Datenstrukturen und Algorithmen

Vorlesung 7: Sortieren (K2)

Joost-Pieter Katoen

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

https://moves.rwth-aachen.de/teaching/ss-18/dsal/

11. Mai 2018

Übersicht

Übersicht

Die Bedeutung des Sortierens

Sortieren ist ein wichtiges Thema

- Sortieren wird häufig benutzt und hat viele Anwendungen.
- Sortierverfahren geben Ideen, wie man Algorithmen verbessern kann.
- Geniale und optimale Algorithmen wurden gefunden.

Die Bedeutung des Sortierens

Sortieren ist ein wichtiges Thema

- Sortieren wird häufig benutzt und hat viele Anwendungen.
- Sortierverfahren geben Ideen, wie man Algorithmen verbessern kann.
- Geniale und optimale Algorithmen wurden gefunden.

- Neben der Funktionsweise der Algorithmen widmen wir uns vor allem der Laufzeitanalyse.

Anwendungen des Sortierens

Beispiel (Suchen)

- Schnellere Suche ist die wohl häufigste Anwendung des Sortierens.
- Binäre Suche findet ein Element in $O(\log n)$.

Anwendungen des Sortierens

Beispiel (Suchen)

- Schnellere Suche ist die wohl häufigste Anwendung des Sortierens.
- Binäre Suche findet ein Element in $O(\log n)$.

Beispiel (Engstes Paar (closest pair))

- Gegeben seien n Zahlen. Finde das Paar mit dem geringstem Abstand.

Anwendungen des Sortierens

Beispiel (Suchen)

- Schnellere Suche ist die wohl häufigste Anwendung des Sortierens.
- Binäre Suche findet ein Element in $O(\log n)$.

Beispiel (Engstes Paar (closest pair))

- Gegeben seien n Zahlen. Finde das Paar mit dem geringstem Abstand.
- Nach dem Sortieren liegen die Paare nebeneinander.
  Der Aufwand ist dann noch $O(n)$.

Noch einige Anwendungen

Beispiel (Eigenschaften von Datenobjekten)

- Sind alle n Elemente einzigartig oder gibt es Duplikate?

Noch einige Anwendungen

Beispiel (Eigenschaften von Datenobjekten)

- Sind alle n Elemente einzigartig oder gibt es Duplikate?
- Das k-t größte Element einer Menge?

Noch einige Anwendungen

Beispiel (Eigenschaften von Datenobjekten)

- Sind alle n Elemente einzigartig oder gibt es Duplikate?
- Das k-t größte Element einer Menge?

Beispiel (Textkompression (Entropiekodierung))

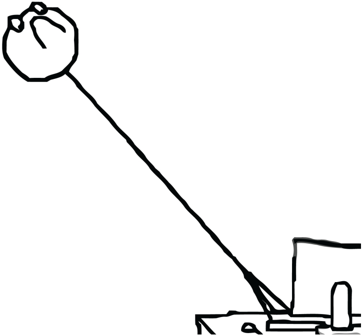
- Sortiere die Buchstaben nach Häufigkeit des Auftretens um sie dann effizient zu kodieren (d.h. mit möglichst kurzen Bitfolgen).

Sortieren ist nicht trivial!

TimSort: Python, Java (OpenJDK + Oracle), Android

Sortieren ist nicht trivial!

TimSort: Python, Java (OpenJDK + Oracle), Android



OpenJDK's java.utils.Collection.sort() is broken: The good, the bad and the worst case*

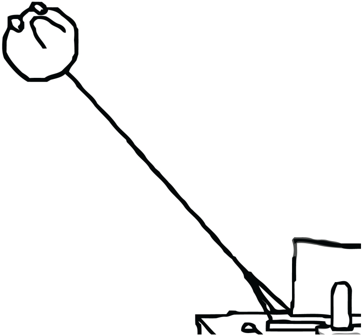
Stijn de Gouw[1,2], Jurriaan Rot[3,1], Frank S. de Boer[1,3], Richard Bubel[4], and Reiner Hähnle[4]

[1] CWI, Amsterdam, The Netherlands
[2] SDL, Amsterdam, The Netherlands
[3] Leiden University, The Netherlands
[4] Technische Universität Darmstadt, Germany

Abstract. We investigate the correctness of TimSort, which is the main sorting algorithm provided by the Java standard library. The goal is functional verification with mechanical proofs. During our verification attempt we discovered a bug which causes the implementation to crash. We characterize the conditions under which the bug occurs, and from this we derive a bug-free version that does not compromise the performance. We formally specify the new version and mechanically verify the absence of this bug with KeY, a state-of-the-art verification tool for Java.

Sortieren ist nicht trivial!

TimSort: Python, Java (OpenJDK + Oracle), Android



OpenJDK's java.utils.Collection.sort() is broken: The good, the bad and the worst case*

Stijn de Gouw[1,2], Jurriaan Rot[3,1], Frank S. de Boer[1,3], Richard Bubel[4], and Reiner Hähnle[4]

[1] CWI, Amsterdam, The Netherlands
[2] SDL, Amsterdam, The Netherlands
[3] Leiden University, The Netherlands
[4] Technische Universität Darmstadt, Germany

Abstract. We investigate the correctness of TimSort, which is the main sorting algorithm provided by the Java standard library. The goal is functional verification with mechanical proofs. During our verification attempt we discovered a bug which causes the implementation to crash. We characterize the conditions under which the bug occurs, and from this we derive a bug-free version that does not compromise the performance. We formally specify the new version and mechanically verify the absence of this bug with KeY, a state-of-the-art verification tool for Java.

Bekannte Fehler [Bearbeiten]

Im Februar 2015 stellte der Amsterdamer Informatiker Stijn de Gouw fest, dass alle Implementierungen des Timsort-Algorithmus einen Fehler enthalten.[6] Dieser Fehler hat keine praktischen Auswirkungen, da er nur auf Rechnern mit sehr viel Speicher auftreten kann, die zurzeit nicht existieren. Dennoch wurde der Fehler in Python[7] und Java[8] behoben.

Einige Hilfsbegriffe

Permutation

Eine Permutation einer Menge $A = \{a_1, \ldots, a_n\}$ ist eine bijektive Abbildung $\pi : A \to A$.

$$\begin{array}{ccccc} 3 & 7 & 12 & 0 & 4 \\ & & \Big\downarrow \pi & & \\ 0 & 7 & 12 & 4 & 3 \end{array}$$

Einige Hilfsbegriffe

Permutation

Eine Permutation einer Menge $A = \{a_1, \ldots, a_n\}$ ist eine bijektive Abbildung $\pi : A \to A$.

Totale Ordnung

Sei $A = \{a_1, \ldots, a_n\}$ eine Menge. Die binäre Relation $\leqslant \subseteq A \times A$ ist eine totale Ordnung (auf A) wenn für alle $a_i, a_j, a_k \in A$ gilt:

Einige Hilfsbegriffe

Permutation

Eine Permutation einer Menge $A = \{a_1, \ldots, a_n\}$ ist eine bijektive Abbildung $\pi : A \to A$.

Totale Ordnung

Sei $A = \{a_1, \ldots, a_n\}$ eine Menge. Die binäre Relation $\leqslant \subseteq A \times A$ ist eine totale Ordnung (auf A) wenn für alle $a_i, a_j, a_k \in A$ gilt:

1. Antisymmetrie: $a_i \leqslant a_j$ und $a_j \leqslant a_i$ impliziert $a_i = a_j$.

Einige Hilfsbegriffe

Permutation

Eine Permutation einer Menge $A = \{a_1, \ldots, a_n\}$ ist eine bijektive Abbildung $\pi : A \to A$.

Totale Ordnung

Sei $A = \{a_1, \ldots, a_n\}$ eine Menge. Die binäre Relation $\leqslant \subseteq A \times A$ ist eine totale Ordnung (auf A) wenn für alle $a_i, a_j, a_k \in A$ gilt:

1. Antisymmetrie: $a_i \leqslant a_j$ und $a_j \leqslant a_i$ impliziert $a_i = a_j$.
2. Transitivität: $a_i \leqslant a_j$ und $a_j \leqslant a_k$ impliziert $a_i \leqslant a_k$.

Einige Hilfsbegriffe

Permutation

Eine Permutation einer Menge $A = \{a_1, \ldots, a_n\}$ ist eine bijektive Abbildung $\pi : A \rightarrow A$.

Totale Ordnung

Sei $A = \{a_1, \ldots, a_n\}$ eine Menge. Die binäre Relation $\leqslant \subseteq A \times A$ ist eine totale Ordnung (auf A) wenn für alle $a_i, a_j, a_k \in A$ gilt:

1. Antisymmetrie: $a_i \leqslant a_j$ und $a_j \leqslant a_i$ impliziert $a_i = a_j$.
2. Transitivität: $a_i \leqslant a_j$ und $a_j \leqslant a_k$ impliziert $a_i \leqslant a_k$.
3. Totalität: $a_i \leqslant a_j$ oder $a_j \leqslant a_i$.

$(\mathbb{N}, \leq)$

totale Ordnung

$\subseteq$

z.B. $\{1,3\} \not\subseteq \{1,4,7\}$

$\not\supseteq$

Einige Hilfsbegriffe

Permutation

Eine Permutation einer Menge $A = \{a_1, \ldots, a_n\}$ ist eine bijektive Abbildung $\pi : A \rightarrow A$.

Totale Ordnung

Sei $A = \{a_1, \ldots, a_n\}$ eine Menge. Die binäre Relation $\leqslant \subseteq A \times A$ ist eine totale Ordnung (auf A) wenn für alle $a_i, a_j, a_k \in A$ gilt:

1. Antisymmetrie: $a_i \leqslant a_j$ und $a_j \leqslant a_i$ impliziert $a_i = a_j$.
2. Transitivität: $a_i \leqslant a_j$ und $a_j \leqslant a_k$ impliziert $a_i \leqslant a_k$.
3. Totalität: $a_i \leqslant a_j$ oder $a_j \leqslant a_i$.

Beispiel

Die lexikographische Ordnung von Zeichenketten und die numerische Ordnung von Zahlen sind totale Ordnungen.

Sortierproblem

Das Sortier-Problem

Eingabe:

1. Ein Array `E` mit `n` Einträgen.
2. Die Einträge gehören zu einer Menge A mit totaler Ordnung $\leqslant$.

Sortierproblem

Das Sortier-Problem

Eingabe:
1. Ein Array `E` mit `n` Einträgen.
2. Die Einträge gehören zu einer Menge A mit totaler Ordnung $\leqslant$.

Ausgabe: Ein Array `F` mit `n` Einträgen, so dass
1. `F[1]`, ..., `F[n]` eine Permutation von `E[1]`, ..., `E[n]` ist

Sortierproblem

Das Sortier-Problem

Eingabe:
1. Ein Array `E` mit `n` Einträgen.
2. Die Einträge gehören zu einer Menge A mit totaler Ordnung $\leqslant$.

Ausgabe: Ein Array `F` mit `n` Einträgen, so dass
1. `F[1]`, ..., `F[n]` eine Permutation von `E[1]`, ..., `E[n]` ist
2. Für alle $0 < i < n$ gilt: `F[i]` $\leqslant$ `F[i+1]`.

Sortierproblem

Das Sortier-Problem

Eingabe:
1. Ein Array `E` mit `n` Einträgen.
2. Die Einträge gehören zu einer Menge A mit totaler Ordnung $\leqslant$.

Ausgabe: Ein Array `F` mit `n` Einträgen, so dass
1. `F[1]`, ..., `F[n]` eine Permutation von `E[1]`, ..., `E[n]` ist
2. Für alle $0 < i < n$ gilt: `F[i]` $\leqslant$ `F[i+1]`.

Annahmen dieser Vorlesung

- Die zu sortierende Sequenz ist als Array organisiert, nicht als Liste.
- Die Elementaroperation ist ein Vergleich von Einträgen. E[i] = E[j] ?

Sortieralgorithmen

Beispiel (Einige Sortieralgorithmen)

Insertionsort, Bubblesort, Shellsort, Mergesort, Heapsort, Quicksort, Countingsort, Bucketsort, Radixsort, Stoogesort, Cocktailsort, Bogosort uvm.

Sortieralgorithmen

Beispiel (Einige Sortieralgorithmen)

Insertionsort, Bubblesort, Shellsort, Mergesort, Heapsort, Quicksort, Countingsort, Bucketsort, Radixsort, Stoogesort, Cocktailsort, Bogosort uvm.

Stabilität

Ein Sortieralgorithmus ist stabil wenn er die Reihenfolge der Elemente, deren Sortierschlüssel gleich sind, bewahrt.

