Algorithmen und Komplexität
Vorlesung 11

Wolfgang Globke

9. Mai 2019

Erinnerung: Sortieren durch Teile-und-Herrsche

Wir haben bereits zwei iterative Sortierverfahren kennengelernt,

- SelectionSort, mit asymptotischen Aufwand $\Theta(n^2)$.
- InsertionSort, mit asymptotischen Aufwand $O(n^2)$.

Quadratischer Aufwand ist nicht sehr gut für große Datenmengen,
aber für kleine Datenmengen können diese Algorithmen trotzdem sehr gut laufen.

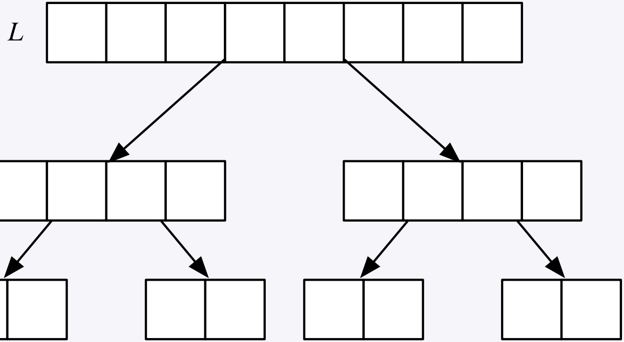
Anderer Ansatz: Teile-und-Herrsche.

- Beobachtung:
  Für Listen der Länge $n = 1$ oder $n = 2$ ist das Sortieren trivial.
- Ansatz:
  Durch wiederholtes Zerteilen der Liste L lässt sich das Sortierproblem rekursiv auf mehrere Instanzen des trivialen Falls zurückführen.
- Schwierigkeit:
  Es ist sicherzustellen, dass sortierte Teillisten in korrekter Weise zur sortierten Gesamtliste L' zusammengefügt werden.

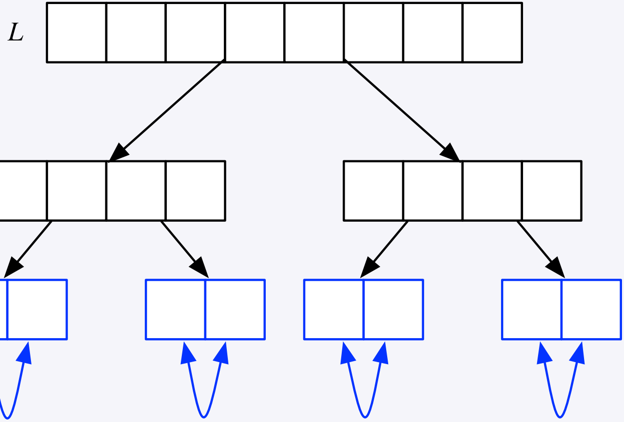
Erinnerung: Sortieren durch Teile-und-Herrsche

L

Erinnerung: Sortieren durch Teile-und-Herrsche

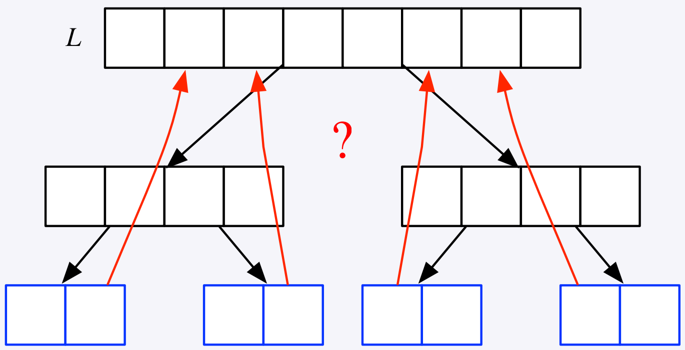


Erinnerung: Sortieren durch Teile-und-Herrsche

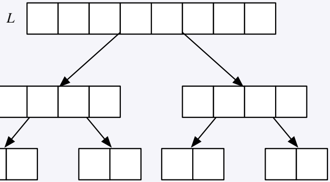


leicht zu sortieren

Erinnerung: Sortieren durch Teile-und-Herrsche



Erinnerung: Sortieren durch Teile-und-Herrsche



Es gibt zwei Sortierverfahren mit unterschiedlichen Ansätzen für das Zusammenfügen der sortierten Teile:

1. QuickSort (oder Sortieren durch Zerlegen) nimmt bei jeder Teilung eine grobe Sortierung vor, so dass bereits alle Elemente im linken Teilbaum kleiner sind als alle Elemente im rechten Teilbaum.
   Am Schluss müssen die einzelnen Teile nur noch andeinander gereiht werden.
2. MergeSort (oder Sortieren durch Mischen ?!) nimmt beim Zerteilen keine Sortierung vor, sondern sortiert beim Zusammenfügen die einzelnen Teile nach dem „Reißverschlussverfahren".

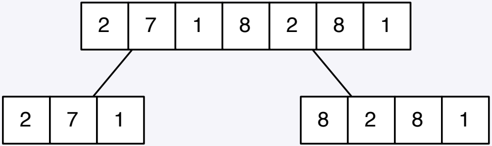
Wir betrachten nun MergeSort. Der Algorithmus sortiert eine Liste L der Länge n folgendermaßen:

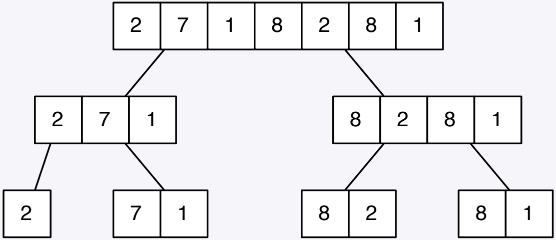
1. Hat die Liste L nur ein Element, so ist sie sortiert.
2. Andernfalls zerteile L in zwei ungefähr gleich große Listen $L_1 = L[0, \ldots, \lfloor \frac{n-1}{2} \rfloor]$ und $L_2 = L[\lfloor \frac{n-1}{2} \rfloor + 1, n-1]$, und wende MergeSort auf diese beiden Teillisten an.
3. Wenn L_1 und L_2 jeweils sortiert sind, rufe eine Hilfsfunktion $\textsc{merge}(L_1, L_2)$ auf, die beide Listen in sortierter Reihenfolge wieder zusammenfügt.

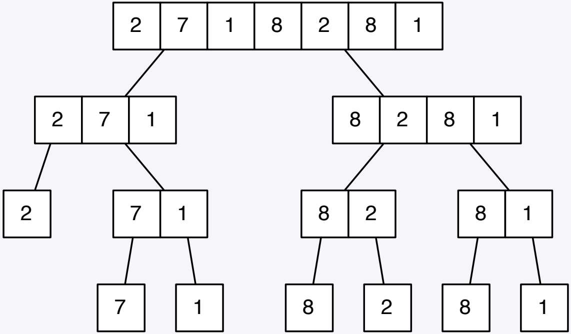
$\textsc{mergeSort}(L)$
- $n := \textsc{length}(L)$
- **if** $n > 0$ **then**
  - $L_1 := L[0, \ldots, \lfloor \frac{n-1}{2} \rfloor]$
  - $L_2 := L[\lfloor \frac{n-1}{2} \rfloor + 1, n-1]$
  - $\textsc{mergeSort}(L_1)$
  - $\textsc{mergeSort}(L_2)$
  - $L := \textsc{merge}(L_1, L_2)$
- **end_if**

