Algorithmen und Datenstrukturen

Teile und herrsche

Prof. Justus Piater, Ph.D.

29. Mai 2024

Dieser Kurs folgt in weiten Teilen dem sehr empfehlenswerten Lehrbuch *Data Structures and Algorithms in Java* [Goodrich u. a. 2014].

Inhaltsverzeichnis

Einführung

Divide et impera auf Latein, englisch *divide and conquer*, bedeutet auf deutsch so viel wie ***Teile und herrsche***. Dieses Prinzip wird dem König Philipp II. von Makedonien zugeschrieben, einem erfolgreichen Feldherrn und Strategen im 4. Jahrhundert vor Christus. Seine Militärreformen trugen wesentlich zu den Erfolgen seines Sohnes Alexander des Großen bei, und später zu denen anderer mächtiger Herrscher wie Julius Caesar und Napoléon Bonaparte.

All diese waren darauf aus, die Heere ihrer Gegner aufzuspalten, um sie zu schwächen. Wir hingegen werden dieses Prinzip anwenden, algorithmische Probleme aufzuspalten, um sie zu lösen.

Typische Beispiele des Teile-und-herrsche-Paradigmas sind Sortieralgorithmen. Daher werden wir uns im Rest dieses Kapitels mit Sortieralgorithmen beschäftigen.

Video 1 beginnt hier.

Algorithmisches Paradigma: Teile und herrsche *(Divide and Conquer)* [Slide 1]

1. **Teile:** Ist das Problem trivial lösbar, löse es unmittelbar. Andernfalls teile es in mehrere ***disjunkte***, ***gleichartige*** Teilprobleme auf.
2. **Herrsche:** Löse die Teilprobleme ***rekursiv***.
3. **Vereine:** Füge die Lösungen der Teilprobleme zu einer Gesamtlösung zusammen.

Definition

Das algorithmische Paradigma *Teile und herrsche* besteht aus diesen drei Schritten:

1. **Teile:** Ist das Problem trivial lösbar, löse es unmittelbar. Andernfalls teile es in mehrere ***disjunkte***, ***gleichartige*** Teilprobleme auf.
2. **Herrsche:** Löse die Teilprobleme ***rekursiv***.
3. **Vereine:** Füge die Lösungen der Teilprobleme zu einer Gesamtlösung zusammen.

1 *Merge-Sort*

Merge-Sort [Slide 2]

Sortiert bedeutet immer *aufsteigend gemäß der verwendeten Totalordnung* $\leq$.

Sortiere eine Sequenz S der Länge n:

1. **Teile:** Ist $n \leq 1$, gib S zurück; andernfalls verschiebe die ersten $\lfloor n/2 \rfloor$ und die letzten $\lceil n/2 \rceil$ Elemente jeweils in eine neue Sequenz S_1 bzw. S_2.
2. **Herrsche:** Sortiere S_1 und S_2 rekursiv.
3. **Vereine:** Spleiße S_1 und S_2 zu einer sortierten Sequenz S zusammen.

Definition

Merge-Sort ist ein Sortieralgorithmus nach dem Teile-und-herrsche-Paradigma. Die drei charakteristischen Schritte von Teile-und-herrsche-Algorithmen sehen hier folgendermaßen aus:

- Im *Teile*-Schritt wird die zu sortierende Sequenz in zwei möglichst gleich große Teilsequenzen aufgeteilt. Oder, falls die zu sortierende Sequenz weniger als zwei Elemente enthält, gilt diese als bereits sortiert und wird sofort zurückgegeben.
- Im *Herrsche*-Schritt ruft der Algorithmus sich selbst rekursiv auf diesen beiden Teilsequenzen auf.
- Im anschließenden *Vereine*-Schritt werden diese beiden, nun sortierten Teilsequenzen zu *einer* sortierten Sequenz zusammengefügt.

Merge-Sort: Beispiel [Slide 3]

Animation

Sehen wir uns hierzu ein Beispiel an. Hier wollen wir die Sequenz `6,5,7,4,1,3,9,8,2` sortieren.

Für uns bedeutet *sortieren* immer aufsteigend gemäß einer Totalordnung $\leq$.

Im *Teile*-Schritt wird die Sequenz in zwei Teilsequenzen aufgeteilt, und im anschließenden *Herrsche*-Schritt ruft sich der Algorithmus auf beiden Teilsequenzen rekursiv selbst auf. Hier sehen wir den rekursiven Aufruf auf der ersten Teilsequenz `6,5,7,4`.

Diese wird nun im *Teile*-Schritt des rekursiven Aufrufs ihrerseits aufgeteilt, unmittelbar gefolgt vom nächsten rekursiven *Herrsche*-Aufruf auf der ersten Teilsequenz `6,5`. Diese wird ihrerseits in zwei Teilsequenzen aufgeteilt.

Wir sehen den rekursiven Aufruf auf der ersten Teilsequenz `6`, der sofort zurückkehrt, da die Sequenz kürzer ist als zwei Elemente. Somit ist sie trivialerweise bereits sortiert. Deshalb wird sie hier grün hervorgehoben. Auch die zweite Teilsequenz `5` wird auf diese Weise als sortiert markiert.

Damit sind die beiden rekursiven Aufrufe des *Herrsche*-Schritts auf der Teilsequenz `6,5` beendet, und es folgt der *Vereine*-Schritt, in dem die beiden sortierten Teilsequenzen zu *einer* sortierten Sequenz zusammengespleißt werden.

Hierzu betrachten wir die beiden ersten Elemente der beiden Teilsequenzen, hier durch die beiden unteren dunkelblauen Ringe markiert, und vergleichen diese. Das kleinere Element `5`, nun rot markiert, platzieren wir im ersten Feld der sortierten Sequenz, hier durch den oberen dunkelblauen Ring markiert.

Nun bestimmen wir, welches Element im nächsten Feld der sortierten Sequenz zu platzieren ist. Hier ist nur mehr eines übrig, nämlich das Element 6 der ersten Teilsequenz. Nun ist die Sequenz `5,6` fertig sortiert und wird daher grün hinterlegt.

