INTERNET NETWORK ARCHITECTURES

Fortgeschrittene Sortierverfahren

Überblick

- Sortieren ist ein sehr intensiv untersuchtes Problem
- Es gibt eine große Zahl von Algorithmen mit jeweils verschiedenen Varianten
- Generell ordnet man Sortierverfahren in zwei Gruppen:
  1) Vergleichende Sortierverfahren
     A. Einfache Sortierverfahren
     B. Fortgeschrittene Sortierverfahren
  2) Nicht vergleichende Sortierverfahren

Überblick

- Einfache vergleichende Sortierverfahren
  - Sortieren durch Einfügen (insertion sort)
  - Sortieren durch Auswählen (selection sort)
  - Sortieren durch Vertauschen (bubble sort)
- Fortgeschrittene vergleichende Sortierverfahren
  - Sortieren durch Mischen (merge sort)
  - Sortieren mittels Heapify (heap sort)
  - Sortieren durch Gruppieren (quick sort)
- Nicht vergleichende Sortierverfahren
  - Sortieren durch Zählen (count sort)
  - Sortieren durch Fachverteilen (radix sort)

Fortgeschrittene Sortierverfahren

- Sortieren durch Gruppieren (quick sort)
  - Berühmt, schnell, breit einsetzbar
  - Nutzt das Prinzip „Teile und Herrsche"
- Arbeitsweise
  - Die Folge wird in zwei Teilfolgen gruppiert (partition), so dass jedes Element der ersten Folge kleiner ist als jedes Element der zweiten.
  - Auf diese Teilfolgen wird das Prinzip dann rekursiv angewendet.
  - Die Rekursion endet, wenn man bei einelementigen Teilfolgen angelangt ist.
  - Als Trennelement (pivot) für das Gruppieren kann ein beliebiges Folgenelement verwendet werden,
z.B. das Letzte, das Erste, oder das Mittlere.

Quicksort: Prinzip

Anfangssituation

start ↓ … pivot, end ↓

x x x x x x x x

Nach Gruppieren:

start ↓ … part ↓ … pivot, end ↓

x x x x | x x x | x

$\underbrace{x\ x\ x\ x}_{\forall i: f_i < f_{pivot}}$ $\underbrace{x\ x\ x}_{\forall i: f_i \geq f_{pivot}}$

vertauschen

Nach Vertauschen:

start ↓ … part ↓ … end ↓

x x x x | x | x x x

$\underbrace{x\ x\ x\ x}_{\forall i: f_i < f_{part}}$ x $\underbrace{x\ x\ x}_{\forall i: f_i \geq f_{part}}$

liegt schon richtig

Komplexität von Quicksort

- Wenn es gelingt, die Teilfolgen durch das Gruppieren jeweils zu halbieren, erhalten wir eine Komplexität von: $T_{bc} = O(n \log n)$

- Wird jeweils nur eine Teilfolge konstanter Länge abgespalten, so wird n mal gruppiert und daher: $T_{wc} = O(n^2)$

- Im Durchschnitt ist allerdings die Länge der Intervalle abhängig von n und man braucht nur zweimal mehr Operationen als im besten Fall:

$$T_{ac} = O(n \log n)$$

Quicksort-Varianten

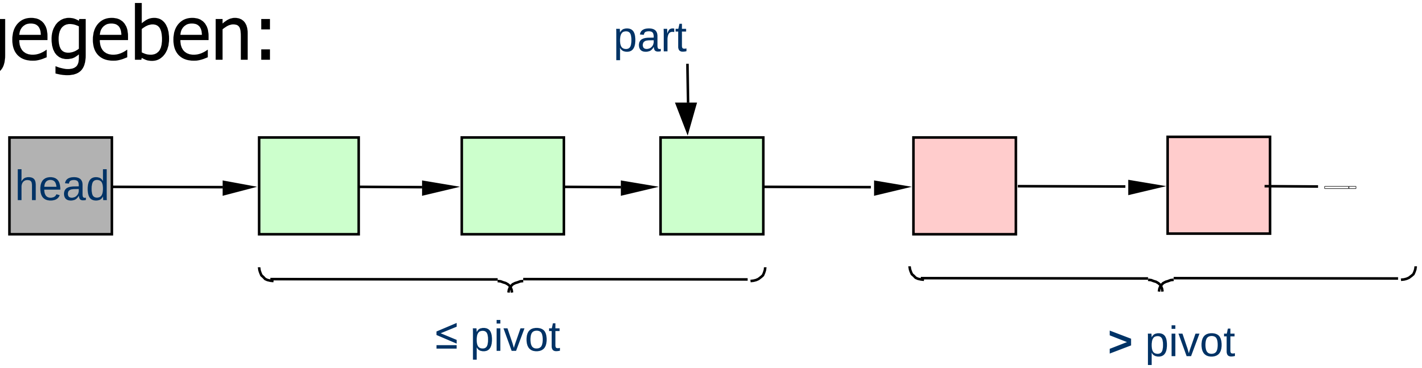
- Vorgeschlagene Varianten betreffen die Wahl des Trennwerts für das Gruppieren, der in der vorliegenden Form für das schlechte Verhalten bei nahezu sortierten Folgen verantwortlich ist:
- Trennindex zufällig aus $[start, end]$
- Trennwert als Median aus drei Elementen $\left\{ f_{start}, f_{\left[\frac{end-start}{2}\right]}, f_{end} \right\}$
- Beide Varianten haben sich gut bewährt und sorgen auch bei fast sortierten Folgen für O(n log n).
- Es ist außerdem vorteilhaft, die kleinere Teilfolge zuerst zu sortieren (Rekursionstiefe minimieren).

Quicksort mit Listen: Gruppieren

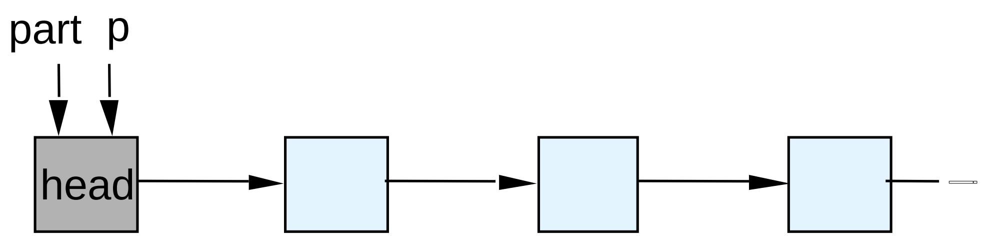
- Zerlegen einer Folge in zwei Teilfolgen, wovon die eine die kleineren (≤) Elemente, die andere die größeren Elemente enthält.
  - Eingabe ist der Trennwert pivot.
  - Ausgabe ein Zeiger auf das Element vor der Gruppengrenze (part).
  - Statt des Vertauschens (swap) findet ein Versetzen (move) statt.
  - Die Komplexität des nachfolgenden Gruppierungsverfahren ist wie bei der dichten Speicherung O(n), da die Liste genau einmal abgelaufen wird und jeweils konstanter Aufwand entsteht.

Gruppieren

- Der Trennwert `pivot` wird als Parameter vorgegeben, das Ende der ersten Gruppe wird als Zeiger `part` zurückgegeben:

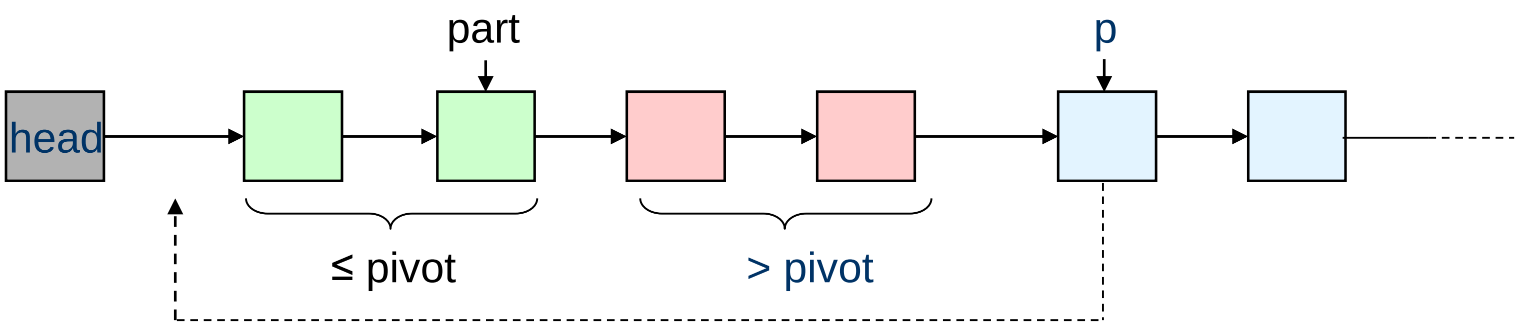


- Initialisierung:



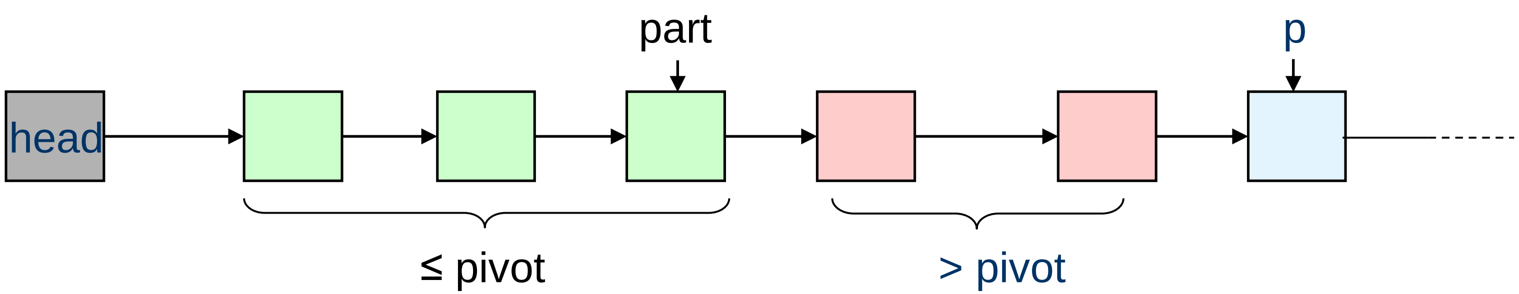
Gruppieren

- Fall 1: p.data <= pivot



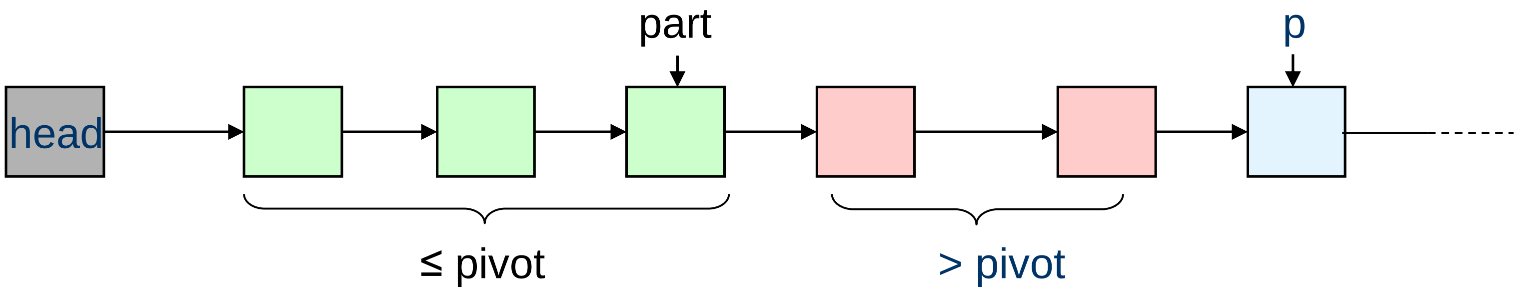
Gruppieren

- Fall 1: p.data <= pivot

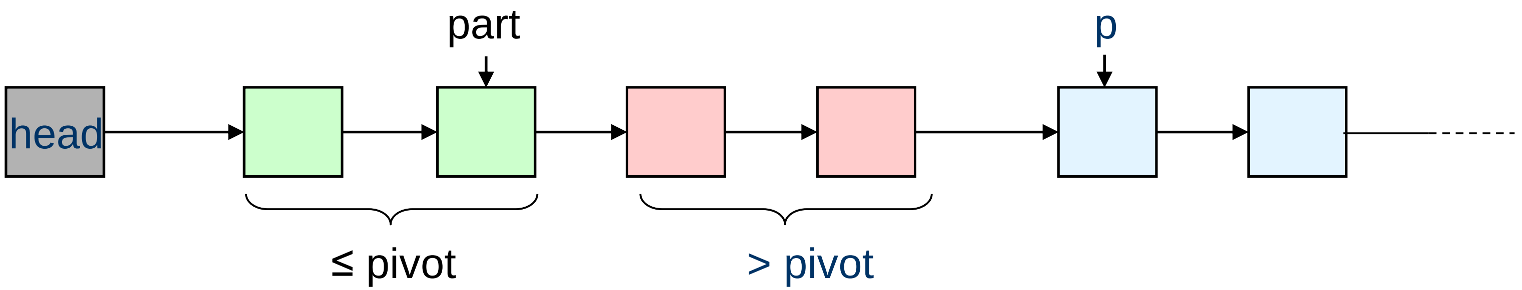


Gruppieren

- Fall 1: p.data <= pivot

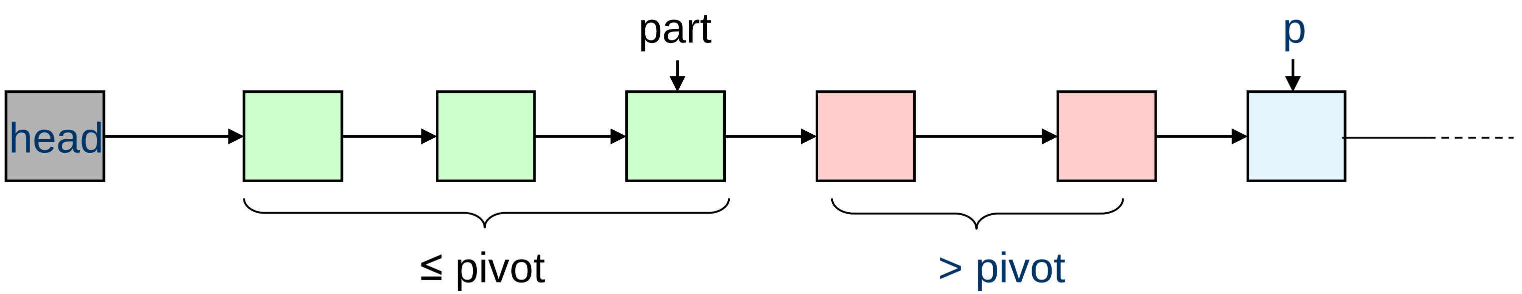


- Fall 2: p.data > pivot:

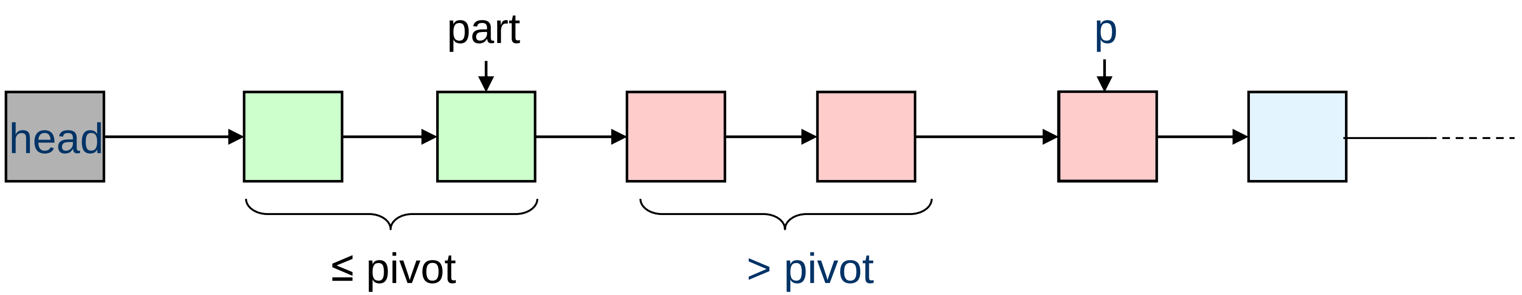


Gruppieren

- Fall 1: p.data <= pivot

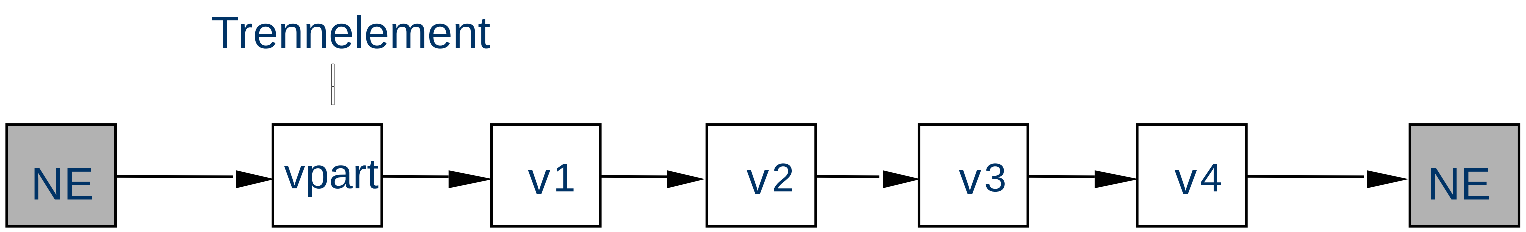


- Fall 2: p.data > pivot:



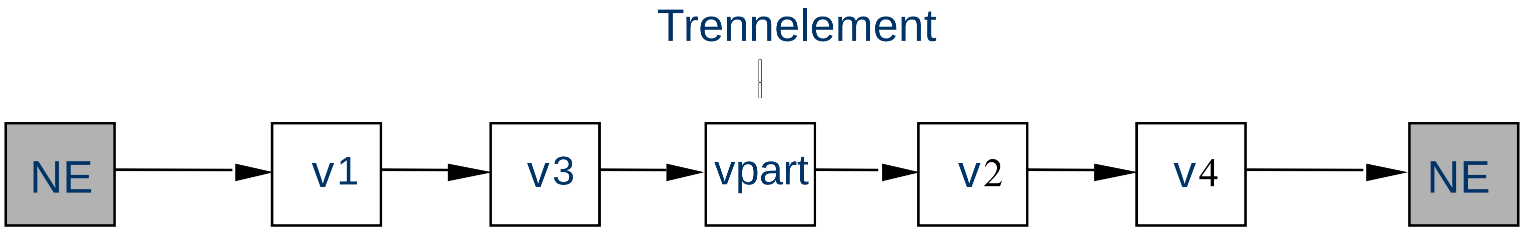
Sortieren durch Gruppieren (Quicksort)

- Komplexität: $T_{avg} = O(n \log n)$.
- Greedy: wir wählen den Wert des ersten Folgenelements als Trennwert.
- Vor Gruppieren:

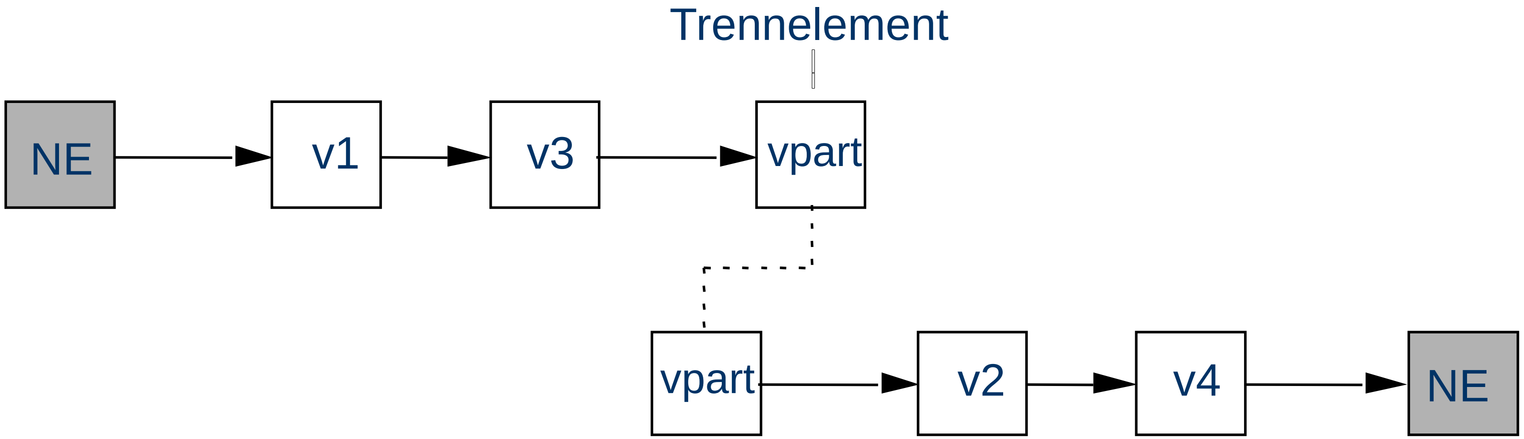


Quicksort

- Nach Gruppieren:



- Die beiden Teillisten:



Sortieren

- Standard Sortierverfahren (Bubble-sort, Insertion-sort, Selection-sort, Quicksort, Merge-sort) sind nicht nur auf Arrays sondern auch auf verkettete Listen anwendbar.
  - Die Komplexität ist dieselbe.
- Lediglich die auf Indexrechnung beruhenden schnellen Verfahren (count sort) können nicht verwendet werden.

Anmerkungen: Quicksort vs. Merge-Sort

- Beide Verfahren, Quicksort und Merge-Sort, benötigen zusätzlichen Speicher
- Merge-Sort, weil das Mischen in linearer Zeit nur mit einem zweiten Array möglich ist.
- Quicksort, weil die rekursiven Aufrufe den Programmstapel (program stack) erheblich anwachsen lassen.
  - (Dies kann man mildern, wenn man das Verfahren so umorganisiert, dass kürzere Teilfolgen immer zuerst bearbeitet werden.)
- Ein Sortierverfahren mit dem gleichen asymptotischen Aufwand und ohne zusätzlichen Speicherbedarf ist Heapsort.

