Algorithmen und Datenstrukturen
Kapitel 5
Sortieren - (k?)eine untere Schranke

Frank Heitmann
heitmann@informatik.uni-hamburg.de

11. November 2015

Sortieren - Die Problemstellung

Definition (Das Sortierproblem)

Eingabe: Eine Sequenz $< a_1, a_2, \ldots, a_n >$ von n Zahlen.
Gesucht: Eine Permutation $< a'_1, a'_2, \ldots, a'_n >$ der Eingabesequenz mit $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Min/MaxSort (SelectionSort)

Algorithmus 1 MaxSort($A[1 \ldots n]$)

```
1: for i = n downto 2 do
2:     idx = max(A[1..i])
3:     swap(A[i], A[idx])
4: end for
```

Algorithmus 2 max($A[1 \ldots n]$)

```
1: idxMax = 1
2: for i = 2 to n do
3:     if A[i] > A[idxMax] then
4:         idxMax = i
5:     end if
6: end for
7: return idxMax;
```

InsertionSort: Der Algorithmus

Algorithmus 3 InsertionSort($A[1 \ldots n]$)

```
1: for j = 2 to n do
2:     key = A[j]
3:     i = j − 1
4:     while i > 0 und A[i] > key do
5:         A[i + 1] = A[i]
6:         i = i − 1
7:     end while
8:     A[i + 1] = key
9: end for
```

BubbleSort

Algorithmus 4 BubbleSort($A[1 \ldots n]$)

```
 1: do
 2:     bSwap = false
 3:     for i = 1 to n − 1 do
 4:         if A[i] > A[i + 1] then
 5:             swap(A[i], A[i + 1])
 6:             bSwap = true
 7:         end if
 8:     end for
 9:     n = n − 1
10: while bSwap && n > 1
```

Divide & Conquer (Wiederholung)

Die drei Schritte des Divide & Conquer (Teile-und-Beherrsche) Paradigmas

1. **Teile** das Problem in n Teilprobleme auf
2. **Beherrsche** die Teilprobleme durch rekursives Lösen (d.h. teile sie weiter; sind die Teilprobleme hinreichend klein, löse sie direkt)
3. **Verbinde** die Lösungen der Teilprobleme zur Lösung des übergeordneten Problems und letztendlich des ursprünglichen Problems.

Die Schritte finden auf jeder Rekursionsebene statt.

Mergesort - Idee

- Teile die Eingabe in zwei Teile gleicher (ca.) Größe.
- Löse die zwei Teilprobleme unabhängig und rekursiv.
- Kombiniere die zwei Ergebnisse in eine Gesamtlösung, wobei wir nur *lineare viel Zeit für die anfängliche Teilung und die abschließende Zusammenführung* nutzen wollen.

Mergesort - Algorithmus

Algorithmus 5 mergesort(A,l,r)

1: **if** $l < r$ **then**
2: $q = (l + r)/2$
3: $mergesort(A, l, q)$
4: $mergesort(A, q + 1, r)$
5: $merge(A, l, q, r)$
6: **end if**

Mergen

- Gegeben zwei sortierte Listen A und B.
- cur_A und cur_B seien Zeiger auf Elemente in A bzw. B. Am Anfang zeigen sie auf das erste Element der jeweiligen Liste.
- Solange beide Listen nicht leer sind:
  - Seien a_i und b_j die Elemente auf die cur_A und cur_B zeigen.
  - Füge das kleinere Element der Ausgabeliste hinzu.
  - Erhöhe den Zeiger der Liste dessen Element gewählt wurde.
- Ist eine Liste leer, füge die andere an die Ausgabeliste an.

QuickSort - Die Idee

- Teile: Das Feld $A[p..r]$ wird in zwei Teilfelder $A[p..q-1]$ und $A[q+1..r]$ zerlegt, wobei alle Elemente im ersten Teilfeld kleiner oder gleich $A[q]$ sind und alle Elemente im zweiten Teilfeld größer oder gleich $A[q]$ sind. Der Index q wird in der Zerlegungsprozedur berechnet.
- Beherrsche: Sortiere die beiden Teilfelder durch rekursive Aufrufe.
- Verbinde: Da die Teilfelder (rekursiv) in-place sortiert werden, brauchen sie nicht weiter verbunden werden. Das Feld $A[p..r]$ ist nun sortiert.

QuickSort - Psuedocode (1/2)

Algorithmus 6 QuickSort(A, p, r)

```
1: if p < r then
2:     q = Partition(A, p, r)
3:     QuickSort(A, p, q − 1)
4:     QuickSort(A, q + 1, r)
5: end if
```

Um ein Array A zu sortieren rufen wir

$$\texttt{QuickSort}(A, 1, \texttt{länge}[A])$$

auf.