Hiermit ist der erste rekursive Aufruf des *Herrsche*-Schritts der Teilsequenz `6,5,7,4` zurückgekehrt, und es folgt der zweite auf der Teilsequenz `7,4`.

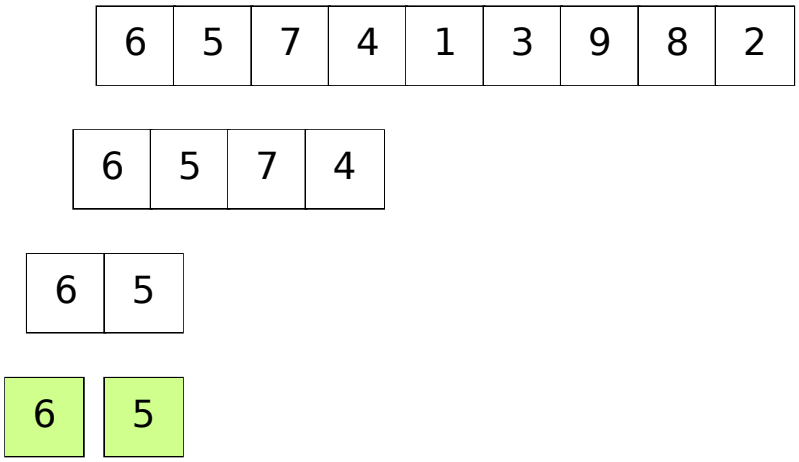
Wenn auch dieser zurückgekehrt ist, folgt der *Vereine*-Schritt des Aufrufs der Teilsequenz `6,5,7,4`. Die beiden unteren dunkelblauen Ringe markieren wiederum die beiden zu vergleichenden Elemente, und das kleinere, rot markiert, wird im nächsten Feld der sortierten Zielsequenz platziert.

Auf diese Weise bewegen sich die Markierungen in allen drei beteiligten Sequenzen monoton von links nach rechts. Sind beide Quellsequenzen abgearbeitet, ist die Zielsequenz komplett sortiert.

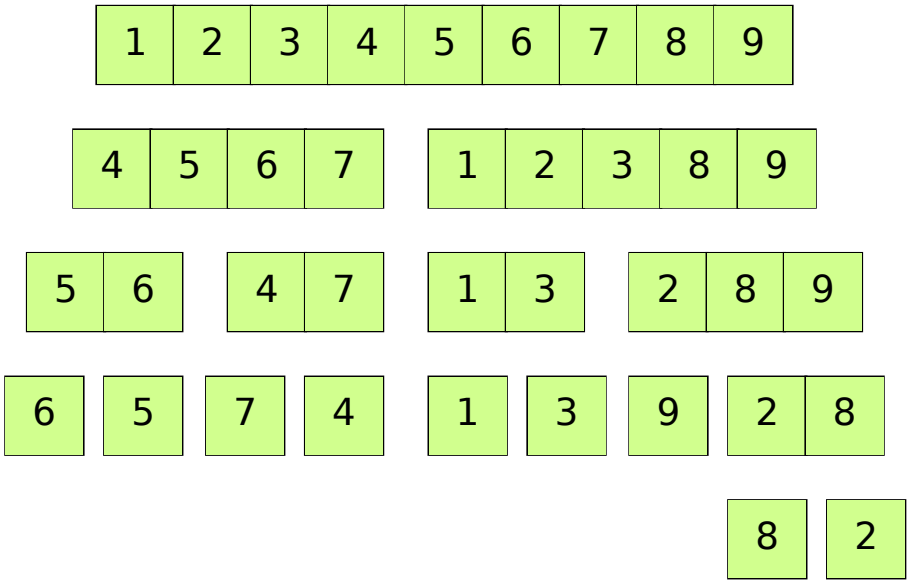
Auf die gleiche Weise wird auch die zweite Teilsequenz `1,3,9,8,2` unserer Eingabesequenz rekursiv sortiert.

Abschließend folgt der *Vereine*-Schritt des 1. Aufrufs, der die beiden sortierten Teilsequenzen `4,5,6,7` und `1,2,3,8,9` zur komplett sortierten Sequenz `1,2,3,4,5,6,7,8,9` zusammengespleißt.

Merge-Sort-Baum nach 1. Abstieg [Slide 4]

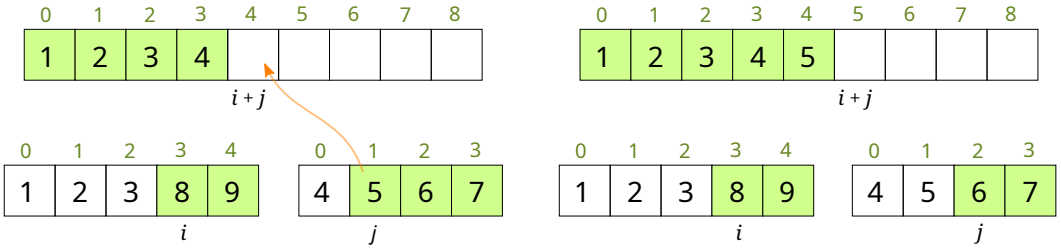


Merge-Sort-Baum nach Aufstieg [Slide 5]



`merge()` [Slide 6]

Algorithm `merge`(S_1, S_2, T)
Require: *sorted source arrays* S_1 *of length* n_1 *and* S_2 *of length* n_2*;*
target array T *of length* $n = n_1 + n_2$.
Ensure: T *contains the sorted elements of* S_1 *and* S_2.
$i \leftarrow 0,\ j \leftarrow 0$
while $i + j < n$ **do**
  if $j = n_2$ **or** $(i < n_1$ **and** $S_1[i] \leq S_2[j])$ **then**
    $T[i + j] \leftarrow S_1[i]$
    $i \leftarrow i + 1$
  else
    $T[i + j] \leftarrow S_2[j]$
    $j \leftarrow j + 1$



Erklärung

Sehen wir hier den `merge()`-Algorithmus im Detail. Im Mittelpunkt stehen zwei Indizes i in der Quellsequenz S_1 und j in der Quellsequenz S_2. Beide zeigen zunächst auf das erste Feld ihrer jeweiligen Sequenz.

