Sortieralgorithmen

Jan Pöschko

18. Januar 2007

Inhalt

1. Problemstellung
   - Definition
   - Warum Sortieren?
2. Einfache Sortieralgorithmen
   - Bubblesort
   - Insertionsort
   - Schranke für einfache Algorithmen
3. Optimale Sortieralgorithmen
   - Mergesort
   - Quicksort
   - Schranke für vergleichbasierte Algorithmen
4. Sortieren in linearer Zeit
   - Counting sort

Inhalt

1. Problemstellung
   - Definition
   - Warum Sortieren?
2. Einfache Sortieralgorithmen
   - Bubblesort
   - Insertionsort
   - Schranke für einfache Algorithmen
3. Optimale Sortieralgorithmen
   - Mergesort
   - Quicksort
   - Schranke für vergleichbasierte Algorithmen
4. Sortieren in linearer Zeit
   - Counting sort

Inhalt

1. Problemstellung
   - Definition
   - Warum Sortieren?
2. Einfache Sortieralgorithmen
   - Bubblesort
   - Insertionsort
   - Schranke für einfache Algorithmen
3. Optimale Sortieralgorithmen
   - Mergesort
   - Quicksort
   - Schranke für vergleichbasierte Algorithmen
4. Sortieren in linearer Zeit
   - Counting sort

Inhalt

1. Problemstellung
   - Definition
   - Warum Sortieren?
2. Einfache Sortieralgorithmen
   - Bubblesort
   - Insertionsort
   - Schranke für einfache Algorithmen
3. Optimale Sortieralgorithmen
   - Mergesort
   - Quicksort
   - Schranke für vergleichbasierte Algorithmen
4. Sortieren in linearer Zeit
   - Counting sort

Definition

Problemdefinition

Input: Folge (Feld) von n Zahlen $\langle a_1, a_2, \ldots, a_n \rangle$

Output: Permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ der Folge, sodass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

In der Praxis:

key: sortiert

satellite data: mitverschoben

Für Analyse: implementationstechnische Details unwichtig

Definition

Problemdefinition

Input: Folge (Feld) von n Zahlen $\langle a_1, a_2, \ldots, a_n \rangle$

Output: Permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ der Folge, sodass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

In der Praxis:

key: sortiert

satellite data: mitverschoben

Für Analyse: implementationstechnische Details unwichtig

Definition

Problemdefinition

Input: Folge (Feld) von n Zahlen $\langle a_1, a_2, \ldots, a_n \rangle$

Output: Permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ der Folge, sodass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

In der Praxis:

key: sortiert

satellite data: mitverschoben

Für Analyse: implementationstechnische Details unwichtig

Warum Sortieren?

Warum beschäftigt man sich mit Sortieralgorithmen?

- zur Informationsaufbereitung
- als Teil eines anderen Algorithmus
- verwendete Techniken generell interessant
- historisches Problem
- untere Laufzeitschranke beweisbar

Warum Sortieren?

Warum beschäftigt man sich mit Sortieralgorithmen?

- zur Informationsaufbereitung
- als Teil eines anderen Algorithmus
- verwendete Techniken generell interessant
- historisches Problem
- untere Laufzeitschranke beweisbar

Warum Sortieren?

Warum beschäftigt man sich mit Sortieralgorithmen?

- zur Informationsaufbereitung
- als Teil eines anderen Algorithmus
- verwendete Techniken generell interessant
- historisches Problem
- untere Laufzeitschranke beweisbar

Warum Sortieren?

Warum beschäftigt man sich mit Sortieralgorithmen?

- zur Informationsaufbereitung
- als Teil eines anderen Algorithmus
- verwendete Techniken generell interessant
- historisches Problem
- untere Laufzeitschranke beweisbar

Warum Sortieren?

Warum beschäftigt man sich mit Sortieralgorithmen?

