Übersicht

Mengen

Der abstrakte Datentyp Menge sollte folgende Operationen unterstützen:

- $x \in M$?
- $M \to M \cup \{x\}$
- $M \to M \setminus \{x\}$
- $M = \emptyset$?
- wähle irgendein $x \in M$

Möglicherweise auch

- $M_1 \to M_2 \cup M_3$
- $M_1 \to M_2 \cap M_3$
- $M_1 \to M_2 \setminus M_3$
- ...

Mengen

Der abstrakte Datentyp Menge sollte folgende Operationen unterstützen:

- $x \in M$?
- $M \to M \cup \{x\}$
- $M \to M \setminus \{x\}$
- $M = \emptyset$?
- wähle irgendein $x \in M$

Möglicherweise auch

- $M_1 \to M_2 \cup M_3$
- $M_1 \to M_2 \cap M_3$
- $M_1 \to M_2 \setminus M_3$
- ...

Mengen können durch assoziative Arrays implementiert werden:

Java

```
public class Set⟨K⟩ {
  private final ADMap < K, ?> h;
  public Set() { h = new Hashtable⟨K, Integer⟩(); }
  public Set(ADMap < K, ?> m) { h = m; }
  public void insert(K k) { h.insert(k, null); }
  public void delete(K k) { h.delete(k); }
  public void union(Set⟨K⟩ U) {
    SimpleIterator⟨K⟩ it;
    for(it = U.iterator(); it.more(); it.step())
      insert(it.key()); }
  public boolean iselement(K k) { return h.containsKey(k); }
  public SimpleIterator⟨K⟩ iterator() {
  return h.simpleiterator(); }
  public List⟨K⟩ list() { return h.list(); }
```

Bitarrays

Wenn das Universum U klein ist, können wir Mengen durch Bitarrays implementieren.

Laufzeiten:

- Suchen, Einfügen, Löschen: $O(1)$
- Vereinigung, Schnitt: $O(|U|)$
- Auswahl: $O(|U|)$ (oder $O(1)$ mit Zusatzzeigern)

Bitarrays

Wenn das Universum U klein ist, können wir Mengen durch Bitarrays implementieren.

Laufzeiten:

- Suchen, Einfügen, Löschen: $O(1)$
- Vereinigung, Schnitt: $O(|U|)$
- Auswahl: $O(|U|)$ (oder $O(1)$ mit Zusatzzeigern)

Übersicht

Insertion Sort

Wir sortieren ein unsortiertes Array, indem wir wiederholt Elemente in ein bereits sortiertes Teilarray einfügen.

Eingabe:

12	73	36	71	79	67	3	17	32	31	14	14	33	4	74	23

Ausgabe:

3	4	12	14	14	17	23	31	32	33	36	67	71	73	74	79

Insertion Sort

12	73	36	71	79	67	3	17	32	31	14	14	33	4	74	23

Algorithmus

procedure insertionsort(n) :
  for i $= 2, ..., n$ **do**
    j := i;
    while j ≥ 2 **and** a[j $- 1$] $>$ a[j] **do**
      vertausche a[j $- 1$] und a[j];
      j := j $- 1$
    od
  od

Insertionsort

Inversionen

Die Laufzeit von Insertion Sort ist $O(n^2)$.

Definition

Sei $\pi \in S_n$ eine Permutation. Die Menge der Inversionen von π ist

$$I(\pi) = \{ (i,j) \in \{1,\ldots,n\}^2 \mid i < j \text{ und } \pi(i) > \pi(j) \}.$$

Beim Sortieren durch Einfügen werden Schlüssel nur durch Vertauschungen benachbarter Elemente bewegt.
So eine Vertauschung verringert die Anzahl der Inversionen höchstens um eins.
Wenn die Eingabe genau falschherum sortiert ist, hat Insertion Sort die Laufzeit $\Theta(n^2)$.
Was ist die durchschnittliche Laufzeit?

Inversionen

Die Laufzeit von Insertion Sort ist $O(n^2)$.

Definition

Sei $\pi \in S_n$ eine Permutation. Die Menge der Inversionen von π ist

$$I(\pi) = \{ (i,j) \in \{1,\ldots,n\}^2 \mid i < j \text{ und } \pi(i) > \pi(j) \}.$$

Beim Sortieren durch Einfügen werden Schlüssel nur durch Vertauschungen benachbarter Elemente bewegt.
So eine Vertauschung verringert die Anzahl der Inversionen höchstens um eins.
Wenn die Eingabe genau falschherum sortiert ist, hat Insertion Sort die Laufzeit $\Theta(n^2)$.
Was ist die durchschnittliche Laufzeit?