In der Schleife wird nun bei jeder Iteration das kleinere verfügbare Element in das nächste Feld der Zielsequenz T kopiert und der Index dieses Elements inkrementiert. Da bei jeder Iteration entweder i oder j inkrementiert wird, ist der Index des zugehörigen Felds in der Zielsequenz $i + j$.

Ist einer der beiden Indizes am Ende seiner Sequenz angelangt, dann wird einfach der Rest der jeweils anderen Sequenz in die Zielsequenz kopiert.

Sie können sich leicht überlegen, dass die Laufzeit von `merge()` $O(n_1 + n_2)$ ist.

`mergeSort()` [Slide 7]

```
Algorithm mergeSort(A)
  Require: array A containing n keys.
  Ensure:  A is sorted.
  if n < 2 then
    return; // array is trivially sorted

  // divide
  m = ⌊n/2⌋
  A_1 ← A[0, ..., m − 1]
  A_2 ← A[m, ..., n − 1]

  // conquer
  mergeSort(A_1)
  mergeSort(A_2)

  // combine
  merge(A_1, A_2, A)
```

Erklärung

Nun ist es trivial, den `mergeSort()`-Algorithmus im Detail auszuformulieren.

Eine Sequenz, die weniger als zwei Elemente enthält, ist bereits sortiert. Ansonsten teilen wir die Sequenz auf zwei Teilsequenzen auf und rufen uns auf jeder der beiden rekursiv auf. Abschließend spleißen wir die beiden sortierten Teilsequenzen mittels des `merge()`-Algorithmus zu *einer* sortierten Sequenz zusammen.

Beachten Sie, dass der *Teile*-Schritt bei `mergeSort()` trivial ist. Vergleiche zwischen Elementen finden lediglich im *Vereine*-Schritt statt. Hier wird also die eigentliche Arbeit von `mergeSort()` getan.

Was ist die asymptotische Laufzeit von `mergeSort()`? Die Laufzeiten der *Teile*- und *Vereine*-Schritte sind jeweils $\Theta(n)$. Entscheidend sind also die beiden rekursiven Aufrufe.

Laufzeit von *Merge-Sort* [Slide 8]

- `merge()`: Alle elementaren Anweisungen haben konstante Laufzeit, und in jeder Iteration der `while`-Schleife wird ein Element kopiert. Die Gesamtlaufzeit ist also $\Theta(n_1 + n_2)$.
- Unter der Annahme $n = 2^k, k \in \mathbb{N}$:

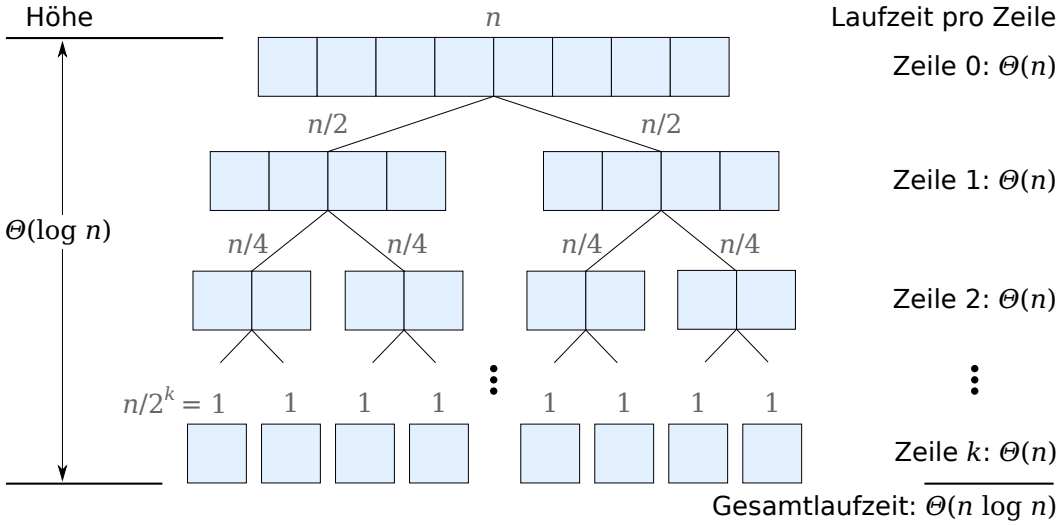
  Das Teilen der Eingangssequenz eines Knotens der Tiefe i dauert $\Theta(n/2^i)$. Es gibt 2^i solche Knoten.

  Die Gesamtzeit aller Teilungsoperationen in Tiefe i ist also $\Theta(n)$.
- Die Rekursionstiefe ist $\Theta(\log n)$.

  Beweis: Übung

  Da alle anderen Operationen in konstanter Zeit ablaufen, ist die Gesamtlaufzeit von `mergeSort()` $\Theta(n \log n)$.

Laufzeit von *Merge-Sort*: Aufrufbaum [Slide 9]



Erklärung

Hier hilft uns wieder der rekursive Aufrufbaum.

Nehmen wir an, die Eingabesequenz habe $n = 2^k$ Elemente für eine natürliche Zahl k. Das Ergebnis für diesen Spezialfall lässt sich dann leicht für beliebige Werte von n generalisieren.

Bei jedem rekursiven Aufruf wird die Sequenz in zwei gleich lange Hälften geteilt. Mit anderen Worten, von einer Zeile des Aufrufbaums zur nächsten verdoppelt sich die Anzahl Aufrufe, aber die Summe der Längen aller Sequenzen auf einer Zeile des Aufrufbaums bleibt immer gleich n.

Da die Laufzeiten der Teilungs- und Vereinigungsoperationen linear sind in der Anzahl der von ihnen behandelten Elemente, ist also die Gesamtlaufzeit auf jeder Zeile des Aufrufbaums $\Theta(n)$.

