

Datenstrukturen und Algorithmen

Vorlesung 7: Sortieren (K2)

Joost-Pieter Katoen

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

<http://www-i2.informatik.rwth-aachen.de/i2/dsa112/>

27. April 2012



Übersicht

- 1 Sortieren - Einführung
 - Bedeutung des Sortierens
 - Dutch National Flag Problem
- 2 Sortieren durch Einfügen
- 3 Mergesort
 - Das Divide-and-Conquer Paradigma
 - Mergesort
- 4 Effizienteres Sortieren?

Übersicht

- 1 Sortieren - Einführung
 - Bedeutung des Sortierens
 - Dutch National Flag Problem
- 2 Sortieren durch Einfügen
- 3 Mergesort
 - Das Divide-and-Conquer Paradigma
 - Mergesort
- 4 Effizienteres Sortieren?

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 $O(\log n)$.

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 $O(n)$.

Einige Hilfsbegriffe

Permutation

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

Totale Ordnung

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

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

Beispiel

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

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).

Sortierproblem

Das Sortier-Problem

- Eingabe:**
1. Ein Array E mit n Einträgen.
 2. Die Einträge gehören zu einer Menge A mit totaler Ordnung \leq .

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

Annahmen dieser Vorlesung

- ▶ Die zu sortierende Sequenz ist als **Array** organisiert, nicht als Liste.
- ▶ Die Elementaroperation ist ein Vergleich von Schlüsseln.

Sortieralgorithmen

Beispiel (Einige Sortieralgorithmen)

Insertionsort, Bubblesort, Shellsort, Mergesort, Heapsort
Quicksort, Countingsort, Bucketsort, Radixsort, Stoogesort, usw.

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.

Dutch National Flag Problem (II)

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

- Eingabe:**
- Ein Array E mit n Einträgen, wobei für alle $0 < i \leq n$
 $E[i] == \text{rot}, E[i] == \text{blau} \text{ oder } E[i] == \text{weiss}$
 - Ordnung: $\text{rot} < \text{weiss} < \text{blau}$

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 (I)



Dutch National Flag Problem (III)

Hauptidee

Zerlege das Array E in 4 Regionen:

- (1) $0 < i \leq r$, (2) $r < i < u$, (3) $u \leq i < b$, und (4) $b \leq i \leq n$

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

- $E[1], \dots, E[r]$ ist die **“rote”** Region, d.h. für alle $0 < i \leq r$
 $E[i] == \text{rot}$.
- $E[r+1], \dots, E[u-1]$ ist die **“weiße”** Region, d.h. für alle $r < i < u$
 $E[i] == \text{weiss}$.
- $E[u], \dots, E[b-1]$ ist **unbekannte** Region, d.h. für alle $u \leq i < b$
 $E[i] == \text{rot} \text{ oder } E[i] == \text{weiss} \text{ oder } E[i] == \text{blau}$.
- $E[b], \dots, E[n]$ ist die **“blaue”** Region, d.h. für alle $b \leq i \leq n$
 $E[i] == \text{blau}$.

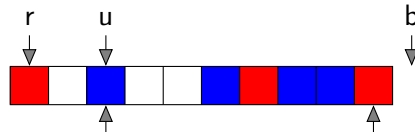
Arrayelemente können mit der swap-Operation vertauscht werden.

Dutch National Flag Problem (IV)

```

1 void DutchNationalFlag(Color E[], int n) {
2   int r = 0, b = n + 1; // rote und blaue Regionen sind leer
3   int u = 1; // weiße Region ist leer, die unbekannte == E
4   while (u < b) {
5     if (E[u] == rot) {
6       swap(E[r + 1], E[u]);
7       r = r + 1; // vergrößere die rote Region
8       u = u + 1; // verkleinere die unbekannte Region
9     }
10    if (E[u] == weiss) {
11      u = u + 1;
12    }
13    if (E[u] == blau) {
14      swap(E[b - 1], E[u]);
15      b = b - 1; // vergrößere die blaue Region
16    }
17  }
18 }

```



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

Übersicht

- 1 Sortieren - Einführung
 - Bedeutung des Sortierens
 - Dutch National Flag Problem
- 2 Sortieren durch Einfügen
- 3 Mergesort
 - Das Divide-and-Conquer Paradigma
 - Mergesort
- 4 Effizienteres Sortieren?