Inversionen

Die Laufzeit von Insertion Sort ist $O(n^2)$.

Definition

Sei $\pi \in S_n$ eine Permutation. Die Menge der Inversionen von π ist

$$I(\pi) = \{ (i,j) \in \{1,\ldots,n\}^2 \mid i < j \text{ und } \pi(i) > \pi(j) \}.$$

Beim Sortieren durch Einfügen werden Schlüssel nur durch Vertauschungen benachbarter Elemente bewegt.
So eine Vertauschung verringert die Anzahl der Inversionen höchstens um eins.
Wenn die Eingabe genau falschherum sortiert ist, hat Insertion Sort die Laufzeit $\Theta(n^2)$.
Was ist die durchschnittliche Laufzeit?

Inversionen

Die Laufzeit von Insertion Sort ist $O(n^2)$.

Definition

Sei $\pi \in S_n$ eine Permutation. Die Menge der Inversionen von π ist

$$I(\pi) = \{ (i,j) \in \{1,\ldots,n\}^2 \mid i < j \text{ und } \pi(i) > \pi(j) \}.$$

Beim Sortieren durch Einfügen werden Schlüssel nur durch Vertauschungen benachbarter Elemente bewegt.
So eine Vertauschung verringert die Anzahl der Inversionen höchstens um eins.
Wenn die Eingabe genau falschherum sortiert ist, hat Insertion Sort die Laufzeit $\Theta(n^2)$.
Was ist die durchschnittliche Laufzeit?

Inversionen

Theorem

Eine zufällig gewählte Permutation $\pi \in S_n$ hat im Erwartungswert $n(n-1)/4$ Inversionen.

Beweis.

Es gibt $n(n-1)/2$ viele Paare (i,j) mit $1 \leq i < j \leq n$.
Wegen

$$\Pr[\pi(i) > \pi(j)] = \frac{1}{2} \text{ falls } i \neq j$$

gilt

$$E(|I(\pi)|) = \sum_{i<j} \Pr[\pi(i) > \pi(j)] = \frac{n(n-1)}{4}.$$

□

Inversionen

Theorem

Eine zufällig gewählte Permutation $\pi \in S_n$ hat im Erwartungswert $n(n-1)/4$ Inversionen.

Beweis.

Es gibt $n(n-1)/2$ viele Paare (i,j) mit $1 \leq i < j \leq n$.
Wegen

$$\Pr[\pi(i) > \pi(j)] = \frac{1}{2} \text{ falls } i \neq j$$

gilt

$$E(|I(\pi)|) = \sum_{i<j} \Pr[\pi(i) > \pi(j)] = \frac{n(n-1)}{4}.$$

□

Inversionen

Theorem

Jedes Sortierverfahren, das Schlüssel nur durch Vertauschungen benachbarter Elemente bewegt, benötigt im Durchschnitt $\Omega(n^2)$ Zeit.

Folgerung:

Wenn wir schneller sein wollen, müssen Schlüssel über weite Strecken bewegt werden.

Inversionen

Theorem

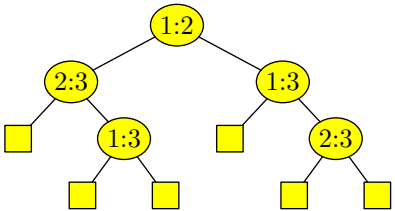
Jedes Sortierverfahren, das Schlüssel nur durch Vertauschungen benachbarter Elemente bewegt, benötigt im Durchschnitt $\Omega(n^2)$ Zeit.

Folgerung:

Wenn wir schneller sein wollen, müssen Schlüssel über weite Strecken bewegt werden.

Vergleichsbäume

Insertion Sort mit $n = 3$:



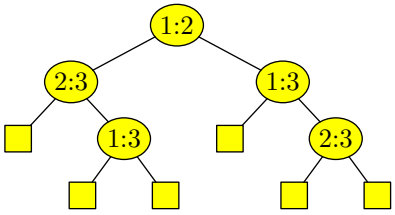
Jeder vergleichsbasierte Sortieralgorithmus hat einen Vergleichsbaum.