Sortieralgorithmen

Beispiel (Einige Sortieralgorithmen)

Insertionsort, Bubblesort, Shellsort, Mergesort, Heapsort, Quicksort, Countingsort, Bucketsort, Radixsort, Stoogesort, Cocktailsort, Bogosort uvm.

Stabilität

Ein Sortieralgorithmus ist stabil wenn er die Reihenfolge der Elemente, deren Sortierschlüssel gleich sind, bewahrt.

Wenn z.B. eine Liste alphabetisch sortierter Personendateien nach dem Geburtsdatum neu sortiert wird, dann bleiben unter einem stabilen Sortierverfahren alle Personen mit gleichem Geburtsdatum alphabetisch sortiert.

Sortieralgorithmen

Beispiel (Einige Sortieralgorithmen)

Insertionsort, Bubblesort, Shellsort, Mergesort, Heapsort, Quicksort, Countingsort, Bucketsort, Radixsort, Stoogesort, Cocktailsort, Bogosort uvm.

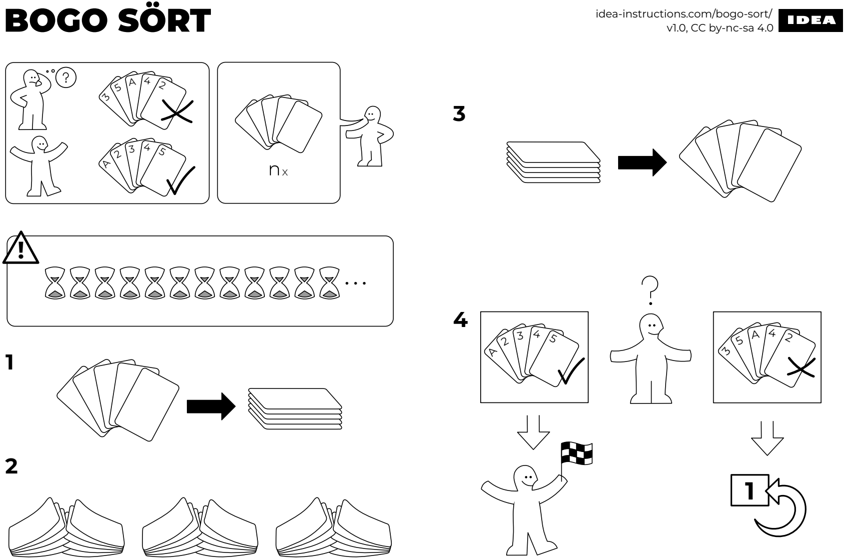
Stabilität

Ein Sortieralgorithmus ist stabil wenn er die Reihenfolge der Elemente, deren Sortierschlüssel gleich sind, bewahrt.

Wenn z.B. eine Liste alphabetisch sortierter Personendateien nach dem Geburtsdatum neu sortiert wird, dann bleiben unter einem stabilen Sortierverfahren alle Personen mit gleichem Geburtsdatum alphabetisch sortiert.

Wir werden erst einen einfachen Sortieralgorithmus betrachten.

Bogosort – IKEA-artige Erklärung



Dutch National Flag Problem (I)

Dutch National Flag Problem (II)

Beispiel (Das niederländische Flaggen-Problem [Dijkstra, 1972])

Eingabe:

1. Ein Array `E` mit `n` Einträgen, wobei für alle $0 < i \leqslant n$
   `E[i] == rot`, `E[i] == blau` *oder*
   `E[i] == weiss`
2. Ordnung: `rot` $<$ `weiss` $<$ `blau`

Dutch National Flag Problem (II)

Beispiel (Das niederländische Flaggen-Problem [Dijkstra, 1972])

Eingabe:
1. Ein Array `E` mit `n` Einträgen, wobei für alle $0 < i \leqslant n$
   `E[i] == rot`, `E[i] == blau` *oder*
   `E[i] == weiss`
2. Ordnung: `rot` $<$ `weiss` $<$ `blau`

Ausgabe: Ein sortiertes Array mit den Einträgen aus `E`.

Dutch National Flag Problem (II)

Beispiel (Das niederländische Flaggen-Problem [Dijkstra, 1972])

Eingabe:

1. Ein Array `E` mit `n` Einträgen, wobei für alle $0 < i \leqslant n$
   `E[i] == rot`, `E[i] == blau` *oder*
   `E[i] == weiss`
2. Ordnung: `rot` $<$ `weiss` $<$ `blau`

Ausgabe: Ein sortiertes Array mit den Einträgen aus `E`.

Erwünschte Worst-Case Zeitkomplexität: $\Theta(n)$.

Dutch National Flag Problem (II)

Beispiel (Das niederländische Flaggen-Problem [Dijkstra, 1972])

Eingabe:
1. Ein Array `E` mit `n` Einträgen, wobei für alle $0 < i \leqslant n$
   `E[i] == rot`, `E[i] == blau` *oder*
   `E[i] == weiss`
2. Ordnung: `rot` $<$ `weiss` $<$ `blau`

Ausgabe: Ein sortiertes Array mit den Einträgen aus `E`.

Erwünschte Worst-Case Zeitkomplexität: $\Theta(n)$.

Erwünschte Worst-Case Speicherkomplexität: $\Theta(1)$.

Dutch National Flag Problem (III)

Hauptidee

Zerlege das Array $\mathtt{E}$ in **4 Regionen:**

$$(1)\ 0 < i \leqslant \mathtt{r}, \quad (2)\ \mathtt{r} < i < \mathtt{u}, \quad (3)\ \mathtt{u} \leqslant i < \mathtt{b}, \quad \textbf{und}\ (4)\ \mathtt{b} \leqslant i \leqslant \mathtt{n}$$

für die Hilfsvariablen $\mathtt{r}$, $\mathtt{u}$ und $\mathtt{b}$,

Dutch National Flag Problem (III)

Hauptidee

Zerlege das Array `E` in **4 Regionen:**

(1) $0 < i \leqslant \mathtt{r}$, (2) $\mathtt{r} < i < \mathtt{u}$, (3) $\mathtt{u} \leqslant i < \mathtt{b}$, **und** (4) $\mathtt{b} \leqslant i \leqslant \mathtt{n}$

für die Hilfsvariablen `r`, `u` und `b`, so dass folgende Invariante gilt:

1. `E[1]`, ..., `E[r]` **ist die "rote" Region, d.h. für alle** $0 < i \leqslant \mathtt{r}$ `E[i] == rot`.

Dutch National Flag Problem (III)

Hauptidee

Zerlege das Array `E` in **4 Regionen:**

$$(1)\ 0 < i \leqslant \mathtt{r}, \quad (2)\ \mathtt{r} < i < \mathtt{u}, \quad (3)\ \mathtt{u} \leqslant i < \mathtt{b}, \quad \textbf{und}\ (4)\ \mathtt{b} \leqslant i \leqslant \mathtt{n}$$

für die Hilfsvariablen `r`, `u` und `b`, so dass folgende Invariante gilt:

1. **`E[1], ..., E[r]` ist die "rote" Region, d.h. für alle $0 < i \leqslant \mathtt{r}$**
   `E[i] == rot.`
2. **`E[r+1], ..., E[u-1]` ist die "weiße" Region, d.h. für alle $\mathtt{r} < i < \mathtt{u}$**
   `E[i] == weiss.`

Dutch National Flag Problem (III)

Hauptidee

Zerlege das Array `E` in **4 Regionen:**

(1) $0 < i \leqslant \mathtt{r}$, (2) $\mathtt{r} < i < \mathtt{u}$, (3) $\mathtt{u} \leqslant i < \mathtt{b}$, **und** (4) $\mathtt{b} \leqslant i \leqslant \mathtt{n}$

für die Hilfsvariablen `r`, `u` und `b`, so dass folgende Invariante gilt:

1. `E[1], ..., E[r]` **ist die "rote" Region, d.h. für alle** $0 < i \leqslant \mathtt{r}$ `E[i] == rot.`
2. `E[r+1], ..., E[u-1]` **ist die "weiße" Region, d.h. für alle** $\mathtt{r} < i < \mathtt{u}$ `E[i] == weiss.`
3. `E[u], ..., E[b-1]` **ist unbekannte Region, d.h. für alle** $\mathtt{u} \leqslant i < \mathtt{b}$ `E[i] == rot` **oder** `E[i] == weiss` **oder** `E[i] == blau.`

Dutch National Flag Problem (III)

Hauptidee

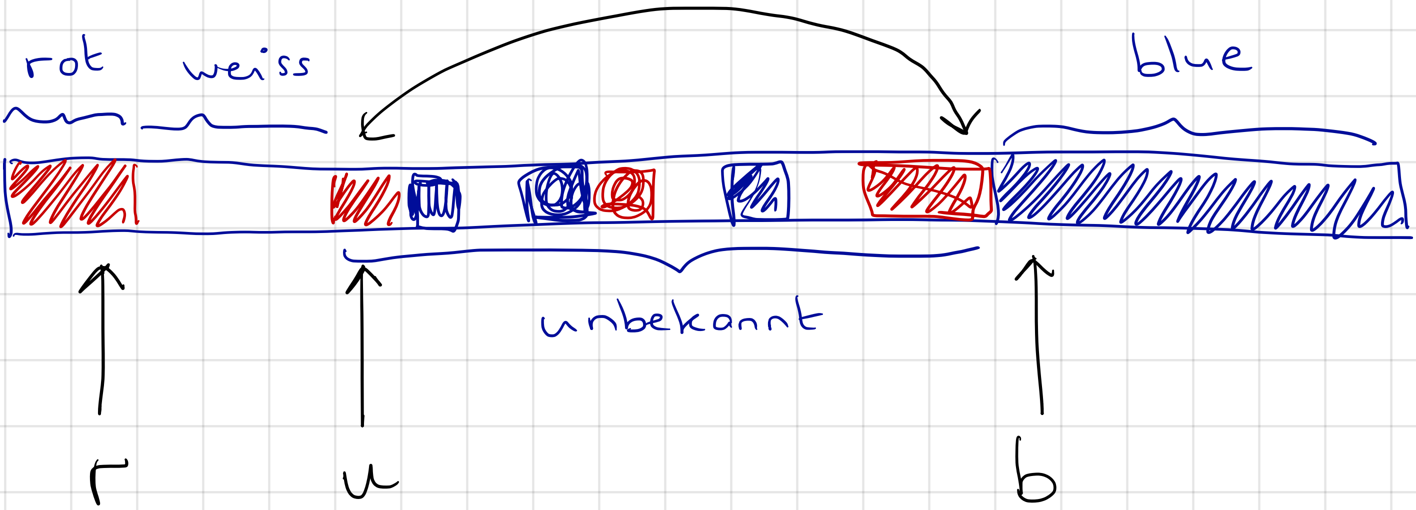
Zerlege das Array `E` in **4 Regionen:**

(1) $0 < i \leqslant \mathtt{r}$, (2) $\mathtt{r} < i < \mathtt{u}$, (3) $\mathtt{u} \leqslant i < \mathtt{b}$, **und** (4) $\mathtt{b} \leqslant i \leqslant \mathtt{n}$

für die Hilfsvariablen `r`, `u` und `b`, so dass folgende Invariante gilt:

1. `E[1], ..., E[r]` **ist die "rote" Region, d.h. für alle** $0 < i \leqslant \mathtt{r}$ `E[i] == rot.`
2. `E[r+1], ..., E[u-1]` **ist die "weiße" Region, d.h. für alle** $\mathtt{r} < i < \mathtt{u}$ `E[i] == weiss.`
3. `E[u], ..., E[b-1]` **ist unbekannte Region, d.h. für alle** $\mathtt{u} \leqslant i < \mathtt{b}$ `E[i] == rot` **oder** `E[i] == weiss` **oder** `E[i] == blau.`
4. `E[b], ..., E[n]` **ist die "blaue" Region, d. h. für alle** $\mathtt{b} \leqslant i \leqslant \mathtt{n}$ `E[i] == blau.`

<u>invariant DNF</u>



$\mathrm{swap}\,(E[u],\ E[b-1])$

Dutch National Flag Problem (III)

Hauptidee

Zerlege das Array `E` in **4 Regionen:**

$(1)\ 0 < i \leqslant \mathtt{r}, \quad (2)\ \mathtt{r} < i < \mathtt{u}, \quad (3)\ \mathtt{u} \leqslant i < \mathtt{b}, \quad$ **und** $(4)\ \mathtt{b} \leqslant i \leqslant \mathtt{n}$

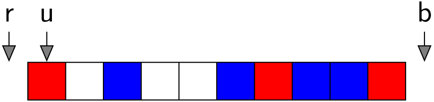
für die Hilfsvariablen `r`, `u` und `b`, so dass folgende Invariante gilt:

1. `E[1], ..., E[r]` **ist die "rote" Region, d.h. für alle** $0 < i \leqslant \mathtt{r}$
   `E[i] == rot.`
2. `E[r+1], ..., E[u-1]` **ist die "weiße" Region, d.h. für alle** $\mathtt{r} < i < \mathtt{u}$
   `E[i] == weiss.`
3. `E[u], ..., E[b-1]` **ist unbekannte Region, d.h. für alle** $\mathtt{u} \leqslant i < \mathtt{b}$
   `E[i] == rot` **oder** `E[i] == weiss` **oder** `E[i] == blau.`
4. `E[b], ..., E[n]` **ist die "blaue" Region, d. h. für alle** $\mathtt{b} \leqslant i \leqslant \mathtt{n}$
   `E[i] == blau.`

Arrayelemente können mit der `swap`-Operation vertauscht werden.