Es bleibt die Frage, wie viele Zeilen der Aufrufbaum hat. Diese ist jedoch schnell beantwortet. Da wir mit $n = 2^k$ Elementen beginnen, sich die Sequenzlängen von Zeile zu Zeile halbieren, und die rekursiven Aufrufe bei Sequenzlängen kleiner 2 enden, hat der Aufrufbaum $\Theta(\log n)$ Zeilen.

Damit ist die Gesamtlaufzeit von `mergeSort()` $\Theta(n \log n)$.

Stabiles Sortieren [Slide 10]

Ein Sortieralgorithmus ist ***stabil***, falls in der sortierten Ausgabesequenz alle identischen Schlüssel in derselben Reihenfolge vorliegen wie in der unsortierten Eingabesequenz.

Ist `mergeSort()` stabil bzw. lässt es sich auf einfache Weise stabil implementieren?

A: ja

B: nein

D: weiß nicht

2 *Quicksort*

Video 2 beginnt hier.

Teile und herrsche: *Quicksort* [Slide 11]

Sortiere eine Sequenz S der Länge n:

1. **Teile:** Ist $n \leq 1$, gib S zurück. Andernfalls wähle ein ***Pivotelement*** $x \in S$ (typischerweise das letzte), und verschiebe alle Elemente $s \in S$ nach

   L für $s < x$,
   E für $s = x$,
   G für $s > x$.

   Es gibt verschiedene Varianten von Quicksort. Der von uns betrachtete Algorithmus platziert genau ein element in E, und alle anderen Elemente $s \geq x$ in G.

2. **Herrsche:** Sortiere L und G rekursiv.

3. **Vereine:** Verkette L, E und G zu S.

   Bei *in-place Quicksort* ist dieser Schritt leer.

- Bei Quicksort findet die Hauptarbeit *vor* dem rekursiven Aufruf statt (im *Teile*-Schritt).
- Bei Merge-Sort findet die Hauptarbeit *nach* dem rekursiven Aufruf statt (im *Vereine*-Schritt).

Definition

Schauen wir uns nun einen anderen *Teile-und-herrsche*-Sortieralgorithmus an: `quickSort()`.

Im Gegensatz zu `mergeSort()` leistet `quickSort()` die Hauptarbeit im *Teile*-Schritt. Hier wird ebenfalls die Eingabesequenz aufgeteilt, aber nicht in der Mitte, sondern so, dass alle Elemente, die kleiner sind als ein so genanntes Pivot-Element, in der ersten Teilsequenz landen, alle, die mit dem Pivot-Element identisch sind, in einer zweiten, und alle, die größer sind, in einer dritten Teilsequenz.

Die erste und dritte Teilsequenz werden nun rekursiv sortiert. Abschließend müssen diese drei Teilsequenzen nur noch in der richtigen Reihenfolge trivial verkettet werden.

Erklärung

Es gibt viele verschiedene Varianten des `quickSort()`-Algorithmus, die sich in der Wahl des Pivot-Elements und in den Details der Sequenzaufteilung unterscheiden.

Im folgenden betrachten wir die wohl einfachste Variante. Sie wählt als Pivot-Element immer das *letzte* Element der Sequenz. Sie platziert dieses als *einziges* Element in E, und G enthält alle anderen Elemente größer als *oder gleich dem* Pivotelement.

Außerdem arbeitet sie *in place*, d.h. sie spart Speicherplatz und Kopieroperationen durch einen Verzicht auf explizite, externe Datenstrukturen. Die Elemente verbleiben während der gesamten Sortieroperation in demselben Array.

`quickSort()` [Slide 12]

```
Algorithm quickSort(A, l, h)
  Require: array A containing at least h + 1 keys, indices l ≥ 0 and h ≥ 0.
  Ensure: A[l, ..., h] is sorted.
  if l < h then
    p ← partition(A, l, h) // divide
    quickSort(A, l, p − 1) // conquer
    quickSort(A, p + 1, h)

Algorithm partition(A, l, h)
  Require: array A containing at least h + 1 keys, indices 0 ≤ l < h.
  Ensure: ∀i, l ≤ i < p : A[i] < A[p]; ∀j, p < j ≤ h : A[p] ≤ A[j]; return pivot p.
  i ← l
  for j ← l to h do
    if A[j] < A[h] then // A[h] is the pivot
      A[i] ↔ A[j]
      i ← i + 1
  A[i] ↔ A[h] // place pivot between lesser and greater/equal keys
  return i
```

Erklärung

Sehen wir uns diesen Algorithmus im Detail an.

Da er *in place* arbeitet, übergeben wir neben dem Array explizit den Anfang und das Ende der Teilsequenz, die der aktuelle Aufruf sortieren soll. Hierzu dienen die Indizes l für *low* und h für *high*.

Falls $l \not< h$, gibt es nichts zu sortieren. Andernfalls wird die zu sortierende Teilsequenz mittels des Algorithmus `partition()` um das Pivot-Element herum partitioniert, und anschließend ruft sich `quickSort()` rekursiv auf den beiden Partitionen auf.

Der Rückgabewert p von `partition()` ist hier der Index des Pivot-Elements im Array A *nach* der Partitionierung. Genauer gesagt, nach dem Aufruf von `partition()` sind alle Elemente in $A[l, ..., p-1]$ kleiner als $A[p]$, und alle Elemente in $A[p+1, ..., h]$ sind größer oder gleich $A[p]$.

Da dieser *Quicksort*-Algorithmus *in place* arbeitet, gibt es im *Vereine*-Schritt nichts zu tun, denn die sortierten Teilsequenzen liegen ja bereits korrekt angeordnet nebeneinander im Array A.

Der `partition()`-Algorithmus verwaltet zwei Indizes i und j, die zu Beginn beide am Anfang der zu partitionierenden Teilsequenz stehen. j durchläuft diese Teilsequenz von Anfang bis zum Ende, und vergleicht jedes Element mit dem Pivot, der sich in $A[h]$ am Ende der Teilsequenz befindet.