Die Wurzel ist der erste Vergleich.

Links folgen die Vergleiche beim Ergebnis kleiner.
Rechts folgen die Vergleiche beim Ergebnis größer.

Vergleichsbäume

Insertion Sort mit $n = 3$:



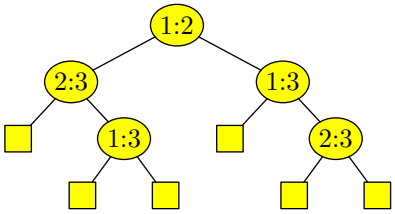
Jeder vergleichsbasierte Sortieralgorithmus hat einen Vergleichsbaum.

Die Wurzel ist der erste Vergleich.

Links folgen die Vergleiche beim Ergebnis kleiner.
Rechts folgen die Vergleiche beim Ergebnis größer.

Vergleichsbäume

Insertion Sort mit $n = 3$:



Jeder vergleichsbasierte Sortieralgorithmus hat einen Vergleichsbaum.

Die Wurzel ist der erste Vergleich.

Links folgen die Vergleiche beim Ergebnis kleiner.
Rechts folgen die Vergleiche beim Ergebnis größer.

Vergleichsbäume

Lemma

Der Vergleichsbaum eines vergleichsbasierten Sortieralgorithmus hat mindestens $n!$ Blätter.

Beweis.

Wenn zwei Permutationen zum gleichen Blatt führen, wird eine von ihnen falsch sortiert.

Es gibt aber $n!$ viele Permutationen. □

Vergleichsbäume

Lemma

Der Vergleichsbaum eines vergleichsbasierten Sortieralgorithmus hat mindestens $n!$ Blätter.

Beweis.

Wenn zwei Permutationen zum gleichen Blatt führen, wird eine von ihnen falsch sortiert.

Es gibt aber $n!$ viele Permutationen. □

Theorem

Jeder vergleichsbasierte Algorithmus benötigt für das Sortieren einer zufällig permutierten Eingabe im Erwartungswert mindestens

$$\log(n!) = n \log n - n \log e - \frac{1}{2} \log n + O(1)$$

viele Vergleiche.

Beweis.

Sei T ein entsprechender Vergleichsbaum. Die mittlere Pfadlänge zu einem Blatt ist am kleinsten, wenn der Baum balanziert ist.
In diesem Fall ist die Höhe $\log(n!)$. Stirling-Formel:

$$n! = \frac{1}{\sqrt{2\pi n}} \left(\frac{n}{e}\right)^n (1 + O(n^{-1})).$$

Theorem

Jeder vergleichsbasierte Algorithmus benötigt für das Sortieren einer zufällig permutierten Eingabe im Erwartungswert mindestens

$$\log(n!) = n \log n - n \log e - \frac{1}{2} \log n + O(1)$$

viele Vergleiche.

Beweis.

Sei T ein entsprechender Vergleichsbaum. Die mittlere Pfadlänge zu einem Blatt ist am kleinsten, wenn der Baum balanziert ist.
In diesem Fall ist die Höhe $\log(n!)$. Stirling-Formel:

$$n! = \frac{1}{\sqrt{2\pi n}} \left(\frac{n}{e}\right)^n \left(1 + O(n^{-1})\right).$$

□

Mergesort

Wir sortieren ein unsortiertes Array durch einen Divide-and-Conquer–Algorithmus:

Eingabe:

12	73	36	71	79	67	3	17	32	31	14	14	33	4	74	23

Ausgabe:

3	4	12	14	14	17	23	31	32	33	36	67	71	73	74	79

Frage: Wie zerteilen wir die Eingabe in zwei unabhängige Teilprobleme?

Mergesort

Wir sortieren ein unsortiertes Array durch einen Divide-and-Conquer–Algorithmus:

Eingabe:

12	73	36	71	79	67	3	17	32	31	14	14	33	4	74	23

Ausgabe:

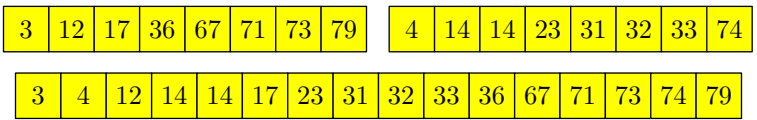
3	4	12	14	14	17	23	31	32	33	36	67	71	73	74	79

Frage: Wie zerteilen wir die Eingabe in zwei unabhängige Teilprobleme?