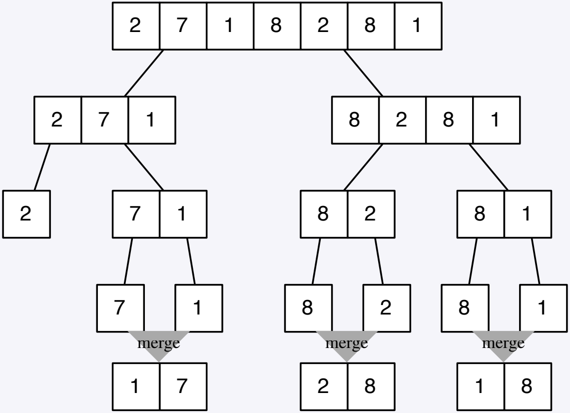
MergeSort – Beispiel

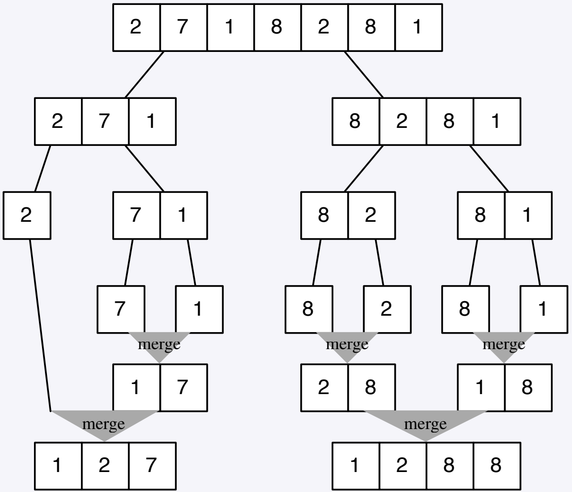
2	7	1	8	2	8	1

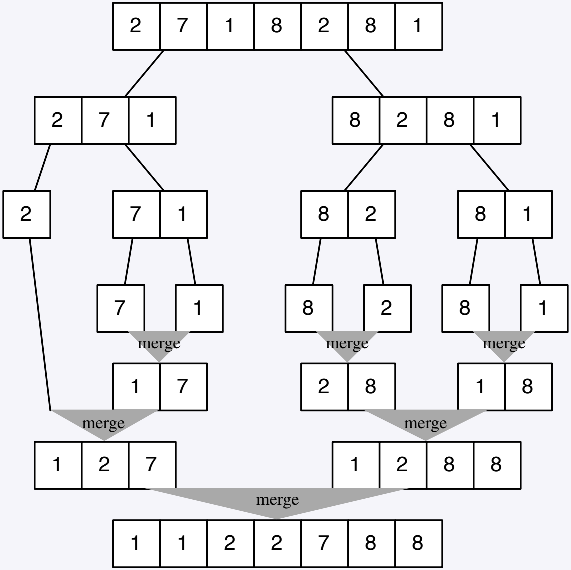

2 7 1 8 2 8 1
2 7 1
8 2 8 1


2 7 1 8 2 8 1
2 7 1
8 2 8 1
2
7 1
8 2
8 1


2 7 1 8 2 8 1
2 7 1
8 2 8 1
2
7 1
8 2
8 1
7
1
8
2
8
1


2 7 1 8 2 8 1
2 7 1
8 2 8 1
2
7 1
8 2
8 1
7
1
8
2
8
1
merge
merge
merge
1 7
2 8
1 8


2 7 1 8 2 8 1
2 7 1
8 2 8 1
2
7 1
8 2
8 1
7
1
8
2
8
1
merge
merge
merge
1 7
2 8
1 8
merge
merge
1 2 7
1 2 8 8


2 7 1 8 2 8 1
2 7 1
8 2 8 1
2
7 1
8 2
8 1
7
1
8
2
8
1
merge
merge
merge
1 7
2 8
1 8
merge
merge
1 2 7
1 2 8 8
merge
1 1 2 2 7 8 8

Die Prozedur MERGE kann für verkettete Listen und Arrays unterschiedlich implementiert werden.

