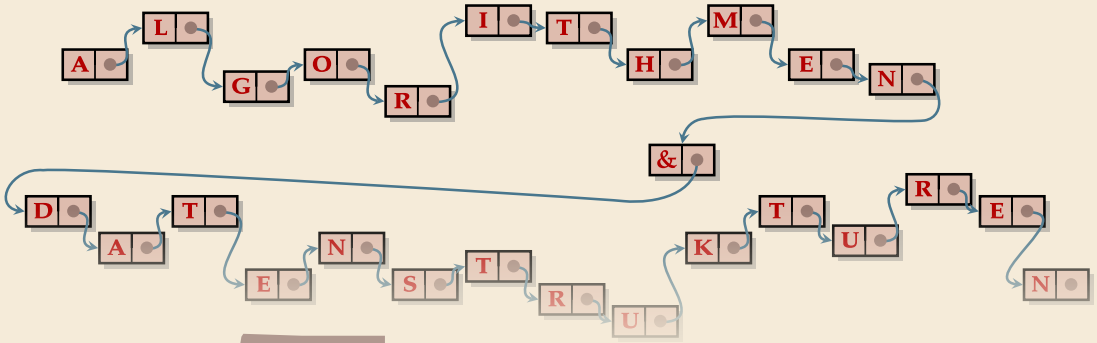


7 Sortieren

Algorithmen & Datenstrukturen · Sommersemester 2026

Prof. Dr. Sebastian Wild

CS210 (Sommersemester 2026)
Philipps-Universität Marburg
version 2026-05-27 00:08 H

Outline

7 Sortieren

7.1 Was bedeutet sortiert?

Sortieren

Gegeben: Array $A[0..n)$ von n Objekten, *Sortierkriterium*

Ziel: Array so permutieren, dass Objekte aufsteigend sortiert sind.

Beispiele:

- Hochschulen in Deutschland nach Gründungsjahr
  `https://de.wikipedia.org/wiki/Liste_der_Hochschulen_in_Deutschland`
- Züge nach Abfahrtszeit
  `https://www.bahnhof.de/marburg-lahn/abfahrt`
- Index in Büchern
- Arbeitsgruppen des FB12, alphabetisch nach Nachnamen der Professor:innen
  `https://www.uni-marburg.de/de/fb12/arbeitsgruppen`
- ...

Typischer Fall: verschiedene Sortierkriterien denkbar/möglich
$\rightsquigarrow$ muss mit angegeben werden

Sample sort client 1

Goal. Sort any type of data.

Ex 1. Sort random real numbers in ascending order.

seems artificial (stay tuned for an application)

```
public class Experiment
{
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      Double[] a = new Double[N];
      for (int i = 0; i < N; i++)
         a[i] = StdRandom.uniform();
      Insertion.sort(a);
      for (int i = 0; i < N; i++)
         StdOut.println(a[i]);
   }
}
```

```
% java Experiment 10
0.08614716385210452
0.09054270895414829
0.10708746304898642
0.21166190071646818
0.363292849257276
0.460954145685913
0.5340026311350087
0.7216129793703496
0.9003500354411443
0.9293994908845686
```

Sample sort client 2

Goal. Sort any type of data.

Ex 2. Sort strings in alphabetical order.

```
public class StringSorter
{
   public static void main(String[] args)
   {
      String[] a = StdIn.readAllStrings();
      Insertion.sort(a);
      for (int i = 0; i < a.length; i++)
         StdOut.println(a[i]);
   }
}
```

```
% more words3.txt
bed bug dad yet zoo ... all bad yes

% java StringSorter < words3.txt
all bad bed bug dad ... yes yet zoo
[suppressing newlines]
```

Sample sort client 3

Goal. Sort any type of data.

Ex 3. Sort the files in a given directory by filename.

```
import java.io.File;

public class FileSorter
{
   public static void main(String[] args)
   {
      File directory = new File(args[0]);
      File[] files = directory.listFiles();
      Insertion.sort(files);
      for (int i = 0; i < files.length; i++)
         StdOut.println(files[i].getName());
   }
}
```

```
% java FileSorter .
Insertion.class
Insertion.java
InsertionX.class
InsertionX.java
Selection.class
Selection.java
Shell.class
Shell.java
ShellX.class
ShellX.java
```

Totale Ordnung

Damit Sortieren wohldefiniert ist, benötigen wir ein sinnvolles Sortierkriterium.

Informell: Alle Objekte sind vergleichbar und Vergleiche ergeben keine Widersprüche.

Formal: Für eine Menge U (Universum) ist eine Relation $\leq \subset U \times U$ eine *totale Ordnung*, falls folgende Bedingungen erfüllt sind

1. **Reflexivität**: $\forall x \in U : x \leq x$
2. **Antisymmetrie**: $\forall x, y \in U : x \leq y \wedge y \leq x \implies x = y$
3. **Transitivität**: $\forall x, y, z \in U : x \leq y \wedge y \leq z \implies x \leq z$
4. **Totalität**: $\forall x, y \in U : x \leq y \vee y \leq x$

Anti-Beispiele:

- Schere-Stein-Papier Gewinne ↯ Transitivität
- $n \mid m$ (Teilbarkeit, „n teilt m“) ↯ Totalität

Callbacks

Goal. Sort any type of data (for which sorting is well defined).

Q. How can `sort()` know how to compare data of type `Double`, `String`, and `java.io.File` without any information about the type of an item's key?

Callback = reference to executable code.

- Client passes array of objects to `sort()` function.
- The `sort()` function calls object's `compareTo()` method as needed.

Implementing callbacks.

- Java: interfaces.
- C: function pointers.
- C++: class-type functors.
- C#: delegates.
- Python, Perl, ML, Javascript: first-class functions.

Callbacks: roadmap

client

```
public class StringSorter
{
   public static void main(String[] args)
   {
      String[] a = StdIn.readAllStrings();
      Insertion.sort(a);
      for (int i = 0; i < a.length; i++)
         StdOut.println(a[i]);
   }
}
```

data-type implementation

```
public class String
implements Comparable<String>
{
   ...
   public int compareTo(String b)
   {
      ...
      return -1;
      ...
      return +1;
      ...
      return 0;
   }
}
```

Comparable interface (built in to Java)

```
public interface Comparable<Item>
{
   public int compareTo(Item that);
}
```

sort implementation

```
public static void sort(Comparable[] a)
{
   int N = a.length;
   for (int i = 0; i < N; i++)
      for (int j = i; j > 0; j--)
         if (a[j].compareTo(a[j-1]) < 0)
            exch(a, j, j-1);
         else break;
}
```

key point: no dependence on `String` data type

Comparable API

Implement `compareTo()` so that `v.compareTo(w)`

- Defines a total order.
- Returns a negative integer, zero, or positive integer
  if v is less than, equal to, or greater than w, respectively.
- Throws an exception if incompatible types (or either is `null`).

less than (return -1) equal to (return 0) greater than (return +1)

Built-in comparable types. `Integer`, `Double`, `String`, `Date`, `File`, ...

User-defined comparable types. Implement the `Comparable` interface.

Implementing the Comparable interface

Date data type. Simplified version of `java.util.Date`.

```
public class Date implements Comparable<Date>
{
   private final int month, day, year;

   public Date(int m, int d, int y)
   {
      month = m;
      day   = d;
      year  = y;
   }

   public int compareTo(Date that)
   {
      if (this.year  < that.year ) return -1;
      if (this.year  > that.year ) return +1;
      if (this.month < that.month) return -1;
      if (this.month > that.month) return +1;
      if (this.day   < that.day  ) return -1;
      if (this.day   > that.day  ) return +1;
      return 0;
   }
}
```

Comparable vs. Comparator

Für die Eingangsbeispiele wäre `Comparable` nicht bequem!

- Möchten oft die **gleichen** Objekte nach unterschiedlichen Kriterien sortieren
- `Comparable` erlaubt ja aber nur **eine** Implementierung von `compareTo`!

⇝ Java erlaubt alternativ, den Sortiermethoden ein **`Comparator`**-Objekt zu übergeben

```java
public interface Comparator<T> {
    int compare(T v, T w);
}
```

Gleiche Semantik:

$$\texttt{c.compare}(v, w) \bowtie 0 \quad \equiv \quad v \bowtie w \iff v - w \bowtie 0$$

7.2 Primitive Sortieralgorithmen

Sortieralgorithmen

Obwohl das Ergebnis eindeutig ist, gibt es *viele* Möglichkeiten für Sortieralgorithmen.

Wir beginnen mit den naheliegendsten und ältesten Ideen.

(siehe Sedgewick & Wayne für mehr Details für diesen Abschnitt)

Diese entstammen oft auch manuellen Verfahren zum Sortieren physischer Objekte

⇝ Typisches Phänomen: Verfügbarkeit und Effizienz von Zugriffsoperationen bestimmt welcher Sortieralgorithmus schneller ist

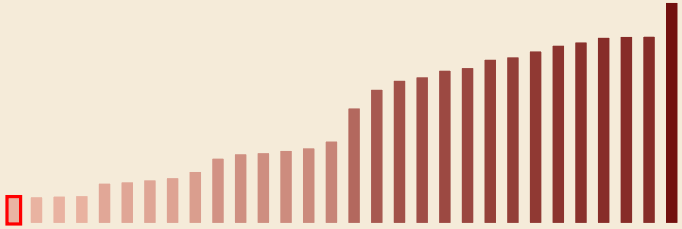
Verwenden hauptsächlich folgende Elementaroperationen

- `less`(v, w) um zu testen of $v < w$ (mittels `Comparator`)
- `swap`(i, j) tauscht $A[i]$ und $A[j]$

```
1 static <Elem> boolean less(final Elem v, final Elem w, final Comparator<Elem> c) {
2     return c.compare(v, w) < 0;
3 }
4
5 static <Elem> void swap(final Elem[] A, final int i, final int j) {
6     Elem tmp = A[i]; A[i] = A[j]; A[j] = tmp;
7 }
```

Selectionsort

Idee: In einer sortierten Liste ist das **Minimum** an erster Stelle.
Finde iterative das Minimum und entferne es.