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.

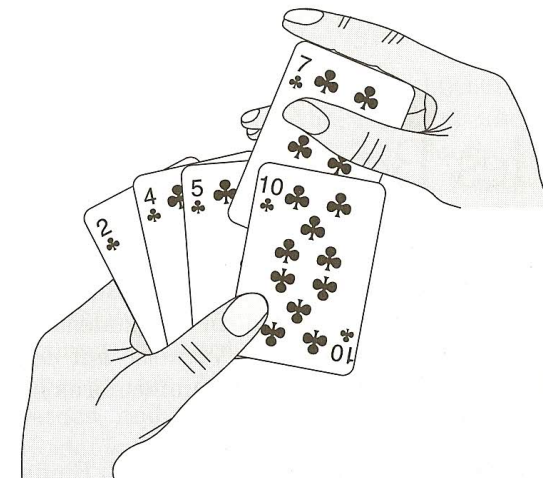
Zeitkomplexität

Betrachte als elementare Operationen Vergleiche der Form $E[i] == \dots$

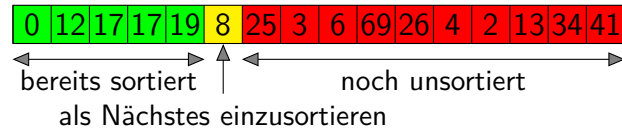
Die Worst-Case Zeitkomplexität ist $\Theta(n)$, da:

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 eins verkleinert wird.

Sortieren durch Einfügen – Insertionsort



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 – Best- und Worst-Case-Analyse

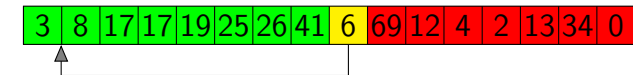
Best-Case

- ▶ 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 – Animation und Algorithmus



```

1 void insertionSort(int E[]) {
2   int i, j;
3   for (i = 1; i < E.length; i++) {
4     int v = E[i]; // speichere E[i]
5     for (j = i; j > 0 && E[j-1] > v; j--) {
6       E[j] = E[j-1]; // schiebe Element j-1 nach rechts
7     }
8     E[j] = v; // füge E[i] an der richtigen Stelle ein
9   }
10 }

```

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

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.

Es gilt:

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

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

Insertionsort – Average-Case-Analyse (II)

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

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

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

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

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

| Vereinfachen

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

Übersicht

- 1 Sortieren - Einführung
 - Bedeutung des Sortierens
 - Dutch National Flag Problem
- 2 Sortieren durch Einfügen
- 3 Mergesort
 - Das Divide-and-Conquer Paradigma
 - Mergesort
- 4 Effizienteres Sortieren?

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

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

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.

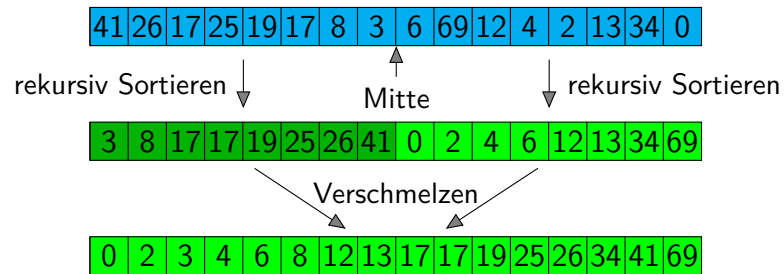
Das Paradigma von Teile-und-Beherrsche umfasst 3 Schritte auf jeder Rekursionsebene:

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 – Strategie

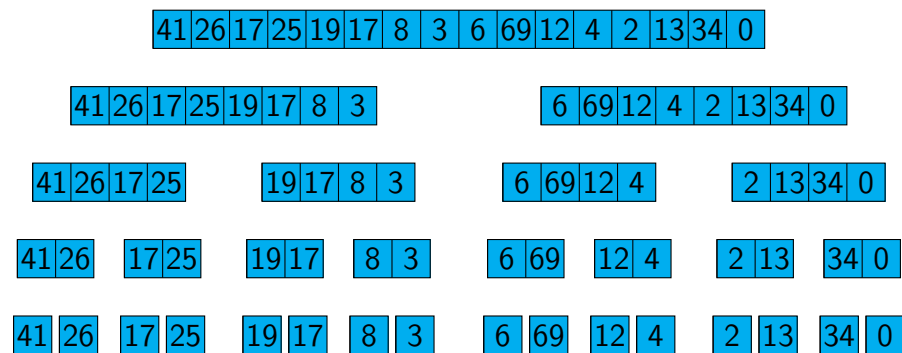


Teile das Array in zwei – möglichst gleichgroße – Hälften.