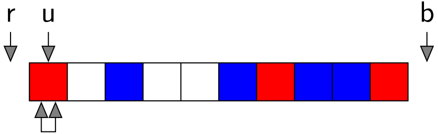
Dutch National Flag Problem (IV)

```
void DutchNationalFlag(Color E[], int n) {
  int r = 0, b = n + 1; // rote und blaue Regionen sind leer
  int u = 1; // weiße Region ist leer, die unbekannte == E
  while (u < b) {
    if (E[u] == rot) {
      swap(E[r + 1], E[u]);
      r = r + 1; // vergrößere die rote Region
      u = u + 1; // verkleinere die unbekannte Region
    }
    if (E[u] == weiss) {
      u = u + 1;
    }
    if (E[u] == blau) {
      swap(E[b - 1], E[u]);
      b = b - 1; // vergrößere die blaue Region
    }
  }
}
```



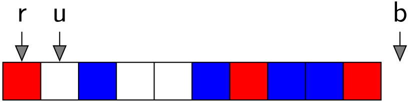
Dutch National Flag Problem (IV)

```
void DutchNationalFlag(Color E[], int n) {
  int r = 0, b = n + 1; // rote und blaue Regionen sind leer
  int u = 1; // weiße Region ist leer, die unbekannte == E
  while (u < b) {
    if (E[u] == rot) {
      swap(E[r + 1], E[u]);
      r = r + 1; // vergrößere die rote Region
      u = u + 1; // verkleinere die unbekannte Region
    }
    if (E[u] == weiss) {
      u = u + 1;
    }
    if (E[u] == blau) {
      swap(E[b - 1], E[u]);
      b = b - 1; // vergrößere die blaue Region
    }
  }
}
```



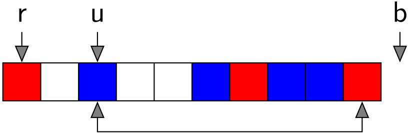
Dutch National Flag Problem (IV)

```
void DutchNationalFlag(Color E[], int n) {
  int r = 0, b = n + 1; // rote und blaue Regionen sind leer
  int u = 1; // weiße Region ist leer, die unbekannte == E
  while (u < b) {
    if (E[u] == rot) {
      swap(E[r + 1], E[u]);
      r = r + 1; // vergrößere die rote Region
      u = u + 1; // verkleinere die unbekannte Region
    }
    if (E[u] == weiss) {
      u = u + 1;
    }
    if (E[u] == blau) {
      swap(E[b - 1], E[u]);
      b = b - 1; // vergrößere die blaue Region
    }
  }
}
```



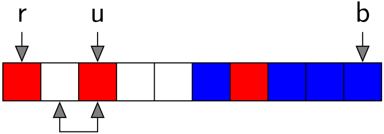
Dutch National Flag Problem (IV)

```
void DutchNationalFlag(Color E[], int n) {
  int r = 0, b = n + 1; // rote und blaue Regionen sind leer
  int u = 1; // weiße Region ist leer, die unbekannte == E
  while (u < b) {
    if (E[u] == rot) {
      swap(E[r + 1], E[u]);
      r = r + 1; // vergrößere die rote Region
      u = u + 1; // verkleinere die unbekannte Region
    }
    if (E[u] == weiss) {
      u = u + 1;
    }
    if (E[u] == blau) {
      swap(E[b - 1], E[u]);
      b = b - 1; // vergrößere die blaue Region
    }
  }
}
```



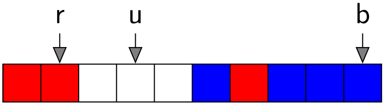
Dutch National Flag Problem (IV)

```
void DutchNationalFlag(Color E[], int n) {
  int r = 0, b = n + 1; // rote und blaue Regionen sind leer
  int u = 1; // weiße Region ist leer, die unbekannte == E
  while (u < b) {
    if (E[u] == rot) {
      swap(E[r + 1], E[u]);
      r = r + 1; // vergrößere die rote Region
      u = u + 1; // verkleinere die unbekannte Region
    }
    if (E[u] == weiss) {
      u = u + 1;
    }
    if (E[u] == blau) {
      swap(E[b - 1], E[u]);
      b = b - 1; // vergrößere die blaue Region
    }
  }
}
```



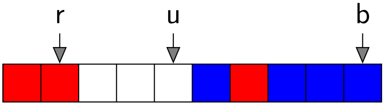
Dutch National Flag Problem (IV)

```
void DutchNationalFlag(Color E[], int n) {
  int r = 0, b = n + 1; // rote und blaue Regionen sind leer
  int u = 1; // weiße Region ist leer, die unbekannte == E
  while (u < b) {
    if (E[u] == rot) {
      swap(E[r + 1], E[u]);
      r = r + 1; // vergrößere die rote Region
      u = u + 1; // verkleinere die unbekannte Region
    }
    if (E[u] == weiss) {
      u = u + 1;
    }
    if (E[u] == blau) {
      swap(E[b - 1], E[u]);
      b = b - 1; // vergrößere die blaue Region
    }
  }
}
```



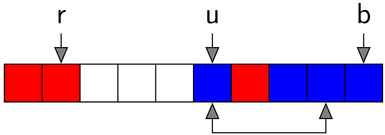
Dutch National Flag Problem (IV)

```
void DutchNationalFlag(Color E[], int n) {
  int r = 0, b = n + 1; // rote und blaue Regionen sind leer
  int u = 1; // weiße Region ist leer, die unbekannte == E
  while (u < b) {
    if (E[u] == rot) {
      swap(E[r + 1], E[u]);
      r = r + 1; // vergrößere die rote Region
      u = u + 1; // verkleinere die unbekannte Region
    }
    if (E[u] == weiss) {
      u = u + 1;
    }
    if (E[u] == blau) {
      swap(E[b - 1], E[u]);
      b = b - 1; // vergrößere die blaue Region
    }
  }
}
```



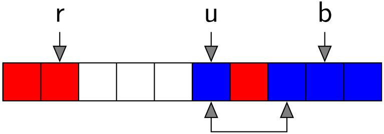
Dutch National Flag Problem (IV)

```
void DutchNationalFlag(Color E[], int n) {
  int r = 0, b = n + 1; // rote und blaue Regionen sind leer
  int u = 1; // weiße Region ist leer, die unbekannte == E
  while (u < b) {
    if (E[u] == rot) {
      swap(E[r + 1], E[u]);
      r = r + 1; // vergrößere die rote Region
      u = u + 1; // verkleinere die unbekannte Region
    }
    if (E[u] == weiss) {
      u = u + 1;
    }
    if (E[u] == blau) {
      swap(E[b - 1], E[u]);
      b = b - 1; // vergrößere die blaue Region
    }
  }
}
```



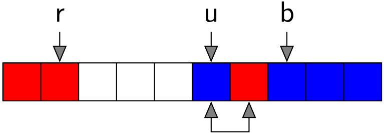
Dutch National Flag Problem (IV)

```
void DutchNationalFlag(Color E[], int n) {
  int r = 0, b = n + 1; // rote und blaue Regionen sind leer
  int u = 1; // weiße Region ist leer, die unbekannte == E
  while (u < b) {
    if (E[u] == rot) {
      swap(E[r + 1], E[u]);
      r = r + 1; // vergrößere die rote Region
      u = u + 1; // verkleinere die unbekannte Region
    }
    if (E[u] == weiss) {
      u = u + 1;
    }
    if (E[u] == blau) {
      swap(E[b - 1], E[u]);
      b = b - 1; // vergrößere die blaue Region
    }
  }
}
```



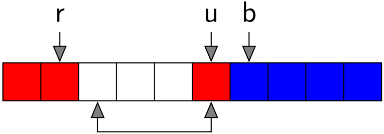
Dutch National Flag Problem (IV)

```
void DutchNationalFlag(Color E[], int n) {
  int r = 0, b = n + 1; // rote und blaue Regionen sind leer
  int u = 1; // weiße Region ist leer, die unbekannte == E
  while (u < b) {
    if (E[u] == rot) {
      swap(E[r + 1], E[u]);
      r = r + 1; // vergrößere die rote Region
      u = u + 1; // verkleinere die unbekannte Region
    }
    if (E[u] == weiss) {
      u = u + 1;
    }
    if (E[u] == blau) {
      swap(E[b - 1], E[u]);
      b = b - 1; // vergrößere die blaue Region
    }
  }
}
```



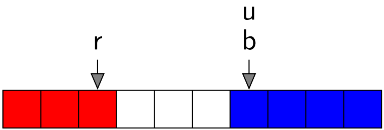
Dutch National Flag Problem (IV)

```
void DutchNationalFlag(Color E[], int n) {
  int r = 0, b = n + 1; // rote und blaue Regionen sind leer
  int u = 1; // weiße Region ist leer, die unbekannte == E
  while (u < b) {
    if (E[u] == rot) {
      swap(E[r + 1], E[u]);
      r = r + 1; // vergrößere die rote Region
      u = u + 1; // verkleinere die unbekannte Region
    }
    if (E[u] == weiss) {
      u = u + 1;
    }
    if (E[u] == blau) {
      swap(E[b - 1], E[u]);
      b = b - 1; // vergrößere die blaue Region
    }
  }
}
```



Dutch National Flag Problem (IV)

```
void DutchNationalFlag(Color E[], int n) {
  int r = 0, b = n + 1; // rote und blaue Regionen sind leer
  int u = 1; // weiße Region ist leer, die unbekannte == E
  while (u < b) {
    if (E[u] == rot) {
      swap(E[r + 1], E[u]);
      r = r + 1; // vergrößere die rote Region
      u = u + 1; // verkleinere die unbekannte Region
    }
    if (E[u] == weiss) {
      u = u + 1;
    }
    if (E[u] == blau) {
      swap(E[b - 1], E[u]);
      b = b - 1; // vergrößere die blaue Region
    }
  }
}
```



Dutch National Flag Problem (IV)

```
void DutchNationalFlag(Color E[], int n) {
  int r = 0, b = n + 1; // rote und blaue Regionen sind leer
  int u = 1; // weiße Region ist leer, die unbekannte == E
  while (u < b) {
    if (E[u] == rot) {
      swap(E[r + 1], E[u]);
      r = r + 1; // vergrößere die rote Region
      u = u + 1; // verkleinere die unbekannte Region
    }
    if (E[u] == weiss) {
      u = u + 1;
    }
    if (E[u] == blau) {
      swap(E[b - 1], E[u]);
      b = b - 1; // vergrößere die blaue Region
    }
  }
}
```

Dutch National Flag Problem (IV)

```
void DutchNationalFlag(Color E[], int n) {
  int r = 0, b = n + 1; // rote und blaue Regionen sind leer
  int u = 1; // weiße Region ist leer, die unbekannte == E
  while (u < b) {
    if (E[u] == rot) {
      swap(E[r + 1], E[u]);
      r = r + 1; // vergrößere die rote Region
      u = u + 1; // verkleinere die unbekannte Region
    }
    if (E[u] == weiss) {
      u = u + 1;
    }
    if (E[u] == blau) {
      swap(E[b - 1], E[u]);
      b = b - 1; // vergrößere die blaue Region
    }
  }
}
```

Frage: Ist DNF-Algorithmus ein stabiles Sortierverfahren?