Ist das aktuelle Element $A[j]$ kleiner als der Pivot, dann werden $A[i]$ und $A[j]$ vertauscht, und i wird inkrementiert.

Auf diese Weise ist garantiert, dass sich *links* von i niemals Elemente befinden können, die größer als oder gleich dem Pivot sind.

Am Anfang ist $i = j$. Dies bleibt so lange der Fall, wie $A[j] < A[h]$ ist, da i und j gemeinsam inkrementiert werden.

Erst, wenn ein Element $A[j] \geq A[h]$ angetroffen wird, bleibt i dort stehen, und j wird allein inkrementiert. Wenn später ein Element $A[j]$ kleiner ist als der Pivot, dann wird es mit $A[i]$ vertauscht, und i bewegt sich wieder weiter.

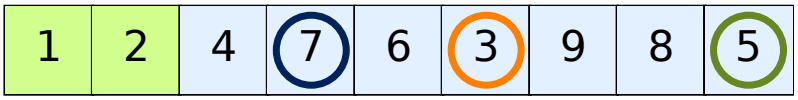
Also ist am Ende der Schleife entweder $i = j = h$ und die Partitionierung hat die Sequenz unverändert gelassen, oder $A[i]$ ist das erste Element der Teilsequenz, das größer als oder gleich dem Pivot ist.

Am Ende der Schleife können sich *rechts* von i keine Elemente mehr befinden, die kleiner

sind als der Pivot, denn diese sind ja alle mit $A[i]$ vertauscht worden.

Wir beenden also die Partitionierung, indem wir $A[i]$, nämlich das erste Element größer als oder gleich dem Pivot, mit dem Pivot-Element $A[h]$ vertauschen. Damit stehen in $A[l, ..., h]$ am Anfang alle Elemente, die kleiner sind als der Pivot, gefolgt vom Pivot-Element, gefolgt von allen Elementen, die größer als oder gleich dem Pivot sind.

Quicksort: $A[i] \leftrightarrow A[j]$ [Slide 13]



- Das hell grüne Subarray ist bereits sortiert; das hellblaue Subarray wird gerade sortiert.
- $A[j]$ (rot) ist kleiner als der Pivot $A[h]$ (dunkelgrün) und wird daher nun mit $A[i]$ (dunkelblau) vertauscht, damit alle $A[i] < A[h]$ links der $A[j] \geq A[h]$ zu liegen kommen.
- Anschließend wird i inkrementiert werden.

Quicksort: Beispiel [Slide 14]

Animation

Sehen wir nun diesen Quicksort-Algorithmus in Aktion. Wir sortieren dieselbe Sequenz wie im *Merge-Sort*-Beispiel. Das Pivot-Element an letzter Stelle der Sequenz ist hier grün markiert.

Die Indizes i und j sind beide durch dunkelblaue Ringe markiert. Zunächst bleibt $i = l = 0$, während j inkrementiert wird, bis es bei $A[j] = 1$ auf ein Element trifft, das kleiner ist als der Pivot 2. Deshalb färbt sich die Ringmarkierung von j nun rot. $A[i]$ und $A[j]$ werden vertauscht, und i wird inkrementiert. Dann beginnt die nächste Iteration der Schleife mit dem nächstgrößeren Wert von j.

Bei seiner Reise durch die Sequenz findet j kein weiteres Element, das kleiner ist als der Pivot. Die Schleife endet, sobald j h übersteigt.

Nun sehen wir, dass i in der Sequenz auf dem ersten Element steht, dass größer als oder gleich dem Pivot ist. Alle Elemente zu seiner Linken sind kleiner als der Pivot, und kein Element zu seiner Rechten ist kleiner als der Pivot. Wir vertauschen also nun $A[i]$ mit dem Pivot $A[h]$, und die Sequenz ist fertig partitioniert.

Wir können nun die Teilsequenzen links und rechts des Pivots unabhängig voneinander sortieren, und wir wissen, dass sich das Pivot-Element bereits im korrekten Feld des Arrays befindet. Daher wird es hier hellgrün hinterlegt.

Nun sortieren wir die linke Teilsequenz, die daher nun hellblau hervorgehoben wird. Diese umfasst jedoch nur ein einziges Element, weshalb der rekursive `quickSort()`-Aufruf unmittelbar zurückkehrt, denn diese Teilsequenz ist trivialerweise bereits fertig sortiert.

Nun sortieren wir die rechte Teilsequenz und beginnen mit der Partitionierung um den Pivot 5.

Das erste Element kleiner als der Pivot ist die 4. Diese wird mit der 7 vertauscht. Anschließend rückt i ein Feld vor, und j setzt seine Wanderung entlang der Sequenz fort. Unterwegs trifft es auf die 3, die mit der nun vorgerückten 7 vertauscht wird.

Keine weiteren Elemente sind kleiner als der Pivot $A[h] = 5$. Daher werden abschließend $A[i]$ und $A[h]$ vertauscht, die Partitionierung der rechten Teilsequenz ist beendet, und das Pivot-Element 5 befindet sich auf seinem endgültigen Feld.

Bei der Partitionierung der nächsten linken Teilsequenz `4,3` ist kein Element kleiner als der Pivot 3. Nach Ende der Schleife ist immer noch $i = l$, und die Vertauschung

von $A[i] = 4$ mit dem Pivotelement $A[h] = 3$ sorgt für das gewünschte Ergebnis der Partitionierung: Alle Elemente links des Pivot – hier gar keine – sind kleiner als der Pivot, und alle Elemente rechts des Pivot – hier nur die 4 – sind größer als oder gleich dem Pivot.

Die linke Teilsequenz ist hiermit leer, und die rechte Teilsequenz enthält lediglich ein Element und ist damit trivialerweise sortiert.

Nun muss nur noch die rechte Teilsequenz `7,9,8,6` der vorhergehenden Partitionierung sortiert werden.

