9. Sortieren II

Mergesort, Quicksort

9.1 Mergesort

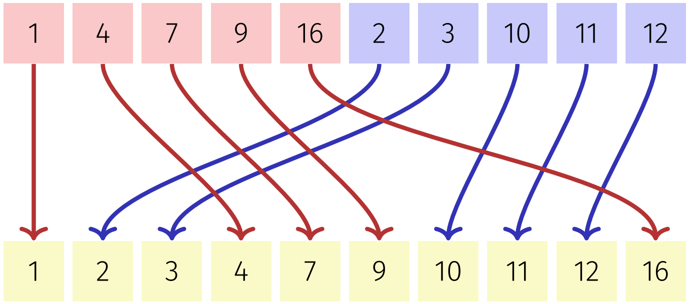
[Ottman/Widmayer, Kap. 2.4, Cormen et al, Kap. 2.3],

Mergesort (Sortieren durch Verschmelzen)

Divide and Conquer!

- Annahme: Zwei Hälften eines Arrays A bereits sortiert.
- Folgerung: Minimum von A kann mit einem Elementvergleich ermittelt werden.
- Iterativ: Füge die beiden vorsortierten Hälften von A zusammen in $\mathcal{O}(n)$.

Merge



Algorithmus `Merge`(A, l, m, r)

Input: Array A der Länge n, Indizes $1 \leq l \leq m \leq r \leq n$.
$A[l, \dots, m]$, $A[m+1, \dots, r]$ sortiert

Output: $A[l, \dots, r]$ sortiert

$B \leftarrow$ new Array$(r - l + 1)$
$i \leftarrow l$; $j \leftarrow m + 1$; $k \leftarrow 1$
while $i \leq m$ and $j \leq r$ **do**
    if $A[i] \leq A[j]$ **then** $B[k] \leftarrow A[i]$; $i \leftarrow i + 1$
    else $B[k] \leftarrow A[j]$; $j \leftarrow j + 1$
    $k \leftarrow k + 1$;
while $i \leq m$ **do** $B[k] \leftarrow A[i]$; $i \leftarrow i + 1$; $k \leftarrow k + 1$
while $j \leq r$ **do** $B[k] \leftarrow A[j]$; $j \leftarrow j + 1$; $k \leftarrow k + 1$
for $k \leftarrow l$ **to** r **do** $A[k] \leftarrow B[k - l + 1]$

Korrektheit

Hypothese: Nach k Durchläufen der Schleife von Zeile 3 ist $B[1,\dots,k]$ sortiert und $B[k] \leq A[i]$, falls $i \leq m$ und $B[k] \leq A[j]$ falls $j \leq r$.

Beweis per Induktion:

Induktionsanfang: Das leere Array $B[1,\dots,0]$ ist trivialerweise sortiert.

Induktionsschluss ($k \to k+1$):

- oBdA $A[i] \leq A[j]$, $i \leq m, j \leq r$.
- $B[1,\dots,k]$ ist nach Hypothese sortiert und $B[k] \leq A[i]$.
- Nach $B[k+1] \leftarrow A[i]$ ist $B[1,\dots,k+1]$ sortiert.
- $B[k+1] = A[i] \leq A[i+1]$ (falls $i+1 \leq m$) und $B[k+1] \leq A[j]$ falls $j \leq r$.
- $k \leftarrow k+1, i \leftarrow i+1$: Aussage gilt erneut.

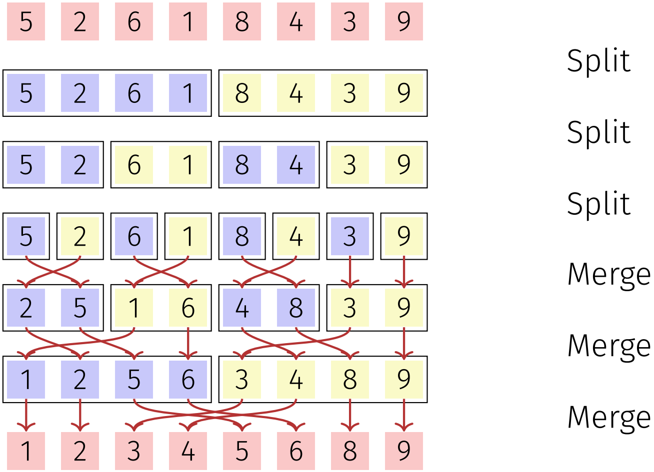
Analyse (Merge)

Lemma 12

Wenn: Array A der Länge n, Indizes $1 \leq l < r \leq n$. $m = \lfloor (l+r)/2 \rfloor$ und $A[l, \ldots, m]$, $A[m+1, \ldots, r]$ sortiert.
Dann: im Aufruf Merge(A, l, m, r) werden $\Theta(r-l)$ viele Schlüsselbewegungen und Vergleiche durchgeführt.

