Sortieralgorithmen im Vergleich

Algorithm Engineering, Prof. Weicker

Matthias Jauernig

Fachbereich Informatik, Mathematik und Naturwissenschaften

05.11.2007

Rahmenbedingungen

- Zeitnahme:
  - Java-Bean `ThreadMxBean`:

```
import java.lang.management.*;
ThreadMXBean tbean = ManagementFactory.getThreadMXBean();
long time = tbean.getCurrentThreadCpuTime();
```

→ Liefert vom Thread verbrauchte CPU-Zeit
→ Trotzdem noch abhängig von anderen Prozessen

  - Durchschnittswerte über 50 Iterationen
  - Keine Benutzerinteraktionen mit dem Rechner
- *Zähler für Vergleiche*: über `compare()`-Methode für Elemente
- *Zähler für Operationen*: jede Zuweisung von Elementen, z.B. 1 Tausch = 3 Operationen

Rahmenbedingungen

- Zeitnahme:
  - Java-Bean `ThreadMxBean`:

```
import java.lang.management.*;
ThreadMXBean tbean = ManagementFactory.getThreadMXBean();
long time = tbean.getCurrentThreadCpuTime();
```

→ Liefert vom Thread verbrauchte CPU-Zeit
→ Trotzdem noch abhängig von anderen Prozessen

  - Durchschnittswerte über 50 Iterationen
  - Keine Benutzerinteraktionen mit dem Rechner
- *Zähler für Vergleiche*: über `compare()`-Methode für Elemente
- *Zähler für Operationen*: jede Zuweisung von Elementen, z.B. 1 Tausch = 3 Operationen

Rahmenbedingungen

- Zeitnahme:
  - Java-Bean `ThreadMxBean`:

```
import java.lang.management.*;
ThreadMXBean tbean = ManagementFactory.getThreadMXBean();
long time = tbean.getCurrentThreadCpuTime();
```

→ Liefert vom Thread verbrauchte CPU-Zeit
→ Trotzdem noch abhängig von anderen Prozessen

  - Durchschnittswerte über 50 Iterationen
  - Keine Benutzerinteraktionen mit dem Rechner
- *Zähler für Vergleiche*: über `compare()`-Methode für Elemente
- *Zähler für Operationen*: jede Zuweisung von Elementen, z.B. 1 Tausch = 3 Operationen

Rahmenbedingungen

- Zeitnahme:
  - Java-Bean `ThreadMxBean`:

```
import java.lang.management.*;
ThreadMXBean tbean = ManagementFactory.getThreadMXBean();
long time = tbean.getCurrentThreadCpuTime();
```

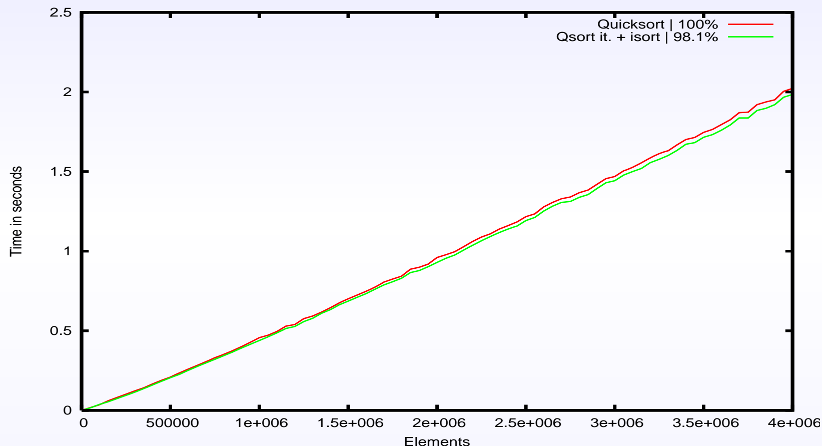
→ Liefert vom Thread verbrauchte CPU-Zeit

→ Trotzdem noch abhängig von anderen Prozessen

  - Durchschnittswerte über 50 Iterationen
  - Keine Benutzerinteraktionen mit dem Rechner
- *Zähler für Vergleiche*: über `compare()`-Methode für Elemente
- *Zähler für Operationen*: jede Zuweisung von Elementen, z.B. 1 Tausch = 3 Operationen