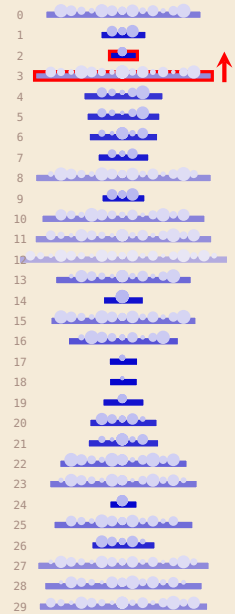
Code: https://visualgo.net/en/sorting?mode=Selection

```
1  static <Elem> void selectionSort(Elem[] A, Comparator<Elem> c) {
2      int n = A.length;
3      for (int i = 0; i < n; ++i) {
4          Elem min = A[i]; int minPos = i;
5          for (int j = i+1; j < n; ++j)
6              if ( less(A[j], min, c) ) {
7                  min = A[j]; minPos = j;
8              }
9          swap(A, i, minPos);
10     }
11 }
```

Korrektheit:
Invariante: $A[0..i)$ enthält die i kleinsten Elemente in sortierter Reihenfolge

Analyse:
Anzahl Vergleiche $\sim \frac{1}{2}n^2$

Bubblesort



Idee: In paarweisen Vergleichen steigt das leichtere Element wie eine Luftblase nach oben. Verfahre in Iterationen über die Eingabe.

in viskosem Gel

Code: https://visualgo.net/en/sorting?mode=Bubble (gespiegelt)

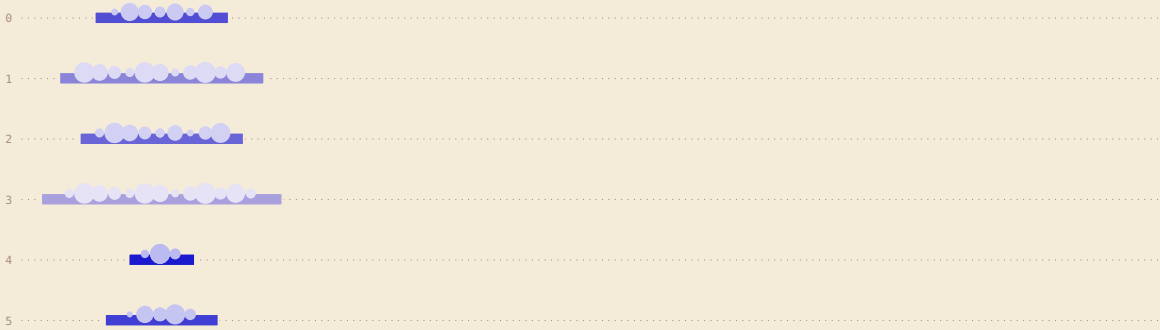
```
static <Elem> void bubbleSort(Elem[] A, Comparator<Elem> c) {
    int n = A.length;
    for (int i = 0; i <= n-2; ++i)
        for (int j = n-1; j >= i+1; --j)
            if (less(A[j],A[j-1],c)) swap(A,j,j-1);
}
```

Korrektheit:
Mit jeder Vertauschung näher an sortiert.
Nach jeder Iteration der äußeren Schleife erreicht nächstes Minimum die Oberfläche und kann von Betrachtung ausgeschlossen werden.

Analyse:
Anzahl Vergleiche $\sim \frac{1}{2}n^2$

Bubblesort – Beispiel

```
1 static <Elem> void bubbleSort(Elem[] A, Comparator<Elem> c) {
2     int n = A.length;
3     for (int i = 0; i <= n-2; ++i)
4         for (int j = n-1; j >= i+1; --j)
5             if (less(A[j], A[j-1], c)) swap(A, j, j-1);
6 }
```



Verbesserter Bubblesort

Obiger Code ist oftmals verschwenderisch / blind!

Beobachtung: Falls in voriger Iteration die letzte Vertauschung zwischen $A[t]$ und $A[t-1]$ stattfand, enthält $A[0..t)$ die kleinsten t Elements in sortierter Reihenfolge.

```
1  static <Elem> void bubbleSortImproved(Elem[] A, Comparator<Elem> c) {
2      int n = A.length; int t = 0;
3      while (t != n) {
4          int i = t; t = n;
5          for (int j = n-1; j >= i+1; --j)
6              if (less(A[j], A[j-1], c)) {
7                  swap(A, j, j-1);
8                  t = j;
9              }
10     }
11 }
```

Korrektheit: Invariante (Z.3): $A[0..t)$ enthält die t kleinsten Elemente in sortierter Reihenfolge

Analyse: später 😉

https://www.wild-inter.net/teaching/ads/bubblesort

Insertionsort

Idee: Kartenspielermethode: Füge Elemente suzessive in sortierte „Hand“ ein.
Hand = Präfix des Arrays
Einfügen = nach links Vertauschen, bis größer

Code:

```
1 static <Elem> void insertionSort(Elem[] A, Comparator<Elem> c) {
2     int n = A.length;
3     for (int i = 1; i < n; ++i)
4         for (int j = i;  j > 0 && less(A[j], A[j-1], c);  --j)
5             swap(A, j, j-1);
6 }
```

https://visualgo.net/en/sorting?mode=Insertion

Korrektheit:
Invariante: $A[0..i)$ sortiert (in Z.3)

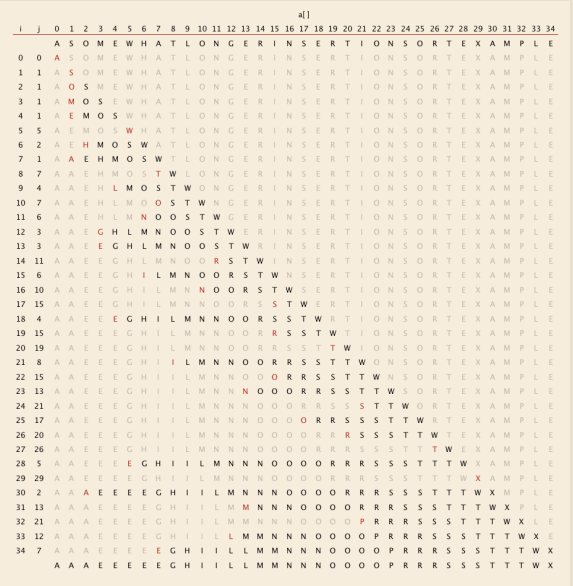
Analyse:
Anzahl Vergleiche $\leq \frac{1}{2}n^2 + O(n)$
Anzahl Swaps $\leq \frac{1}{2}n^2 + O(n)$
(nach jedem Vergleich außer letztem in Z.4)

Optimierung: Einfügeposition per binärer Suche finden! $\rightsquigarrow$ *Binary Insertionsort*
Da wir aber trotzdem per Swaps Elemente bewegen müssen, hilft das nur für Vergleiche . . .

Sowohl Bubblesort als auch Insertionsort sind datenabhängig. Wie gut sind sie?

https://www.wild-inter.net/teaching/ads/insertionsort

Insertion sort: trace



7.3 Analyse der primitiven Verfahren

Datenabhängigkeit

Sowohl Bubblesort als auch Insertionsort sind datenabhängig. Wie gut sind sie?

Guter Anfang: Visualisieren und experimentieren!

Hoffnung: Bekommen dadurch eine Idee für eine Hypothese

Permutationen und Inversionen

Für die Analyse benötigen wir etwas Notation

- Wir schreiben $[n] = [1..n] = \{1, 2, \ldots, n\}$
- Eine *Permutation* ist eine bijektive Abbildung $\pi : [n] \to [n]$
  wir identifizieren die Funktion π mit der Sequenz ihrer Bilder: $\pi(1), \ldots, \pi(n)$

  Beispiel: Die Permutation $1, 2, 3, 4, 5$ ist die Identitätsfunktion auf $[5]$
- Für eine Permutation $\pi : [n] \to [n]$ ist $(i, j) \in [n]^2$ mit eine *Inversion*,
  wenn $i < j$, aber $\pi(i) > \pi(j)$ gilt.

  $2, 1, 4, 3$ hat genau 2 Inversionen: $(1, 2)$ und $(3, 4)$

Permutation und Algebra

Permutationen sind ein gut untersuchtes Gebiet ...

- Für eine Permutation $\pi : [n] \to [n]$ heißt die eindeutig bestimmte Permutation π^{-1} mit $\pi^{-1}(i) = j$ gdw. $\pi(j) = i$ die *inverse Permutation* zu π
- Für eine Permutation $\pi : [n] \to [n]$ heißt die eindeutig bestimmte Permutation π^R mit $\pi^R = \pi(n), \pi(n-1), \ldots, \pi(2), \pi(1)$ die *reverse Permutation* zu π

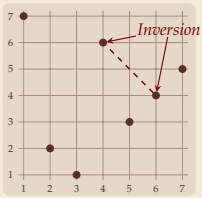
$\rightsquigarrow$ **Algebra:** Die Menge aller Permutationen von $[n]$ mit der Operation $\pi \circ \pi'$ (Funktionskomposition) bildet eine *Gruppe*, die *symmetrische Gruppe* $\mathcal{S}_n$, mit neutralem Element $\mathrm{id}_n = 1, 2, \ldots, n$.

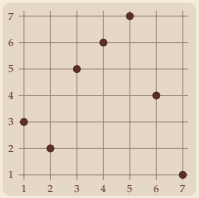
- Oft am einfachsten graphisch! (i auf x-Achse, $\pi(i)$ auf y-Achse)

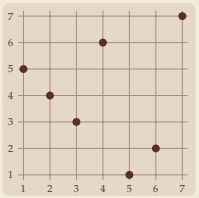
$\pi = [7,2,1,6,3,4,5]$

$\pi^{-1} = [3,2,5,6,7,4,1]$

$\pi^R = [5,4,3,6,1,2,7]$

$\pi =$ 

$\pi^{-1} =$ 

$\pi^R =$ 

Zurück zum Sortieren

ohne Beschränkung der Allgemeinheit

Wir sortieren **Vergleichs-basiert** $\rightsquigarrow$ o. B. d. A. sortieren Permutation von $[n]$

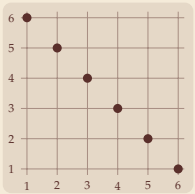
keine Duplikate!