Dutch National Flag Problem (IV)

```
void DutchNationalFlag(Color E[], int n) {
  int r = 0, b = n + 1; // rote und blaue Regionen sind leer
  int u = 1; // weiße Region ist leer, die unbekannte == E
  while (u < b) {
    if (E[u] == rot) {
      swap(E[r + 1], E[u]);
      r = r + 1; // vergrößere die rote Region
      u = u + 1; // verkleinere die unbekannte Region
    }
    if (E[u] == weiss) {
      u = u + 1;
    }
    if (E[u] == blau) {
      swap(E[b - 1], E[u]);
      b = b - 1; // vergrößere die blaue Region
    }
  }
}
```

Frage: Ist DNF-Algorithmus ein stabiles Sortierverfahren? Antwort: Nein.

Dutch National Flag Problem (V)

Speicherkomplexität

Die Worst-Case Speicherkomplexität vom DNF-Algorithmus ist $\Theta(1)$, da die einzigen extra Variablen `r`, `u` und `b` sind.

Dutch National Flag Problem (V)

Speicherkomplexität

Die Worst-Case Speicherkomplexität vom DNF-Algorithmus ist $\Theta(1)$, da die einzigen extra Variablen `r`, `u` und `b` sind.

DNF ist in-place, d. h. der Algorithmus arbeitet ohne zusätzlichen Speicherplatz.

Dutch National Flag Problem (V)

Speicherkomplexität

Die Worst-Case Speicherkomplexität vom DNF-Algorithmus ist $\Theta(1)$, da die einzigen extra Variablen `r`, `u` und `b` sind.

DNF ist in-place, d. h. der Algorithmus arbeitet ohne zusätzlichen Speicherplatz.

Zeitkomplexität

Betrachte als elementare Operationen Vergleiche der Form `E[i] == ....`

Die Worst-Case Zeitkomplexität ist $\Theta(n)$, da:

Dutch National Flag Problem (V)

Speicherkomplexität

Die Worst-Case Speicherkomplexität vom DNF-Algorithmus ist $\Theta(1)$, da die einzigen extra Variablen `r`, `u` und `b` sind.

DNF ist in-place, d. h. der Algorithmus arbeitet ohne zusätzlichen Speicherplatz.

Zeitkomplexität

Betrachte als elementare Operationen Vergleiche der Form `E[i] == ...`.

Die Worst-Case Zeitkomplexität ist $\Theta(n)$, da:

1. in jedem Durchlauf werden konstant viele Vergleiche durchgeführt

E[i] == weiss

Dutch National Flag Problem (V)

Speicherkomplexität

Die Worst-Case Speicherkomplexität vom DNF-Algorithmus ist $\Theta(1)$, da die einzigen extra Variablen `r`, `u` und `b` sind.

DNF ist in-place, d. h. der Algorithmus arbeitet ohne zusätzlichen Speicherplatz.

Zeitkomplexität

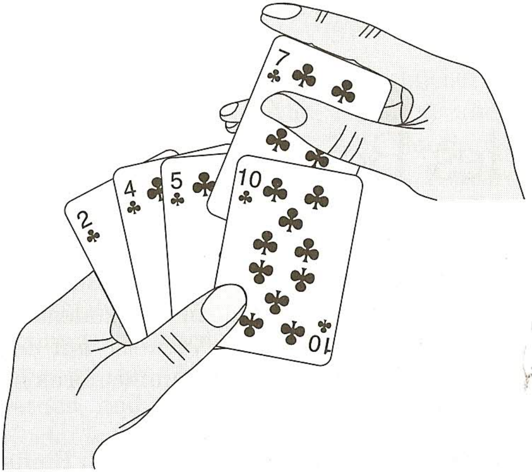
Betrachte als elementare Operationen Vergleiche der Form `E[i] == ...`.

Die Worst-Case Zeitkomplexität ist $\Theta(n)$, da:

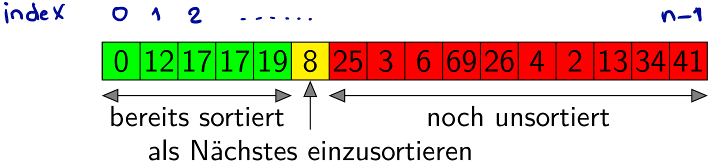
1. in jedem Durchlauf werden konstant viele Vergleiche durchgeführt
2. die Anzahl der Durchläufe ist $\Theta(n)$, da in jedem Durchlauf die Größe des unbekannten Gebiets `b - u` um mindestens eins und höchstens zwei verkleinert wird.

Übersicht

Sortieren durch Einfügen – Insertionsort

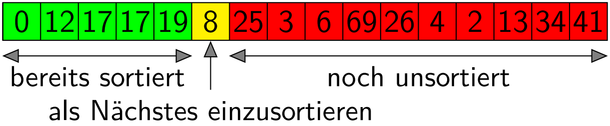


Sortieren durch Einfügen – Insertionsort



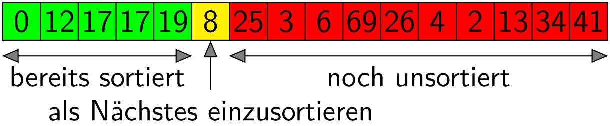
- Durchlaufen des (unsortierten) Arrays von links nach rechts.

Sortieren durch Einfügen – Insertionsort



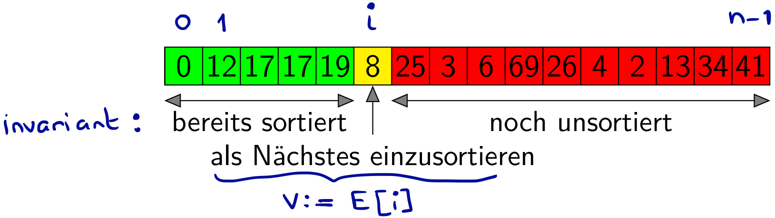
- Durchlaufen des (unsortierten) Arrays von links nach rechts.
- Gehe zum ersten bisher noch nicht berücksichtigte Element.

Sortieren durch Einfügen – Insertionsort



- Durchlaufen des (unsortierten) Arrays von links nach rechts.
- Gehe zum ersten bisher noch nicht berücksichtigte Element.
- Füge es im sortierten Teil (links) nach elementweisem Vergleichen ein.

Sortieren durch Einfügen – Insertionsort



- Durchlaufen des (unsortierten) Arrays von links nach rechts.
- Gehe zum ersten bisher noch nicht berücksichtigte Element.
- Füge es im sortierten Teil (links) nach elementweisem Vergleichen ein.
- Dieser Algorithmus funktioniert auch mit anderen linearen Anordnungen, etwa Listen.

Insertionsort – Animation und Algorithmus

0	1	2													n-1
41	26	17	25	19	17	8	3	6	69	12	4	2	13	34	0

=v

i

j

```
void insertionSort(int E[]) {
  int i,j;
  for (i = 1; i < E.length; i++) {
    int v = E[i]; // speichere E[i]
    for (j = i; j > 0 && E[j-1] > v; j--) {
      E[j] = E[j-1]; // schiebe Element j-1 nach rechts
    }
    E[j] = v; // füge E[i] an der richtigen Stelle ein
  }
}
```

Insertionsort – Animation und Algorithmus

26	41	17	25	19	17	8	3	6	69	12	4	2	13	34	0

```java
void insertionSort(int E[]) {
  int i,j;
  for (i = 1; i < E.length; i++) {
    int v = E[i]; // speichere E[i]
    for (j = i; j > 0 && E[j-1] > v; j--) {
      E[j] = E[j-1]; // schiebe Element j-1 nach rechts
    }
    E[j] = v; // füge E[i] an der richtigen Stelle ein
  }
}
```

Insertionsort – Animation und Algorithmus

17	26	41	25	19	17	8	3	6	69	12	4	2	13	34	0

$i = 4$

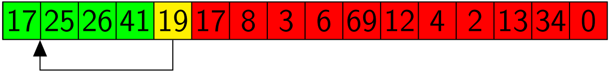
```java
void insertionSort(int E[]) {
  int i,j;
  for (i = 1; i < E.length; i++) {
    int v = E[i]; // speichere E[i]
    for (j = i; j > 0 && E[j-1] > v; j--) {
      E[j] = E[j-1]; // schiebe Element j-1 nach rechts
    }
    E[j] = v; // füge E[i] an der richtigen Stelle ein
  }
}
```

$j = 4 \quad \underbrace{41}_{E[j-1]} > 25$

$j = 3 \quad 26 > 25$

$j = 2 \quad 17 \leq 25$

Insertionsort – Animation und Algorithmus



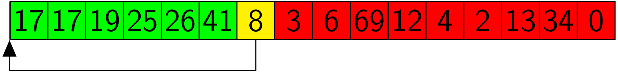

```java
void insertionSort(int E[]) {
  int i,j;
  for (i = 1; i < E.length; i++) {
    int v = E[i]; // speichere E[i]
    for (j = i; j > 0 && E[j-1] > v; j--) {
      E[j] = E[j-1]; // schiebe Element j-1 nach rechts
    }
    E[j] = v; // füge E[i] an der richtigen Stelle ein
  }
}
```

Insertionsort – Animation und Algorithmus

17	19	25	26	41	17	8	3	6	69	12	4	2	13	34	0

```
void insertionSort(int E[]) {
  int i,j;
  for (i = 1; i < E.length; i++) {
    int v = E[i]; // speichere E[i]
    for (j = i; j > 0 && E[j-1] > v; j--) {
      E[j] = E[j-1]; // schiebe Element j-1 nach rechts
    }
    E[j] = v; // füge E[i] an der richtigen Stelle ein
  }
}
```

Insertionsort – Animation und Algorithmus



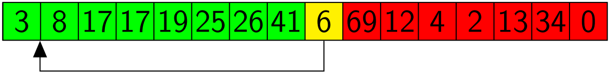

```
void insertionSort(int E[]) {
  int i,j;
  for (i = 1; i < E.length; i++) {
    int v = E[i]; // speichere E[i]
    for (j = i; j > 0 && E[j-1] > v; j--) {
      E[j] = E[j-1]; // schiebe Element j-1 nach rechts
    }
    E[j] = v; // füge E[i] an der richtigen Stelle ein
  }
}
```

Insertionsort – Animation und Algorithmus

8	17	17	19	25	26	41	3	6	69	12	4	2	13	34	0

```
void insertionSort(int E[]) {
  int i,j;
  for (i = 1; i < E.length; i++) {
    int v = E[i]; // speichere E[i]
    for (j = i; j > 0 && E[j-1] > v; j--) {
      E[j] = E[j-1]; // schiebe Element j-1 nach rechts
    }
    E[j] = v; // füge E[i] an der richtigen Stelle ein
  }
}
```

Insertionsort – Animation und Algorithmus




```
void insertionSort(int E[]) {
  int i,j;
  for (i = 1; i < E.length; i++) {
    int v = E[i]; // speichere E[i]
    for (j = i; j > 0 && E[j-1] > v; j--) {
      E[j] = E[j-1]; // schiebe Element j-1 nach rechts
    }
    E[j] = v; // füge E[i] an der richtigen Stelle ein
  }
}
```

Insertionsort – Animation und Algorithmus

3	6	8	17	17	19	25	26	41	69	12	4	2	13	34	0

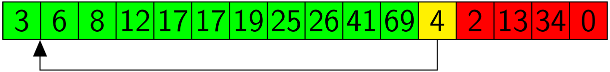
```
void insertionSort(int E[]) {
  int i,j;
  for (i = 1; i < E.length; i++) {
    int v = E[i]; // speichere E[i]
    for (j = i; j > 0 && E[j-1] > v; j--) {
      E[j] = E[j-1]; // schiebe Element j-1 nach rechts
    }
    E[j] = v; // füge E[i] an der richtigen Stelle ein
  }
}
```

Insertionsort – Animation und Algorithmus

3	6	8	17	17	19	25	26	41	69	12	4	2	13	34	0

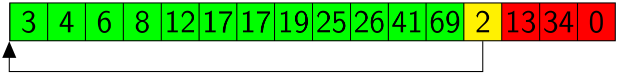
```
void insertionSort(int E[]) {
  int i,j;
  for (i = 1; i < E.length; i++) {
    int v = E[i]; // speichere E[i]
    for (j = i; j > 0 && E[j-1] > v; j--) {
      E[j] = E[j-1]; // schiebe Element j-1 nach rechts
    }
    E[j] = v; // füge E[i] an der richtigen Stelle ein
  }
}
```

Insertionsort – Animation und Algorithmus



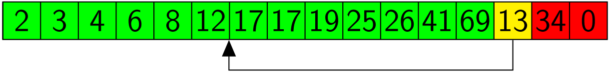

```
void insertionSort(int E[]) {
  int i,j;
  for (i = 1; i < E.length; i++) {
    int v = E[i]; // speichere E[i]
    for (j = i; j > 0 && E[j-1] > v; j--) {
      E[j] = E[j-1]; // schiebe Element j-1 nach rechts
    }
    E[j] = v; // füge E[i] an der richtigen Stelle ein
  }
}
```