- zur Informationsaufbereitung
- als Teil eines anderen Algorithmus
- verwendete Techniken generell interessant
- historisches Problem
- untere Laufzeitschranke beweisbar

Bubblesort

Prinzip

- Zwei benachbarte Elemente vergleichen und ggf. vertauschen
- Vorgang solange wiederholen, bis keine Vertauschungen mehr nötig

Algorithmus

procedure BUBBLESORT(A)
  for $i \leftarrow 1$ to $n-1$ **do**
    for $j \leftarrow 1$ to $i-1$ **do**
      if $A[j] > A[j+1]$ **then**
        exchange $A[j] \leftrightarrow A[j+1]$

Bubblesort

Prinzip

- Zwei benachbarte Elemente vergleichen und ggf. vertauschen
- Vorgang solange wiederholen, bis keine Vertauschungen mehr nötig

Algorithmus

procedure BUBBLESORT(A)
  for $i \leftarrow 1$ to $n-1$ **do**
    for $j \leftarrow 1$ to $i-1$ **do**
      if $A[j] > A[j+1]$ **then**
        exchange $A[j] \leftrightarrow A[j+1]$

Bubblesort

Analyse:

- Laufzeit: $O(n^2)$
- *in place*: kein zusätzlicher Speicherbedarf
- *stabil*: gleiche Elemente behalten Reihenfolge

Varianten:

Bubblesort

Analyse:

- Laufzeit: $O(n^2)$
- *in place*: kein zusätzlicher Speicherbedarf
- *stabil*: gleiche Elemente behalten Reihenfolge

Varianten:

- Mit Abbruchbedingung in innerer Schleife: für sortierte Felder sehr schnell

Bubblesort

Analyse:

- Laufzeit: $O(n^2)$
- *in place*: kein zusätzlicher Speicherbedarf
- *stabil*: gleiche Elemente behalten Reihenfolge

Varianten:

- Mit Abbruchbedingung in innerer Schleife: für sortierte Felder sehr schnell
- Shakersort oder Cocktailsort: Richtung des Schleifendurchlaufs jedesmal geändert (gleicher Zeitbedarf)

Bubblesort

Analyse:

- Laufzeit: $O(n^2)$
- *in place*: kein zusätzlicher Speicherbedarf
- *stabil*: gleiche Elemente behalten Reihenfolge

Varianten:

- Mit Abbruchbedingung in innerer Schleife: für sortierte Felder sehr schnell
- Shakersort oder Cocktailsort: Richtung des Schleifendurchlaufs jedesmal geändert (gleicher Zeitbedarf)

Bubblesort

Analyse:

- Laufzeit: $O(n^2)$
- *in place*: kein zusätzlicher Speicherbedarf
- *stabil*: gleiche Elemente behalten Reihenfolge

Varianten:

- Mit Abbruchbedingung in innerer Schleife: für sortierte Felder sehr schnell
- Shakersort oder Cocktailsort: Richtung des Schleifendurchlaufs jedesmal geändert (gleicher Zeitbedarf)

Bubblesort

Analyse:

- Laufzeit: $O(n^2)$
- *in place*: kein zusätzlicher Speicherbedarf
- *stabil*: gleiche Elemente behalten Reihenfolge

Varianten:

- Mit Abbruchbedingung in innerer Schleife: für sortierte Felder sehr schnell
- Shakersort oder Cocktailsort: Richtung des Schleifendurchlaufs jedesmal geändert (gleicher Zeitbedarf)

Insertionsort

Prinzip

- Nimm schrittweise Element aus Feld und füge es an richtiger Stelle im bereits sortierten Teil ein
- Vergleiche es dazu mit allen bisher sortierten Elementen

Laufzeit: $O(n^2)$

Verwandte Sortieralgorithmen:

- Shellsort: beginnt mit größerer Schrittweite
- Combsort: „Durchkämmen" des Feldes

Insertionsort

Prinzip

- Nimm schrittweise Element aus Feld und füge es an richtiger Stelle im bereits sortierten Teil ein
- Vergleiche es dazu mit allen bisher sortierten Elementen

Laufzeit: $O(n^2)$

Verwandte Sortieralgorithmen:

- Shellsort: beginnt mit größerer Schrittweite
- Combsort: „Durchkämmen" des Feldes

Insertionsort

Prinzip

- Nimm schrittweise Element aus Feld und füge es an richtiger Stelle im bereits sortierten Teil ein
- Vergleiche es dazu mit allen bisher sortierten Elementen

Laufzeit: $O(n^2)$

Verwandte Sortieralgorithmen:

- SHELLSORT: beginnt mit größerer Schrittweite
- COMBSORT: „Durchkämmen" des Feldes

Insertionsort

Prinzip

- Nimm schrittweise Element aus Feld und füge es an richtiger Stelle im bereits sortierten Teil ein
- Vergleiche es dazu mit allen bisher sortierten Elementen

Laufzeit: $O(n^2)$

Verwandte Sortieralgorithmen:

- SHELLSORT: beginnt mit größerer Schrittweite
- COMBSORT: „Durchkämmen“ des Feldes

Schranke für einfache Algorithmen

Satz

Jeder Sortieralgorithmus, der nur benachbarte Elemente vertauscht, benötigt $\Omega(n^2)$ Vergleiche im worst case.