- In Bubblesort und Insertionsort:
  - jeder Swap vertauscht benachbarte Indices $j, j+1$ ← Definition von *primitiven* Sortierverfahren
  - Vergleich davor stellt sicher: $A[j] > A[j+1]$
  - $\rightsquigarrow$ Beheben **eine Inversion** der Eingabe
  - Aber auch **genau** eine Inversion! (benachbarte Indices!)

$\rightsquigarrow$ *Bubblesort und Insertionsort verwenden genau so viele Swaps wie die Eingabe Inversionen hat.*

$\rightsquigarrow$ Laufzeit $\geq \mathrm{Inv}(A[0..n)) = \#\text{Inversionen in } A = \left|\{(i,j) : 0 \leq i < j < n : A[i] > A[j]\}\right|$

Worst-Case:



Jedes Paar $i < j$ ist Inversion $\rightsquigarrow$ $\mathrm{Inv} = \binom{n}{2} \sim \frac{1}{2}n^2$ ☹

Average-Case-Analyse von Inversionen

Im Worst-Case ist die Anzahl Inversionen quadratisch

$\rightsquigarrow$ Alle primitiven Sortierverfahren benötigen Laufzeit in $\Omega(n^2)$ im Worst-Case

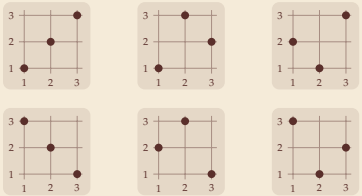
Wie sieht es im Average-Case aus?

- ***Random permutation model***: Alle $n!$ Permutation gleich wahrscheinlich

$\rightsquigarrow$ $\mathbb{E}[\mathrm{Inv}(\pi)] = ?$ (scheint knifflig?)

- **Trick:** Betrachte Paar von Permutation π und π^R gemeinsam.

Für $n = 3$



Inv = 3 Inv = 3 Inv = 3

$\rightsquigarrow$ $\mathrm{Inv}(\pi) + \mathrm{Inv}(\pi^R) = \binom{n}{2}$ gilt allgemein!

Im Mittel haben also beide $\binom{n}{2}\Big/2$ Inversionen

$$\mathbb{E}\big[\mathrm{Inv}(A[0..n)\big] \;=\; \binom{n}{2}\Big/2 \;\sim\; \mathbf{\frac{1}{4}n^2}$$

Primitive Sortierverfahren – Diskussion

- 👍 Einfache Algorithmen, kompakter Code
- 👍 Kein zusätzlicher Speicher/Puffer nötig
- 👎 $\Omega(n^2)$ im Worst-Case
- 👎 Average-Case um Faktor 2 besser also Worst-Case . . . aber trotzdem quadratisch

- ⇝ Gute Wahl für sehr kleine Eingaben (minimaler Overhead)
- ϟ nicht skalierbar

7.4 Mergesort

Two classic sorting algorithms: mergesort and quicksort

Critical components in the world's computational infrastructure.

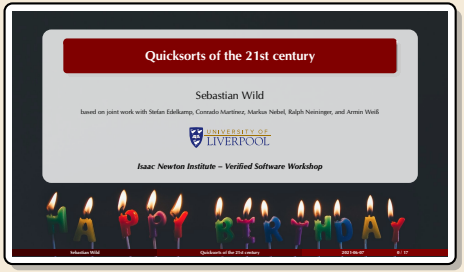
- Full scientific understanding of their properties has enabled us to develop them into practical system sorts.
- Quicksort honored as one of top 10 algorithms of 20th century in science and engineering.

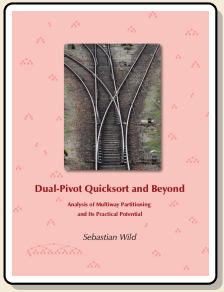
Mergesort. [this lecture]

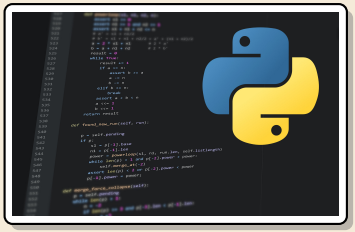
Quicksort. [next lecture]

— Werbung —

Sortieralgorithmen in Standardbibliotheken haben mich viel beschäftigt . . .







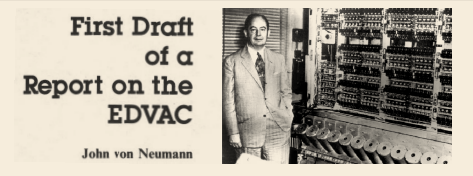
Mergesort

Basic plan.

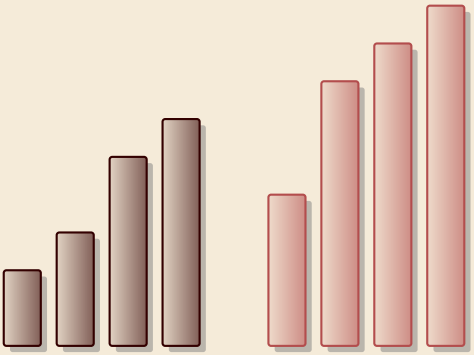
- Divide array into two halves.
- Recursively sort each half.
- Merge two halves.

input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Mergesort overview

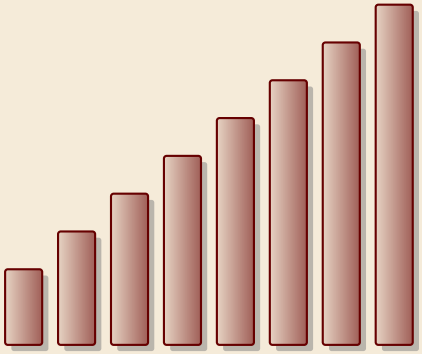


Merging runs – Verschmelzen sortierter Teile



run1 run2 result

Merging runs – Verschmelzen sortierter Teile



run1 run2 result

Merge – In-place API

Merge (aus Sicht des Benutzers):

- **Eingabe:** Zwei sortierte Teilarrays
- **Ausgabe:** Sortiertes Array
- Bequem z. B. mit physischen beweglichen Objekten zu machen

Wie machen wir das aber mit einem Array $A[0..n)$ von Objekten?

Merge (in Arrays, als Teil unseres Mergesort):

- **Gegeben:** zwei *benachbarte* sortierte Bereiche $A[\texttt{lo..mid}]$ und $A(\texttt{mid..hi}]$ in Array $A[0..n)$
- **Ziel:** $A[\texttt{lo..hi}]$ sortiert

ϟ extrem knifflig ohne einen Pufferbereich, in den wir Objekte auslagern können!

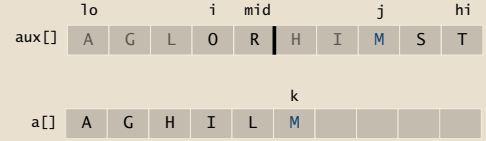
⇝ intern: (1) Verschiebe Elemente in Hilfsarray $\texttt{aux}[\texttt{lo..hi}]$
(2) Merge zurück von $\texttt{aux}$ nach $A[\texttt{lo..hi}]$

Merging: Java implementation

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
   for (int k = lo; k <= hi; k++)
      aux[k] = a[k];

   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)
   {
      if      (i > mid)              a[k] = aux[j++];
      else if (j > hi)               a[k] = aux[i++];
      else if (less(aux[j], aux[i])) a[k] = aux[j++];
      else                           a[k] = aux[i++];
   }
}
```

copy

merge



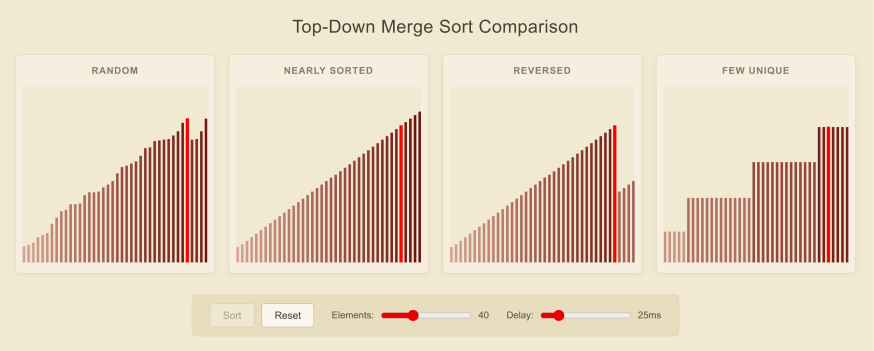
Mergesort: Java implementation

```
public class Merge
{
   private static void merge(...)
   {  /* as before */  }

   private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
   {
      if (hi <= lo) return;
      int mid = lo + (hi - lo) / 2;
      sort(a, aux, lo, mid);
      sort(a, aux, mid+1, hi);
      merge(a, aux, lo, mid, hi);
   }