Insertionsort – Animation und Algorithmus



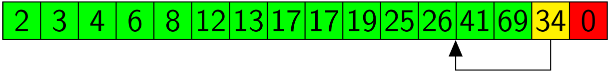

```
void insertionSort(int E[]) {
  int i,j;
  for (i = 1; i < E.length; i++) {
    int v = E[i]; // speichere E[i]
    for (j = i; j > 0 && E[j-1] > v; j--) {
      E[j] = E[j-1]; // schiebe Element j-1 nach rechts
    }
    E[j] = v; // füge E[i] an der richtigen Stelle ein
  }
}
```

Insertionsort – Animation und Algorithmus




```
void insertionSort(int E[]) {
  int i,j;
  for (i = 1; i < E.length; i++) {
    int v = E[i]; // speichere E[i]
    for (j = i; j > 0 && E[j-1] > v; j--) {
      E[j] = E[j-1]; // schiebe Element j-1 nach rechts
    }
    E[j] = v; // füge E[i] an der richtigen Stelle ein
  }
}
```

Insertionsort – Animation und Algorithmus




```
void insertionSort(int E[]) {
  int i,j;
  for (i = 1; i < E.length; i++) {
    int v = E[i]; // speichere E[i]
    for (j = i; j > 0 && E[j-1] > v; j--) {
      E[j] = E[j-1]; // schiebe Element j-1 nach rechts
    }
    E[j] = v; // füge E[i] an der richtigen Stelle ein
  }
}
```

Insertionsort – Animation und Algorithmus

2	3	4	6	8	12	13	17	17	19	25	26	34	41	69	0

```
void insertionSort(int E[]) {
  int i,j;
  for (i = 1; i < E.length; i++) {
    int v = E[i]; // speichere E[i]
    for (j = i; j > 0 && E[j-1] > v; j--) {
      E[j] = E[j-1]; // schiebe Element j-1 nach rechts
    }
    E[j] = v; // füge E[i] an der richtigen Stelle ein
  }
}
```

Insertionsort – Animation und Algorithmus

0	2	3	4	6	8	12	13	17	17	19	25	26	34	41	69

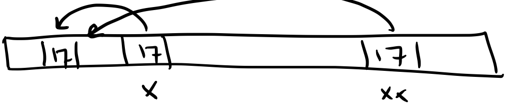
```java
void insertionSort(int E[]) {
  int i,j;
  for (i = 1; i < E.length; i++) {
    int v = E[i]; // speichere E[i]
    for (j = i; j > 0 && E[j-1] > v; j--) {
      E[j] = E[j-1]; // schiebe Element j-1 nach rechts
    }
    E[j] = v; // füge E[i] an der richtigen Stelle ein
  }
}
```

Insertionsort – Animation und Algorithmus

```
void insertionSort(int E[]) {
  int i,j;
  for (i = 1; i < E.length; i++) {
    int v = E[i]; // speichere E[i]
    for (j = i; j > 0 && E[j-1] > v; j--) {
      E[j] = E[j-1]; // schiebe Element j-1 nach rechts
    }
    E[j] = v; // füge E[i] an der richtigen Stelle ein
  }
}
```

- Insertionsort ist in-place, d.h. der Algorithmus arbeitet ohne zusätzlichen Speicherplatz.

Insertionsort – Animation und Algorithmus




```
void insertionSort(int E[]) {
  int i,j;
  for (i = 1; i < E.length; i++) {
    int v = E[i]; // speichere E[i]
    for (j = i; j > 0 && E[j-1] > v; j--) {
      E[j] = E[j-1]; // schiebe Element j-1 nach rechts
    }
    E[j] = v; // füge E[i] an der richtigen Stelle ein
  }
}
```

- Insertionsort ist in-place, d.h. der Algorithmus arbeitet ohne zusätzlichen Speicherplatz.
- Insertionsort ist stabil, da die Reihenfolge der gleichwertigen Arrayelemente unverändert bleibt.

Insertionsort – Best- und Worst-Case-Analyse

Best-Case

- Im Best-Case ist das Array bereits sortiert.

Insertionsort – Best- und Worst-Case-Analyse

Best-Case

- Im Best-Case ist das Array bereits sortiert.
- Pro Element ist daher nur ein Vergleich nötig.

Insertionsort – Best- und Worst-Case-Analyse

Best-Case

- Im Best-Case ist das Array bereits sortiert.
- Pro Element ist daher nur ein Vergleich nötig.

$\Rightarrow$ Es gilt: $B(n) = n - 1 \in \Theta(n)$

Insertionsort – Best- und Worst-Case-Analyse

Best-Case

- Im Best-Case ist das Array bereits sortiert.
- Pro Element ist daher nur ein Vergleich nötig.

$\Rightarrow$ Es gilt: $B(n) = n - 1 \in \Theta(n)$

Worst-Case

- Im Worst-Case wird das einzusortierende Element immer ganz vorne eingefügt.

7 6 4 3 1 0

6 7 4 3 1 0

Insertionsort – Best- und Worst-Case-Analyse

Best-Case

- Im Best-Case ist das Array bereits sortiert.
- Pro Element ist daher nur ein Vergleich nötig.

$\Rightarrow$ Es gilt: $B(n) = n - 1 \in \Theta(n)$

Worst-Case

- Im Worst-Case wird das einzusortierende Element immer ganz vorne eingefügt.
- Es muss mit allen vorhergehenden Elementen verglichen werden.

Insertionsort – Best- und Worst-Case-Analyse

Best-Case

- Im Best-Case ist das Array bereits sortiert.
- Pro Element ist daher nur ein Vergleich nötig.

$\Rightarrow$ Es gilt: $B(n) = n - 1 \in \Theta(n)$

Worst-Case

- Im Worst-Case wird das einzusortierende Element immer ganz vorne eingefügt.
- Es muss mit allen vorhergehenden Elementen verglichen werden.
- Das tritt etwa auf, wenn das Array umgekehrt vorsortiert war.

Insertionsort – Best- und Worst-Case-Analyse

Best-Case

- Im Best-Case ist das Array bereits sortiert.
- Pro Element ist daher nur ein Vergleich nötig.

⇒ Es gilt: $B(n) = n - 1 \in \Theta(n)$

Worst-Case

- Im Worst-Case wird das einzusortierende Element immer ganz vorne eingefügt.
- Es muss mit allen vorhergehenden Elementen verglichen werden.
- Das tritt etwa auf, wenn das Array umgekehrt vorsortiert war.

⇒ Zum Einsortieren des i-ten Elements sind im schlimmsten Fall i Vergleiche nötig:

Insertionsort – Best- und Worst-Case-Analyse

Best-Case

- Im Best-Case ist das Array bereits sortiert.
- Pro Element ist daher nur ein Vergleich nötig.

$\Rightarrow$ Es gilt: $B(n) = n - 1 \in \Theta(n)$

Worst-Case

- Im Worst-Case wird das einzusortierende Element immer ganz vorne eingefügt.
- Es muss mit allen vorhergehenden Elementen verglichen werden.
- Das tritt etwa auf, wenn das Array umgekehrt vorsortiert war.

$\Rightarrow$ Zum Einsortieren des i-ten Elements sind im schlimmsten Fall i Vergleiche nötig: $W(n) = \sum_{i=1}^{n-1} i =$

Insertionsort – Best- und Worst-Case-Analyse

Best-Case

- Im Best-Case ist das Array bereits sortiert.
- Pro Element ist daher nur ein Vergleich nötig.

⇒ Es gilt: $B(n) = n - 1 \in \Theta(n)$

Worst-Case

- Im Worst-Case wird das einzusortierende Element immer ganz vorne eingefügt.
- Es muss mit allen vorhergehenden Elementen verglichen werden.
- Das tritt etwa auf, wenn das Array umgekehrt vorsortiert war.

⇒ Zum Einsortieren des i-ten Elements sind im schlimmsten Fall i Vergleiche nötig: $W(n) = \sum_{i=1}^{n-1} i = \frac{n \cdot (n-1)}{2} \in \Theta(n^2)$

Insertionsort – Average-Case-Analyse (I)

Annahmen für die Average-Case-Analyse

- Alle Permutationen von Elementen treten in gleicher Häufigkeit auf.
- Die zu sortierenden Elemente sind alle verschieden.
- Die Array-Plätze sind `E[0]` bis `E[n-1]`.

Insertionsort – Average-Case-Analyse (I)

Annahmen für die Average-Case-Analyse

- Alle Permutationen von Elementen treten in gleicher Häufigkeit auf.
- Die zu sortierenden Elemente sind alle verschieden.
- Die Array-Plätze sind `E[0]` bis `E[n-1]`.

Es gilt:

$$A(n) = \sum_{i=1}^{n-1} \underbrace{\text{erwartete Anzahl an Vergleichen, um } \texttt{E[i]} \text{ einzusortieren}}$$

$$\sum_{\cancel{i=0}}^{n-1}$$

Insertionsort – Average-Case-Analyse (I)

Annahmen für die Average-Case-Analyse

- Alle Permutationen von Elementen treten in gleicher Häufigkeit auf.
- Die zu sortierenden Elemente sind alle verschieden.
- Die Array-Plätze sind `E[0]` bis `E[n-1]`.

Es gilt:

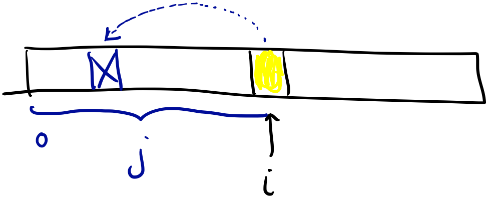
$$A(n) = \sum_{i=1}^{n-1} \overbrace{\underbrace{\text{erwartete Anzahl an Vergleichen, um } \texttt{E[i]} \text{ einzusortieren}}_{*}}^{\text{Pr} \cdot \text{kosten}}$$

Frage: wieso wird der Fall $i=0$ nicht betrachtet?

Die erwartete Anzahl an Vergleichen, um den richtigen Platz für `E[i]` zu finden wird dann wie folgt hergeleitet:

Insertionsort – Average-Case-Analyse (II)

$$\sum_{j=0}^{i} \underbrace{\Pr\left\{\begin{array}{l} i\text{-tes Element wird}\\ \text{an Position } j \text{ eingefügt}\end{array}\right\}}_{1} \cdot \underbrace{\begin{array}{l}\text{Anzahl Vergleiche, um } \texttt{E[i]}\\ \text{an Position } j \text{ einzufügen}\end{array}}_{\text{kosten}}$$

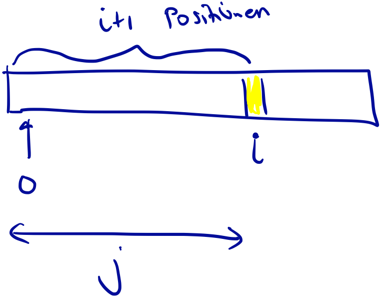


Insertionsort – Average-Case-Analyse (II)

$$\sum_{j=0}^{i} \underbrace{\Pr\left\{\begin{array}{l} i\text{-tes Element wird} \\ \text{an Position } j \text{ eingefügt} \end{array}\right\}}_{1} \cdot \begin{array}{l} \text{Anzahl Vergleiche, um } \texttt{E[i]} \\ \text{an Position } j \text{ einzufügen} \end{array} \quad 2$$

E[i] wird an beliebiger Position j
mit gleicher W'lichkeit eingefügt

$$= \sum_{j=0}^{i} \underbrace{\frac{1}{i+1}} \cdot \begin{array}{l} \text{Anzahl Vergleiche, um } \texttt{E[i]} \\ \text{an Position } j \text{ einzufügen} \end{array}$$



Insertionsort – Average-Case-Analyse (II)

$$\sum_{j=0}^{i} \Pr\left\{\begin{array}{l} i\text{-tes Element wird} \\ \text{an Position } j \text{ eingefügt}\end{array}\right\} \cdot \begin{array}{l}\text{Anzahl Vergleiche, um } \texttt{E[i]} \\ \text{an Position } j \text{ einzufügen}\end{array}$$

E[i] wird an beliebiger Position j mit gleicher W'lichkeit eingefügt

$$= \sum_{j=0}^{i} \frac{1}{i+1} \cdot \begin{array}{l}\text{Anzahl Vergleiche, um } \texttt{E[i]} \\ \text{an Position } j \text{ einzufügen}\end{array}$$

Anzahl Vergleiche, um an Position 0 einzufügen ist i, sonst $i-j+1$.

$$= \underbrace{\overbrace{\frac{1}{i+1}}^{\Pr} \cdot i}_{E[0]} + \underbrace{\overbrace{\frac{1}{i+1}}^{\Pr} \cdot \sum_{j=1}^{i}(i-j+1)}_{E[1] \ldots E[i]}$$

E[0] = E[i] → i Vergleiche

E[1] = E[i] → i Vergleiche
E[2] = E[i] → i−1 Vergl.
E[3] = E[i] → i−2 Vergl.
...