KOMPLEXITÄT VON VERGLEICHENDEN SORTIERVERFAHREN

Komplexität von vergleichenden Sortierverfahren

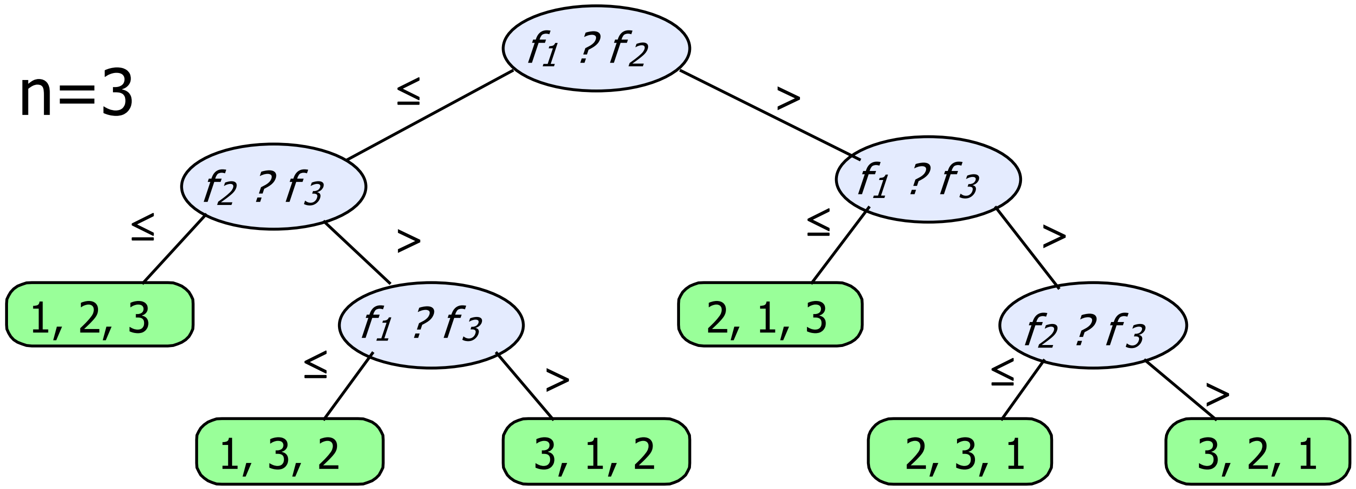
- Alle bisher besprochenen Sortieralgorithmen gehören zur Klasse der Vergleichssortierverfahren
  - Elementaroperation ist der paarweise Größenvergleich
- Damit ergibt sich Folgendes:
- Satz: Jedes Sortierverfahren, das auf dem paarweisen Vergleich von Elementen beruht, hat eine Komplexität von: $\Omega(n \log n)$

Komplexität von vergleichenden Sortierverfahren

- Alle bisher besprochenen Sortieralgorithmen gehören zur Klasse der Vergleichssortierverfahren
  - Elementaroperation ist der paarweise Größenvergleich
- Der Ablauf kann in einem binären Entscheidungsbaum wie folgt dargestellt werden:
  - Jeder interne Knoten entspricht dem Vergleich zweier Elemente.
  - Die jeweiligen Teilbäume enthalten die noch erforderlichen restlichen Vergleiche, so dass eine Sortierung gelingt.
  - Jedes Blatt repräsentiert eine der n! möglichen Permutationen.
  - Der Ablauf einer konkreten Sortierung entspricht einem Pfad von der Wurzel bis zu einem Blatt.

Entscheidungsbaum

- Beispiel für n=3



- Der längste Pfad von der Wurzel bis zu einem Blatt (= Höhe des Baumes) kennzeichnet die größtmögliche Anzahl von Vergleichen und damit die Laufzeit des Algorithmus.
  - Die Worst-case-Komplexität aller solcher vergleichsbasierter Sortierverfahren entspricht daher der Höhe des Baumes.
  - Eine untere Schranke für die Höhe des Entscheidungsbaums ist daher auch eine untere Schranke für jedes vergleichsbasierte Sortierverfahren.

Höhe des Entscheidungsbaums

- Wie groß ist die Höhe eines Entscheidungsbaumes für n Elemente mindestens?
- Es gibt n! Permutationen, also mindestens n! Blätter.
- Ein Binärbaum der Höhe h hat höchstens 2^h Blätter, daher gilt $n! \leq 2^h$
- Durch Logarithmieren $h \geq \log(n!)$
- Nach der Stirlingschen Formel gilt $n! > \left(\frac{n}{e}\right)^n$
- Also $h \geq \log\left(\left(\frac{n}{e}\right)^n\right) = n\log n - n\log e = O(n\log n)$

Vergleichende Sortierverfahren

- Satz: Jedes Sortierverfahren, das auf dem paarweisen Vergleich von Elementen beruht, hat eine Komplexität von: $\Omega(n \log n)$

Überblick

- Einfache Sortierverfahren
  - Sortieren durch Auswählen (selection sort)
  - Sortieren durch Vertauschen (bubble sort)
  - Sortieren durch Einfügen (insertion sort)
- Fortgeschrittene Sortierverfahren
  - Sortieren durch Gruppieren (quick sort)
  - Sortieren durch Mischen (merge sort)
- Nicht vergleichende Sortierverfahren
  - Sortieren durch Zählen (count sort)
  - Sortieren durch Fachverteilen (radix sort)

