87 897 61 908 170 503

Beispiel QuickSort auf Liste(n)

512	87	897	61	908	170	503
87	61	170	503	**512**	897	908

Beispiel QuickSort auf Liste(n)

512	87	897	61	908	170	503
87	61	170	503	**512**	897	908
87	61	170	503	**512**	897	908

Beispiel QuickSort auf Liste(n)

512	87	897	61	908	170	503
87	61	170	503	**512**	897	908
87	61	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908

Beispiel QuickSort auf Liste(n)

512	87	897	61	908	170	503
87	61	170	503	**512**	897	908
87	61	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908

Beispiel QuickSort auf Liste(n)

512	87	897	61	908	170	503
87	61	170	503	**512**	897	908
87	61	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	**170**	503	**512**	897	908

Beispiel QuickSort auf Liste(n)

512	87	897	61	908	170	503
87	61	170	503	**512**	897	908
87	61	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	**170**	503	**512**	897	908
61	**87**	**170**	**503**	**512**	897	908

Beispiel QuickSort auf Liste(n)

512	87	897	61	908	170	503
87	61	170	503	**512**	897	908
87	61	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	**170**	503	**512**	897	908
61	**87**	**170**	**503**	**512**	897	908
61	**87**	**170**	**503**	**512**	897	908

Beispiel QuickSort auf Liste(n)

512	87	897	61	908	170	503
87	61	170	503	**512**	897	908
87	61	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	**170**	503	**512**	897	908
61	**87**	**170**	**503**	**512**	897	908
61	**87**	**170**	**503**	**512**	897	908
61	**87**	**170**	**503**	**512**	**897**	908

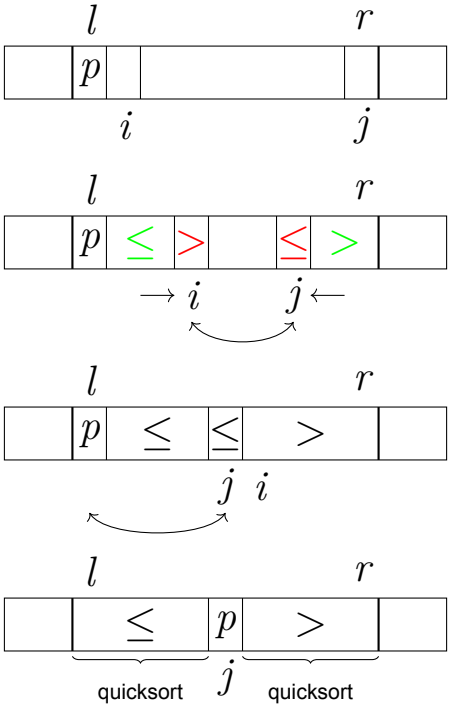
Beispiel QuickSort auf Liste(n)

512	87	897	61	908	170	503
87	61	170	503	**512**	897	908
87	61	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	**170**	503	**512**	897	908
61	**87**	**170**	**503**	**512**	897	908
61	**87**	**170**	**503**	**512**	897	908
61	**87**	**170**	**503**	**512**	**897**	908
61	**87**	**170**	**503**	**512**	**897**	**908**

Quicksort auf einem Array

- Speicher-effizient
- in-place Verfahren
- Verschieben des Pivotelements
  - Suche von links nach einem Element u_i mit $s(u_i) > s(u_p)$
  - Suche von rechts nach einem Element u_j mit $s(u_j) < s(u_p)$
  - Vertauschen der beiden Elemente
  - Solange wiederholen, bis sich i und j treffen
  - Pivotelement mit Element an Position j vertauschen, falls $s(u_p) < s(u_i)$

Quicksort auf einem Array - grafisch



Beispiel QuickSort auf Array

512	87	897	61	908	170	503
p	i					j

Beispiel QuickSort auf Array

512	87	897	61	908	170	503
p	i					j
512	87	897	61	908	170	503
p		i				j

Beispiel QuickSort auf Array

512	87	897	61	908	170	503
p	i					j
512	87	897	61	908	170	503
p		i				j
512	87	503	61	908	170	897
p		i				j

Beispiel QuickSort auf Array

512	87	897	61	908	170	503
p	i					j
512	87	897	61	908	170	503
p		i				j
512	87	503	61	908	170	897
p		i				j
512	87	503	61	908	170	897
p				i	j	

Beispiel QuickSort auf Array

512	87	897	61	908	170	503
p	i					j
512	87	897	61	908	170	503
p		i				j
512	87	503	61	908	170	897
p		i				j
512	87	503	61	908	170	897
p				i	j	
512	87	503	61	170	908	897
p				j	i	

Beispiel QuickSort auf Array

512	87	897	61	908	170	503
p	i					j
512	87	897	61	908	170	503
p		i				j
512	87	503	61	908	170	897
p		i				j
512	87	503	61	908	170	897
p				i	j	
512	87	503	61	170	908	897
p				j	i	
170	87	503	61	512	908	897
p	i		j			