Laufzeit von *Quicksort* [Slide 15]

- Die Zeit, die `quickSort()` in einem Knoten des *Quicksort*-Aufrufbaums verbringt (d.h. ein Aufruf ohne rekursive Aufrufe), ist linear in seiner Eingabegröße n.
- Sei s_i die Summe der Eingabegrößen aller Knoten der Tiefe i. Dann sind $s_0 = n$ und $s_i < s_{i-1}$, da mindestens das Pivotelement jedes Knotens nicht nach unten weitergereicht wird.
- Insgesamt ist die Laufzeit also $O(nh)$. Allerdings ist schlimmstenfalls $h = n-1$; im schlechtesten Fall ist die Laufzeit also $\Theta(n^2)$.

  Wann tritt der schlechteste Fall auf?

Erklärung

Was ist die asymptotische Laufzeit von *Quicksort*?

Die Argumentation ist ähnlich wie bei *Merge-Sort*: Wir schauen uns an, wie viel Zeit der Algorithmus in jeder Zeile des Aufrufbaums verbringt. Diese Zeit ist genau die Laufzeit des `partition()`-Algorithmus aufsummiert über alle Knoten der Zeile. Die Laufzeit von `partition()` ist linear in seiner Eingabegröße, und diese Eingabegrößen summieren sich in jeder Zeile des Aufrufbaums zur Gesamtlänge n der Sequenz.

Hiervon können wir die Pivotelemente abziehen, die bei den rekursiven Aufrufen ausgeschlossen werden. Dies hat jedoch keine Auswirkungen auf die asymptotische Laufzeit unserer Quicksort-Variante, da wir pro Aufruf lediglich eine konstante Anzahl Elemente einsparen, nämlich eines.

Folglich ist die Laufzeit von `quickSort()` in jeder Zeile des Aufrufbaums $O(n)$, und insgesamt $O(nh)$ bei einer Rekursionstiefe von h.

Falls die Pivots unglücklicherweise so fallen, dass immer eine der beiden Teilsequenzen der Partitionierungen leer ist, dann reduzieren wir die zu sortierende Sequenz bei jedem rekursiven Aufruf lediglich um das Pivotelement. In diesem Fall ist also $h = n - 1$, und damit die Laufzeit von `quickSort()` $\Theta(n^2)$ im schlechtesten Fall.

Verbesserte Wahl des Pivotelements [Slide 16]

- zufällig (*Randomized Quicksort*)

  Erwartete Laufzeit: $\Theta(n \log n)$
- Median dreier Werte (z.B. des ersten, mittleren und letzten Elements der Eingabesequenz)

  Laufzeit in der Praxis meist $\Theta(n \log n)$; keine Zufallszahlen erforderlich

Erklärung

Diese schlechte Laufzeit wird bei ungünstigen Konstellationen der Pivots erreicht. Insbesondere treten solche Konstellationen auf, wenn die Eingabesequenz bereits ganz oder teilweise sortiert ist, was in der Praxis recht häufig vorkommt.

Es ist also wichtig, die Pivots so zu wählen, dass die Teilsequenzen der Partitionierungen im Erwartungsfall möglichst gleich lang sind. Dies ist dann der Fall, wenn wir als Pivot den Median der Sequenz wählen. Die Bestimmung des Median erfordert jedoch eine Zeit von $\Theta(n)$. Eine in der Praxis oft gute Näherung ist der Median des ersten, mittleren und letzten Elements des Arrays. Dieser hängt nur von einer konstanten Anzahl direkt zugänglicher Elemente ab und lässt sich daher in konstanter Zeit berechnen.

Ein anderes, beliebtes Verfahren ist die zufällige Wahl des Pivots. Diese Variante ist als *Randomized Quicksort* bekannt und hat eine erwartete Laufzeit von $\Theta(n \log n)$.

Quiz [Slide 17]

Ist `Quicksort()` stabil bzw. lässt es sich auf einfache Weise stabil implementieren?

A: ja

B: nein

D: weiß nicht

3 Vergleichsbasiertes Sortieren: Minimale Laufzeit

Video 3 beginnt hier.

Vergleichsbasiertes Sortieren: Minimale Laufzeit [Slide 18]

Geht es schneller als $O(n \log n)$?

Zählen wir die Vergleichsoperationen, die beim Sortieren einer Sequenz n paarweise verschiedener Elemente ausgeführt werden:

- Jeder Sortiervorgang beinhaltet eine gewisse Folge von Vergleichsoperationen, die jeweils in *wahr* oder *falsch* resultieren.
- Die Gesamtheit der möglichen Vergleiche eines gegebenen Sortieralgorithmus können wir also als binären Entscheidungsbaum darstellen; der Sortiervorgang einer gegebenen Eingabepermutation folgt einem Pfad von der Wurzel bis zu einem Blatt.
- Es ist unmöglich, dass zwei *verschiedene* Eingabepermutationen *demselben* Pfad folgen.
- Es gibt $n!$ mögliche Permutationen; der Entscheidungsbaum muss also $n_E = n!$ Blätter besitzen.

$$
\begin{array}{rcll}
2^h & \geq & n_E & \text{wie wir bereits gezeigt haben} \\
h & \geq & \log n_E & \\
h & \geq & \log n! & \\
\log n! & \in & \Omega(n \log n) & \text{nach der Stirling-Näherungsformel}
\end{array}
$$

