7. Sortieren I

Einfache Sortierverfahren

7.1 Einfaches Sortieren

Sortieren durch Auswahl, Sortieren durch Einfügen, Bubblesort
[Ottman/Widmayer, Kap. 2.1, Cormen et al, Kap. 2.1, 2.2, Exercise 2.2-2, Problem 2-2

Problemstellung

Eingabe: Ein Array $A = (A[1], ..., A[n])$ der Länge n.

Ausgabe: Eine Permutation A' von A, die sortiert ist: $A'[i] \leq A'[j]$ für alle $1 \leq i \leq j \leq n$.

Algorithmus: IsSorted(A)

Input: Array $A = (A[1], ..., A[n])$ der Länge n.
Output: Boolesche Entscheidung "sortiert" oder "nicht sortiert"

```
for i ← 1 to n − 1 do
    if A[i] > A[i + 1] then
        return "nicht sortiert";
return "sortiert";
```

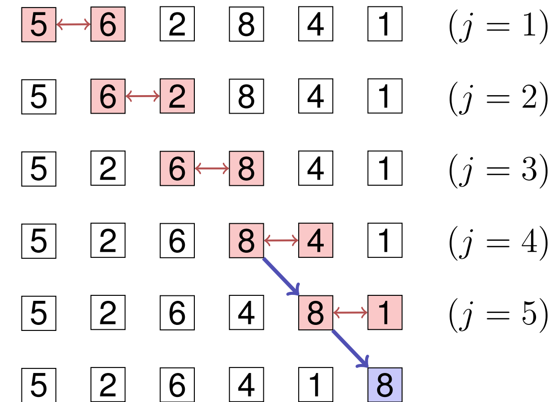
Beobachtung

IsSorted(A):“nicht sortiert”, wenn $A[i] > A[i+1]$ für ein i.

⇒ Idee:

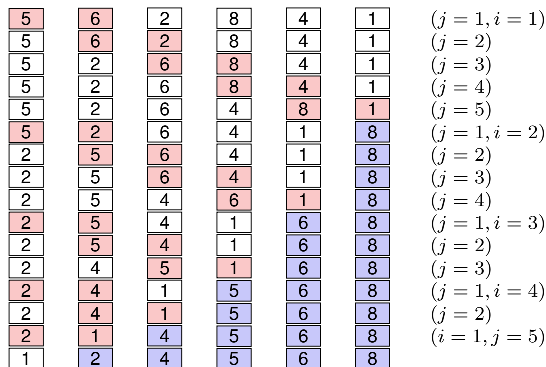
for $j \leftarrow 1$ **to** $n-1$ **do**
  if $A[j] > A[j+1]$ **then**
    swap($A[j], A[j+1]$);

Ausprobieren



- Nicht sortiert! ☹.
- Aber das grösste Element wandert ganz nach rechts.
  ⇒ Neue Idee! ☺

Ausprobieren



- Wende das Verfahren iterativ an.
- Für $A[1,\ldots,n]$, dann $A[1,\ldots,n-1]$, dann $A[1,\ldots,n-2]$, etc.

Algorithmus: Bubblesort

Input: Array $A = (A[1],\ldots,A[n])$, $n \geq 0$.
Output: Sortiertes Array A

for $i \leftarrow 1$ **to** $n-1$ **do**
  for $j \leftarrow 1$ **to** $n-i$ **do**
    if $A[j] > A[j+1]$ **then**
      swap($A[j], A[j+1]$);

Analyse

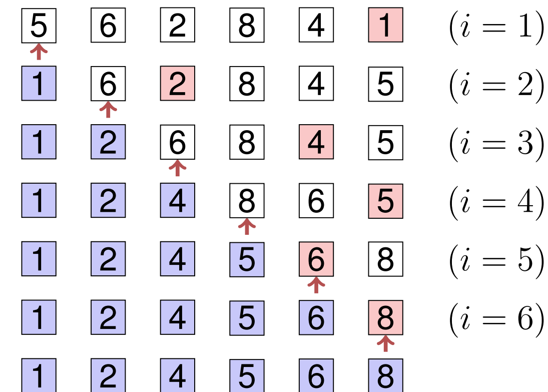
Anzahl Schlüsselvergleiche $\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} = \Theta(n^2)$.