Beweis: (Inspektion des Algorithmus und Zählen der Operationen).

Mergesort


5 2 6 1 8 4 3 9
5 2 6 1 8 4 3 9
5 2 6 1 8 4 3 9
5 2 6 1 8 4 3 9
2 5 1 6 4 8 3 9
1 2 5 6 3 4 8 9
1 2 3 4 5 6 8 9
Split
Split
Split
Merge
Merge
Merge

Algorithmus (Rekursives 2-Wege) `Mergesort`(A, l, r)

Input: Array A der Länge n. $1 \leq l \leq r \leq n$
Output: $A[l, \dots, r]$ sortiert.

```
if l < r then
    m ← ⌊(l + r)/2⌋          // Mittlere Position
    Mergesort(A, l, m)       // Sortiere vordere Hälfte
    Mergesort(A, m + 1, r)   // Sortiere hintere Hälfte
    Merge(A, l, m, r)        // Verschmelzen der Teilfolgen
```

Analyse

Rekursionsgleichung für die Anzahl Vergleiche und Schlüsselbewegungen:

$$T(n) = T(\left\lceil \frac{n}{2} \right\rceil) + T(\left\lfloor \frac{n}{2} \right\rfloor) + \Theta(n) \in \Theta(n \log n)$$

Algorithmus `StraightMergesort`(A)

Rekursion vermeiden: Verschmelze Folgen der Länge $1, 2, 4...$ direkt

Input: Array A der Länge n
Output: Array A sortiert

```
length ← 1
while length < n do                       // Iteriere über die Längen n
    r ← 0
    while r + length < n do               // Iteriere über die Teilfolgen
        l ← r + 1
        m ← l + length − 1
        r ← min(m + length, n)
        Merge(A, l, m, r)
    length ← length · 2
```

Analyse

Wie rekursives Mergesort führt reines 2-Wege-Mergesort immer $\Theta(n \log n)$ viele Schlüsselvergleiche und -bewegungen aus.

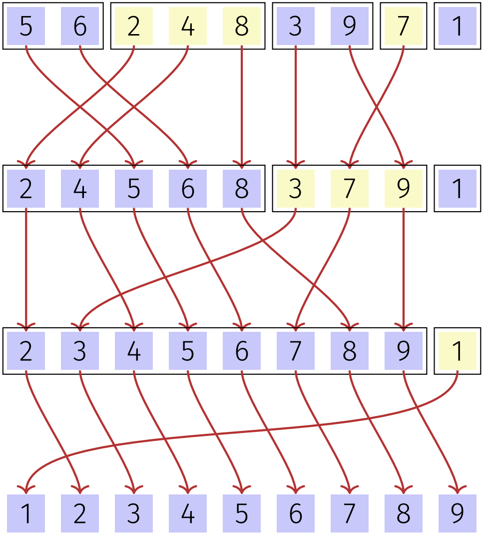
Natürliches 2-Wege Mergesort

Beobachtung: Obige Varianten nutzen nicht aus, wenn vorsortiert ist und führen immer $\Theta(n \log n)$ viele Bewegungen aus.

Wie kann man teilweise vorsortierte Folgen besser sortieren?

❗ Rekursives Verschmelzen von bereits vorsortierten Teilen (*Runs) von* A.

Natürliches 2-Wege Mergesort



Algorithmus `NaturalMergesort`(A)

Input: Array A der Länge $n > 0$
Output: Array A sortiert

repeat
  $r \leftarrow 0$
  while $r < n$ **do**
    $l \leftarrow r + 1$
    $m \leftarrow l$; **while** $m < n$ **and** $A[m+1] \geq A[m]$ **do** $m \leftarrow m + 1$
    if $m < n$ **then**
      $r \leftarrow m + 1$; **while** $r < n$ **and** $A[r+1] \geq A[r]$ **do** $r \leftarrow r + 1$
      Merge(A, l, m, r);
    else
      $r \leftarrow n$
until $l = 1$

Analyse

Ist es auch im Mittel asymptotisch besser als StraightMergesort?