Erklärung

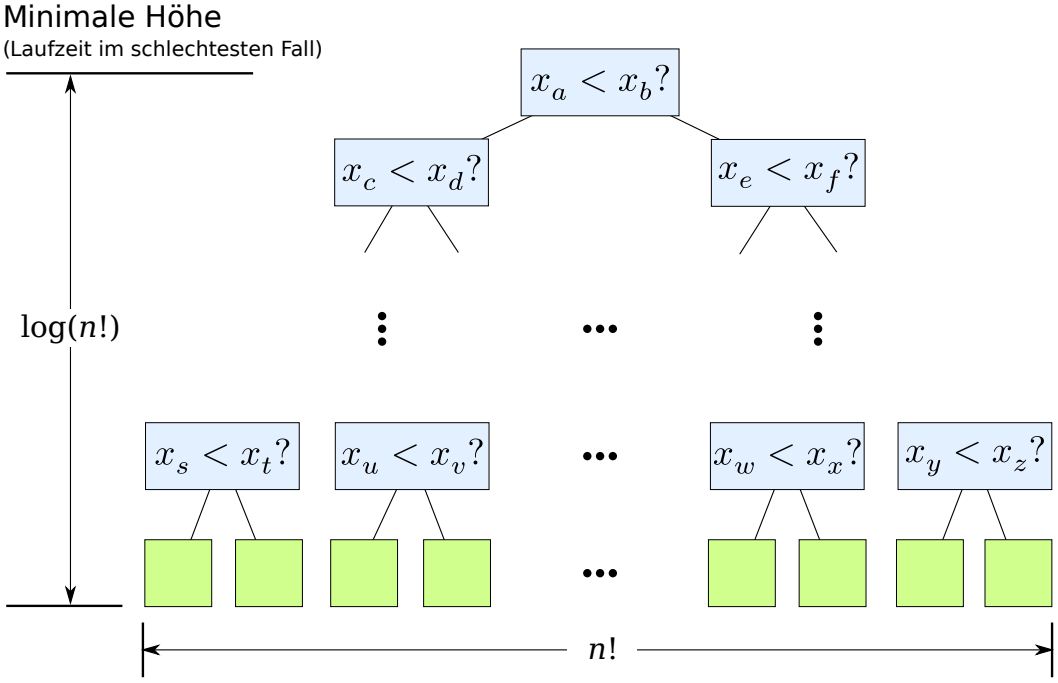
Wir haben nun verschiedene Sortieralgorithmen kennengelernt. Die schnellsten hatten eine Laufzeit von $O(n \log n)$, und die langsamsten von $O(n^2)$. Geht es schneller als $O(n \log n)$?

Wir skizzieren nun einen Beweis für die Tatsache, dass es keinen Sortieralgorithmus geben kann, der auf paarweisen Vergleichen beruht und eine bessere Laufzeit als $O(n \log n)$ garantiert.

Hierzu betrachten wir die Anzahl Vergleichsoperationen, die beim Sortieren einer Sequenz ausgeführt werden. Jeder Sortiervorgang einer Sequenz beinhaltet eine Folge von Vergleichsoperationen, die jeweils in *wahr* oder *falsch* resultieren.

Nun ist es unmöglich, dass beim Sortieren zweier *verschiedener* Sequenzen exakt *dieselbe* Folge von Vergleichen mit jeweils *denselben* Resultaten ausgeführt wird. Schließlich können nicht dieselben Vertauschungen zwei unterschiedliche Sequenzen mit demselben Ergebnis sortieren. Jede mögliche Permutation einer Sequenz muss beim Sortieren also seine *eigene* Folge von Vergleichen und deren Resultaten nach sich ziehen.

Folgen von Vergleichen als Entscheidungsbaum [Slide 19]



Erklärung

Die Gesamtheit aller möglichen Folgen von Vergleichsoperationen können wir als Entscheidungsbaum veranschaulichen. Die Wurzel repräsentiert den ersten Vergleich, der beim Sortiervorgang auftritt. Je nachdem, ob er negativ oder positiv ausfällt, folgen wir dem Pfad zum linken oder zum rechten Kind. Dieses repräsentiert den zweiten Vergleich, und so weiter. Ein Blatt markiert das Resultat des letzten Vergleichs.

Jeder Pfad von der Wurzel zu einem Blatt entspricht also einer Folge von Vergleichen und deren Ergebnissen. Da es sich um eine Baumstruktur handelt, repräsentiert jedes Blatt seinen eindeutigen Pfad von der Wurzel.

Eine Sequenz von n Elementen kann in $n!$ verschiedenen Permutationen angeordnet sein. Daher muss unser Entscheidungsbaum mindestens $n!$ Blätter besitzen, um jeder dieser Permutationen ein eigenes Blatt und damit eine eigene Folge von Vergleichs- und Vertauschungsoperationen zuordnen zu können.

Damit ein Binärbaum $n!$ Blätter haben kann, muss seine Höhe mindestens $\log(n!)$ betragen. Nach der Stirling-Näherungsformel, die wir hier nicht näher betrachten, ist $\log n! \in \Omega(n \log n)$.

Damit haben wir gezeigt, dass $\Omega(n \log n)$ Vergleiche erforderlich sind, um eine beliebige Sequenz der Länge n zu sortieren.

Etliche der Sortieralgorithmen, die wir betrachtet haben, besitzen also optimales asymptotisches Laufzeitverhalten.

4 Vergleichsbasiertes Sortieren: Gegenüberstellung

Video 4 beginnt hier.

Vergleich verschiedener Sortieralgorithmen [Slide 20]

Algorithmus	**Laufzeit**	**Eigenschaften**
Selection Sort	$\Theta(n^2)$	zu vermeiden
Insertion Sort	$O(n^2); O(n+m)$	*in place*; stabil möglich; gut für kurze oder *beinahe sortierte* Sequenzen (m klein)
Heap-Sort	$O(n \log n)$	*in place*; nicht stabil; gut für Sequenzen, die komplett in RAM/Cache passen
Quicksort	$O(n^2)$; $O(n \log n)$ erwartet	*in place* möglich; nicht stabil; in der Praxis oft unter den schnellsten
Merge-Sort	$\Theta(n \log n)$	stabil möglich; gut für lange Sequenzen (Cache-freundliche Zugriffsmuster)
Tim-Sort*	$O(n \log n)$	baut auf Merge-Sort und Insertion Sort; stabil; populärer Allround-Algorithmus

m ist die Anzahl *Inversionen*, d.h. die Anzahl der Elementpaare in umgekehrter Reihenfolge.

* Tim-Sort haben wir in der LV nicht behandelt, wie auch viele andere Sortieralgorithmen.