Insertionsort – Average-Case-Analyse (II)

$$\sum_{j=0}^{i} \Pr\left\{\begin{array}{l} i\text{-tes Element wird} \\ \text{an Position } j \text{ eingefügt}\end{array}\right\} \cdot \begin{array}{l}\text{Anzahl Vergleiche, um } \texttt{E[i]} \\ \text{an Position } j \text{ einzufügen}\end{array}$$

| E[i] wird an beliebiger Position j mit gleicher W'lichkeit eingefügt

$$= \sum_{j=0}^{i} \frac{1}{i+1} \cdot \begin{array}{l}\text{Anzahl Vergleiche, um } \texttt{E[i]} \\ \text{an Position } j \text{ einzufügen}\end{array}$$

| Anzahl Vergleiche, um an Position 0 einzufügen ist i, sonst $i-j+1$.

$$= \frac{1}{i+1} \cdot i + \frac{1}{i+1} \cdot \sum_{j=1}^{i} (i-j+1)$$

i

$i-1$

$\vdots$

1

| Vereinfachen

$$= \frac{i}{i+1} + \frac{1}{i+1} \cdot \sum_{j=1}^{i} j = \frac{i}{i+1} + \frac{1}{i+1} \cdot \frac{i\cdot(i+1)}{2} = \boxed{\frac{i}{2} + 1 - \frac{1}{i+1}} \quad (*)$$

Insertionsort – Average-Case-Analyse (III)

Damit gilt für $A(n)$:

$$A(n) = \sum_{i=1}^{n-1} \underbrace{\left(\frac{i}{2} + 1 - \frac{1}{i+1}\right)}_{\text{letzte Folie}} \quad (*)$$

Insertionsort – Average-Case-Analyse (III)

Damit gilt für $A(n)$:

$$A(n) = \sum_{i=1}^{n-1} \left(\frac{i}{2} + 1 - \frac{1}{i+1} \right)$$

Auseinanderziehen und Indexverschiebung

$$= \frac{n \cdot (n-1)}{4} + (n-1) - \sum_{i=2}^{n} \frac{1}{i}$$

$$\underbrace{\sum_{i=1}^{n-1} \frac{i}{2}} + \underbrace{\sum_{i=1}^{n-1} 1} - \underbrace{\sum_{i=1}^{n-1} \frac{1}{i+1}}$$

$$\underbrace{\frac{1}{2} \cdot \sum_{i=1}^{n-1} i} \qquad \frac{n(n-1)}{2} \qquad = \sum_{i=2}^{n} \frac{1}{i}$$

Insertionsort – Average-Case-Analyse (III)

Damit gilt für $A(n)$:

$$A(n) = \sum_{i=1}^{n-1} \left(\frac{i}{2} + 1 - \frac{1}{i+1} \right)$$

| Auseinanderziehen und Indexverschiebung

$$= \frac{n \cdot (n-1)}{4} + (n - 1) - \sum_{i=2}^{n} \frac{1}{i}$$

| Verschieben des Summenstarts

$$= \frac{n \cdot (n-1)}{4} + n - \sum_{i=1}^{n} \frac{1}{i}$$

Insertionsort – Average-Case-Analyse (III)

Damit gilt für $A(n)$:

$$A(n) = \sum_{i=1}^{n-1} \left(\frac{i}{2} + 1 - \frac{1}{i+1} \right)$$

| Auseinanderziehen und Indexverschiebung

$$= \frac{n \cdot (n-1)}{4} + (n-1) - \sum_{i=2}^{n} \frac{1}{i}$$

| Verschieben des Summenstarts

$$= \frac{n \cdot (n-1)}{4} + n - \sum_{i=1}^{n} \frac{1}{i}$$

| Harmonische Reihe: $\sum_{i=1}^{n} (1/i) \approx \ln n$

Dominant

$$A(n) \approx \frac{n \cdot (n-1)}{4} + n - \ln n \quad \in \Theta(n^2)$$

Übersicht

Divide-and-Conquer

Teile-und-Beherrsche Algorithmen (divide-and-conquer) teilen das Problem in mehrere Teilprobleme auf, $\underbrace{\text{die dem Ausgangsproblem ähneln}}$, jedoch von kleinerer Größe sind.

Rekursion!

Divide-and-Conquer

Teile-und-Beherrsche Algorithmen (divide-and-conquer) teilen das Problem in mehrere Teilprobleme auf, die dem Ausgangsproblem ähneln, jedoch von kleinerer Größe sind.

Sie lösen die Teilprobleme rekursiv und kombinieren diese Lösungen dann, um die Lösung des eigentlichen Problems zu erstellen.

Divide-and-Conquer

Teile-und-Beherrsche Algorithmen (divide-and-conquer) teilen das Problem in mehrere Teilprobleme auf, die dem Ausgangsproblem ähneln, jedoch von kleinerer Größe sind.

Sie lösen die Teilprobleme rekursiv und kombinieren diese Lösungen dann, um die Lösung des eigentlichen Problems zu erstellen.

Das Paradigma von Teile-und-Beherrsche umfasst 3 Schritte auf jeder Rekursionsebene:

Divide-and-Conquer

Teile-und-Beherrsche Algorithmen (divide-and-conquer) teilen das Problem in mehrere Teilprobleme auf, die dem Ausgangsproblem ähneln, jedoch von kleinerer Größe sind.

Sie lösen die Teilprobleme rekursiv und kombinieren diese Lösungen dann, um die Lösung des eigentlichen Problems zu erstellen.

Das Paradigma von Teile-und-Beherrsche umfasst 3 Schritte auf jeder Rekursionsebene:

Teile das Problem in eine Anzahl von Teilproblemen auf.

Divide-and-Conquer

Teile-und-Beherrsche Algorithmen (divide-and-conquer) teilen das Problem in mehrere Teilprobleme auf, die dem Ausgangsproblem ähneln, jedoch von kleinerer Größe sind.

Sie lösen die Teilprobleme rekursiv und kombinieren diese Lösungen dann, um die Lösung des eigentlichen Problems zu erstellen.

Das Paradigma von Teile-und-Beherrsche umfasst 3 Schritte auf jeder Rekursionsebene:

Teile das Problem in eine Anzahl von Teilproblemen auf.

Beherrsche die Teilprobleme durch rekursives Lösen. Hinreichend kleine Teilprobleme werden direkt gelöst.

Divide-and-Conquer

Teile-und-Beherrsche Algorithmen (divide-and-conquer) teilen das Problem in mehrere Teilprobleme auf, die dem Ausgangsproblem ähneln, jedoch von kleinerer Größe sind.

Sie lösen die Teilprobleme rekursiv und kombinieren diese Lösungen dann, um die Lösung des eigentlichen Problems zu erstellen.

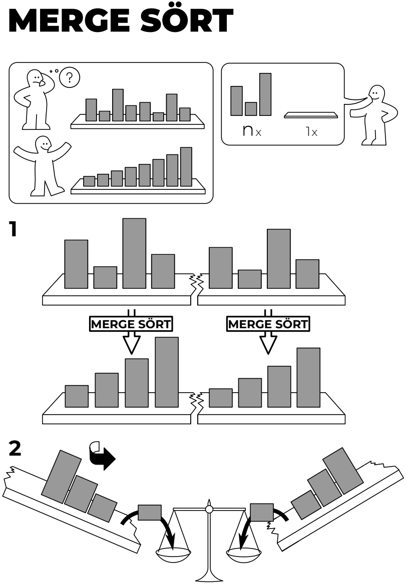
Das Paradigma von Teile-und-Beherrsche umfasst 3 Schritte auf jeder Rekursionsebene:

Teile das Problem in eine Anzahl von Teilproblemen auf.

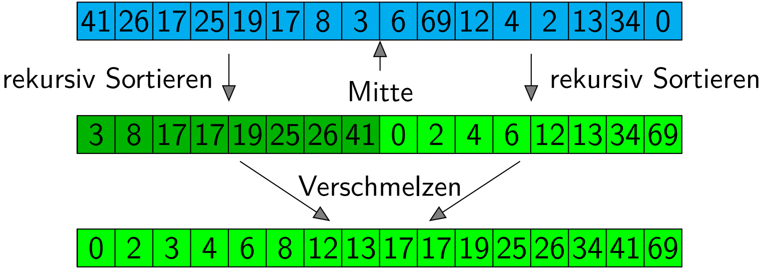
Beherrsche die Teilprobleme durch rekursives Lösen. Hinreichend kleine Teilprobleme werden direkt gelöst.

Verbinde die Lösungen der Teilprobleme zur Lösung des Ausgangsproblems.

Mergesort – IKEA-artige Erklärung



Mergesort – Strategie



Teile das Array in zwei – möglichst gleichgroße – Hälften.

Beherrsche: Sortiere die Teile durch rekursive Mergesort-Aufrufe.

Verbinde: Mische je 2 sortierte Teilsequenzen zu einem einzigen, sortierten Array.

Mergesort – Algorithm

```
void mergeSort(int E[], int left, int right) {
  if (left < right) {
    int mid = (left + right) / 2; // finde Mitte
    mergeSort(E, left, mid);       // sortiere linke Hälfte
    mergeSort(E, mid + 1, right); // sortiere rechte Hälfte
    // Verschmelzen der sortierten Hälften
    merge(E, left, mid, right);
  }
}
// Aufruf: mergeSort(E, 0, E.length-1);
```

- Verschmelzen kann man in Linearzeit. – Wie?

Mergesort – Animation

41	26	17	25	19	17	8	3	6	69	12	4	2	13	34	0

Mergesort – Animation

41	26	17	25	19	17	8	3	6	69	12	4	2	13	34	0

41	26	17	25	19	17	8	3

6	69	12	4	2	13	34	0

Mergesort – Animation

41	26	17	25	19	17	8	3	6	69	12	4	2	13	34	0

41	26	17	25	19	17	8	3

6	69	12	4	2	13	34	0

41	26	17	25

19	17	8	3

6	69	12	4

2	13	34	0

Mergesort – Animation

41	26	17	25	19	17	8	3	6	69	12	4	2	13	34	0

41	26	17	25	19	17	8	3

6	69	12	4	2	13	34	0

41	26	17	25

19	17	8	3

6	69	12	4

2	13	34	0

41	26

17	25

19	17

8	3

6	69

12	4

2	13

34	0

Mergesort – Animation

41	26	17	25	19	17	8	3	6	69	12	4	2	13	34	0

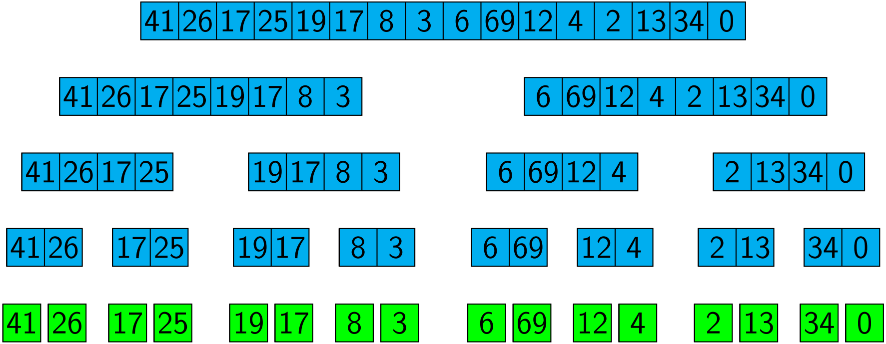
41 26 17 25 19 17 8 3   6 69 12 4 2 13 34 0