Zunächst nehmen wir an, dass L eine verkettete Liste ist.

```
List MERGE(L1, L2)
    L' := EMPTYLIST()
    while not ISEMPTY(L1) and not ISEMPTY(L2) do
        if HEAD(L1) ≤ HEAD(L2) then
            APPEND(L', HEAD(L1))
            DELETE(L1, HEAD(L1))
        else
            APPEND(L', HEAD(L2))
            DELETE(L2, HEAD(L2))
        end_if
    end_while
    CONCAT(L', L1)
    CONCAT(L', L2)
    return L'
```

Zum Aufwand von MERGE für verkettete Listen:

- Wenn wir für die verketteten Listen noch einen Zeiger auf das letzte Element der Liste mitführen, laufen die Operationen HEAD, APPEND, DELETE und CONCAT alle in O(1).
- Die Funktion MERGE(L_1, L_2) iteriert im ungünstigsten Fall über alle Elemente von L_1 und L_2.
- L_1 und L_2 enthalten maximal $n = |L|$ Elemente.

Satz

Für verkettete Listen läuft MERGE(L_1, L_2) *mit* O(n) *Vergleichen, wenn* $\max\{|L_1|, |L_2|\} \leq n$.

Satz

MERGESORT *für verkettete Listen ist korrekt.*

Beweisskizze:

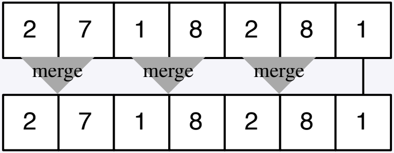
- Mit Induktion über $n = |L|$ ist es klar, dass MERGESORT korrekt ist, wenn MERGE korrekt ist.
- Die Schleifeninvariante von MERGE ist:
  L' ist sortiert, enthält alle Elemente, die nicht mehr in L_1 oder L_2 enthalten sind, und alle Elemente in L' sind kleiner-gleich den verbliebenen Elementen in L_1, L_2. Dies folgt mit der Voraussetzung, dass L_1 und L_2 jeweils sortiert sind.
- Nach der Schleife ist eine der Listen leer (etwa L_2), und L_1 wird an L' angehängt.
  Da L_1 und L' vorher sortiert sind, folgt, dass L' auch nach dem Anhängen von L_1 sortiert ist. $\square$

MergeSort für Arrays

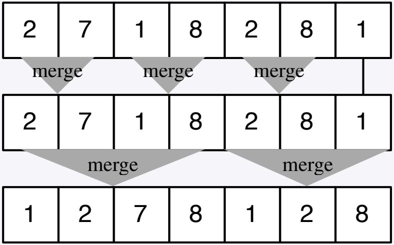
Wenn wir einen Array L sortieren, können wir MergeSort als iterativen Algorithmus interpretieren.

2	7	1	8	2	8	1

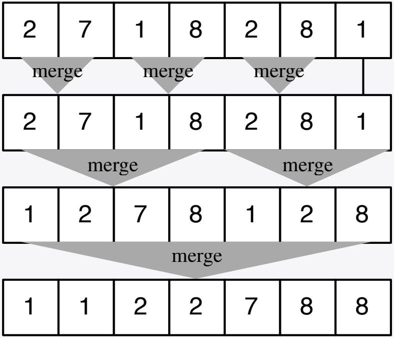
Wenn wir einen Array L sortieren, können wir MergeSort als iterativen Algorithmus interpretieren.



Wenn wir einen Array L sortieren, können wir MergeSort als iterativen Algorithmus interpretieren.



Wenn wir einen Array L sortieren, können wir MergeSort als iterativen Algorithmus interpretieren.



Wenn wir einen Array L der Länge n sortieren, können wir MergeSort als iterativen Algorithmus interpretieren.
Für MERGE benötigen wir hier einen zweiten Array P der Größe $n = |L|$ als Puffer.