Beherrsche: Sortiere die Teile durch rekursive Mergesort-Aufrufe.

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

Mergesort – Animation



Mergesort – Algorithm

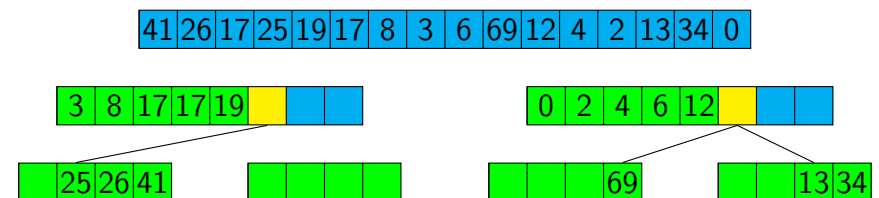
```

1 void mergeSort(int E[], int left, int right) {
2   if (left < right) {
3     int mid = (left + right) / 2; // finde Mitte
4     mergeSort(E, left, mid);     // sortiere linke Hälfte
5     mergeSort(E, mid + 1, right); // sortiere rechte Hälfte
6     // Verschmelzen der sortierten Hälften
7     merge(E, left, mid, right);
8   }
9 }
10 // Aufruf: mergeSort(E, 0, E.length-1);

```

► Verschmelzen kann man in Linearzeit. – Wie?

Mergesort – Animation



Mergesort – Verschmelzen in Linearzeit

```

1 void merge(int E[], int left, int mid, int right) {
2   int a = left, b = mid + 1;
3   int Eold[] = E;
4   for (; left <= right; left++) {
5     if (a > mid) { // Wir wissen (Widerspruch): b <= right
6       E[left] = Eold[b];
7       b++;
8     } else if (b > right || Eold[a] <= Eold[b]) { // stabil: <=
9       E[left] = Eold[a];
10      a++;
11    } else { // Eold[a] > Eold[b]
12      E[left] = Eold[b];
13      b++;
14    }
15  }
16 }

```

- ▶ Mergesort ist **stabil** (vgl. Zeile 8), d. h. die Reihenfolge von Elementen mit *gleichem* Schlüssel bleibt erhalten.

Übersicht

- 1 Sortieren - Einführung
 - Bedeutung des Sortierens
 - Dutch National Flag Problem
- 2 Sortieren durch Einfügen
- 3 Mergesort
 - Das Divide-and-Conquer Paradigma
 - Mergesort
- 4 Effizienteres Sortieren?

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

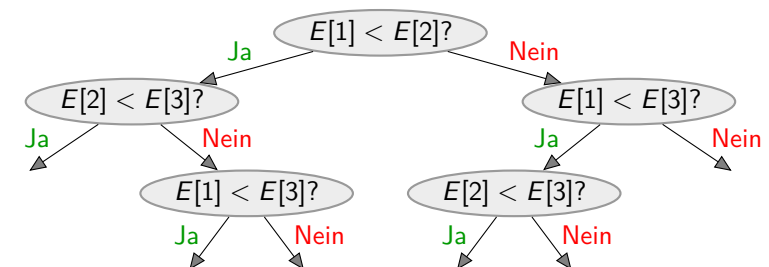
Das Worst-Case-Ergebnis gilt genauso im Best-Case, und damit auch im Average-Case: $W(n) = B(n) = A(n) \in \Theta(n \cdot \log n)$.

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.

Wie effizient kann man sortieren? (I)



Betrachte vergleichsbasierte Sortialgorithmen 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:

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

⇒ Mit $n! \leq 2^k$ erhält man $k \geq \lceil \log(n!) \rceil$ **Vergleiche** im Worst-Case.

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