Beispiel QuickSort auf Array

512	87	897	61	908	170	503
p	i					j
512	87	897	61	908	170	503
p		i				j
512	87	503	61	908	170	897
p		i				j
512	87	503	61	908	170	897
p				i	j	
512	87	503	61	170	908	897
p				j	i	
170	87	503	61	512	908	897
p	i		j			
170	87	503	61	512	908	897
p		i	j			

Beispiel QuickSort auf Array

Beispiel QuickSort auf Array

170	87	61	503	512	908	897
p		j	i			
61	87	170	503	512	908	897
p	i,j					

Beispiel QuickSort auf Array

170	87	61	503	512	908	897
p		j	i			
61	87	170	503	512	908	897
p	i,j					
61	87	170	503	512	908	897
p,j	i					

Beispiel QuickSort auf Array

170	87	61	503	512	908	897
p		j	i			
61	87	170	503	512	908	897
p	i,j					
61	87	170	503	512	908	897
p,j	i					
61	**87**	170	503	512	908	897
61	87	170	**503**	512	908	897
61	87	170	503	512	**908**	897
					p	i,j

Beispiel QuickSort auf Array

170	87	61	503	512	908	897	
p		j	i				
61	87	170	503	512	908	897	
p	i,j						
61	87	170	503	512	908	897	
p,j	i						
61	**87**	170	503	512	908	897	
61	87	170	**503**	512	908	897	
61	87	170	503	512	**908**	897	
					p	i,j	
61	87	170	503	512	**908**	897	
					p	j	i

Beispiel QuickSort auf Array

170	87	61	503	512	908	897	
p		j	i				
61	87	170	503	512	908	897	
p	i,j						
61	87	170	503	512	908	897	
p,j	i						
61	**87**	170	503	512	908	897	
61	87	170	**503**	512	908	897	
61	87	170	503	512	**908**	897	
					p	i,j	
61	87	170	503	512	**908**	897	
					p	j	i
61	87	170	503	512	897	**908**	
61	87	170	503	512	897	908	

Quicksort auf einem Array - Pseudocode

Algorithm 1: QuickSort

Aufruf: `quicksort(`$M, 1, n$`)`

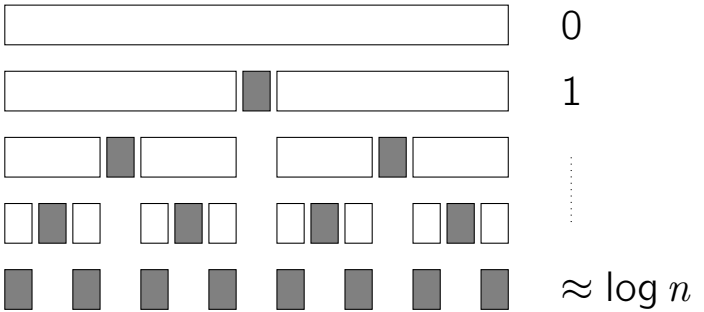
```
quicksort(M, l, r) begin
    if l < r then
        p ← M[l]; i ← l + 1                          // pivot wählen, dann von links nach zu großen
        j ← r                                        // und von rechts nach zu kleinen Elementen suchen
        while i ≤ j do                               // solange i noch links von j
            while i ≤ j and M[i] ≤ p do              // falls klein genug
                i ← i + 1                            // laufe vorbei
            while i ≤ j and M[j] > p do              // falls groß genug
                j ← j − 1                            // laufe vorbei
            if i < j then                            // falls i und j noch nicht aneinander vorbei
                vertausche M[i] und M[j]             // zu großes mit zu kleinem vertauschen

        if l < j then                                // j steht auf dem rechtesten Element, das kleiner als Pivot
            vertausche M[l] und M[j]                 // tausche Pivot an endgültige Position
            quicksort(M, l, j − 1)                   // Aufruf für die kleineren Elemente
        if j < r then quicksort(M, j + 1, r)         // Aufruf für die grösseren Elemente
```