1. Führe Laufvariable m und k ein, beginnend mit $m = 1$ und $k = n$.
2. Solange $m < n$:
   - Teile L in k Teilarrays der Länge m ein, $L[0, \dots, m-1]$, $L[m, \dots, 2m-1]$, $\dots$, $L[(k-1)m, n-1]$ (der letzte Teilarray kann kürzer als m sein).
   - Kopiere L in den Puffer P.
   - Für $i = 0, \dots, k-1$:
     Wende MERGE auf je zwei nebeneinanderstehende Teilarrays im Puffer P an:
     MERGE($P[2im, (2i+1)m-1], P[(2i+1)m, r]$) und schreibe das Ergebnis in $L[2im, r]$, wobei $r = \min\{(2i+2)m-1, n-1\}$.
   - Nun sind alle Teilarrays $L[0, \dots, m-1]$, $L[m, \dots, 2m-1]$, $\dots$, $L[(k-1)m, n-1]$ sortiert.
     Ersetze m durch $2m$ und k durch $\lfloor \frac{k+1}{2} \rfloor$, und führe Schritt 2 erneut aus.

```
mergeSort(L)
    n :=length(L)
    P := allocate (n· sizeof(dataType))
    m := 1
    k := n
    while n > m do
        copyArray(L, P)
        k := ⌊(k+1)/2⌋
        for i = 0 to k − 1 do
            if n > (2i + 1)m then
                r := min{(2i + 2)m, n}
                merge (P[2im, ..., (2i + 1)m − 1], P[(2i + 1)m, r − 1],
                       L[2im, ..., r − 1])
            end_if
        end_for
        m := 2m
    end_while
```

Merge für Arrays

```
MERGE(P1, P2, L)
    n1 := LENGTH(P1)
    n2 := LENGTH(P2)
    i1 := i2 := k := 0
    while i1 < n1 and i2 < n2 do
        if P1[i1] ≤ P2[i2] then
            L[k] := P1[i1]
            i1 := i1 + 1
        else
            L[k] := P2[i2]
            i2 := i2 + 1
        end_if
        k := k + 1
    end_while
        ⋮
```

```
    ⋮
if i1 = n1 then
    for j = i2 to n2 − 1 do
        L[k] := P2[j]
        k := k + 1
    end_for
else
    for j = i1 to n1 − 1 do
        L[k] := P1[j]
        k := k + 1
    end_for
end_if
```

```
MERGESORT(L)
    n := LENGTH(L)
    if n > 0 then
        L_1 = L[0, ..., ⌊(n-1)/2⌋]
        L_2 = L[⌊(n-1)/2⌋ + 1, n - 1]
        MERGESORT(L_1)
        MERGESORT(L_2)
        L := MERGE(L_1, L_2)
    end_if
```

Anhand der MERGESORT-Variant für Listen sehen wir:
Hat L die Länge $n = 2^k$, so gilt für den Aufwand $T_{\mathrm{ms}}(n)$ von MERGESORT:

$$T_{\mathrm{ms}}(n) = T(\text{ZERLEGEN}) + T(\text{MERGE}) + 2T_{\mathrm{ms}}\left(\frac{n}{2}\right).$$

Da Zerlegen der Liste und MERGE in $O(n)$ ablaufen, folgt mit dem Master-Theorem:

$$T_{\mathrm{ms}}(n) \in \Theta(n \log(n)).$$

Falls $m = 2^k < n < 2^{k+1} = 2m$,

$$\Theta(m \log(m)) \leq T_{\mathrm{ms}}(n) \leq \Theta(2m(\log(m) + \log(2))) = \Theta(m \log(m)).$$

Satz
Der Algorithmus MERGESORT *hat Aufwand* $\Theta(n \log(n))$.