Nein. Unter Annahme der Gleichverteilung der paarweise unterschiedlichen Schlüssel haben wir im Mittel $n/2$ Stellen i mit $k_i > k_{i+1}$, also $n/2$ Runs und sparen uns lediglich einen Durchlauf, also n Vergleiche.

Natürliches Mergesort führt im schlechtesten und durchschnittlichen Fall $\Theta(n \log n)$ viele Vergleiche und Bewegungen aus.

9.2 Quicksort

[Ottman/Widmayer, Kap. 2.2, Cormen et al, Kap. 7]

Quicksort

Was ist der Nachteil von Mergesort?

Benötigt zusätzlich $\Theta(n)$ Speicherplatz für das Verschmelzen.

Wie könnte man das Verschmelzen einsparen?

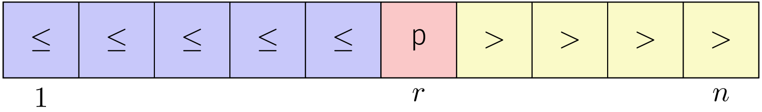
Sorge dafür, dass jedes Element im linken Teil kleiner ist als im rechten Teil.

Wie?

Pivotieren und Aufteilen!

Pivotieren

1. Wähle ein (beliebiges) Element p als Pivotelement
2. Teile A in zwei Teile auf: einen Teil L der Elemente mit $A[i] \leq p$ und einen Teil R der Elemente mit $A[i] > p$.
3. Quicksort: Rekursion auf Teilen L und R



Algorithmus `Partition`(A, l, r, p)

Input: Array A, welches den Pivot p in $A[l, \dots, r]$ mindestens einmal enthält.
Output: Array A partitioniert in $A[l, \dots, r]$ um p. Rückgabe der Position von p.

```
while l ≤ r do
    while A[l] < p do
        l ← l + 1
    while A[r] > p do
        r ← r − 1
    swap(A[l], A[r])
    if A[l] = A[r] then
        l ← l + 1
return l-1
```

Algorithmus $\texttt{Quicksort}(A, l, r)$

Input: Array A der Länge n. $1 \leq l \leq r \leq n$.
Output: Array A, sortiert in $A[l, \dots, r]$.

if $l < r$ **then**
- Wähle Pivot $p \in A[l, \dots, r]$
- $k \leftarrow \texttt{Partition}(A, l, r, p)$
- $\texttt{Quicksort}(A, l, k-1)$
- $\texttt{Quicksort}(A, k+1, r)$

Quicksort (willkürlicher Pivot)

2	4	5	6	8	$\boxed{3}$	7	9	1
$\boxed{2}$	1	3	6	8	$\boxed{5}$	7	9	4
1	2	3	4	5	8	$\boxed{7}$	9	6
1	2	3	4	5	6	7	9	$\boxed{8}$
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9

Analyse: Anzahl Vergleiche

Schlechtester Fall. Pivotelement = Minimum oder Maximum; Anzahl Vergleiche:

$$T(n) = T(n-1) + c \cdot n,\ T(1) = d \quad \Rightarrow \quad T(n) \in \Theta(n^2)$$

Analyse: Anzahl Vertauschungen

Resultat eines Aufrufes an Partition (Pivot 3):

2	1	3	6	8	5	7	9	4

? Wie viele Vertauschungen haben hier maximal stattgefunden?

! 2. Die maximale Anzahl an Vertauschungen ist gegeben durch die Anzahl Schlüssel im kleineren Bereich.

Analyse: Anzahl Vertauschungen

Gedankenspiel

- Jeder Schlüssel aus dem kleineren Bereich zahlt bei einer Vertauschung eine Münze.
- Wenn ein Schlüssel eine Münze gezahlt hat, ist der Bereich, in dem er sich befindet maximal halb so gross wie zuvor.
- Jeder Schlüssel muss also maximal $\log n$ Münzen zahlen. Es gibt aber nur n Schlüssel.

Folgerung: Es ergeben sich $\mathcal{O}(n \log n)$ viele Schlüsselvertauschungen im schlechtesten Fall!

Randomisiertes Quicksort

Quicksort wird trotz $\Theta(n^2)$ Laufzeit im schlechtesten Fall oft eingesetzt.

Grund: Quadratische Laufzeit unwahrscheinlich, sofern die Wahl des Pivots und die Vorsortierung nicht eine ungünstige Konstellation aufweisen.

Vermeidung: Zufälliges Ziehen eines Pivots. Mit gleicher Wahrscheinlichkeit aus $[l, r]$.