Überlegungen zum Aufwand - grafisch

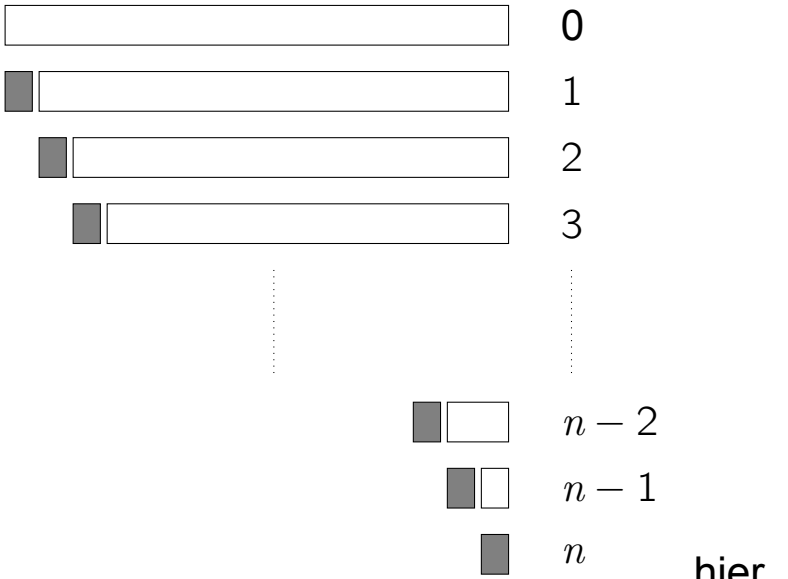
Überlegungen zum Aufwand - grafisch

günstigste Teilung



hier Zeilenzahl =
Rekursionstiefe $\log n$

ungünstigste Teilung



hier

Zeilenzahl =
Rekursionstiefe n

Aufwandsabschätzung - *in-place* Variante

- Idealerweise teilt das Pivotelement das Array in zwei gleich große Teilbereiche, die rekursiv sortiert werden

$\Rightarrow$ Baumstruktur wie bei Mergesort mit $\leq \lceil log_2(n) \rceil$ Ebenen

- In jeder Ebene wird jedes übrig gebliebene Element einmal mit dem Pivotelement verglichen
- In einigen Fällen findet zusätzlich eine Vertauschung statt
- Im Mittel Gesamtaufwand $\approx 1,44 \cdot n \cdot log_2(n)$
- Bei schlechter Wahl des Pivotelements
  - „Baum" hat nun n Ebenen $\Rightarrow$ Gesamtaufwand $\approx \frac{1}{2}n^2$

Weitere Überlegungen

Korrektheit

Weitere Überlegungen

Korrektheit

- Durch das Anordnen wird das Pivot Element an seine endgültige Position gelegt
- Jedes Element wird einmal Pivot oder steht allein
- Auf diese Weise bekommen alle Elemente ihre korrekte Position

Weitere Überlegungen

Korrektheit

- Durch das Anordnen wird das Pivot Element an seine endgültige Position gelegt
- Jedes Element wird einmal Pivot oder steht allein
- Auf diese Weise bekommen alle Elemente ihre korrekte Position

Speicher und Stabilität

Weitere Überlegungen

Korrektheit

- Durch das Anordnen wird das Pivot Element an seine endgültige Position gelegt
- Jedes Element wird einmal Pivot oder steht allein
- Auf diese Weise bekommen alle Elemente ihre korrekte Position

Speicher und Stabilität

- Beim Vertauschen: nur *ein* Speicherplatz für Daten benötigt
  (allerdings wird wegen der Rekursion auch noch Speicher auf dem Stapel benötigt)
- Algorithmus ist nicht stabil
  - Beim Umordnen werden evtl. die Reihenfolge von Elementen mit gleichem Wert verändert

Quicksortvarianten

- Zufällige Auswahl des Pivotelements
  - reduziert Probleme mit (teil)sortieren Listen deutlich

Quicksortvarianten

- Zufällige Auswahl des Pivotelements
  - reduziert Probleme mit (teil)sortieren Listen deutlich
- *Median-aus-3*
  - Pivotelement ist der Median, des linken, rechten und mittleren Elemente der Liste
  - Annahme, dass dieser Median auch die gesamte Liste gleichmäßig teilt

Quicksortvarianten

- Zufällige Auswahl des Pivotelements
  - reduziert Probleme mit (teil)sortieren Listen deutlich
- *Median-aus-3*
  - Pivotelement ist der Median, des linken, rechten und mittleren Elemente der Liste
  - Annahme, dass dieser Median auch die gesamte Liste gleichmäßig teilt
- Vorzeitiger Rekursionsabbruch
  - für wenige Elemente ist QuickSort aufwändiger als z.B. SelectionSort
  - rekursive Aufrufe an sich kosten Zeit
    - in der vorletzten Ebene des Baumes werden $\frac{n}{2}$ rekursive Aufrufe getätigt
  - Abbruch der Rekursion bei z.B. nur noch fünf Elementen und Umschalten auf SelectionSort
  - Laufzeitgewinn rund 10% bei $5 \leq$ MIN_SIZE ≤ 25

Literatur

T. Ottmann und P. Widmayer.
Algorithmen und Datenstrukturen — Kapitel 2.
Spektrum Akademischer Verlag, 4. Ausgabe, 2002, ISBN 978-3-8274-1029-0.

Robert Sedgewick.
Algorithms in Java – Parts 1-4 – Kapitel 6–9.
Addison-Wesley Longman, Amsterdam, 3. Auflage, 2003, ISBN 0-210-36120-5.

H. P. Gumm und M. Sommer.
Einführung in die Informatik — Kapitel 4.2, 4.3.
Oldenburg Verlag, 7. Ausgabe, 2006, ISBN 978-3-486-58115-7.