Anzahl Vertauschungen im schlechtesten Fall: $\Theta(n^2)$

? Was ist der schlechteste Fall?

! Wenn A absteigend sortiert ist.

Sortieren durch Auswahl



- Auswahl des kleinsten Elementes durch Suche im unsortierten Teil $A[i..n]$ des Arrays.
- Tausche kleinstes Element an das erste Element des unsortierten Teiles.
- Unsortierter Teil wird ein Element kleiner ($i \rightarrow i+1$). Wiederhole bis alles sortiert. ($i = n$)

Algorithmus: Sortieren durch Auswahl

Input: Array $A = (A[1], \dots, A[n])$, $n \geq 0$.
Output: Sortiertes Array A

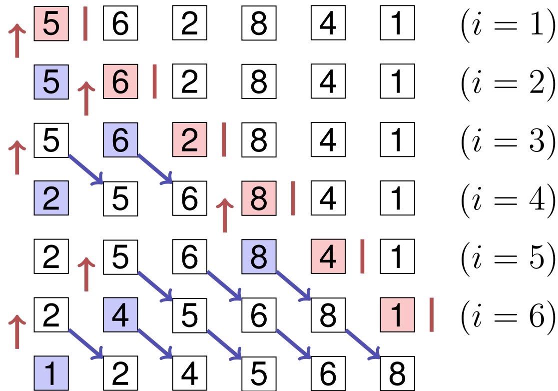
for $i \leftarrow 1$ **to** $n-1$ **do**
  $p \leftarrow i$
  for $j \leftarrow i+1$ **to** n **do**
    if $A[j] < A[p]$ **then**
      $p \leftarrow j$;
  swap($A[i], A[p]$)

Analyse

Anzahl Vergleiche im schlechtesten Fall: $\Theta(n^2)$.

Anzahl Vertauschungen im schlechtesten Fall: $n - 1 = \Theta(n)$

Sortieren durch Einfügen



- Iteratives Vorgehen: $i = 1...n$
- Einfügeposition für Element i bestimmen.
- Element i einfügen, ggfs. Verschiebung nötig.

Sortieren durch Einfügen

? Welchen Nachteil hat der Algorithmus im Vergleich zum Sortieren durch Auswahl?

! Im schlechtesten Fall viele Elementverschiebungen.

? Welchen Vorteil hat der Algorithmus im Vergleich zum Sortieren durch Auswahl?

! Der Suchbereich (Einfügebereich) ist bereits sortiert. Konsequenz: binäre Suche möglich.

Algorithmus: Sortieren durch Einfügen

Input: Array $A = (A[1], \dots, A[n])$, $n \geq 0$.
Output: Sortiertes Array A

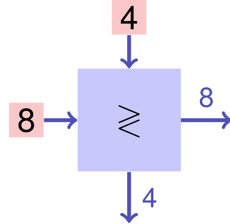
for $i \leftarrow 2$ **to** n **do**
  $x \leftarrow A[i]$
  $p \leftarrow$ BinarySearch($A[1...i-1], x$); // Kleinstes $p \in [1, i]$ mit $A[p] \geq x$
  for $j \leftarrow i - 1$ **downto** p **do**
    $A[j + 1] \leftarrow A[j]$
  $A[p] \leftarrow x$

Analyse

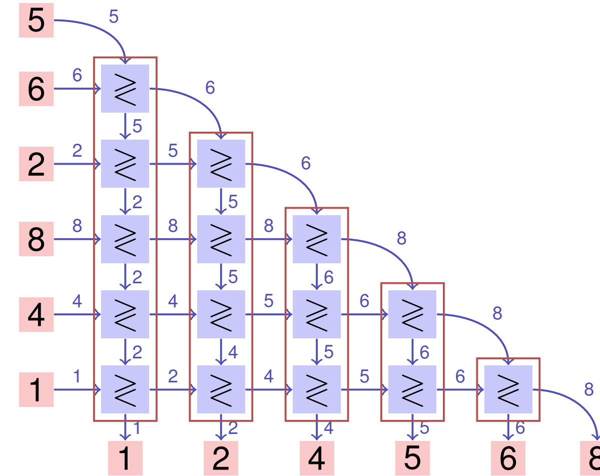
Anzahl Vergleiche im schlechtesten Fall:
$\sum_{k=1}^{n-1} a \cdot \log k = a \log((n-1)!) \in \mathcal{O}(n \log n)$.
Anzahl Vertauschungen im schlechtesten Fall: $\sum_{k=2}^{n}(k-1) \in \Theta(n^2)$