Analyse (Randomisiertes Quicksort)

Erwartete Anzahl verglichener Schlüssel bei Eingabe der Länge n:

$$T(n) = (n-1) + \frac{1}{n}\sum_{k=1}^{n}(T(k-1) + T(n-k)),\ T(0) = T(1) = 0$$

Behauptung $T(n) \leq 4n\log n$.

Beweis per Induktion:
Induktionsanfang: klar für $n = 0$ (mit $0\log 0 := 0$) und für $n = 1$.
Hypothese: $T(n) \leq 4n\log n$ für ein n.
Induktionsschritt: $(n-1 \to n)$

Analyse (Randomisiertes Quicksort)

$$
\begin{aligned}
T(n) &= n-1+\frac{2}{n}\sum_{k=0}^{n-1} T(k) \overset{\mathsf{H}}{\leq} n-1+\frac{2}{n}\sum_{k=0}^{n-1} 4k\log k \\
&= n-1+\sum_{k=1}^{n/2} 4k \underbrace{\log k}_{\leq \log n-1} + \sum_{k=n/2+1}^{n-1} 4k \underbrace{\log k}_{\leq \log n} \\
&\leq n-1+\frac{8}{n}\left((\log n-1)\sum_{k=1}^{n/2} k + \log n \sum_{k=n/2+1}^{n-1} k\right) \\
&= n-1+\frac{8}{n}\left((\log n)\cdot\frac{n(n-1)}{2} - \frac{n}{4}\left(\frac{n}{2}+1\right)\right) \\
&= 4n\log n - 4\log n - 3 \leq 4n\log n
\end{aligned}
$$

Analyse (Randomisiertes Quicksort)

Theorem 13

Im Mittel benötigt randomisiertes Quicksort $\mathcal{O}(n \log n)$ Vergleiche.

Praktische Anmerkungen

Rekursionstiefe im schlechtesten Fall: $n - 1$[10]. Dann auch Speicherplatzbedarf $\mathcal{O}(n)$.

Kann vermieden werden: Rekursion nur auf dem kleineren Teil. Dann garantiert $\mathcal{O}(\log n)$ Rekursionstiefe und Speicherplatzbedarf.

[10] Stack-Overflow möglich!

Quicksort mit logarithmischem Speicherplatz

Input: Array A der Länge n. $1 \leq l \leq r \leq n$.
Output: Array A, sortiert zwischen l und r.

while $l < r$ **do**
  Wähle Pivot $p \in A[l, \dots, r]$
  $k \leftarrow$ Partition(A, l, r, p)
  if $k - l < r - k$ **then**
    Quicksort$(A[l, \dots, k-1])$
    $l \leftarrow k + 1$
  else
    Quicksort$(A[k+1, \dots, r])$
    $r \leftarrow k - 1$

Der im ursprünglichen Algorithmus verbleibende Aufruf an Quicksort$(A[l, \dots, r])$ geschieht iterativ (Tail Recursion ausgenutzt!): die If-Anweisung wurde zur While Anweisung.

Praktische Anmerkungen

- Für den Pivot wird in der Praxis oft der Median von drei Elementen genommen. Beispiel: Median3$(A[l], A[r], A[\lfloor l + r/2 \rfloor])$.
- Es existiert eine Variante von Quicksort mit konstanten Speicherplatzbedarf. Idee: Zwischenspeichern des alten Pivots am Ort des neuen Pivots.
- Komplizierte Divide-And-Conquer-Algorithmen verwenden oft als Basisfall einen trivialen ($\Theta(n^2)$) Algorithmus für kleine Problemgrössen.

9.3 Anhang

Herleitung einiger mathematischen Formeln

$\log n! \in \Theta(n \log n)$

$$\log n! = \sum_{i=1}^{n} \log i \leq \sum_{i=1}^{n} \log n = n \log n$$

$$\begin{aligned}
\sum_{i=1}^{n} \log i &= \sum_{i=1}^{\lfloor n/2 \rfloor} \log i + \sum_{\lfloor n/2 \rfloor + 1}^{n} \log i \\
&\geq \sum_{i=2}^{\lfloor n/2 \rfloor} \log 2 + \sum_{\lfloor n/2 \rfloor + 1}^{n} \log \frac{n}{2} \\
&= (\underbrace{\lfloor n/2 \rfloor}_{> n/2 - 1} - 2 + 1) + (\underbrace{n - \lfloor n/2 \rfloor}_{\geq n/2})(\log n - 1) \\
&> \frac{n}{2} \log n - 2.
\end{aligned}$$