   public static void sort(Comparable[] a)
   {
      Comparable[] aux = new Comparable[a.length];
      sort(a, aux, 0, a.length - 1);
   }
}
```

Mergesort: trace

Datenabhängigkeit?

Tatsächlich hängt die Laufzeit von Mergesort (fast) nicht von den Daten ab!



`https://www.wild-inter.net/teaching/ads/mergesort`

Mergesort: empirical analysis

Running time estimates:

- Laptop executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (N^2)			mergesort (N log N)		
computer	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

Bottom line. Good algorithms are better than supercomputers.

Mergesort: number of compares

Proposition. Mergesort uses $\leq N \lg N$ compares to sort an array of length N.

Pf sketch. The number of compares $C(N)$ to mergesort an array of length N satisfies the recurrence:

$$C(N) \leq C(\lceil N/2 \rceil) + C(\lfloor N/2 \rfloor) + N \quad \text{for } N > 1, \text{ with } C(1) = 0.$$

left half | right half | merge

We solve the recurrence when N is a power of 2: ← result holds for all N (analysis cleaner in this case)

$$D(N) = 2\,D(N/2) + N, \text{ for } N > 1, \text{ with } D(1) = 0.$$

Wir erinnern uns: Master-Theorem

Laufzeit von Mergesort

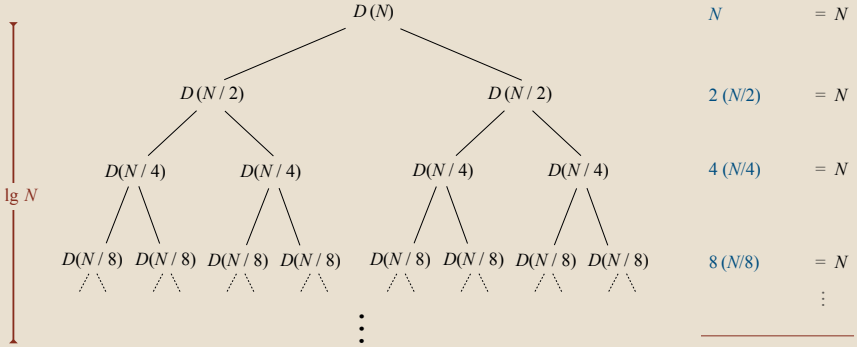
$$T(n) = \begin{cases} \mathbf{0} & n \leq 1 \\ \mathbf{2} \cdot T(n/\mathbf{2}) + \boldsymbol{n} & n \geq 2 \end{cases}$$

Für $n = 2^k$, $k \in \mathbb{N}_0$

Divide-and-conquer recurrence: proof by picture

Proposition. If $D(N)$ satisfies $D(N) = 2\,D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

Pf 1. [assuming N is a power of 2]



Divide-and-conquer recurrence: proof by induction

Proposition. If $D(N)$ satisfies $D(N) = 2\,D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

Pf 2. [assuming N is a power of 2]

- Base case: $N = 1$.
- Inductive hypothesis: $D(N) = N \lg N$.
- Goal: show that $D(2N) = (2N) \lg (2N)$.

$$
\begin{aligned}
D(2N) &= 2\,D(N) + 2N && \text{given} \\
&= 2\,N \lg N + 2N && \text{inductive hypothesis} \\
&= 2\,N\,(\lg (2N) - 1) + 2N && \text{algebra} \\
&= 2\,N \lg (2N) && \text{QED}
\end{aligned}
$$

Mergesort: number of array accesses

Proposition. Mergesort uses $\leq 6 N \lg N$ array accesses to sort an array of length N.

Pf sketch. The number of array accesses $A(N)$ satisfies the recurrence:

$$A(N) \leq A(\lceil N/2 \rceil) + A(\lfloor N/2 \rfloor) + 6N \text{ for } N > 1, \text{ with } A(1) = 0.$$

Key point. Any algorithm with the following structure takes $N \log N$ time:

```
public static void linearithmic(int N)
{
    if (N == 0) return;
    linearithmic(N/2);   <-- solve two problems
    linearithmic(N/2);   <-- of half the size
    linear(N);           <-- do a linear amount of work
}
```

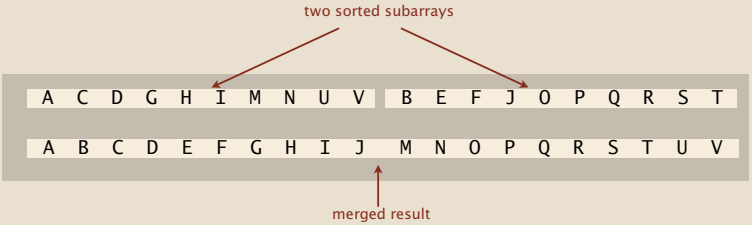
Notable examples. FFT, hidden-line removal, Kendall-tau distance, ...

7.5 Mergesort Optimierungen

Mergesort analysis: memory

Proposition. Mergesort uses extra space proportional to N.

Pf. The array `aux[]` needs to be of length N for the last merge.



Def. A sorting algorithm is in-place if it uses $\leq c \log N$ extra memory.

Ex. Insertion sort, selection sort, shellsort.

Challenge 1 (not hard). Use `aux[]` array of length $\sim \frac{1}{2} N$ instead of N.

Challenge 2 (very hard). In-place merge. [Kronrod 1969]

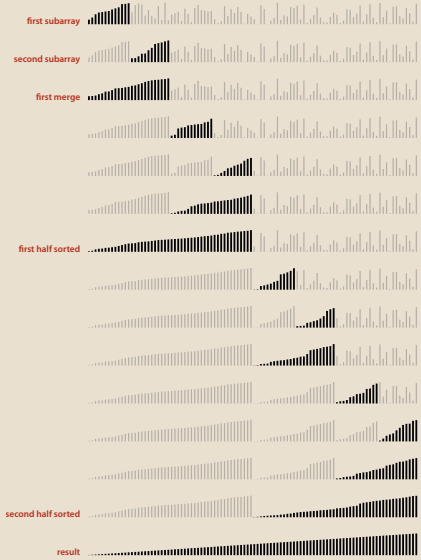
Mergesort: practical improvements

Use insertion sort for small subarrays.

- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 10 items.

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo + CUTOFF - 1)
   {
      Insertion.sort(a, lo, hi);
      return;
   }
   int mid = lo + (hi - lo) / 2;
   sort (a, aux, lo, mid);
   sort (a, aux, mid+1, hi);
   merge(a, aux, lo, mid, hi);
}
```

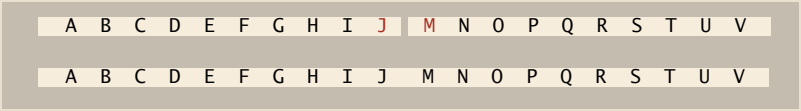
Mergesort with cutoff to insertion sort: visualization



Mergesort: practical improvements

Stop if already sorted.

- Is largest item in first half ≤ smallest item in second half?
- Helps for partially-ordered arrays.




```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo) return;
   int mid = lo + (hi - lo) / 2;
   sort (a, aux, lo, mid);
   sort (a, aux, mid+1, hi);
   if (!less(a[mid+1], a[mid])) return;
   merge(a, aux, lo, mid, hi);
}
```

Mergesort: practical improvements

Eliminate the copy to the auxiliary array. Save time (but not space) by switching the role of the input and auxiliary array in each recursive call.

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)
   {
      if      (i > mid)          aux[k] = a[j++];
      else if (j > hi)           aux[k] = a[i++];
      else if (less(a[j], a[i])) aux[k] = a[j++];
      else                       aux[k] = a[i++];
   }
}

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo) return;
   int mid = lo + (hi - lo) / 2;
   sort (aux, a, lo, mid);
   sort (aux, a, mid+1, hi);
   merge(a, aux, lo, mid, hi);
}
```

merge from a[] to aux[]

assumes aux[] is initialize to a[] once, before recursive calls

switch roles of aux[] and a[]

Iterativer Mergesort

Bisher: Mergesort als rekursive Methode

- meistens effizienteste Methode
- manchmal unbequem, dass man Platz für call stack braucht
- manchmal hilfreich, Sortieren zu können, ohne n zu kennen!

Mergesort kann auch ohne Rekursion auskommen!

Bottom-up mergesort

Basic plan.

- Pass through array, merging subarrays of size 1.
- Repeat for subarrays of size 2, 4, 8, ...

		a[i]														
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sz = 1																	
	merge(a, aux, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
	merge(a, aux, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
	merge(a, aux, 4, 4, 5)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
	merge(a, aux, 6, 6, 7)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
	merge(a, aux, 8, 8, 9)	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
	merge(a, aux, 10, 10, 11)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
	merge(a, aux, 12, 12, 13)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
	merge(a, aux, 14, 14, 15)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L
sz = 2																	
	merge(a, aux, 0, 1, 3)	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L
	merge(a, aux, 4, 5, 7)	E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L
	merge(a, aux, 8, 9, 11)	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
	merge(a, aux, 12, 13, 15)	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P
sz = 4																	
	merge(a, aux, 0, 3, 7)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
	merge(a, aux, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
sz = 8																	
	merge(a, aux, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Bottom-up mergesort: Java implementation

```
public class MergeBU
{
   private static void merge(...)
   {  /* as before */  }

   public static void sort(Comparable[] a)
   {
      int N = a.length;
      Comparable[] aux = new Comparable[N];
      for (int sz = 1; sz < N; sz = sz+sz)
         for (int lo = 0; lo < N-sz; lo += sz+sz)
            merge(a, aux, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
   }
}
```

but about 10% slower than recursive, top-down mergesort on typical systems

Bottom line. Simple and non-recursive version of mergesort.

Stability

A typical application. First, sort by name; then sort by section.

```
Selection.sort(a, new Student.ByName());
```

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

```
Selection.sort(a, new Student.BySection());
```

Furia	1	A	766-093-9873	101 Brown
Rohde	2	A	232-343-5555	343 Forbes
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Andrews	3	A	664-480-0023	097 Little
Kanaga	3	B	898-122-9643	22 Brown
Gazsi	4	B	766-093-9873	101 Brown
Battle	4	C	874-088-1212	121 Whitman

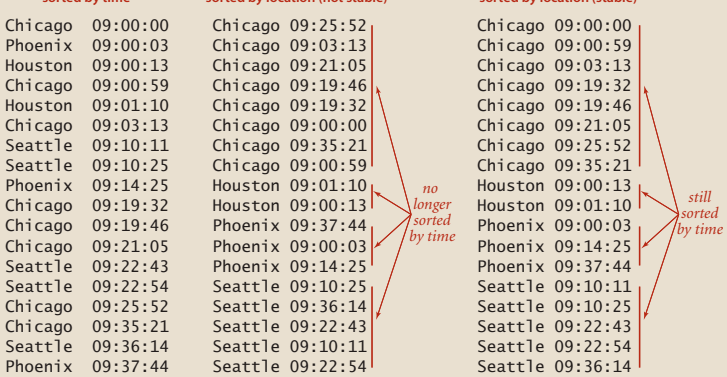
@#%&@! Students in section 3 no longer sorted by name.

A stable sort preserves the relative order of items with equal keys.

Stability

Q. Which sorts are stable?

A. Need to check algorithm (and implementation).



Stability: mergesort

Proposition. Mergesort is stable.