Anderer Blickwinkel

Sortierknoten:

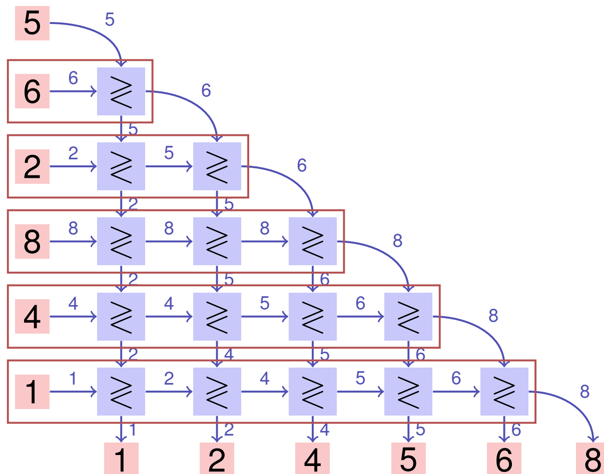


Anderer Blickwinkel



- Wie Selection Sort [und wie Bubble Sort]

Anderer Blickwinkel



- Wie Insertion Sort

Schlussfolgerung

Selection Sort, Bubble Sort und Insertion Sort sind in gewissem Sinne dieselben Sortieralgorithmen. Wird später präzisiert. [7]

[7] Im Teil über parallele Sortiernetzwerke. Für sequentiellen Code gelten natürlich weiterhin die zuvor gemachten Feststellungen.

Shellsort (Donald Shell 1959)

Insertion Sort auf Teilfolgen der Form $(A_{k \cdot i})$ $(i \in \mathbb{N})$ mit absteigenden Abständen k. Letzte Länge ist zwingend $k = 1$.

Worst-case Performance hängt kritisch von den gewählten Teilfolgen ab. Beispiele:

- Ursprünglich mit Folge $1, 2, 4, 8, ..., 2^k$ konzipiert. Laufzeit: $\mathcal{O}(n^2)$
- Folge $1, 3, 7, 15, ..., 2^{k-1}$ (Hibbard 1963). $\mathcal{O}(n^{3/2})$
- Folge $1, 2, 3, 4, 6, 8, ..., 2^p 3^q$ (Pratt 1971). $\mathcal{O}(n \log^2 n)$

Shellsort

9	8	7	6	5	4	3	2	1	0	
1	8	7	6	5	4	3	2	9	0	insertion sort, $k = 4$
1	0	7	6	5	4	3	2	9	8	
1	0	3	6	5	4	7	2	9	8	
1	0	3	2	5	4	7	6	9	8	
1	0	3	2	5	4	7	6	9	8	insertion sort, $k = 2$
1	0	3	2	5	4	7	6	9	8	
0	1	2	3	4	5	6	7	8	9	insertion sort, $k = 1$

8. Sortieren II

Heapsort, Quicksort, Mergesort

8.1 Heapsort

[Ottman/Widmayer, Kap. 2.3, Cormen et al, Kap. 6]

Heapsort

Inspiration von Selectsort: Schnelles Einfügen

Inspiration von Insertionsort: Schnelles Finden der Position

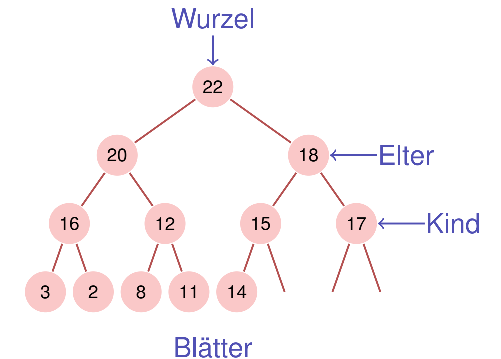
? Können wir das beste der beiden Welten haben?