Beweisidee:

- Betrachte verkehrt sortierte Folge
- Erstes Element benötigt $n-1$ Vertauschungen, zweites $n-2$ usw.

Schranke für einfache Algorithmen

Satz

Jeder Sortieralgorithmus, der nur benachbarte Elemente vertauscht, benötigt $\Omega(n^2)$ Vergleiche im worst case.

Beweisidee:

- Betrachte verkehrt sortierte Folge
- Erstes Element benötigt $n-1$ Vertauschungen, zweites $n-2$ usw.

Mergesort

Prinzip

Divide: Liste in kleinere Listen zerlegen

Conquer: Listen für sich sortieren

Combine: Zusammenführen (*merge*)

Algorithmus

procedure $\textsc{MergeSort}(A, p, r)$
  if $p < r$ **then**
    $q \leftarrow \lfloor (p+r)/2 \rfloor$
    $\textsc{MergeSort}(A, p, q)$
    $\textsc{MergeSort}(A, q+1, r)$
    $\textsc{Merge}$(A,p,q,r)

Mergesort

Prinzip

Divide: Liste in kleinere Listen zerlegen

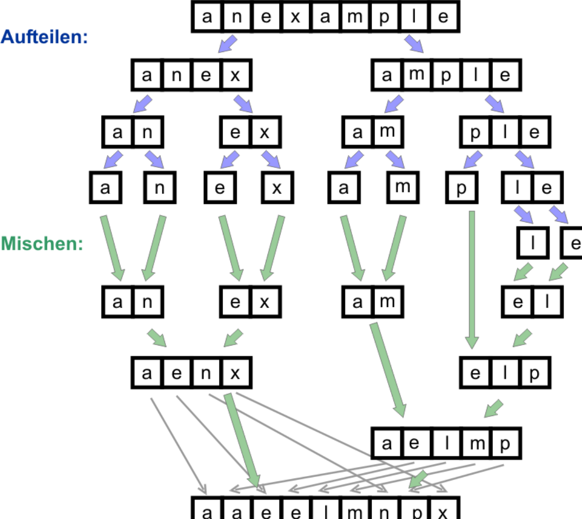
Conquer: Listen für sich sortieren

Combine: Zusammenführen (*merge*)

Algorithmus

procedure MERGESORT(A, p, r)
  if $p < r$ **then**
    $q \leftarrow \lfloor (p + r)/2 \rfloor$
    MERGESORT(A, p, q)
    MERGESORT($A, q + 1, r$)
    MERGE(A,p,q,r)

Mergesort



Mergesort

Analyse:

- Laufzeit: $O(n \log n)$
- Zusätzlicher Speicherbedarf: bei günstiger Datenstruktur *in place*, bei Arrays $O(n)$
- Optimal für *externes Sortieren*: auf Daten kann nicht beliebig zugegriffen werden, sondern nur mittels „Band“ (in eine Richtung lesbar)

Mergesort

Analyse:

- Laufzeit: $O(n \log n)$
- Zusätzlicher Speicherbedarf: bei günstiger Datenstruktur *in place*, bei Arrays $O(n)$
- Optimal für *externes Sortieren*: auf Daten kann nicht beliebig zugegriffen werden, sondern nur mittels „Band“ (in eine Richtung lesbar)

Mergesort

Analyse:

- Laufzeit: $O(n \log n)$
- Zusätzlicher Speicherbedarf: bei günstiger Datenstruktur *in place*, bei Arrays $O(n)$
- Optimal für *externes Sortieren*: auf Daten kann nicht beliebig zugegriffen werden, sondern nur mittels „Band“ (in eine Richtung lesbar)

Quicksort

Prinzip

Divide: Zerlege Feld $A[p \dots r]$ in zwei Teilfelder $A[p \dots q-1]$ und $A[q+1 \dots r]$, sodass erstere $\leq A[q]$ und letztere $\geq A[q]$

Conquer: Sortiere die zwei Teilfelder durch rekursiven Aufruf von QUICKSORT

Algorithmus

```
procedure QUICKSORT(A, p, r)
    if p < r then
        q ← PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
```

Quicksort

Prinzip

Divide: Zerlege Feld $A[p \dots r]$ in zwei Teilfelder $A[p \dots q-1]$ und $A[q+1 \dots r]$, sodass erstere $\leq A[q]$ und letztere $\geq A[q]$