```
public class Merge
{
   private static void merge(...)
   {  /* as before */  }

   private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
   {
      if (hi <= lo) return;
      int mid = lo + (hi - lo) / 2;
      sort(a, aux, lo, mid);
      sort(a, aux, mid+1, hi);
      merge(a, aux, lo, mid, hi);
   }

   public static void sort(Comparable[] a)
   {  /* as before */  }
}
```

Pf. Suffices to verify that merge operation is stable.

Stability: mergesort

Proposition. Merge operation is stable.

```
private static void merge(...)
{
   for (int k = lo; k <= hi; k++)
      aux[k] = a[k];

   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)
   {
      if      (i > mid)              a[k] = aux[j++];
      else if (j > hi)               a[k] = aux[i++];
      else if (less(aux[j], aux[i])) a[k] = aux[j++];
      else                           a[k] = aux[i++];
   }
}
```

0	1	2	3	4	5	6	7	8	9	10
A_1	A_2	A_3	B	D	A_4	A_5	C	E	F	G

Pf. Takes from left subarray if equal keys.

7.6 Quicksort

A brief history of quicksort

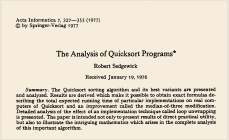
Tony Hoare.

- Developed quicksort in 1960 to translate Russian into English.
- Later implemented it in Algol 60, using recursion.


Algorithms

ALGORITHM 64
QUICKSORT
C. A. R. Hoare
Computer Unit, Borehamwood, Hertfordshire, Eng.


Programming Techniques
Implementing Quicksort Programs
Robert Sedgewick
Brown University


Acta Informatica 7, 327—355 (1977)
© by Springer-Verlag 1977

The Analysis of Quicksort Programs*

Robert Sedgewick

Received January 10, 1976

Tony Hoare
(Turing Award, 1980)

Bob Sedgewick.

- Refined and popularized quicksort in the 1970s.
- Analyzed many versions of quicksort.

Bob Sedgewick
(longtime COS 226 instructor)

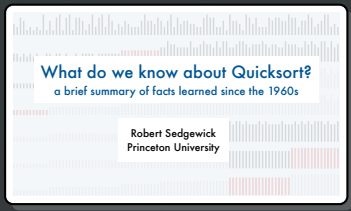
Quicksort – 60th Anniversary Workshop in 2021

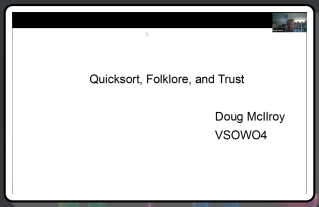
The workshop will include a special event marking the 60th anniversary of Tony Hoare's invention of Quicksort.

Given the current restrictions on movement, due to the ongoing pandemic, the INI intends to run this workshop virtually across 5 days. All talks and discussion sessions will be available virtually.

MONDAY 7TH JUNE 2021

16:00 to 16:30	**Douglas McIlroy** ***Dartmouth College*** Chair: Cliff Jones Folklore of Computational Complexity
16:30 to 17:00	**Robert Sedgewick** ***Princeton University*** Chair: Cliff Jones What Do We Know About Quicksort?
17:00 to 17:30	Break
17:30 to 18:00	**Sebastian Wild** ***University of Liverpool*** Chair: Cliff Jones Quicksorts of the 21st century
18:00 to 18:30	**Tony Hoare** ***Isaac Newton Institute*** Chair: Cliff Jones Tony Hoare Answers Questions



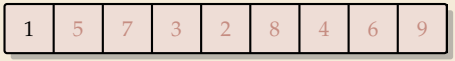


Talk Recordings: `https://tiny.cc/quicksort-60`

Irgendwas von Selectionsort zu lernen?

Selectionsort

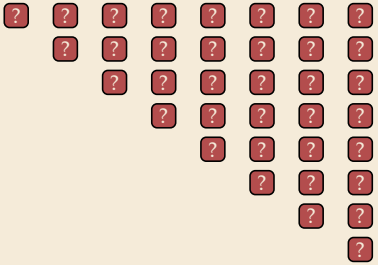
Finde Minimum und tausche nach links.



Irgendwas von Selectionsort zu lernen?

Selectionsort

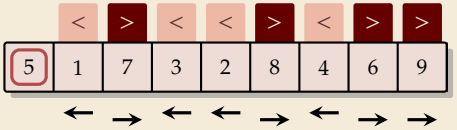
Finde Minimum und tausche nach links.



n − 1 *Vergleiche um* 1 *Element loszuwerden . . .*
können wir damit nicht mehr machen?

Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

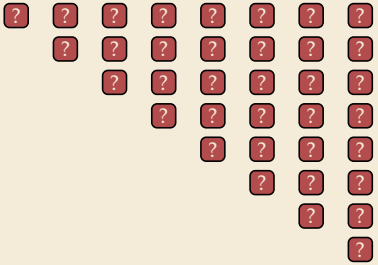


Irgendwas von Selectionsort zu lernen?

Selectionsort

Finde Minimum und tausche nach links.

1	2	3	4	5	6	7	8	9



$n - 1$ *Vergleiche um* 1 *Element loszuwerden . . .*
können wir damit nicht mehr machen?

Rang finden

Zähle, wie viele Elements kleiner als *Pivot*.

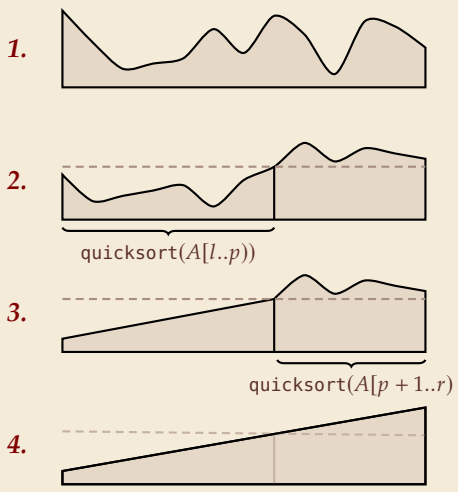
1	2	3	4	5	6	7	8	9

$n - 1$ *Vergleiche um* 1 *Rang zu bestimmen . . .*
viel mehr Fortschritt!

Quicksort

Mit einer **rekursiven** Methode ist Verwaltung der to-do-Teile einfach!

```java
static <Elem> void quicksort(Elem[] A, Comparator<Elem> c) {
    quicksort(A, 0, A.length, c);
}

/** Sortiert A[l..r) */
static <Elem> void quicksort(Elem[] A, int l, int r,
                             Comparator<Elem> c) {
    if (r - l <= 1) return; // n ≤ 1 -> schon sortiert
    // 1. unsortiert
    int p = partition(A, l, r, c); // Details unten
    // 2. partitioniert, A[p] an endgültiger Position!
    quicksort(A, l,   p, c );
    // 3. A[l..p] sortiert
    quicksort(A, p+1, r, c);
    // 4. alles sortiert
}
```



"Writing code for partition wasn't too difficult, but when it came to keeping a record of the ranges awaiting treatment, I couldn't manage it. [. . .] [When I read about recursion], I realized that I had the solution to my coding problem: If I use recursion, everything would look very simple."

Tony Hoare, https://youtu.be/WZqsjdbW_Es&t=348s

Partitioniern – Einfache Version

Erste Version: *Wir separieren Elemente die größer bzw. kleiner als ein „Pivot-Element“ sind*

```java
@SuppressWarnings("unchecked")
public static <Elem> int partition(Elem[] A, int l, int r,
                                   Comparator<Elem> c) {
    Elem[] smaller = (Elem[]) new Object[r-l];
    Elem[] larger  = (Elem[]) new Object[r-l];
    Elem pivot = A[l]; // fixe Pivotwahl
    int s = 0, g = 0;
    for (int i = l; i < r; ++i) {
        if (less(A[i], pivot, c))
            smaller[s++] = A[i];
        else larger [g++] = A[i];
    }
    System.arraycopy(smaller, 0, A, l,   s);
    System.arraycopy(larger,  0, A, l+s, g);
    return l + s;
}
```

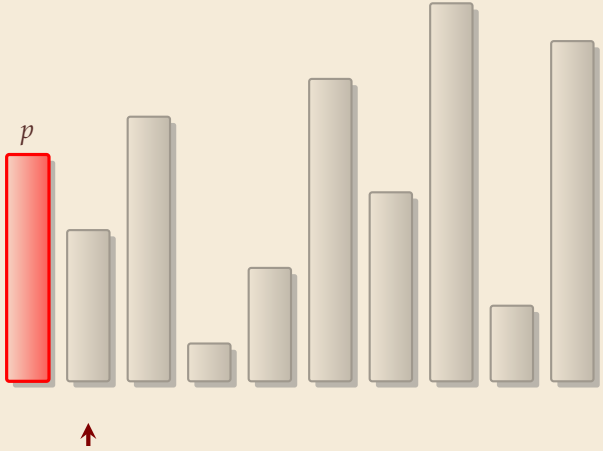
- Verwenden Zusatzspeicher, um Elemente zwischenzulagern
- ⇝ Müssen Sie am Ende zurückkopieren
- Rückgabewert ist Position des Pivots
  - ⇝ brauchen wir für die Rekursion!
- *Pivot wie hier zu wählen, ist eine ganz schlechte Idee . . . mehr dazu später!*
- Java: `@SuppressWarnings` und cast wegen generic arrays . . .

- Java: `System.arraycopy` schneller zum Kopieren großer Arrays als eine Schleife