! Ja, aber nicht ganz so einfach...

[Max-]Heap[8]

Binärer Baum mit folgenden Eigenschaften

1. vollständig, bis auf die letzte Ebene
2. Lücken des Baumes in der letzten Ebene höchstens rechts.
3. *Heap-Bedingung:* Max-(Min-)Heap: Schlüssel eines Kindes kleiner (grösser) als der des Elternknotens

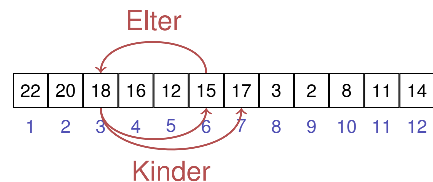


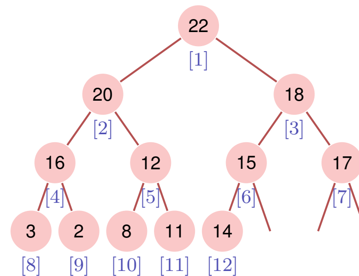
[8] Heap (Datenstruktur), nicht: wie in "Heap und Stack" (Speicherallokation)

Heap als Array

Baum $\rightarrow$ Array:

- Kinder$(i) = \{2i, 2i+1\}$
- Elter$(i) = \lfloor i/2 \rfloor$



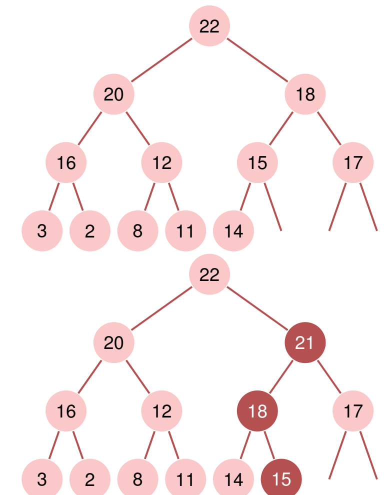


Abhängig von Startindex![9]

[9] Für Arrays, die bei 0 beginnen: $\{2i, 2i+1\} \rightarrow \{2i+1, 2i+2\}$, $\lfloor i/2 \rfloor \rightarrow \lfloor (i-1)/2 \rfloor$

Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.
- Stelle Heap Eigenschaft wieder her: Sukzessives Aufsteigen.
- Anzahl Operationen im schlechtesten Fall: $\mathcal{O}(\log n)$



Algorithmus Aufsteigen(A, m)

Input: Array A mit mindestens $m + 1$ Elementen und Max-Heap-Struktur auf $A[0, \ldots, m - 1]$
Output: Array A mit Max-Heap-Struktur auf $A[0, \ldots, m]$.

```
v ← A[m] // Wert
c ← m // derzeitiger Knoten
p ← ⌊(c − 1)/2⌋ // Elternknoten
while c > 0 and v > A[p] do
    A[c] ← A[p] // Wert Elternknoten → derzeitiger Knoten
    c ← p // Elternknoten → derzeitiger Knoten
    p ← ⌊(c − 1)/2⌋
A[c] ← v // Wert → derzeitiger Knoten
```


Höhe eines Heaps

Vollständiger binärer Baum der Höhe[10] h hat

$$1 + 2 + 4 + 8 + ... + 2^{h-1} = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

Knoten. Somit gilt für einen Heap der Höhe h:

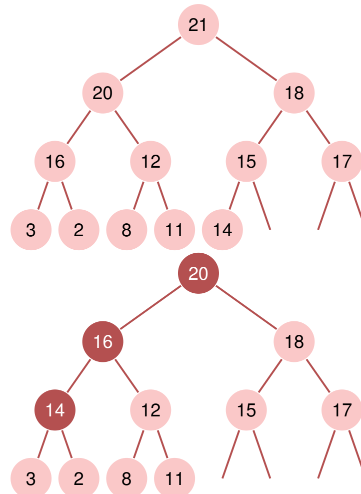
$$2^{h-1} - 1 < n \leq 2^h - 1$$
$$\Leftrightarrow \quad 2^{h-1} < n + 1 \leq 2^h$$

Also insbesondere $h(n) = \lceil \log_2(n + 1) \rceil$ und $h(n) \in \Theta(\log n)$.