Quicksort rekursiv vs. (iterativ + Insertion Sort)

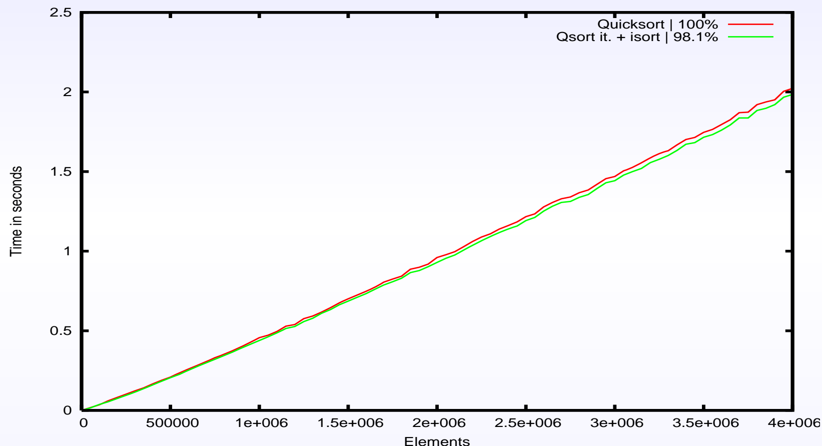
- Benötigte Zeit:



- Kaum Verbesserung!
- Und direkter Vergleich iterativ vs. rekursiv?

Quicksort rekursiv vs. (iterativ + Insertion Sort)

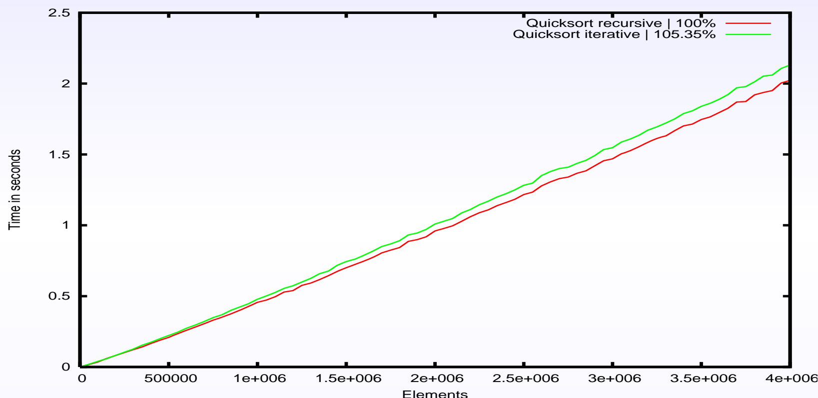
- Benötigte Zeit:



- Kaum Verbesserung!
- Und direkter Vergleich iterativ vs. rekursiv?

Quicksort rekursiv vs. iterativ

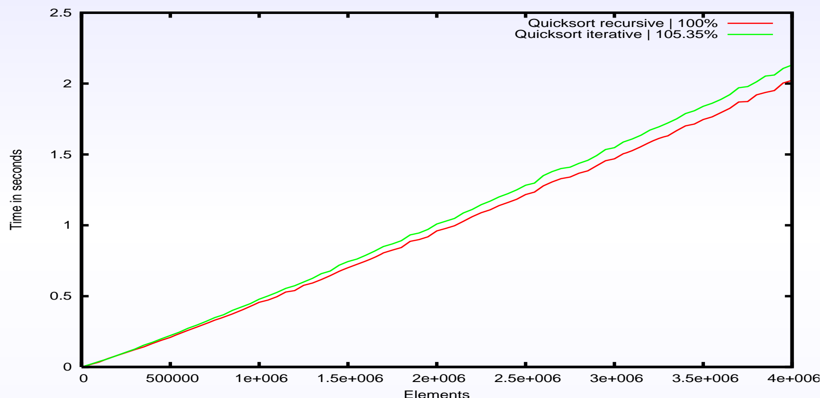
- Benötigte Zeit:



- Iterative Variante langsamer!
- Anscheinend ist zusätzlicher Vergleich `p<(from+to)/2` und Neusetzen der Grenzen zeitaufwendiger als rekursiver Aufruf
- Compiler-bedingt? Keine „Auffälligkeiten" im Bytecode

Quicksort rekursiv vs. iterativ

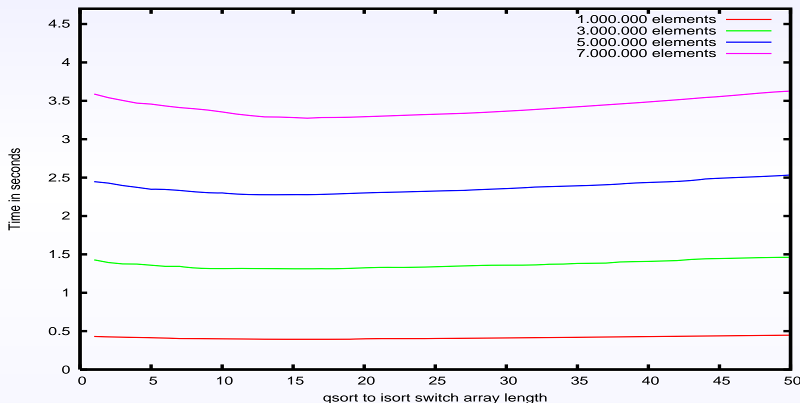
- Benötigte Zeit:



- Iterative Variante langsamer!
- Anscheinend ist zusätzlicher Vergleich `p<(from+to)/2` und Neusetzen der Grenzen zeitaufwendiger als rekursiver Aufruf
- Compiler-bedingt? Keine „Auffälligkeiten" im Bytecode

Quicksort + Insertion Sort

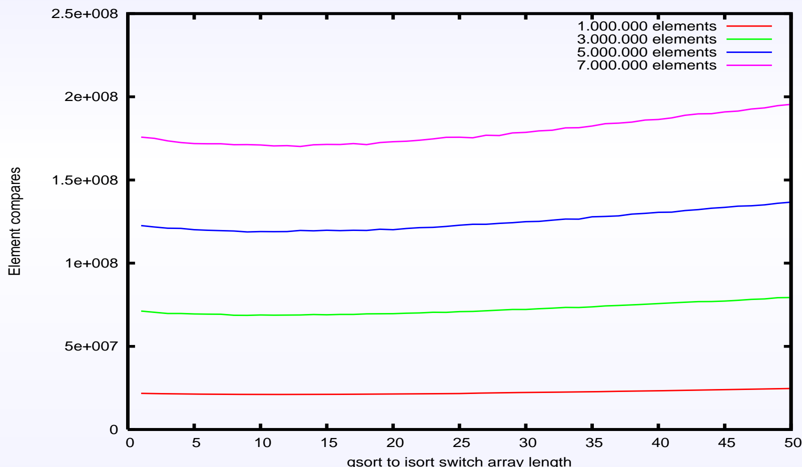
- Finde optimales m
- Benötigte Zeit:



- „Optimales“ m unabhängig von Elementanzahl im Bereich 14–17, z. B. $m{=}15$
- für 7.000.000 Elemente: m=0 … 100%, m=15 … 91.46%

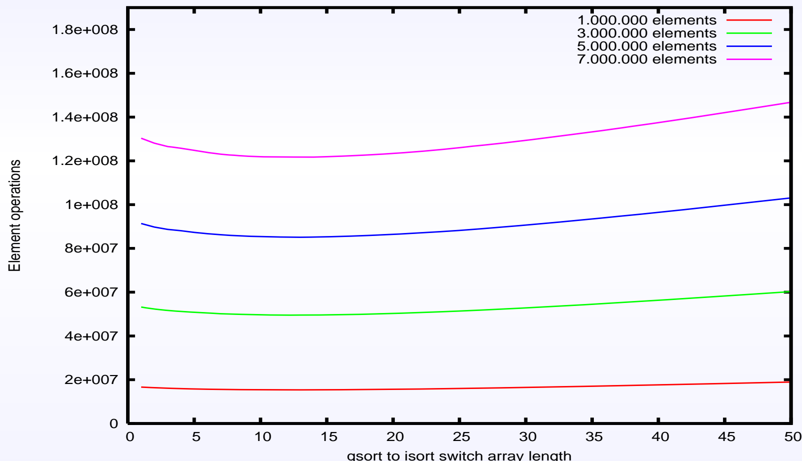
Quicksort + Insertion Sort

- Finde optimales m
- Anzahl an Vergleichen:



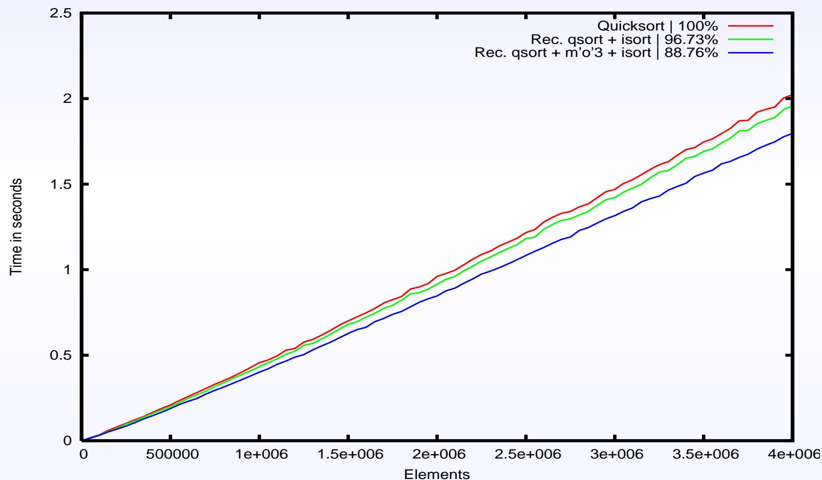
Quicksort + Insertion Sort

- Finde optimales m
- Anzahl an Operationen:



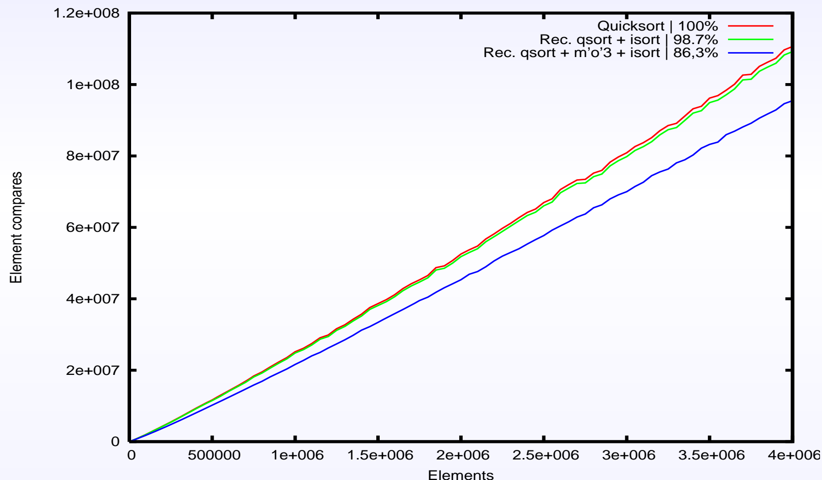
Quicksort Tuning

- Benötigte Zeit:



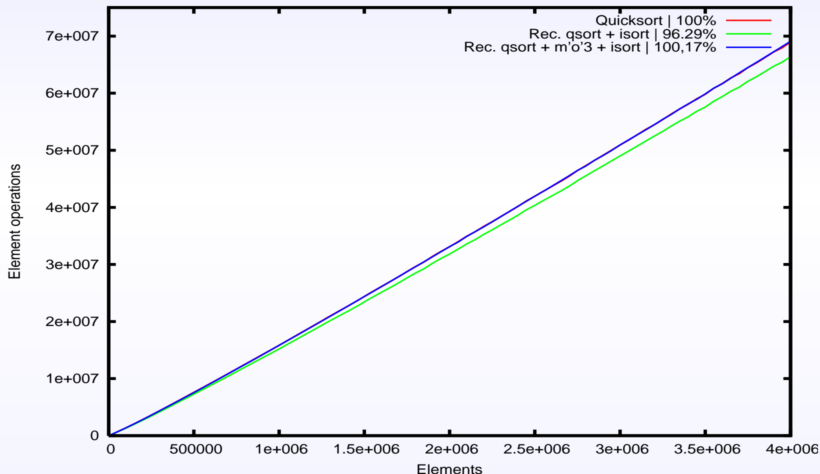
Quicksort Tuning

- Anzahl an Vergleichen:



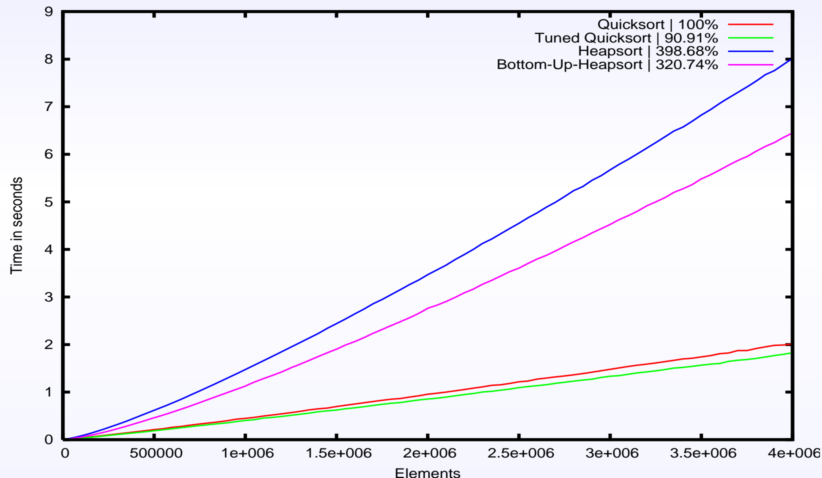
Quicksort Tuning

- Anzahl an Operationen:



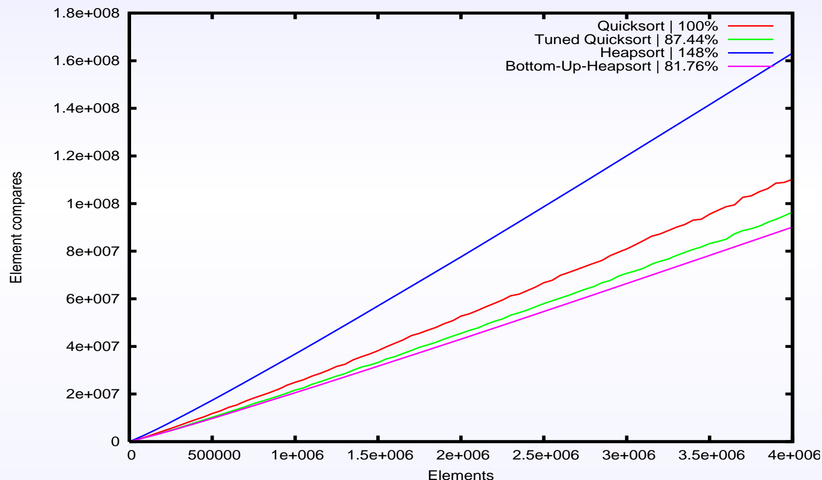
Quicksort vs. Heapsort

- Benötigte Zeit:



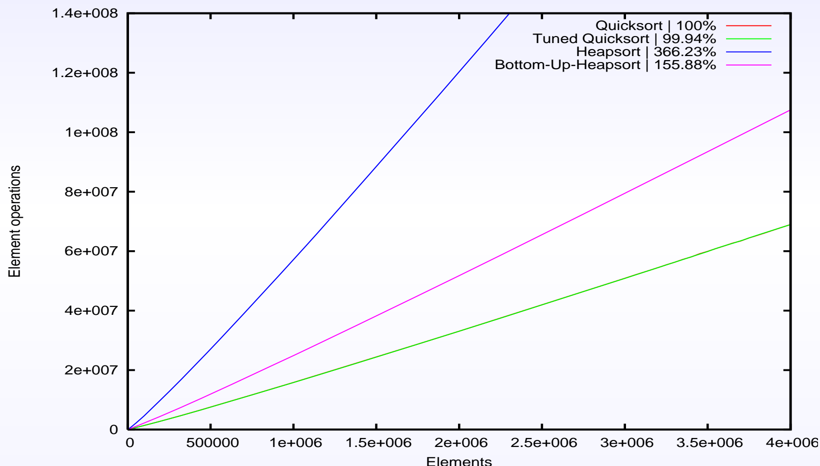
Quicksort vs. Heapsort

- Anzahl an Vergleichen:



Quicksort vs. Heapsort

- Anzahl an Operationen:



- Quicksort ungefähr gleich mit Tuned Quicksort

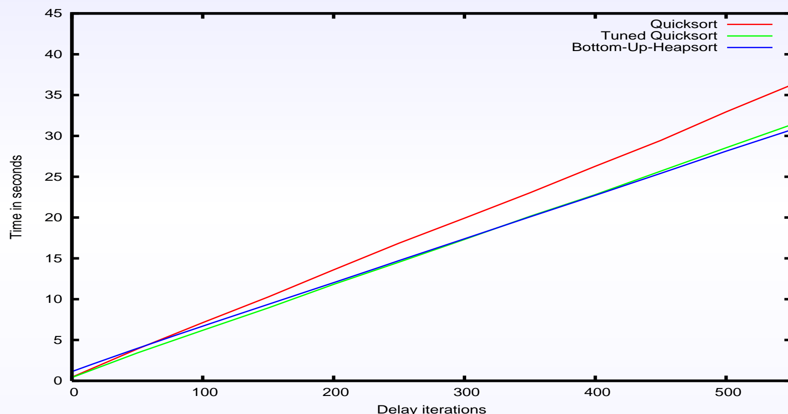
Künstliche Erhöhung der Vergleichszeit

Einfache Zählschleife in Vergleichsmethode compare():

```
private int compare(int a, int b){
  if(this.cmpIts > 0){
    int i = 0;
    while(i < this.cmpIts)
      i++;
  }
  ...
}
```

Künstliche Erhöhung der Vergleichszeit

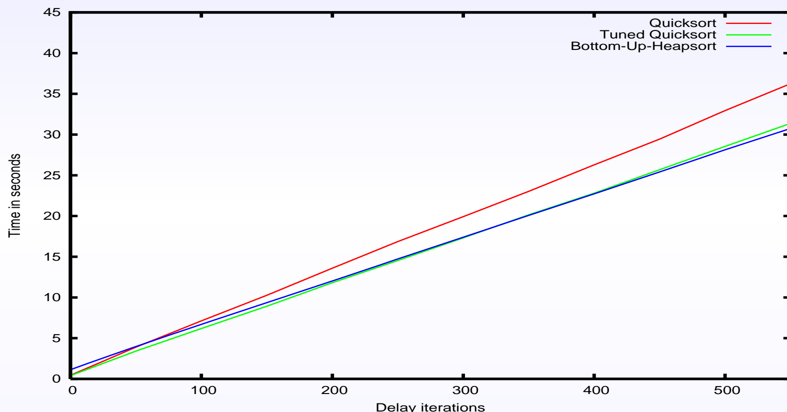
- Benötigte Zeit für 1.000.000 Elemente:



- B-Up-Heapsort schneller als Quicksort: ab ca. 60 Iterationen
- B-Up-Heapsort schneller als Tuned Quicksort: ab ca. 305 Iterationen

Künstliche Erhöhung der Vergleichszeit

- Benötigte Zeit für 1.000.000 Elemente:



- B-Up-Heapsort schneller als Quicksort: ab ca. 60 Iterationen
- B-Up-Heapsort schneller als Tuned Quicksort: ab ca. 305 Iterationen