$[n! \in o(n^n)\ ]$

$$
\begin{aligned}
n \log n &\geq \sum_{i=1}^{\lfloor n/2 \rfloor} \log 2i + \sum_{i=\lfloor n/2 \rfloor + 1}^{n} \log i \\
&= \sum_{i=1}^{n} \log i + \left\lfloor \frac{n}{2} \right\rfloor \log 2 \\
&> \sum_{i=1}^{n} \log i + n/2 - 1 = \log n! + n/2 - 1
\end{aligned}
$$

$$
n^n = 2^{n \log_2 n} \geq 2^{\log_2 n!} \cdot 2^{n/2} \cdot 2^{-1} = n! \cdot 2^{n/2-1}
$$

$$
\Rightarrow \frac{n!}{n^n} \leq 2^{-n/2+1} \stackrel{n\to\infty}{\longrightarrow} 0 \Rightarrow n! \in o(n^n) = \mathcal{O}(n^n) \backslash \Omega(n^n)
$$

[Sogar $n! \in o((n/c)^n)\,\forall\, 0 < c < e$]

Konvergenz oder Divergenz von $f_n = \frac{n!}{(n/c)^n}$.
Quotientenkriterium

$$\frac{f_{n+1}}{f_n} = \frac{(n+1)!}{\left(\frac{n+1}{c}\right)^{n+1}} \cdot \frac{\left(\frac{n}{c}\right)^n}{n!} = c \cdot \left(\frac{n}{n+1}\right)^n \longrightarrow c \cdot \frac{1}{e} \lesseqgtr 1 \,\text{wenn}\; c \lesseqgtr e$$

denn $\left(1 + \frac{1}{n}\right)^n \to e$. Sogar die Reihe $\sum_{i=1}^n f_n$ konvergiert / divergiert für $c \lesseqgtr e$.
f_n divergiert für $c = e$, denn (Stirling): $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$.

[Quotientenkriterium]

Quotientenkriterium für eine Folge $(f_n)_{n\in\mathbb{N}}$: Wenn $\frac{f_{n+1}}{f_n} \xrightarrow[n\to\infty]{} \lambda$, dann sind die Folge f_n und auch die Reihe $\sum_{i=1}^{n} f_i$

- konvergent, falls $\lambda < 1$ und
- divergent, falls $\lambda > 1$.

[Quotientenkriterium Herleitung]

Quotientenkriterium ergibt sich aus: Geometrische Reihe

$$S_n(r) := \sum_{i=0}^{n} r^i = \frac{1 - r^{n+1}}{1 - r}.$$

konvergiert für $n \to \infty$ genau dann wenn $-1 < r < 1$.
Sei nämlich $0 \le \lambda < 1$:

$$\begin{aligned} &\forall \varepsilon > 0 \, \exists n_0 : f_{n+1}/f_n < \lambda + \varepsilon \, \forall n \ge n_0 \\ \Rightarrow &\exists \varepsilon > 0, \exists n_0 : f_{n+1}/f_n \le \mu < 1 \, \forall n \ge n_0 \end{aligned}$$

Somit

$$\sum_{n=n_0}^{\infty} f_n \le f_{n_0} \cdot \sum_{n=n_0}^{\infty} \cdot \mu^{n-n_0} \quad \text{konvergiert.}$$

(Analog für Divergenz)

Regel von de L'Hospital

Theorem 14

Seien $f, g : \mathbb{R}^+ \to \mathbb{R}$ *differenzierbare Funktionen mit* $g'(x) \neq 0 \; \forall x > 0$.
Falls

$$\lim_{x\to\infty} f(x) = \lim_{x\to\infty} g(x) = 0,$$

oder

$$\lim_{x\to\infty} f(x) = \pm\infty \text{ und } \lim_{x\to\infty} g(x) = \pm\infty,$$

dann

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$$

sofern der Grenzwert von $f'(x)/g'(x)$ *im eigentlichen oder uneigentlichen Sinne existiert.*

Regel von de L'Hospital

Beispiel

Es gilt $\log^k(n) \in o(n)$, denn mit $f(x) = \log^k(x)$, $g(n) = x$, können wir die Regel von de L'Hospital anwenden und erhalten

$$\lim_{x\to\infty} \frac{\log^k(x)}{x} = \lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)} = \lim_{x\to\infty} k\frac{\log^{k-1}(x)}{x}$$

Nach k Iterationen erhalten wir

$$\lim_{x\to\infty} \frac{\log^k(x)}{x} = \lim_{x\to\infty} k!\frac{1}{x} = 0.$$