[10] Hier: Anzahl Kanten von der Wurzel zu einem Blatt

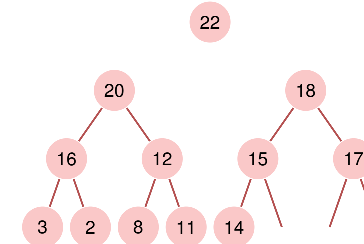
Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her: Sukzessives Absinken (in Richtung des grösseren Kindes).
- Anzahl Operationen im schlechtesten Fall: $\mathcal{O}(\log n)$



Warum das korrekt ist: Rekursive Heap-Struktur

Ein Heap besteht aus zwei Teilheaps:



Algorithmus Versickern(A, i, m)

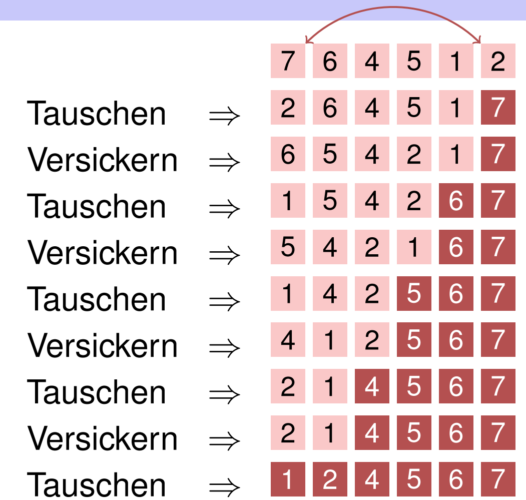
Input: Array A mit Heapstruktur für die Kinder von i. Letztes Element m.
Output: Array A mit Heapstruktur für i mit letztem Element m.

while $2i \leq m$ **do**
  $j \leftarrow 2i$; // j linkes Kind
  if $j < m$ and $A[j] < A[j+1]$ **then**
    $j \leftarrow j + 1$; // j rechtes Kind mit grösserem Schlüssel
  if $A[i] < A[j]$ **then**
    swap($A[i], A[j]$)
    $i \leftarrow j$; // weiter versickern
  else
    $i \leftarrow m$; // versickern beendet

Heap Sortieren

$A[1, ..., n]$ ist Heap.
Solange $n > 1$

- swap($A[1]$, $A[n]$)
- Versickere($A, 1, n-1$);
- $n \leftarrow n - 1$



Heap erstellen

Beobachtung: Jedes Blatt eines Heaps ist für sich schon ein korrekter Heap.

Folgerung: Induktion von unten!

Algorithmus HeapSort(A, n)

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

// Heap Bauen.
for $i \leftarrow n/2$ **downto** 1 **do**
  Versickere(A, i, n);
// Nun ist A ein Heap.
for $i \leftarrow n$ **downto** 2 **do**
  swap($A[1], A[i]$)
  Versickere($A, 1, i-1$)
// Nun ist A sortiert.

Analyse: Sortieren eines Heaps

Versickere durchläuft maximal $\log n$ Knoten. An jedem Knoten 2 Schlüsselvergleiche. $\Rightarrow$ Heap Sortieren kostet im schlechtesten Fall $2n \log n$ Vergleiche.

Anzahl der Bewegungen vom Heap Sortieren auch $\mathcal{O}(n \log n)$.

Analyse: Heap bauen

Aufrufe an Versickern: $n/2$. Also Anzahl Vergleiche und Bewegungen $v(n) \in \mathcal{O}(n \log n)$.
Versickerpfade sind aber im Mittel viel kürzer:

$$v(n) = \sum_{l=0}^{\lfloor \log n \rfloor} \underbrace{2^l}_{\text{Anzahl Heaps auf Level l}} \cdot \underbrace{(\lfloor \log n \rfloor - l)}_{\text{Höhe Heaps auf Level l}} = \sum_{k=0}^{\lfloor \log n \rfloor} 2^{\lfloor \log n \rfloor - k} \cdot k$$

$$\leq \sum_{k=0}^{\lfloor \log n \rfloor} \frac{n}{2^k} \cdot k = n \cdot \sum_{k=0}^{\lfloor \log n \rfloor} \frac{k}{2^k} \in \mathcal{O}(\mathbf{n})$$

mit $s(x) := \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \quad (0 < x < 1)$ [11] und $s(\frac{1}{2}) = 2$