```java
/** Kopiert B[j..j+n) := A[i..i+n) */
static void System.arraycopy(Object[] A, int i, Object[] B, int j, int n) {
    // direkter Aufruf von JVM an Betriebssystem, daher idR schneller
}
```

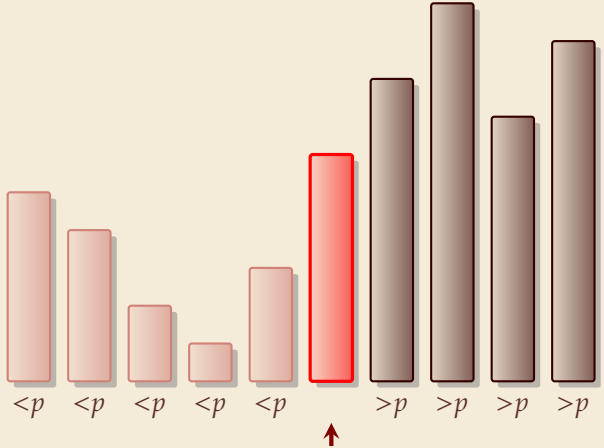
Inplace Partition

*Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!*



Inplace Partition

*Mit etwas mehr Cleverness geht es auch **ohne** Extraspeicher!*



- kein Zusatzspeicher!
- nur zwei Indizes + Pivot
- betrachten jedes Element genau einmal
- wo Indizes sich treffen gehört das Pivot hin

Hoare-Sedgewick Partitionierung

Idee mit kreuzenden Indizes von Tony Hoare, mit Verfeinerungen von Robert Sedgewick

Achtung: Voll subtiler Details! Verwenden Sie exakt diesen Code (falls Sie das je brauchen)

```
1  static <Elem> int partition(Elem[] A, int l, int r,
2                              Comparator<Elem> c) {
3      swap(A, l, choosePivot(l, r));
4      int i = l, j = r;
5      while (true) {
6          do ++i; while (i < r && less(A[i], A[l], c));
7          do --j; while (j > l && less(A[l], A[j], c));
8          if (i >= j) break;
9          swap(A, i, j);
10     }
11     swap(A, l, j);
12     return j;
13 }
```

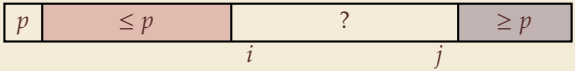
- Wahl des Pivots über einen Index `choosePivot(l, r)` $\in [l..r)$

$\rightsquigarrow$ Details dazu gleich

Subtilitäten ($\notin$ exam)

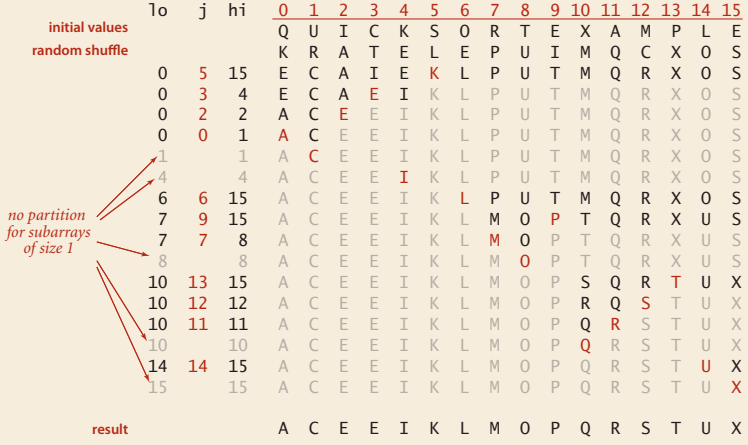
- Beide do-while-Schleifen stoppen bei Duplikaten des Pivots
- Check `j > l` könnte man weglassen (immer wahr)
- für paarweise verschiedene Elemente, verlassen äußere Schleife immer mit `i = j + 1`;
  `i = j` aber auch möglich, wenn $A[i] = A[j] = A[l]$

Loop invariant (5 – 10): $A[l..r)$



7.7 Quicksort Analyse & Randomisierte Algorithmen

Quicksort trace



Quicksort trace (array contents after each partition)

Quicksort: empirical analysis (1961)

Running time estimates:

- Algol 60 implementation.
- National Elliott 405 computer.

Table 1

NUMBER OF ITEMS	MERGE SORT	QUICKSORT
500	2 min 8 sec	1 min 21 sec
1,000	4 min 48 sec	3 min 8 sec
1,500	8 min 15 sec*	5 min 6 sec
2,000	11 min 0 sec*	6 min 47 sec

* These figures were computed by formula, since they cannot be achieved on the 405 owing to limited store size.

sorting n 6-word items with 1-word keys

Elliott 405 magnetic disc (16K words)

Quicksort: empirical analysis

Running time estimates:

- Home PC executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

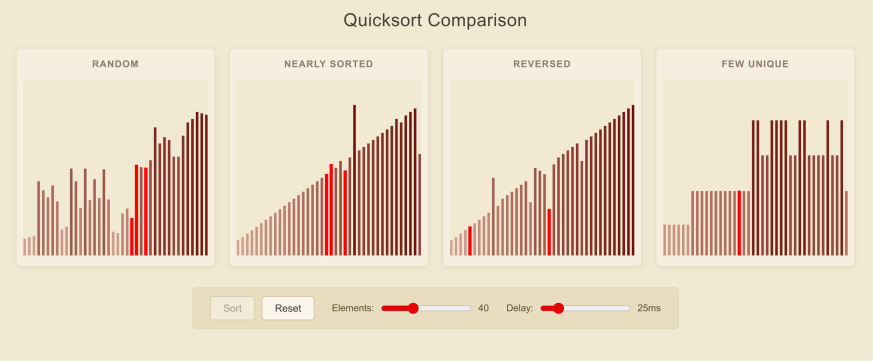
	insertion sort (n^2)			mergesort (n log n)			quicksort (n log n)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

Lesson 1. Good algorithms are better than supercomputers.

Lesson 2. Great algorithms are better than good ones.

Datenabhängigkeit?

Die Laufzeit von Quicksort hängt scheinbar wenig von den Daten ab!



`https://www.wild-inter.net/teaching/ads/quicksort`

Doch der Schein trügt! 😱

Quicksort: best-case analysis

Best case. Number of compares is $\sim n \lg n$.

			a[ ]													
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
random shuffle			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
0	7	14	D	A	C	B	F	E	G	H	L	I	K	J	N	M	O
0	3	6	B	A	C	D	F	E	G	H	L	I	K	J	N	M	O
0	1	2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
0		0	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
2		2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
4	5	6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
4		4	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
6		6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
8	11	14	A	B	C	D	E	F	G	H	J	I	K	L	N	M	O
8	9	10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
8		8	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
10		10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
12	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12		12	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Quicksort: worst-case analysis

Worst case. Number of compares is $\sim \frac{1}{2} n^2$.

			a[ ]													
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
random shuffle			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0	0	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	1	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
2	2	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
3	3	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
4	4	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	5	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
6	6	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
7	7	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
8	8	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
9	9	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
10	10	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
11	11	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12	12	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
13	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Das Quicksort-Puzzle

Üblicherweise ist Quicksort der schnellste generische Sortieralgorithmus

- äußerst schlanke innere Schleifen
- kein Extraspeicher
- sequentielle Scans über Eingabe
- besucht jedes Element nur einmal pro Partitionierungsschritt
  (Mergesort muss lesen und schreiben pro Merge!)

$\rightsquigarrow$ weit und breit in Verwendung in Softwarebibliotheken

Gleichzeitig ist der Worst-Case quadratisch! ***Wie passt das zusammen?***

Average-Case Analyse von Quicksort

Betrachten wieder das *Random-permutation model*

- Sortieren die Zahlen $[n]$
- jede der $n!$ möglichen Reihenfolgen ist gleich wahrscheinlich

$\rightsquigarrow$ *Jede Zahl $p \in [n]$ hat Chance $\frac{1}{n}$, als Pivot gewählt zu werden!*

Damit können wir für die erwarteten Kosten eine **Rekursionsgleichung** aufstellen
$C(n)$ = erwartete Anzahl Vergleiche, wenn wir zufällige Permutation von $[n]$ sortieren.

$$
\begin{aligned}
C(0) &= 0 \\
C(1) &= 0 \\
C(n) &= n+1 + \sum_{p=1}^{n} \frac{1}{n} \cdot \Big(C(p-1) + C(n-p) \Big) \qquad (n \geq 2)
\end{aligned}
$$

Kosten einer Partitionierung

linker rek. Aufruf

Wahrscheinlichkeit, p also Pivot zu wählen

rechter rek. Aufruf

Sieht fies aus! Nicht wie eine Rekursion, die man einfach lösen kann . . .

Wider Erwarten geht das hier exakt(!) $C(n) = 2(n+1)\big(H_{n+1} - \frac{4}{3}\big)$ $\qquad H_n = \sum_{i=1}^{n} \frac{1}{i}$

Wenn Sie das genauer wissen wollen $\rightsquigarrow$ *Effiziente Algorithmen!*

Quicksort throws a Tantrum

Wir werden stattdessen eine einfachere und robustere Analyse anstellen.

- **Trick:** Analysieren anderen Algorithmus, der leichter zu analysieren ist
- Zeigen, dass der `quicksort` höchstens besser ist

⇝ obere Schranke für Kosten

Betrachten hier den etwas launischen **Tobsucht-Quicksort**

```
1  static <Elem> void tobsuchtQuicksort(Elem[] A, int l, int r,
2  /** Sortiert A[l..r) */                Comparator<Elem> c) {
3      int n = r-l;
4      if (n <= 1) return;  // fertig
5      while (true) {
6          int p = partition(A, l, r, c);  // n+1 Vergleiche
7          if (l + n/4 <= p && p <= l + 3*n/4) break;
8          //  #%!&  Schon wieder so ein Mist-Pivot!?
9          // Bekomme Tobsuchtanfall
10         // Mische A[l..r) im Ärger gut durch
11     }
12     tobsuchtQuicksort(A, l,   p, c );
13     tobsuchtQuicksort(A, p+1, r, c);
14 }
```

Theorem 7.1
Für die erwartete Anzahl $T(n)$ an Vergleichen in Tobsucht-Quicksort gilt $C(n) \leq T(n) = O(n \log n)$. ◀

Corollary 7.2
Im Average-Case benötigt Quicksort $O(n \log n)$ Vergleiche. ◀

Analyse von Tobsucht-Quicksort

Randomisierte Algorithmen

- ⇝ Quicksort ist gut im Average-Case & Average-Case is repräsentativ
- **aber:** Analyse gilt nur im *Random Permutation Model*

Können wir uns darauf verlassen?

- ⇝ **Gültig, aber gefährlich**

```
1 static int choosePivotBad(int l, int r) {
2     return l;  // feste Position
3 }
```

Unsere Wahl ist vorhersehbar
- ⇝ Kann ausgenutzt werden!

- ⇝ **Robuste Wahl:**

```
1 static int choosePivot(int l, int r) {
2     return l + random.nextInt(r - l);
3 }
```

Niemand weiß welches Pivot als nächstes kommt (wir auch nicht!)

strategic ambiguity

Man kann zeigen

- Uniform zufällige Pivotposition verhält sich *genauso(!)* wie *Random Permutation Model*
- ⇝ Analyse von oben bleibt gültig, aber jetzt für **beliebige Eingaben**

ohne Duplikate . . .

Average-Case Analyse vs. Randomisierter Algorithmus

Randomisierung: Gezielter Einsatz von zufälligen Entscheidungen im Algorithmus

Average-Case-Analyse

- Algorithmus ist **deterministisch** gleiche Eingabe, gleiche Berechnung
- Nehmen an, dass Eingabe nach **Wahrscheinlichkeits-Verteilung** zufällig gezogen
- Kosten sind Erwartungswert über **zufällige Eingabe**

Randomisierte Algorithmen

- Algorithmus ist **nicht** deterministisch gleiche Eingabe, evtl. andere Berechnung
- Eingabe als Worst-Case gewählt potentiell von Gegner, der unseren Algorithmus kennt
- Kosten sind Erwartungswert über **zufällige Aktionen unseres Algorithmus**

Verwirrenderweise sind die Analysetechniken für beides oft identisch!

Aber: Implikation sehr unterschiedlich! Randomisierung liefert viel stärkere Garantien.

Quicksort: practical improvements

Insertion sort small subarrays.

- Even quicksort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 10 items.