NICHT VERGLEICHENDE SORTIERVERFAHREN

Schnelle, digitale Sortierverfahren

- Die bisher diskutierten Sortierverfahren beruhen auf dem paarweisen Vergleich der Werte der einzelnen Elemente.
- Eine Verbesserung der asymptotischen Komplexität dieser Verfahren ist nicht möglich.
  (Siehe untere Schranke von $\Omega(n \log n)$)

Schnelle, digitale Sortierverfahren

- Unter gewissen Einschränkungen des Wertebereichs können die Werte dazu verwendet werden, den endgültigen Platz direkt anzusteuern.
  - Sortieren durch Zählen (count sort)
  - Sortieren durch Fachverteilen (radix sort)
- Diese Verfahren sind jedoch nicht immer sinnvoll einsetzbar, z.B. wenn
  - Das Sortieren stabil sein soll
  - Der Wertebereich zu groß ist

Wiederholung: Sortieren durch Zählen (count sort)

- Annahme:
  - Die Werte stammen aus einem kleinen Wertebereich, d.h. sie liegen so dicht, dass sie zum Indizieren eines Arrays verwendet werden können.
  - Es ist wahrscheinlich, dass Werte mehrfach auftreten.
- Idee
  - Die Häufigkeit jedes Elements wird ermittelt und daraus wird die endgültige Lage im Zielarray berechnet (streuendes Umspeichern).
  - Zum Schluss kann die Folge in das ursprüngliche Array zurückkopiert werden.

Beispiel: Count-sort

j:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A_in:	3	1	3	3	5	7	2	8	9	7	6	3	8	1	8	8

j:	1	2	3	4	5	6	7	8	9	10	Werte
C:	2	1	4	0	1	1	2	4	1	0	Zähler
	1	3	4	8	8	9	10	12	16	17	Position

i

A_out:	1	1	2	3	3	3	3	5	6	7	7	8	8	8	8	9
k:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Komplexität: Count-sort

- Anzahl der Schritte
  - Linear in der Zahl der Elemente n
  - Linear in der Zahl der Werte N
- Also O(n), wenn N = O(n)
- Problem
  - In vielen praktischen Fällen ist der Wertebereich viel größer als die Zahl der Elemente

Sortieren durch Fachverteilen (radix sort)

- Idee:
  - Wir wenden count sort nacheinander auf die Stellen der Werte an
  - Begonnen wird bei der kleinsten signifikanten Stelle
  - Da count sort stabil ist, bleibt die Ordnung bezüglich der weniger signifikanter Stellen erhalten
- Voraussetzungen:
  - Werte können in einem Alphabet dargestellt werden
  - Die Buchstaben des Alphabets sind total geordnet
  - Typische Alphabete: Binär, Dezimal, Hexadezimal

Sortieren durch Fachverteilen (radix sort)

- Beispielfolge (Dezimal): 333, 78, 77, 3, 37, 38
- Sortieren nach der Einerstelle: 333, 3, 77, 37, 78, 38
  - 2 x 3 (333, 3), 2 x 7 (77, 37), 2 x 8 (78, 38)
- Sortieren nach der Zehnerstelle: 3, 333, 37, 38, 77, 78
  - 1 x 0 (3), 3 x 3 (333, 37, 38), 2 x 7 (77, 78)
- Sortieren nach den Hunderterstelle: 3, 37, 38, 77, 78, 333
  - 5 x 0 (3, 37, 38, 77, 78), 1 x 3 (333)
- Notwendige Größe für das Zählerfeld ist immer 10

Sortieren durch Fachverteilen (radix sort)

- Komplexität
  - Annahmen:
    - Der Größte Wert ist N
    - Die Größe des Wertebereichs für eine Stelle ist b
  - Laufzeit:
    - Dann ist die Anzahl der Stellen proportional zu $\log_b N$
    - Für jede Stelle wird count sort verwendet: d.h. der Aufwand pro Stelle ist O(n)
    - Also insgesamt O(n log N), mit typischerweise log N << n