[11] $f(x) = \frac{1}{1-x} = 1 + x + x^2 ... \Rightarrow f'(x) = \frac{1}{(1-x)^2} = 1 + 2x + ...$

Zwischenstand

Heapsort: $\mathcal{O}(n \log n)$ Vergleiche und Bewegungen.

? Nachteile von Heapsort?

- ! Wenig Lokalität: per Definition springt Heapsort im sortierten Array umher (Negativer Cache Effekt).
- ! Zwei Vergleiche vor jeder benötigten Bewegung.

8.2 Mergesort

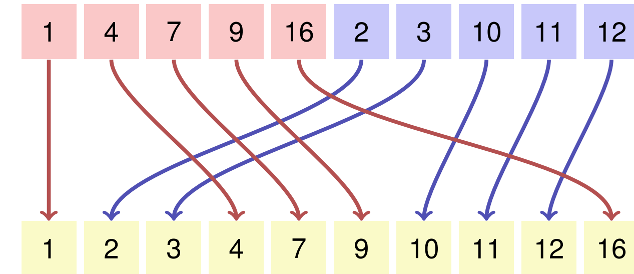
[Ottman/Widmayer, Kap. 2.4, Cormen et al, Kap. 2.3],

Mergesort (Sortieren durch Verschmelzen)

Divide and Conquer!

- Annahme: Zwei Hälften eines Arrays A bereits sortiert.
- Folgerung: Minimum von A kann mit 2 Vergleichen 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 $1 \le l \le m \le r \le n$. $A[l, \dots, m]$, $A[m+1, \dots, r]$ sortiert
Output: $A[l, \dots, r]$ sortiert

```
1 B ← new Array(r − l + 1)
2 i ← l; j ← m + 1; k ← 1
3 while i ≤ m and j ≤ r do
4     if A[i] ≤ A[j] then B[k] ← A[i]; i ← i + 1
5     else B[k] ← A[j]; j ← j + 1
6     k ← k + 1;
7 while i ≤ m do B[k] ← A[i]; i ← i + 1; k ← k + 1
8 while j ≤ r do B[k] ← A[j]; j ← j + 1; k ← k + 1
9 for k ← l to r do A[k] ← B[k − l + 1]
```


Korrektheit

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

Beweis per Induktion:
Induktionsanfang: Das leere Array $B[1, \dots, 0]$ ist trivialerweise sortiert.
Induktionsschluss ($k \to k+1$):

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

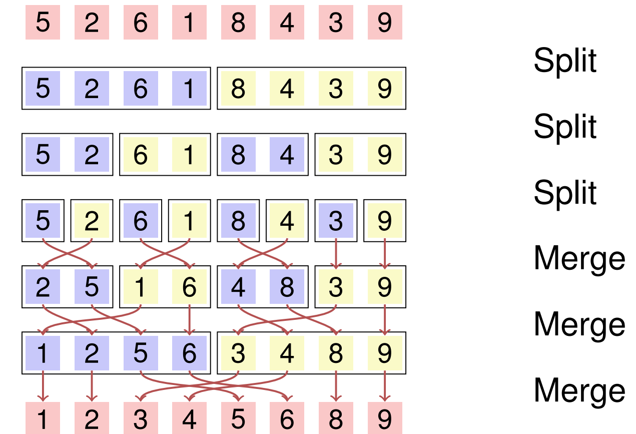
Analyse (Merge)

Lemma

Wenn: Array A der Länge n, Indizes $1 \leq l < r \leq n$. $m = \lfloor (l+r)/2 \rfloor$ und $A[l,\dots,m]$, $A[m+1,\dots,r]$ 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



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: Array $A[l,\dots,r]$ sortiert.
if $l < r$ **then**
  $m \leftarrow \lfloor (l+r)/2 \rfloor$ // 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:

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

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.