```
private static void sort(Comparable[] a, int lo, int hi)
{
   if (hi <= lo + CUTOFF - 1)
   {
      Insertion.sort(a, lo, hi);
      return;
   }
   int j = partition(a, lo, hi);
   sort(a, lo, j-1);
   sort(a, j+1, hi);
}
```

Quicksort – Praktische Verbesserung

Pivot als Median eines kleine Samples

- ideales Pivot: Median der Eingabe (aber teuer zu finden)

⇝ Wählen kleines Sample der Eingabe und finden Median vom Sample

- beliebte Wahl: Median-of-3

```
1  /** Median of 3 random elements */
2  public static <Elem> int choosePivot(Elem[] A, int l, int r, Comparator<Elem> c) {
3      int n = r - l;
4      int i1 = l + random.nextInt(n);
5      int i2 = l + random.nextInt(n);
6      int i3 = l + random.nextInt(n);
7      sort3(A, c, i1, i2, i3);
8      return i2; // median is now at i2
9  }
10
11 public static <Elem> void sort3(Elem[] A, Comparator<Elem>c, int i1, int i2, int i3) {
12     if (less(A[i2], A[i1], c)) swap(A,i1,i2);
13     if (less(A[i3], A[i1], c)) swap(A,i1,i3);
14     if (less(A[i3], A[i2], c)) swap(A,i2,i3);
15 }
```

⇝ Senkt erwartete Vergleichszahl von $\sim 2n \ln n$ auf $\sim \frac{12}{7} n \ln n$

7.8 Untere Schranke

Geht's noch besser?

Mit Mergesort und Quicksort haben wir 2 Verfahren mit (erwartet) $\Theta(n \log n)$ Laufzeit gesehen.

Ist das Zufall? Geht's nicht auch noch schneller?

Wie wir sehen werden, Nein! Tatsächlich zeigen wir:

Theorem 7.3 (Informationstheoretische untere Schranke fürs Sortieren)
Jeder *vergleichsbasierte* Sortieralgorithmus benötigt im Worst-Case mindestens
$\lceil \lg(n!) \rceil = n \lg n - n \underbrace{\lg e}_{\approx 1.443} \pm O(\log n)$ Vergleiche um n Elemente zu sortieren. ◀

$\rightsquigarrow$ *Mergesort optimal bis auf $O(n)$ Vergleiche!*

Für den Beweis müssen wir Aussagen über **alle** möglichen Algorithmen treffen
$\rightsquigarrow$ geht nur mit einem mathematischen Maschinenmodell!

Für allgemeine word-RAM Programme unklar
$\rightsquigarrow$ betrachten stärker eingeschränktes Maschinenmodell

Vergleichsbasierte Algorithmen

Im *comparison model* kann auf Eingabe $A[0..n)$ **nur** über `less` und `swap` zugegriffen werden

- Für alle unsere Sortiertverfahren der Fall!
- dürfen keine Annahme über Typ der sortieren Daten treffen
- Objekte können nur als "black box" bewegt oder verglichen werden

Zusätzliche Vereinfachung: **Kosten** einer Ausführung = Anzahl Aufrufe an `less`

- sämtliche sonstige Berechnung geschenkt!

$\rightsquigarrow$ (zu) optimistische Annahme, aber:
wenn schon mit dieser Annahme Kosten $\Omega(n \log n)$ unvermeidbar, erst recht ohne!

Wichtige Konsequenz: **„Normalform" für Sortieralgorithmen**

- Für jeden vergleichsbasierten Sortieralgorithmen gibt es einen bzgl. der Kosten **äquivalenten** Algorithmus mit den folgenden Einschränkungen
- **Alle** gewünschten **Vergleiche** werden **zu Beginn** ausführt (und Ergebnisse gespeichert)
- danach nur noch Swaps, um Eingabe sortiert anzuordnen

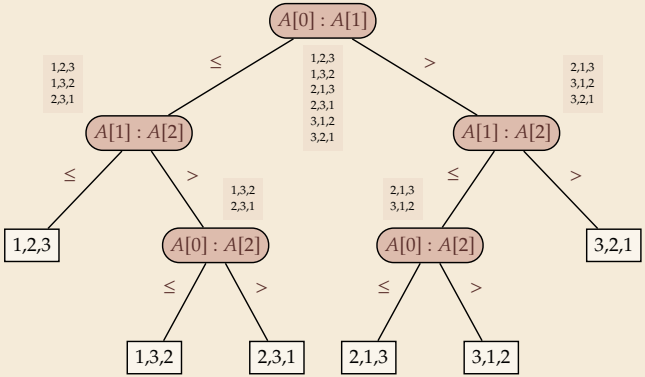
$\rightsquigarrow$ *Komplettes **Wissen** über Eingabe muss bereits vor dem ersten Swap vorliegen*

Vergleichsbäume

⇝ Ein solcher vergleichsbasierter Algorithmus in Normalform ist bestimmt durch einen *Vergleichsbaum* (*decision tree*).

- Knoten im Baum = Vergleiche, die der Algorithmus (*bei irgendeiner Eingabe*) ausführt
- Kanten zu Kindknoten = Ergebnis des Vergleichs
- Blatt = eindeutig bestimmte Permutation der Eingabe
- nächster Vergleich darf von Historie abhängen

Beispiel: Vergleichsbaum für hypothetischen Algorithmus zum Sortieren von $A[0..3)$



Optimale Anzahl an Vergleichen

Annahme: Sortierverfahren basiert auf Schlüsselvergleichen.
$\rightsquigarrow$ Jedem Verfahren kann ein Vergleichsbaum zugeordnet werden, dessen innere Knoten *das Wissen des Algorithmus* zum jeweiligen Zeitpunkt repräsentieren.

Wurzel: Initialzustand, in dem der Algorithmus noch nichts weiß (jede der $n!$ Permutationen als Eingabe möglich).

Verzweigung: Jedem Knoten ist noch ein Vergleich `A[i]<A[j]` zweier Schlüssel zugeordnet[8]; der linke (rechte) Nachfolger des Knotens steht dann für die Situation, die aus einem positiven (negativen) Ergebnis des Vergleichs resultiert.

*War dieser Vergleich gut gewählt, so kann je nach seinem Ausgang die ein oder andere Permutation ausgeschlossen werden (ist z.B. `A[1]<A[2]` so ist $2, 1, \ldots$ unmöglich).

Optimale Anzahl an Vergleichen

Annahme: Sortierverfahren basiert auf Schlüsselvergleichen.
$\rightsquigarrow$ Jedem Verfahren kann ein Vergleichsbaum zugeordnet werden, dessen innere Knoten *das Wissen des Algorithmus* zum jeweiligen Zeitpunkt repräsentieren.
Wurzel: Initialzustand, in dem der Algorithmus noch nichts weiß (jede der $n!$ Permutationen als Eingabe möglich).
Verzweigung: Jedem Knoten ist noch ein Vergleich `A[i]<A[j]` zweier Schlüssel zugeordnet[8]; der linke (rechte) Nachfolger des Knotens steht dann für die Situation, die aus einem positiven (negativen) Ergebnis des Vergleichs resultiert.

[*]War dieser Vergleich gut gewählt, so kann je nach seinem Ausgang die ein oder andere Permutation ausgeschlossen werden (ist z.B. `A[1]<A[2]` so ist $2, 1, \ldots$ unmöglich).

Optimale Anzahl an Vergleichen

Annahme: Sortierverfahren basiert auf Schlüsselvergleichen.
$\rightsquigarrow$ Jedem Verfahren kann ein Vergleichsbaum zugeordnet werden, dessen innere Knoten *das Wissen des Algorithmus* zum jeweiligen Zeitpunkt repräsentieren.
Wurzel: Initialzustand, in dem der Algorithmus noch nichts weiß (jede der $n!$ Permutationen als Eingabe möglich).
Verzweigung: Jedem Knoten ist noch ein Vergleich `A[i]<A[j]` zweier Schlüssel zugeordnet[8]; der linke (rechte) Nachfolger des Knotens steht dann für die Situation, die aus einem positiven (negativen) Ergebnis des Vergleichs resultiert.

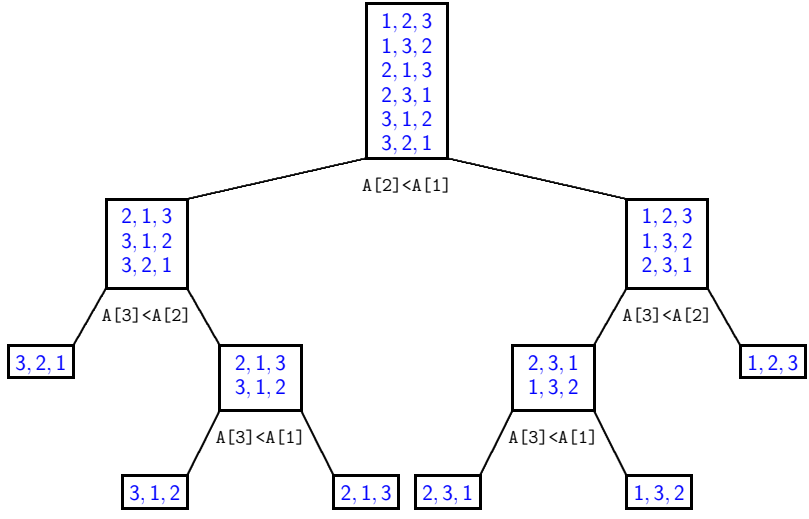
*War dieser Vergleich gut gewählt, so kann je nach seinem Ausgang die ein oder andere Permutation ausgeschlossen werden (ist z.B. `A[1]<A[2]` so ist $2, 1, \ldots$ unmöglich).

Wir beenden auf einem Pfad die Konstruktion, wenn der zuletzt erzeugte Knoten nur noch eine Permutation besitzt.

$\rightsquigarrow$ Erweiterter binärer Baum.

Wir beenden auf einem Pfad die Konstruktion, wenn der zuletzt erzeugte Knoten nur noch eine Permutation besitzt.

$\rightsquigarrow$ Erweiterter binärer Baum.



Führt Algorithmus auch noch Vergleiche in den Blättern durch $\rightsquigarrow$ nutzlos, für untere Schranke damit uninteressant.