Mergesort

Eine einfache Strategie: Teilen in der Mitte.

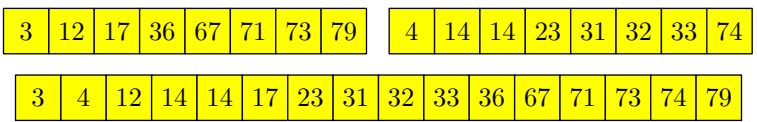
1. Teile das Array in der Mitte.
2. Sortiere beide Hälften.
3. Mische beide in eine sortierte Folge.



Mergesort

Eine einfache Strategie: Teilen in der Mitte.

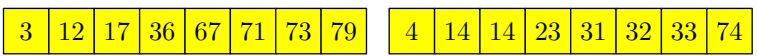
1. Teile das Array in der Mitte.
2. Sortiere beide Hälften.
3. Mische beide in eine sortierte Folge.

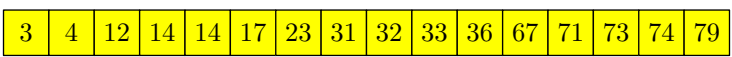


Mergesort

Eine einfache Strategie: Teilen in der Mitte.

1. Teile das Array in der Mitte.
2. Sortiere beide Hälften.
3. Mische beide in eine sortierte Folge.

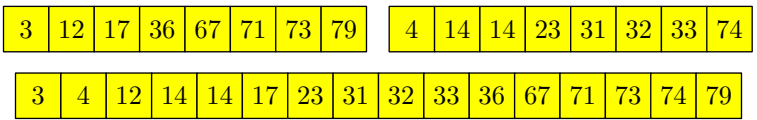




Mergesort

Eine einfache Strategie: Teilen in der Mitte.

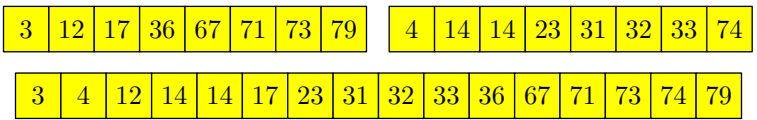
1. Teile das Array in der Mitte.
2. Sortiere beide Hälften.
3. Mische beide in eine sortierte Folge.



Mergesort

Eine einfache Strategie: Teilen in der Mitte.

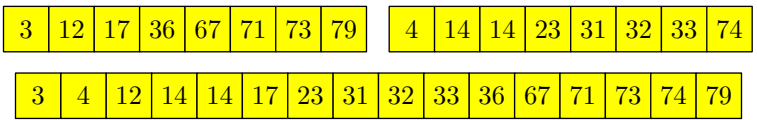
1. Teile das Array in der Mitte.
2. Sortiere beide Hälften.
3. Mische beide in eine sortierte Folge.



Mergesort

Eine einfache Strategie: Teilen in der Mitte.

1. Teile das Array in der Mitte.
2. Sortiere beide Hälften.
3. Mische beide in eine sortierte Folge.

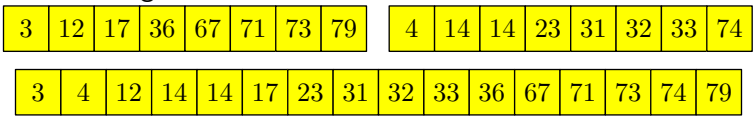


Mergesort

Bottom-up Mergesort

Mergesort

Mischen ist der schwierige Teil.



Algorithmus, der $a[l], \ldots, a[m-1]$ mit $a[m], \ldots, a[r]$ mischt:

Algorithmus

```
i := l; j := m; k := l;
while k ≤ r do
  if a[i] ≤ a[j] and i < m or j > r
  then b[k] := a[i]; k := k + 1; i := i + 1
  else b[k] := a[j]; k := k + 1; j := j + 1 fi
od;
for i = l, ..., r do a[k] := b[k] od
```

Mergesort

Algorithmus

```
procedure mergesort(l, r) :
  if l ≥ r then return fi;
  m := ⌈(r + l)/2 ⌉;
  mergesort(l, m − 1);
  mergesort(m, r);
  i := l; j := m; k := l;
  while k ≤ r do
    if a[i] ≤ a[j] and i < m or j > r
    then b[k] := a[i]; k := k + 1; i := i + 1
    else b[k] := a[j]; k := k + 1; j := j + 1 fi
  od;
  for k = l, ..., r do a[k] := b[k] od
```

Analyse von Mergesort

Das Mischen dauert $\Theta(n)$.