Conquer: Sortiere die zwei Teilfelder durch rekursiven Aufruf von Quicksort

Algorithmus

procedure Quicksort(A, p, r)
- **if** $p < r$ **then**
  - $q \leftarrow$ Partition(A, p, r)
  - Quicksort$(A, p, q-1)$
  - Quicksort$(A, q+1, r)$

Quicksort

Quicksort

Wahl des *Pivot-Elements* $A[q]$:

- erstes, letztes oder mittleres Element
- Median dieser drei Elemente
- Median der gesamten Liste
- zufälliges Element

Laufzeitanalyse:

Quicksort

Wahl des *Pivot-Elements* $A[q]$:

- erstes, letztes oder mittleres Element
- Median dieser drei Elemente
- Median der gesamten Liste
- zufälliges Element

Laufzeitanalyse:

Quicksort

Wahl des *Pivot-Elements* $A[q]$:

- erstes, letztes oder mittleres Element
- Median dieser drei Elemente
- Median der gesamten Liste
- zufälliges Element

Laufzeitanalyse:

- Worst case: $O(n^2)$

Quicksort

Wahl des *Pivot-Elements* $A[q]$:

- erstes, letztes oder mittleres Element
- Median dieser drei Elemente
- Median der gesamten Liste
- zufälliges Element

Laufzeitanalyse:

- *Worst case:* $O(n^2)$
- *Average case:* $O(n \log n)$

Quicksort

Wahl des *Pivot-Elements* $A[q]$:

- erstes, letztes oder mittleres Element
- Median dieser drei Elemente
- Median der gesamten Liste
- zufälliges Element

Laufzeitanalyse:

- *Worst case*: $O(n^2)$
- *Average case*: $O(n \log n)$

Quicksort

Wahl des *Pivot-Elements* $A[q]$:

- erstes, letztes oder mittleres Element
- Median dieser drei Elemente
- Median der gesamten Liste
- zufälliges Element

Laufzeitanalyse:

- *Worst case*: $O(n^2)$
- *Average case*: $O(n \log n)$

Quicksort

Wahl des *Pivot-Elements* $A[q]$:

- erstes, letztes oder mittleres Element
- Median dieser drei Elemente
- Median der gesamten Liste
- zufälliges Element

Laufzeitanalyse:

- *Worst case*: $O(n^2)$
- *Average case*: $O(n \log n)$

Schranke für vergleichbasierte Algorithmen

Satz

Jeder auf Vergleichen basierende Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche im worst case.

Beweisidee:

- Vergleiche erzeugen *Entscheidungsbaum*
- Jede der $n!$ Permutationen der Folge entspricht einem Blatt
- Höhe des Baumes: $h \geq \log(n!) = \Omega(n \log n)$

Schranke für vergleichbasierte Algorithmen

Satz

Jeder auf Vergleichen basierende Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche im worst case.

Beweisidee:

- Vergleiche erzeugen *Entscheidungsbaum*
- Jede der $n!$ Permutationen der Folge entspricht einem Blatt
- Höhe des Baumes: $h \geq \log(n!) = \Omega(n \log n)$

Counting sort

Voraussetzung: Werte sind ganze Zahlen $\leq k$

Prinzip

- Bestimme für jedes Element x Anzahl der Elemente $\leq x$
- Füge mit dieser Information jedes Element an seiner richtigen Position ein

Analyse:

- Laufzeit: $O(n+k)$
- Speicherbedarf: $O(n+k)$

Counting sort

Voraussetzung: Werte sind ganze Zahlen $\leq k$

Prinzip

- Bestimme für jedes Element x Anzahl der Elemente $\leq x$
- Füge mit dieser Information jedes Element an seiner richtigen Position ein

Analyse:

- Laufzeit: $O(n + k)$
- Speicherbedarf: $O(n + k)$

Counting sort

Voraussetzung: Werte sind ganze Zahlen $\leq k$

Prinzip

- Bestimme für jedes Element x Anzahl der Elemente $\leq x$
- Füge mit dieser Information jedes Element an seiner richtigen Position ein

Analyse:

- Laufzeit: $O(n+k)$
- Speicherbedarf: $O(n+k)$

Literatur

- T. H. Cormen, C. E. Leisserson, R. L. Rivest, C. Stein: Introduction to Algorithms
- Donald E. Knuth: The Art of Computer Programming, Volume 3. Searching and Sorting
- Wikipedia: Sortieralgorithmus