Sortieralgorithmus muss für alle möglichen Eingaben funktionieren $\rightsquigarrow$ Vergleichsbaum muss für jedes Verfahren mindestens $n!$ Blätter besitzen[9].

Hat der Baum mehr als $n!$ Blätter $\rightsquigarrow$ unnötige Vergleiche, wir können entsprechende Blätter löschen.

Anzahl innerer Knoten auf Pfad von Wurzel zu Blatt $\leftrightarrow$ Anzahl Vergleiche, die Algorithmus zum Sortieren der Permutation aus dem Blatt benötigt.

[9] Andernfalls existieren Eingaben, die das Verfahren nicht sortieren kann.

Führt Algorithmus auch noch Vergleiche in den Blättern durch $\rightsquigarrow$ nutzlos, für untere Schranke damit uninteressant.

Sortieralgorithmus muss für alle möglichen Eingaben funktionieren $\rightsquigarrow$ Vergleichsbaum muss für jedes Verfahren mindestens $n!$ Blätter besitzen[9].

Hat der Baum mehr als $n!$ Blätter $\rightsquigarrow$ unnötige Vergleiche, wir können entsprechende Blätter löschen.

Anzahl innerer Knoten auf Pfad von Wurzel zu Blatt $\leftrightarrow$ Anzahl Vergleiche, die Algorithmus zum Sortieren der Permutation aus dem Blatt benötigt.

[9] Andernfalls existieren Eingaben, die das Verfahren nicht sortieren kann.

Führt Algorithmus auch noch Vergleiche in den Blättern durch $\rightsquigarrow$ nutzlos, für untere Schranke damit uninteressant.

Sortieralgorithmus muss für alle möglichen Eingaben funktionieren $\rightsquigarrow$ Vergleichsbaum muss für jedes Verfahren mindestens $n!$ Blätter besitzen[9].

Hat der Baum mehr als $n!$ Blätter $\rightsquigarrow$ unnötige Vergleiche, wir können entsprechende Blätter löschen.

Anzahl innerer Knoten auf Pfad von Wurzel zu Blatt $\leftrightarrow$ Anzahl Vergleiche, die Algorithmus zum Sortieren der Permutation aus dem Blatt benötigt.

[9] Andernfalls existieren Eingaben, die das Verfahren nicht sortieren kann.

Führt Algorithmus auch noch Vergleiche in den Blättern durch $\rightsquigarrow$ nutzlos, für untere Schranke damit uninteressant.

Sortieralgorithmus muss für alle möglichen Eingaben funktionieren $\rightsquigarrow$ Vergleichsbaum muss für jedes Verfahren mindestens $n!$ Blätter besitzen[9].

Hat der Baum mehr als $n!$ Blätter $\rightsquigarrow$ unnötige Vergleiche, wir können entsprechende Blätter löschen.

Anzahl innerer Knoten auf Pfad von Wurzel zu Blatt $\leftrightarrow$ Anzahl Vergleiche, die Algorithmus zum Sortieren der Permutation aus dem Blatt benötigt.

[9] Andernfalls existieren Eingaben, die das Verfahren nicht sortieren kann.

$\rightsquigarrow$ Ein optimales Verfahren hat also für alle Blätter möglichst kurze Distanzen zur Wurzel.

Ein Binärbaum, der die Distanzen minimiert, ist möglichst ausgeglichen d. h. er besitzt auf jedem Niveau möglichst viele Knoten.
$\rightsquigarrow$ Auf Niveau 1 einen, auf Niveau 2 zwei und allgemein auf Niveau i genau 2^{i-1} viele Knoten.

Außerdem: Gibt es einen Knoten auf Niveau p $\rightsquigarrow$ es gibt Eingabe, für die das Verfahren mindestens p Vergleiche benötigt.

Sei $V(n)$ die minimale Anzahl an Vergleichen, die notwendig sind, n verschiedene Elemente zu sortieren.
$\rightsquigarrow$ $n! \leq 2^{V(n)}$; logarithmieren liefert $V(n) \geq \mathrm{ld}(n!)$.

$\rightsquigarrow$ Ein optimales Verfahren hat also für alle Blätter möglichst kurze Distanzen zur Wurzel.

Ein Binärbaum, der die Distanzen minimiert, ist möglichst ausgeglichen d. h. er besitzt auf jedem Niveau möglichst viele Knoten.
$\rightsquigarrow$ Auf Niveau 1 einen, auf Niveau 2 zwei und allgemein auf Niveau i genau 2^{i-1} viele Knoten.

Außerdem: Gibt es einen Knoten auf Niveau p $\rightsquigarrow$ es gibt Eingabe, für die das Verfahren mindestens p Vergleiche benötigt.

Sei $V(n)$ die minimale Anzahl an Vergleichen, die notwendig sind, n verschiedene Elemente zu sortieren.
$\rightsquigarrow n! \leq 2^{V(n)}$; logarithmieren liefert $V(n) \geq \mathrm{ld}(n!)$.

$\rightsquigarrow$ Ein optimales Verfahren hat also für alle Blätter möglichst kurze Distanzen zur Wurzel.

Ein Binärbaum, der die Distanzen minimiert, ist möglichst ausgeglichen d. h. er besitzt auf jedem Niveau möglichst viele Knoten.
$\rightsquigarrow$ Auf Niveau 1 einen, auf Niveau 2 zwei und allgemein auf Niveau i genau 2^{i-1} viele Knoten.

Außerdem: Gibt es einen Knoten auf Niveau p $\rightsquigarrow$ es gibt Eingabe, für die das Verfahren mindestens p Vergleiche benötigt.

Sei $V(n)$ die minimale Anzahl an Vergleichen, die notwendig sind, n verschiedene Elemente zu sortieren.
$\rightsquigarrow$ $n! \leq 2^{V(n)}$; logarithmieren liefert $V(n) \geq \mathrm{ld}(n!)$.

$\rightsquigarrow$ Ein optimales Verfahren hat also für alle Blätter möglichst kurze Distanzen zur Wurzel.

Ein Binärbaum, der die Distanzen minimiert, ist möglichst ausgeglichen d. h. er besitzt auf jedem Niveau möglichst viele Knoten.
$\rightsquigarrow$ Auf Niveau 1 einen, auf Niveau 2 zwei und allgemein auf Niveau i genau 2^{i-1} viele Knoten.

Außerdem: Gibt es einen Knoten auf Niveau $p \rightsquigarrow$ es gibt Eingabe, für die das Verfahren mindestens p Vergleiche benötigt.

Sei $V(n)$ die minimale Anzahl an Vergleichen, die notwendig sind, n verschiedene Elemente zu sortieren.
$\rightsquigarrow n! \leq 2^{V(n)}$; logarithmieren liefert $V(n) \geq \mathrm{ld}(n!)$.

Da $V(n)$ eine natürliche Zahl sein muss $\rightsquigarrow V(n) \geq \lceil \mathrm{ld}(n!) \rceil$.
Nun ist $n! = (n/e)^n \sqrt{2\pi n}(1 + \mathcal{O}(n^{-1}))$ (STIRLINGs Formel) und damit $\lceil \mathrm{ld}(n!) \rceil = n\mathrm{ld}(n) - n/\ln(2) + \frac{1}{2}\mathrm{ld}(n) + \mathcal{O}(1)$, also $V(n) \gtrsim n\mathrm{ld}(n)$. Damit haben wir bewiesen:

Satz

Jedes auf Vergleichen basierende Verfahren, dass für beliebige Eingabegröße n alle $n!$ möglichen Permutationen sortieren kann, muss im Worst-Case mindestens $n\mathrm{ld}(n)$ viele Vergleiche durchführen.

7.9 Priority Queues & Heapsort

Sorting summary

	inplace?	stable?	best	average	worst	remarks
selection	✔		$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	n exchanges
insertion	✔	✔	n	$\frac{1}{4} n^2$	$\frac{1}{2} n^2$	use for small n or partially ordered
shell	✔		$n \log_3 n$	?	$c\, n^{3/2}$	tight code; subquadratic
merge		✔	$\frac{1}{2} n \lg n$	$n \lg n$	$n \lg n$	$n \log n$ guarantee; stable
timsort		✔	n	$n \lg n$	$n \lg n$	improves mergesort when preexisting order
quick	✔		$n \lg n$	$2 n \ln n$	$\frac{1}{2} n^2$	$n \log n$ probabilistic guarantee; fastest in practice
3-way quick	✔		n	$2 n \ln n$	$\frac{1}{2} n^2$	improves quicksort when duplicate keys
?	✔	✔	n	$n \lg n$	$n \lg n$	holy sorting grail

7.10 System sorts