Sei $T(n)$ die Laufzeit. Wir erhalten die Gleichung

$$T(n) = \Theta(n) + T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil).$$

Falls n eine Zweierpotenz ist, gilt

$$T(n) \leq cn + 2T(n/2).$$

Wiederholtes Einsetzen liefert

$$T(n) \leq c\left(n + 2\frac{n}{2} + 4\frac{n}{4} + \cdots\right) = O(n \log n).$$

Mergesort

Mergesort hat interessante Eigenschaften:

1. Der Teile-Teil ist sehr einfach.
2. Der Conquer-Teil ist kompliziert.
3. Er verbraucht viel Speicherplatz (nicht „in-place“)
4. Er ist stabil (gleiche Schlüssel behalten ihre Reihenfolge)
5. ...

Quicksort

Wir sortieren ein unsortiertes Array durch einen Divide-and-Conquer–Algorithmus:

Eingabe:

67	32	17	36	3	4	79	14	31	23	71	74	73	33	14	12

Ausgabe:

3	4	12	14	14	17	23	31	32	33	36	67	71	73	74	79

Frage: Wie zerteilen wir die Eingabe in zwei unabhängige Teilprobleme auf eine andere Art?

Quicksort

Wir sortieren ein unsortiertes Array durch einen Divide-and-Conquer–Algorithmus:

Eingabe:

67	32	17	36	3	4	79	14	31	23	71	74	73	33	14	12

Ausgabe:

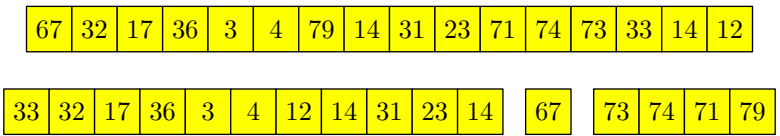
3	4	12	14	14	17	23	31	32	33	36	67	71	73	74	79

Frage: Wie zerteilen wir die Eingabe in zwei unabhängige Teilprobleme auf eine andere Art?

Quicksort

Anstatt in der Mitte zu teilen, wählen wir ein Pivot-Element p und teilen in drei Teile:

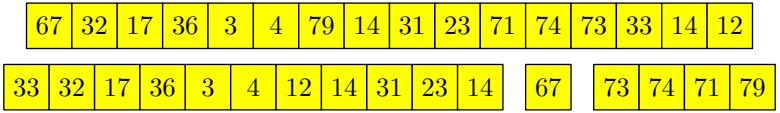
1. Alle Schlüssel kleiner als p
2. p selbst
3. Alle Schlüssel größer als p



Sortiere dann rekursiv den ersten und dritten Teil.

Quicksort

Quicksort



Algorithmus

procedure quicksort(L, R) :
  if $R \leq L$ **then return fi**;
  $p := a[L]$; $l := L$; $r := R + 1$;
  do
    do $l := l + 1$ **while** $a[l] < p$;
    do $r := r - 1$ **while** $p < a[r]$;
    vertausche a[l] und a[r];
  while $l < r$;
  $temp := a[r]$; $a[L] := a[l]$; $a[l] := temp$; $a[r] := p$;
  quicksort(L, r − 1); quicksort(r + 1, R)

Analyse von Quicksort

Wir nehmen an, die Eingabe besteht aus n paarweise verschiedenen Zahlen und daß jede Permutation gleich wahrscheinlich ist.

Was ist der Erwartungswert der Laufzeit?

Wir werden nur die Anzahl der Vergleiche analysieren.

Sei C_n die erwartete Anzahl von Vergleichen für eine Eingabe der Länge n.

Analyse von Quicksort

1. Offensichtlich ist $C_0 = C_1 = 0$.
2. Die Anzahl der direkten Vergleiche ist $n + 1$.
3. Falls k die endgültige Position des Pivot-Elements ist, dann gibt es noch C_{k-1} und C_{n-k} Vergleiche in den beiden rekursiven Aufrufen.
4. Falls $n \geq 2$, dann

$$C_n = n + 1 + \frac{1}{n} \sum_{k=1}^{n} (C_{k-1} + C_{n-k}).$$

Um einen geschlossenen Ausdruck für C_n zu erhalten, lösen wir diese Rekursionsgleichung.