In place bedeutet nicht zwangsläufig *mit konstantem zusätzlichen Speicherbedarf.* Quicksort wird bspw. als *in-place*-Algorithmus betrachtet, benötigt jedoch zusätzlichen Speicher proportional zur Rekursionstiefe. Äquivalente nicht-rekursive Formulierungen benötigen äquivalenten Speicher für einen expliziten Stack.

Erklärung

Fassen wir hier einmal die verschiedenen Sortieralgorithmen zusammen, die wir betrachtet haben.

Selection Sort muss bei jedem Schritt die gesamte Quellsequenz nach dem kleinsten Element durchsuchen, unabhängig vom Inhalt dieser Sequenz. Damit ist seine Laufzeit $\Theta(n^2)$, selbst im besten Fall. Damit hebt sich dieser Algorithmus von allen anderen negativ ab, die wir betrachtet haben. Auch sonst spricht praktisch nichts für ihn. *Selection Sort* sollte also niemals verwendet werden.

Insertion Sort greift sich bei jeder Iteration das nächste Element der Quellsequenz, muss diese also nicht jedes Mal durchlaufen. Beim Einfügen in die sortierte Zielsequenz muss diese nur so weit durchlaufen werden, bis das richtige Feld gefunden ist. Im Idealfall ist dies gleich das erste, wo in einem zyklischen Array in konstanter Zeit eingefügt werden kann. Damit ist *Insertion Sort* im besten Fall $\Theta(n)$, was optimal ist. Im schlechtesten Fall sowie im Erwartungsfall erfordert die Suche jedoch $\Theta(n)$ Schritte. Damit ist die Gesamtlaufzeit von *Insertion Sort* $\Theta(n^2)$ im schlechtesten und im erwarteten Fall.

Sein Laufzeitverhalten lässt sich allgemein anhand der Anzahl m der *Inversionen* in der Quellsequenz charakterisieren. Eine Inversion ist ein Elementpaar, das in der verkehrten Reihenfolge auftritt. Eine sortierte Sequenz enthält 0 Inversionen, und eine in umgekehrter Reihenfolge sortierte Sequenz von n Elementen enthält n^2 Inversionen, denn jedes Element bildet mit jedem seiner Nachfolger eine Inversion. Insgesamt ist die Laufzeit von *Insertion Sort* $O(n+m)$. Die Abhängigkeit von n besteht, weil die Quellsequenz komplett durchlaufen werden muss, und m liegt irgendwo zwischen 0 und n^2. Da in der

Praxis vorkommende Sequenzen oft bereits teilweise sortiert sind und wegen der extremen Einfachheit des Algorithmus kann *Insertion Sort* durchaus brauchbar sein, insbesondere für kurze Sequenzen.

Insertion Sort lässt sich so implementieren, dass Elemente gleichen Schlüssels beim Sortierprozess ihre originale Reihenfolge beibehalten. Diese Eigenschaft bezeichnet man als ***Stabilität*** eines Sortieralgorithmus, und ist in der Praxis oft gefragt.

Darüber hinaus lässt sich *Insertion Sort in place* implementieren, also ohne explizite Zuhilfenahme externer Datenstrukturen, deren Größe von n abhängt.

Heap-Sort arbeitet ebenfalls *in place*, ist aber nicht stabil. Es garantiert eine Laufzeit von $O(n \log n)$, die sich auf $\Theta(n)$ reduziert, falls alle Schlüssel identisch sind.

Beim Herstellen der Heap-Ordnung verteilen sich die Zugriffsmuster über das gesamte Array. Daher ist *Heap-Sort* vor allem für kleine bis mittelgroße Datensätze geeignet, die komplett ins RAM bzw. in Prozessor-nahe Caches passen.

Die Laufzeit von *Quicksort* ist zwar $O(n^2)$ im schlechtesten Fall, lässt sich aber mit einfachen Mitteln auf $O(n \log n)$ im Erwartungsfall reduzieren, z.B. durch zufällige Wahl des Pivots. Wie *Heap-Sort* ist *Quicksort* nicht stabil und arbeitet *in place*. Im Gegensatz zu *Heap-Sort* ist *Quicksort* jedoch rekursiv und benötigt daher zusätzlichen Speicher proportional zur Rekursionstiefe.

Es gibt Varianten von *Quicksort*, die mit relativ wenigen Vergleichen auskommen und die Mehrfachbehandlung identischer Schlüssel effektiv vermeiden, so dass die Laufzeit im besten Fall auf $\Theta(n)$ gehalten werden kann, wenn alle Schlüssel identisch sind. Daher ist *Quicksort* in der Praxis oft am schnellsten.

Merge-Sort lässt sich auf natürliche Weise stabil implementieren. Da es hauptsächlich sequenziell auf die Elemente zugreift, profitiert es ausgezeichnet von Caches. Daher eignet es sich hervorragend für sehr große Datensätze, auch solche, die nicht komplett ins RAM geladen werden können. Ein Nachteil von *Merge-Sort* in seiner ursprünglichen Form ist, dass bereits sortierte Teilsequenzen sich nicht systematisch zur Beschleunigung nutzen lassen. Damit ist seine Laufzeit $\Omega(n \log n)$ auch im besten Fall.

Wir sehen, dass alle betrachteten Algorithmen ihre Vor- und Nachteile haben. *Selection Sort* ist der einzige klare Verlierer, aber es gibt keinen klaren Gewinner. Daher gibt es Bemühungen, verschiedene Basis-Algorithmen so zu kombinieren, dass deren Vorteile kombiniert und deren Nachteile aufgehoben werden. Ein solcher Algorithmus ist *Tim-Sort*. Aufbauend auf *Merge-Sort* und *Insertion Sort* sucht Tim-Sort bereits sortierte Teilsequenzen und nutzt diese zur Reduktion der Laufzeit. Tim-Sort wurde seinerzeit als Standard-Sortieralgorithmus für die Programmiersprache Python entwickelt, und erfreut sich bis heute großer Beliebtheit weit über Python hinaus.

5 Literatur

Literatur [Slide 21]

Goodrich, Michael, Roberto Tamassia und Michael Goldwasser (Aug. 2014). *Data Structures and Algorithms in Java*. Wiley.