41 26 17 25   19 17 8 3   6 69 12 4   2 13 34 0

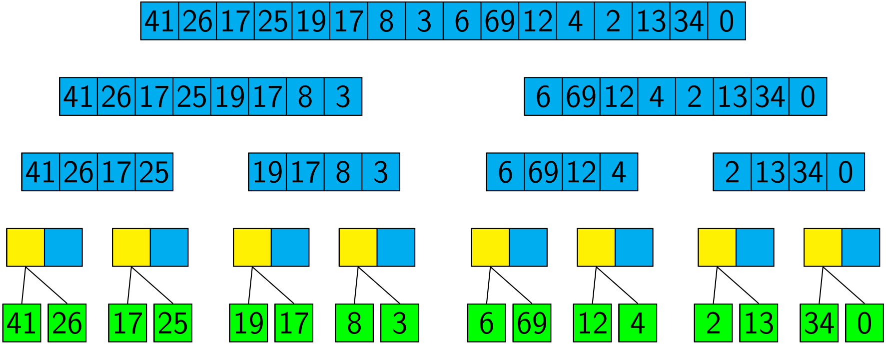
41 26   17 25   19 17   8 3   6 69   12 4   2 13   34 0

41   26   17   25   19   17   8   3   6   69   12   4   2   13   34   0

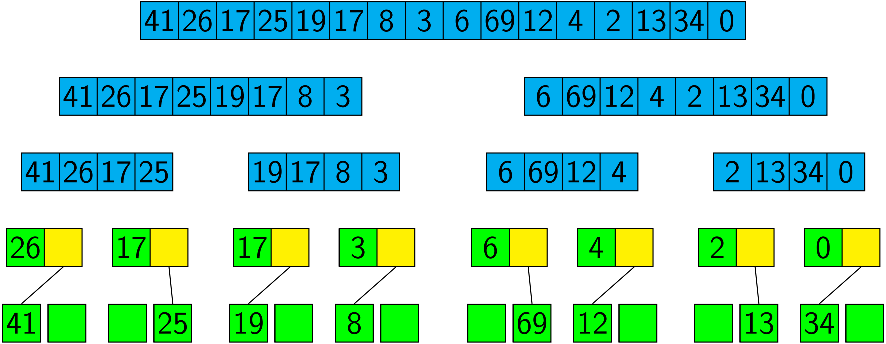
Mergesort – Animation



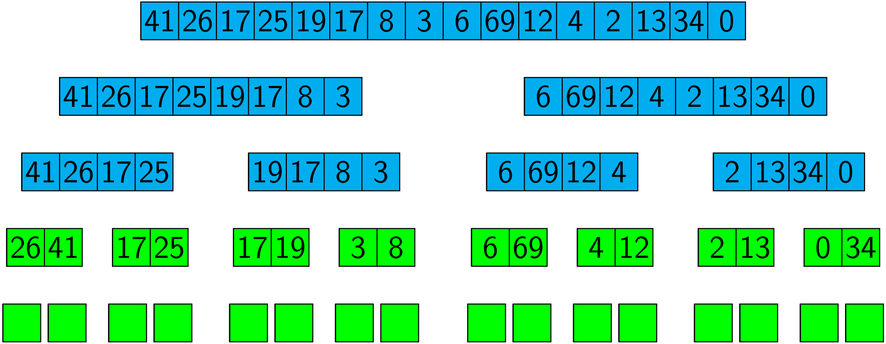
Mergesort – Animation



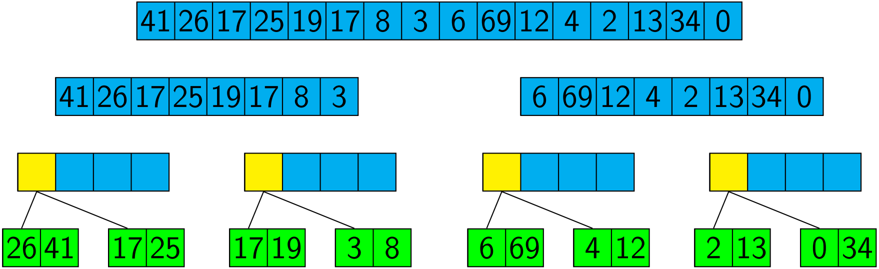
Mergesort – Animation



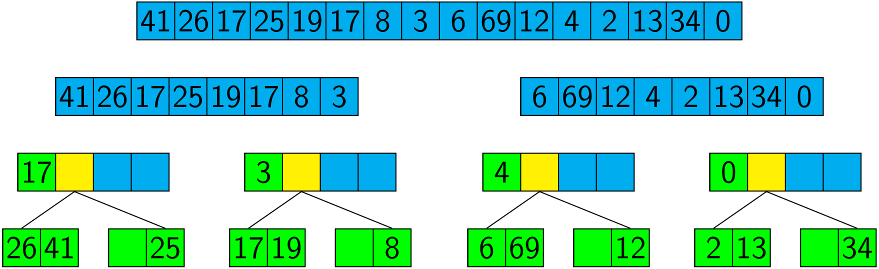
Mergesort – Animation



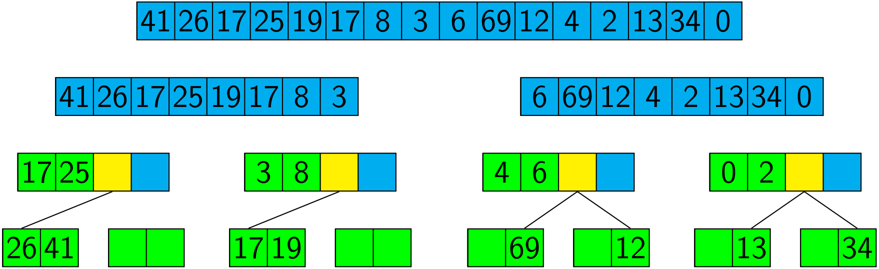
Mergesort – Animation



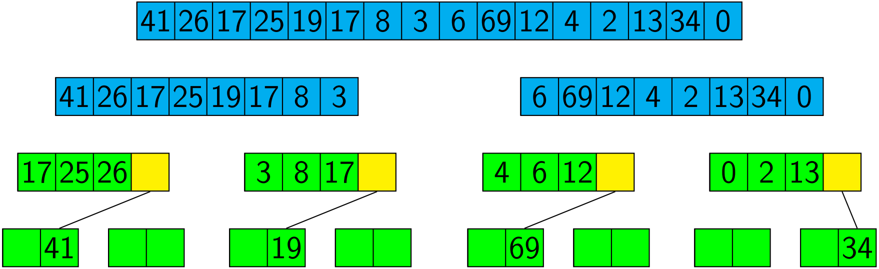
Mergesort – Animation



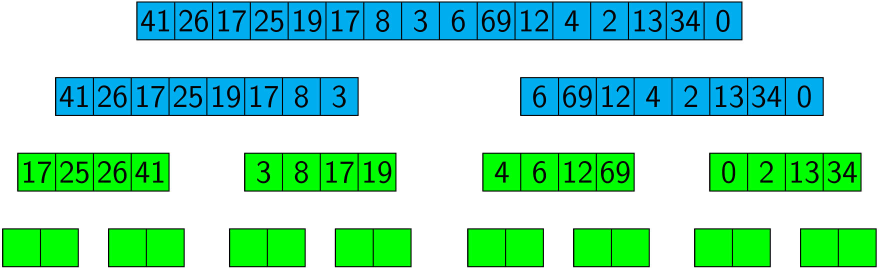
Mergesort – Animation



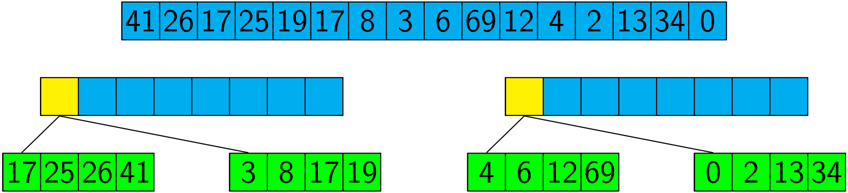
Mergesort – Animation



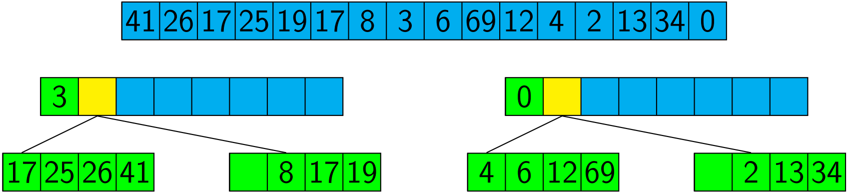
Mergesort – Animation



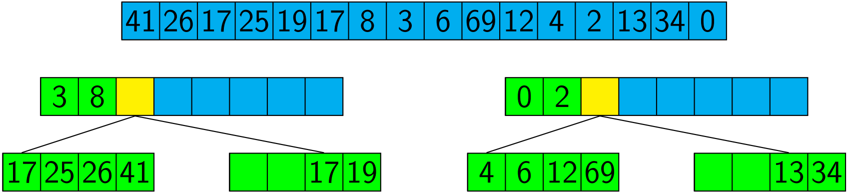
Mergesort – Animation



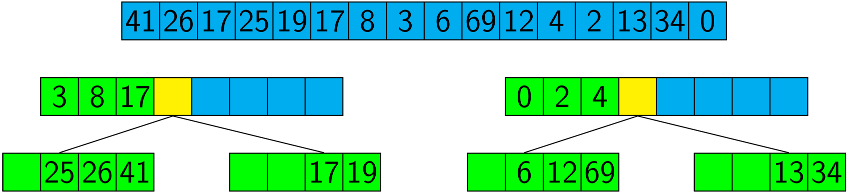
Mergesort – Animation



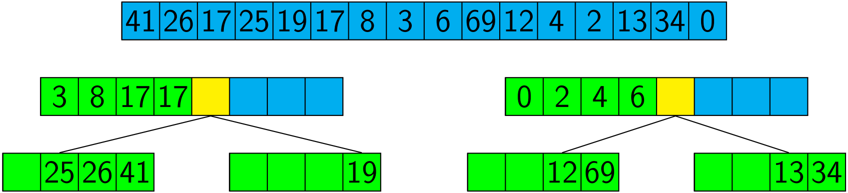
Mergesort – Animation



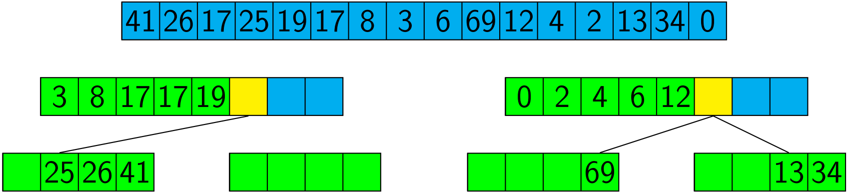
Mergesort – Animation



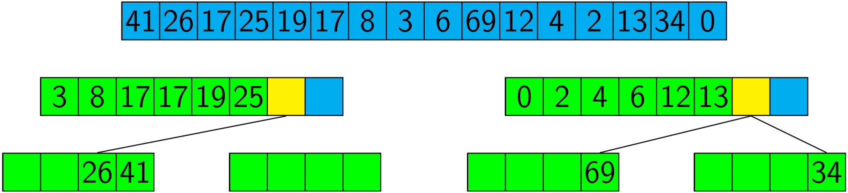
Mergesort – Animation



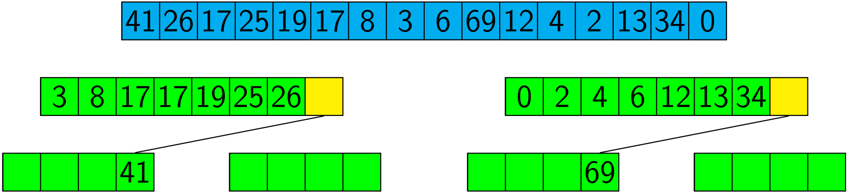
Mergesort – Animation



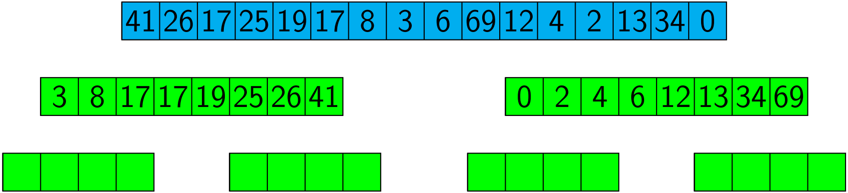
Mergesort – Animation



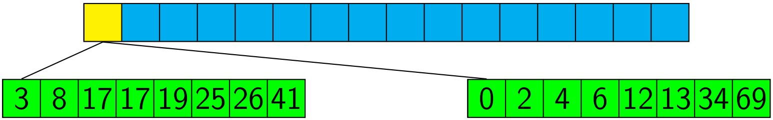
Mergesort – Animation



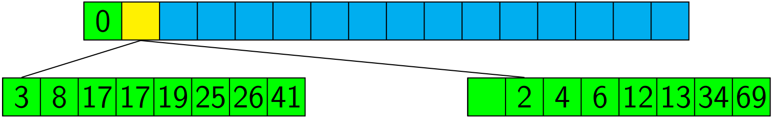
Mergesort – Animation



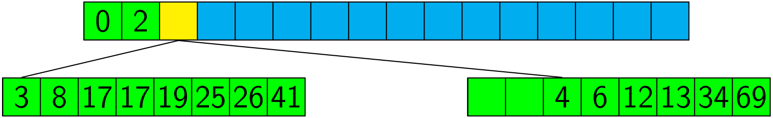
Mergesort – Animation



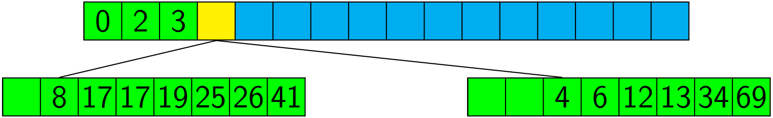
Mergesort – Animation



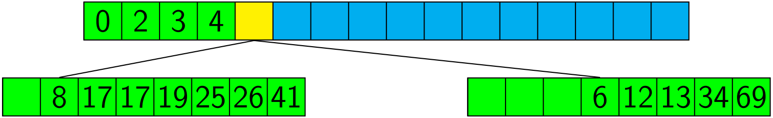
Mergesort – Animation



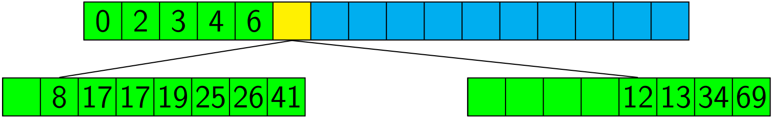
Mergesort – Animation



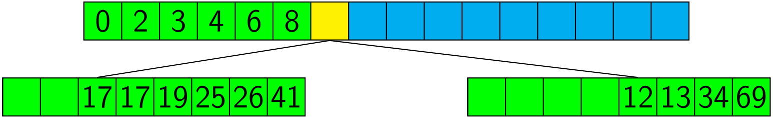
Mergesort – Animation



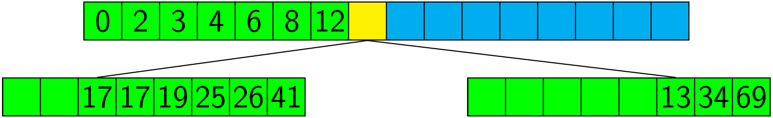
Mergesort – Animation



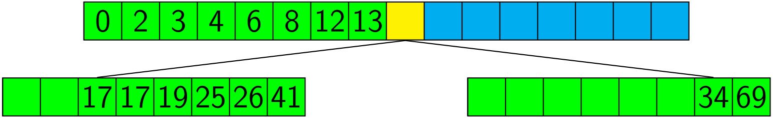
Mergesort – Animation



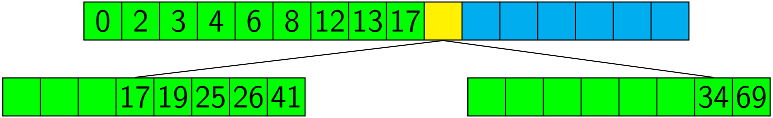
Mergesort – Animation



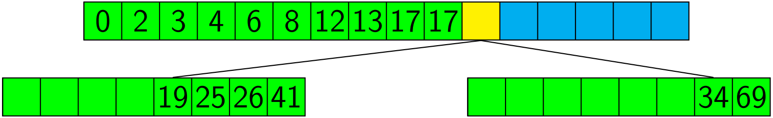
Mergesort – Animation



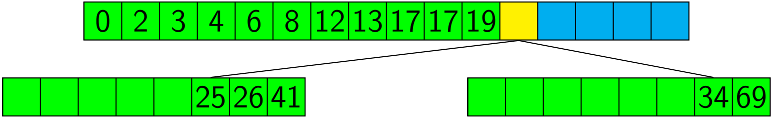
Mergesort – Animation



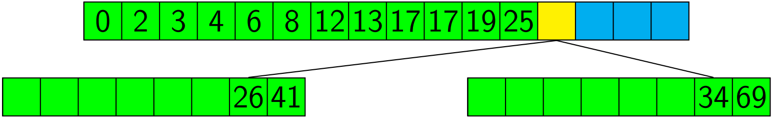
Mergesort – Animation



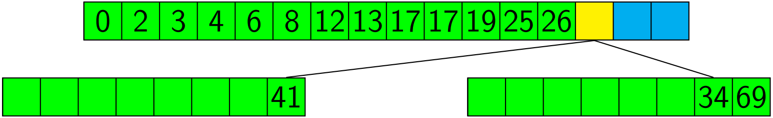
Mergesort – Animation



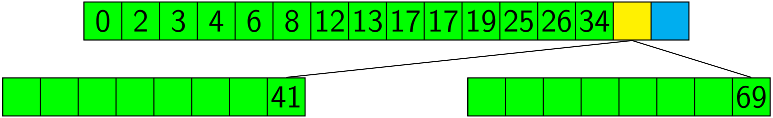
Mergesort – Animation



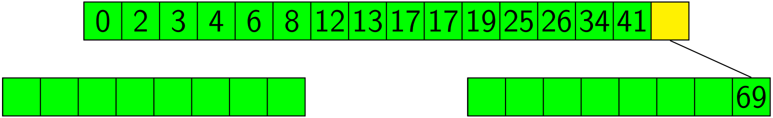
Mergesort – Animation



Mergesort – Animation



Mergesort – Animation



Mergesort – Animation

0	2	3	4	6	8	12	13	17	17	19	25	26	34	41	69

Mergesort – Verschmelzen in Linearzeit

```
void merge(int E[], int left, int mid, int right) {
  int a = left, b = mid + 1;
  int Eold[] = E;   // kopie von E
  for (; left <= right; left++) {
    if (a > mid) { // Wir wissen (Widerspruch): b <= right
      E[left] = Eold[b];
      b++;
    } else if (b > right || Eold[a] <= Eold[b]) { // stabil: <=
      E[left] = Eold[a];
      a++;
    } else { // Eold[a] > Eold[b]
      E[left] = Eold[b];
      b++;
    }
  }
}
```

Speicherkompl. $\Theta(n)$

Mergesort – Analyse

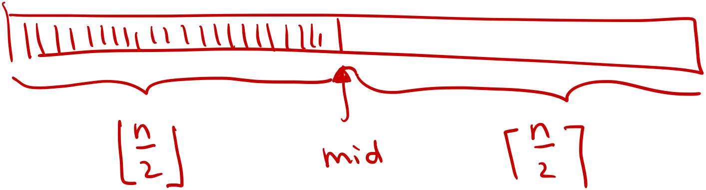
"Waage" = merge

Worst-Case

Wir erhalten:

$$W(n) = \underbrace{W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil)} + n - 1 \quad \text{mit} \quad W(1) = 0.$$

2 Teilprobleme $\longrightarrow W(n) = 2 \cdot W(\frac{n}{2})$



Mergesort – Analyse

Worst-Case

Wir erhalten:

$$W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n - 1 \quad \text{mit} \quad W(1) = 0.$$

Mit Hilfe des Mastertheorems ergibt sich: $W(n) \in \Theta(n \cdot \log n)$.

Mergesort – Analyse

Worst-Case

Wir erhalten:

$$W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n - 1 \quad \text{mit} \quad W(1) = 0.$$

Mit Hilfe des Mastertheorems ergibt sich: $W(n) \in \Theta(n \cdot \log n)$.

Best-Case, Average-Case

Das Worst-Case-Ergebnis gilt genauso im Best-Case, und damit auch im Average-Case: $W(n) = B(n) = A(n) \in \Theta(n \cdot \log n)$.

worst case = best case = average case Komplexität

Mergesort – Analyse

Worst-Case

Wir erhalten:

$$W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n - 1 \quad \text{mit} \quad W(1) = 0.$$

Mit Hilfe des Mastertheorems ergibt sich: $W(n) \in \Theta(n \cdot \log n)$.

Best-Case, Average-Case

Das Worst-Case-Ergebnis gilt genauso im Best-Case, und damit auch im Average-Case: $W(n) = B(n) = A(n) \in \Theta(n \cdot \log n)$.

Speicherbedarf

$\Theta(n)$ für die Kopie des Arrays beim Mergen. $\Theta(\log n)$ für den Stack.

Mergesort – Analyse

Worst-Case

Wir erhalten:

$$W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n - 1 \quad \text{mit} \quad W(1) = 0.$$

Mit Hilfe des Mastertheorems ergibt sich: $W(n) \in \Theta(n \cdot \log n)$.

Best-Case, Average-Case

Das Worst-Case-Ergebnis gilt genauso im Best-Case, und damit auch im Average-Case: $W(n) = B(n) = A(n) \in \Theta(n \cdot \log n)$.

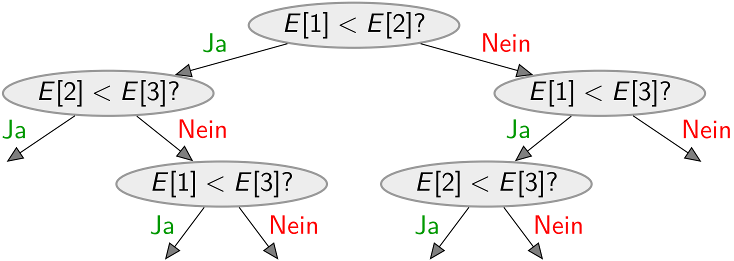
Speicherbedarf

$\Theta(n)$ für die Kopie des Arrays beim Mergen. $\Theta(\log n)$ für den Stack.

- Mergesort ist nicht in-place.
- Mit zusätzlichen Verschiebungen ist die Kopie des Arrays nicht nötig.

Übersicht

Wie effizient kann man sortieren? (I)



Betrachte vergleichsbasierte Sortieralgorithmen als Entscheidungsbaum:

- Dieser beschreibt die Abfolge der durchgeführten Vergleiche.
- Sortieren verschiedener Eingabepermutationen ergibt also verschiedene Pfade im Baum.

Wie effizient kann man sortieren? (II)

Theorem

Vergleichsbasiertes Sortieren benötigt im Worst-Case mindestens $O(n \log n)$ Vergleiche.

Beweis.

Es gilt:

Wie effizient kann man sortieren? (II)

Theorem

Vergleichsbasiertes Sortieren benötigt im Worst-Case mindestens $O(n \log n)$ Vergleiche.

Beweis.

Es gilt:

- Anzahl Vergleiche im Worst-Case = Länge des längsten Pfades = Baumhöhe k.

Wie effizient kann man sortieren? (II)

Theorem

Vergleichsbasiertes Sortieren benötigt im Worst-Case mindestens $O(n \log n)$ Vergleiche.

Beweis.

Es gilt:

- Anzahl Vergleiche im Worst-Case = Länge des längsten Pfades = Baumhöhe k.
- Da wir binäre Vergleiche verwenden, ergibt sich ein Binärbaum mit $n!$ Blättern.

Wie effizient kann man sortieren? (II)

Theorem

Vergleichsbasiertes Sortieren benötigt im Worst-Case mindestens $O(n \log n)$ Vergleiche.

Beweis.

Es gilt:

- Anzahl Vergleiche im Worst-Case = Länge des längsten Pfades = Baumhöhe k.
- Da wir binäre Vergleiche verwenden, ergibt sich ein Binärbaum mit $n!$ Blättern.

$\Rightarrow$ Mit $n! \leqslant 2^k$ erhält man $k \geqslant \lceil \log(n!) \rceil$ Vergleiche im Worst-Case.

Wie effizient kann man sortieren? (II)

Theorem

Vergleichsbasiertes Sortieren benötigt im Worst-Case mindestens $O(n \log n)$ Vergleiche.

Beweis.

Es gilt:

- Anzahl Vergleiche im Worst-Case = Länge des längsten Pfades = Baumhöhe k.
- Da wir binäre Vergleiche verwenden, ergibt sich ein Binärbaum mit $n!$ Blättern.

$\Rightarrow$ Mit $n! \leqslant 2^k$ erhält man $k \geqslant \lceil \log(n!) \rceil$ Vergleiche im Worst-Case.

Da $\lceil \log(n!) \rceil \approx n \cdot \log n - 1.4 \cdot n$ geht es nicht besser als $O(n \cdot \log n)$! □

Nächste Vorlesung

Nächste Vorlesung

Montag 14. Mai, 08:30 (Hörsaal H01). Bis dann!