QuickSort vs. MergeSort

QuickSort	MergeSort
Extremfall $O(n^2)$ Erwartung $\Theta(n \log(n))$	$\Theta(n \log(n))$
wahlfreier Zugriff auf Datenelemente	sequentieller Zugriff auf Datenelemente
(relativ) speichereffizient für Arrays	speichereffizient für verkettete Listen
Hauptarbeit in SPLIT, SPLIT ist schnell	Hauptarbeit in MERGE, aufwändig für Arrays
	stabil
braucht Speicher im Call-Stack, „schwaches" in-place	nicht in-place, Speichermehraufwand $O(n)$

Ausblick

- QuickSort aufgrund der kompakten SPLIT-Prozedur die Methode der Wahl zur Sortierung von Daten im Hauptspeicher.
- MergeSort aufgrund des sequentiellen Zugriffs besser für das Sortieren externer Daten (Festplatte) geeignet.
- Raffinierte Kombinationen von MergeSort und QuickSort sind an Speicherhierarchien angepasst.

Untere Schranke für vergleichsbasiertes Sortieren

Ein Sortieralgorithmus ist vergleichsbasiert, wenn alle Entscheidungen über das Sortieren der Elemente in $L = [x_1, \ldots, x_n]$ in der Ergebnisliste L' durch Vergleiche der Form $x_j \geq x_k$? getroffen werden.

SelectionSort, InsertionSort, QuickSort und MergeSort sind alle vergleichsbasierte Algorithmen.

Bisher war $\Theta(n \log(n))$ der beste asymptotische Aufwand, den wir für ein vergleichsbasiertes Sortiervefahren gefunden haben.

Können wir diese asymptotische Schranke verbessern?
Nein.

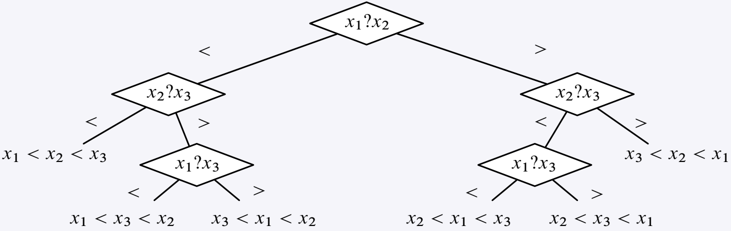
Vergleichsbaum

Es sei $L = [x_1, \ldots, x_n]$.
Vereinfachende Annahme: Alle x_i, x_j sind paarweise verschieden.

Dann gibt es genau eine Permutation σ der Indizes $1, \ldots, n$, so dass

$$x_{\sigma(1)} < x_{\sigma(2)} < \ldots < x_{\sigma(n-1)} < x_{\sigma(n)}.$$

Wir können uns diese Permutation σ als Blatt in einem Vergleichsbaum vorstellen.



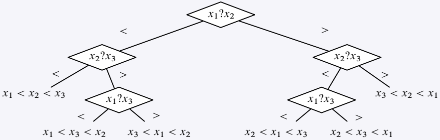
Untere Schranke für Vergleiche

Satz

Jeder vergleichsbasierte Sortieralgorithmus benötigt im ungünstigsten Fall $\Omega(n \log(n))$ *Vergleiche.*

Beweisskizze:

- Ein vergleichsbasierter Sortieralgorithmus bestimmt für eine Eingabe $L = [x_1, \ldots, x_n]$ ein eindeutiges Blatt im Sortierbaum, dass der korrekten Permutation σ mit $x_{\sigma(1)} < x_{\sigma(2)} < \ldots < x_{\sigma(n-1)} < x_{\sigma(n)}$ entspricht.



- Offenbar entsprechen verschiedenen Permutationen verschiedene Blätter.
- Da ein Binärbaum der Tiefe T maximal 2^T Blätter hat, gilt $2^T \geq \#(\text{Permutationen von } \{1, \ldots, n\}) = n!$ bzw. $T \geq \log(n!)$.
- Mit Hilfe der Stirlingschen Formel $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ folgt $T \geq \log(n!) \geq \log((\frac{n}{e})^n) = n\log(n) - n\log(e) \in \Omega(n\log(n))$.
- Die Höhe des Vergleichsbaumes ist die Mindestanzahl an Vergleichen, die ein Algorithmus im ungünstigsten Fall durchführen muss, um die gesuchte Permutation σ zu identifizieren. $\square$