Analyse von Quicksort

Sei $n \geq 2$. Sei $D_n = nC_n$. Dann

$$D_{n+1} - D_n = \Big((n+1)(n+2) + \sum_{k=1}^{n+1}(C_{k-1} + C_{n+1-k})\Big) - \Big(n(n+1) + \sum_{k=1}^{n}(C_{k-1} + C_{n-k})\Big)$$
$$= 2(n+1) + C_n + C_n = 2(n+1) + 2D_n/n$$

und wir erhalten die Rekursionsgleichung

$$D_{n+1} = 2(n+1) + \frac{n+2}{n} D_n.$$

Dividieren durch $(n+1)(n+2)$ ergibt

$$\frac{D_{n+1}}{(n+1)(n+2)} = \frac{2}{n+2} + \frac{D_n}{n(n+1)} \text{ für } n \geq 2.$$

Analyse von Quicksort

$$\frac{D_{n+1}}{(n+1)(n+2)} = \frac{2}{n+2} + \frac{D_n}{n(n+1)} \text{ für } n \geq 2.$$

Wiederholtes Einsetzen ergibt für $n \geq 3$

$$\frac{D_n}{n(n+1)} = \frac{2}{n+1} + \frac{2}{n} + \cdots + \frac{2}{5} + \frac{D_2}{6}$$

oder

$$\begin{aligned} C_n =& (n+1)\Big(\frac{2}{n+1} + \frac{2}{n} + \cdots + \frac{2}{5}\Big) + (n+1)\frac{C_2}{3} \\ =& 2nH_n + O(n) = 2n\ln(n) + O(n). \end{aligned}$$

Die durchschnittliche Laufzeit von Quicksort ist $O(n \log n)$.

Java

```
public void quicksort() {
  Stack⟨Pair⟨Integer, Integer⟩⟩ stack = new Stack⟨Pair⟨Integer, Integer⟩⟩();
  stack.push(new Pair⟨Integer, Integer⟩(1, size() − 1));
  int min = 0;
  for(int i = 1; i < size(); i++) if(less(i, min)) min = i;
  D t = get(0); set(0, get(min)); set(min, t);
  while(!stack.isempty()) {
    Pair⟨Integer, Integer⟩ p = stack.pop();
    int l = p.first(), r = p.second();
    int i = l − 1, j = r, pivot = j;
      do { i++; } while(less(i, pivot));
      do { j−−; } while(less(pivot, j));
      t = get(i); set(i, get(j)); set(j, t);
    } while(i < j);
    set(j, get(i)); set(i, get(r)); set(r, t);
    if(r − i > 1) stack.push(new Pair⟨Integer, Integer⟩(i + 1, r));
    if(i − l > 1) stack.push(new Pair⟨Integer, Integer⟩(l, i − 1));
  }
}
```

Java

```
public void quicksort(int a[]) {
  Stack⟨Pair⟨Integer, Integer⟩⟩ stack = new Stack⟨Pair⟨Integer, Integer⟩⟩();
  stack.push(new Pair⟨Integer, Integer⟩(1, a.length − 1));
  int min = 0;
  for(int i = 1; i < a.length; i++) if(a[i] < a[min]) min = i;
  int t = a[0]; a[0] = a[min]; a[min] = t;
  while(!stack.isempty()) {
    Pair⟨Integer, Integer⟩ p = stack.pop();
    int l = p.first(), r = p.second();
    int i = l − 1, j = r, pivot = j;
      do { i++; } while(a[i] < a[pivot]);
      do { j−−; } while(a[j] > a[pivot]);
      t = a[i]; a[i] = a[j]; a[j] = t;
    } while(i < j);
    a[j] = a[i]; a[i] = a[r]; a[r] = t;
    if(r − i > 1) stack.push(new Pair⟨Integer, Integer⟩(i + 1, r));
    if(i − l > 1) stack.push(new Pair⟨Integer, Integer⟩(l, i − 1));
  }
}
```

Quicksort

Quicksort hat evenfalls interessante Eigenschaften:

1. Der Teile-Teil ist schwierig.
2. Der Herrsche-Teil ist sehr einfach.
3. Die durchschnittliche Laufzeit ist sehr gut.
4. Die worst-case Laufzeit ist sehr schlecht.
5. Die innere Schleife ist sehr schnell.