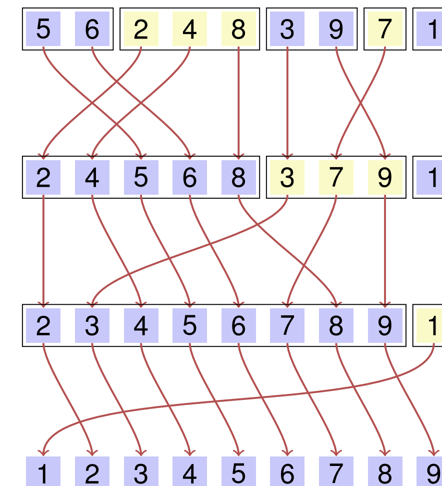
Natürliches 2-Wege Mergesort

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 ← 0
    while r < n do
        l ← r + 1
        m ← l; while m < n and A[m + 1] ≥ A[m] do m ← m + 1
        if m < n then
            r ← m + 1; while r < n and A[r + 1] ≥ A[r] do r ← r + 1
            Merge(A, l, m, r);
        else
            r ← 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.

8.3 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?

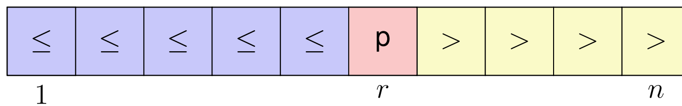
! Sorge dafür, dass jedes Element im linken Teil kleiner ist als im rechten Teil.

? 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 im Intervall $[l, r]$ mindestens einmal enthält.
Output: Array A partitioniert in $[l..r]$ um p. Rückgabe der Position von p.

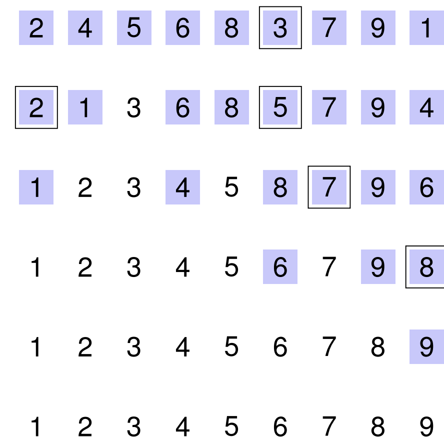
while $l \leq r$ **do**
  while $A[l] < p$ **do**
    $l \leftarrow l + 1$
  while $A[r] > p$ **do**
    $r \leftarrow r - 1$
  swap($A[l]$, $A[r]$)
  if $A[l] = A[r]$ **then**
    $l \leftarrow l + 1$
return l-1

Algorithmus Quicksort($A[l, \dots, r]$)

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

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

Quicksort (willkürlicher Pivot)



Analyse: Anzahl Vergleiche

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

$$T(n) = T(n-1) + c \cdot n, \; T(1) = 0 \quad \Rightarrow \quad T(n) \in \Theta(n^2)$$

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:

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

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)

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

Analyse (Randomisiertes Quicksort)

Theorem

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

Praktische Anmerkungen

Rekursionstiefe im schlechtesten Fall: $n-1$[12]. Dann auch Speicherplatzbedarf $\mathcal{O}(n)$.

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

[12] Stack-Overflow möglich!

Quicksort mit logarithmischem Speicherplatz

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

```
while l < r do
    Wähle Pivot p ∈ A[l,...,r]
    k ← Partition(A[l,...,r], p)
    if k − l < r − k then
        Quicksort(A[l,...,k − 1])
        l ← k + 1
    else
        Quicksort(A[k + 1,...,r])
        r ← 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.

8.4 Anhang

Herleitung einiger mathematischen Formeln

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

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

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

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

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

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

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

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

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

Quotientenkriterium

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

denn $\left(1 + \frac{1}{n}\right)^n \to e$. Sogar die Reihe $\sum_{i=1}^{n} f_n$ konvergiert / divergiert für $c \lessgtr e$.

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 $\frac{f_{n+1}}{f_n} \underset{n\to\infty}{\longrightarrow} \lambda$, 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 $0 \leq \lambda < 1$:

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

Somit

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

(Analog für Divergenz)