QuickSort - Pseudocode (2/2)

Algorithmus 7 Partition(A, p, r)

```
 1: x = A[r]      // dies ist das Pivotelement
 2: i = p − 1
 3: for j = p to r − 1 do
 4:     if A[j] ≤ x then
 5:         i = i + 1
 6:         swap(A[i], A[j])
 7:     end if
 8: end for
 9: swap(A[i + 1], A[r])
10: return  i + 1
```

HeapSort - Die Idee

1. Gegeben ein Array, stelle zunächst einen Heap her (BuildMaxHeap).
2. Vertausche die Wurzel (maximales Element!) mit dem Element ganz rechts unten (das Element steht im Array ganz rechts! (Von den noch nicht behandelten Elementen)).
3. Verringere die Heap-Größe um 1 und führe MaxHeapify auf die Wurzel aus (die und nur die verletzt jetzt möglicherweise die Heap-Eigenschaft).
4. Fahre bei 2. fort.

HeapSort - Pseudocode

Algorithmus 8 HeapSort(A)

```
1: BuildMaxHeap(A)
2: for i = länge[A] downto 2 do
3:     swap(A[1], A[i])
4:     heap-größe[A] = heap-größe[A] − 1
5:     MaxHeapify(A, 1)
6: end for
```

Anmerkung

Ist das Array gar nicht vollständig gefüllt, sollte in der `for`-Schleife nicht länge[A] genutzt werden, sondern heap-größe[A] (die weiteren Plätze im Array werden ja gar nicht genutzt!).

BuildMaxHeap - Die Idee und Pseudocode

1. Suche im Baum von unten nach oben und von rechts nach links (im Array also von rechts nach links) den ersten Knoten, der als Wurzel eines Teilbaumes betrachtet, kein Heap mehr ist.
2. Stelle die Heap-Eigenschaft her (mit einem Aufruf von `MaxHeapify`).
3. Fahre bei 1. fort.

Algorithmus 9 BuildMaxHeap(A, i)

```
1: heap-größe[A] = länge[A]
2: for i = ⌊länge[A]/2⌋ downto 1 do
3:     MaxHeapify(A, i)
4: end for
```

MaxHeapify - Die Idee

- Vergleiche $A[i]$ mit $A[Left(i)]$ und $A[Right(i)]$.
- Ist $A[i]$ am größten, so sind wir fertig.
- Sonst sei max der Index des größeren Elementes, tausche $A[i]$ und $A[max]$ (d.h. tausche $A[i]$ mit dem größeren Kind).
- Fahre nun mit $A[max]$ (hat den Wert von $A[i]$!) so wie eben mit $A[i]$ fort.

Anmerkung

Der Wert von $A[i]$ wandert im Baum nach unten. Bei allen Tauschoperationen ist dieser Wert beteiligt. Es finden sonst keine Tauschoperationen statt!

MaxHeapify - Pseudocode

Algorithmus 10 MaxHeapify(A, i)

```
1:  l = Left(i)
2:  r = Right(i)
3:  if l ≤ heap-größe[A] und A[l] > A[i] then
4:      max = l
5:  else
6:      max = i
7:  end if
8:  if r ≤ heap-größe[A] und A[r] > A[max] then
9:      max = r
10: end if
11: if max ≠ i then
12:     swap(A[i], A[max])
13:     MaxHeapify(A, max)
14: end if
```

Wie schnell geht es?

Wir kennen jetzt Verfahren,

- die in $O(n^2)$ sortieren (Min/MaxSort (SelectionSort), InsertionSort, BubbleSort, Quicksort) und welche,
- die sogar in $O(n \cdot \log n)$ sortieren (MergeSort, HeapSort).

Geht es *noch* schneller?

Nein! Zumindest nicht bei *vergleichenden Sortierverfahren*...

Vergleichendes Sortieren

Alle bisherigen Sortierverfahren basieren auf *Vergleichen zwischen den Eingabeelementen*. Diese Sortierverfahren werden daher als *vergleichende Sortierverfahren* bezeichnet.

Wir zeigen nun für solche Sortierverfahren eine untere Schranke von mindestens $\Omega(n \cdot \log n)$ nötigen Vergleichsoperationen im schlechtesten Fall.

Wichtige Anmerkung

Damit sind MergeSort und HeapSort asymptotisch (!) optimal!

Vergleichende Sortierverfahren

Bei vergleichenden Sortierverfahren sind folgende Tests möglich

- $a_i < a_j$
- $a_i \leq a_j$
- $a_i = a_j$
- $a_i \geq a_j$
- $a_i > a_j$

Nur Aufgrund der Ergebnisse dieser Tests wird die Reihenfolge bestimmt (insb. wurde z.B. nie mit den Werten der Elemente selbst gearbeitet).

Anmerkung

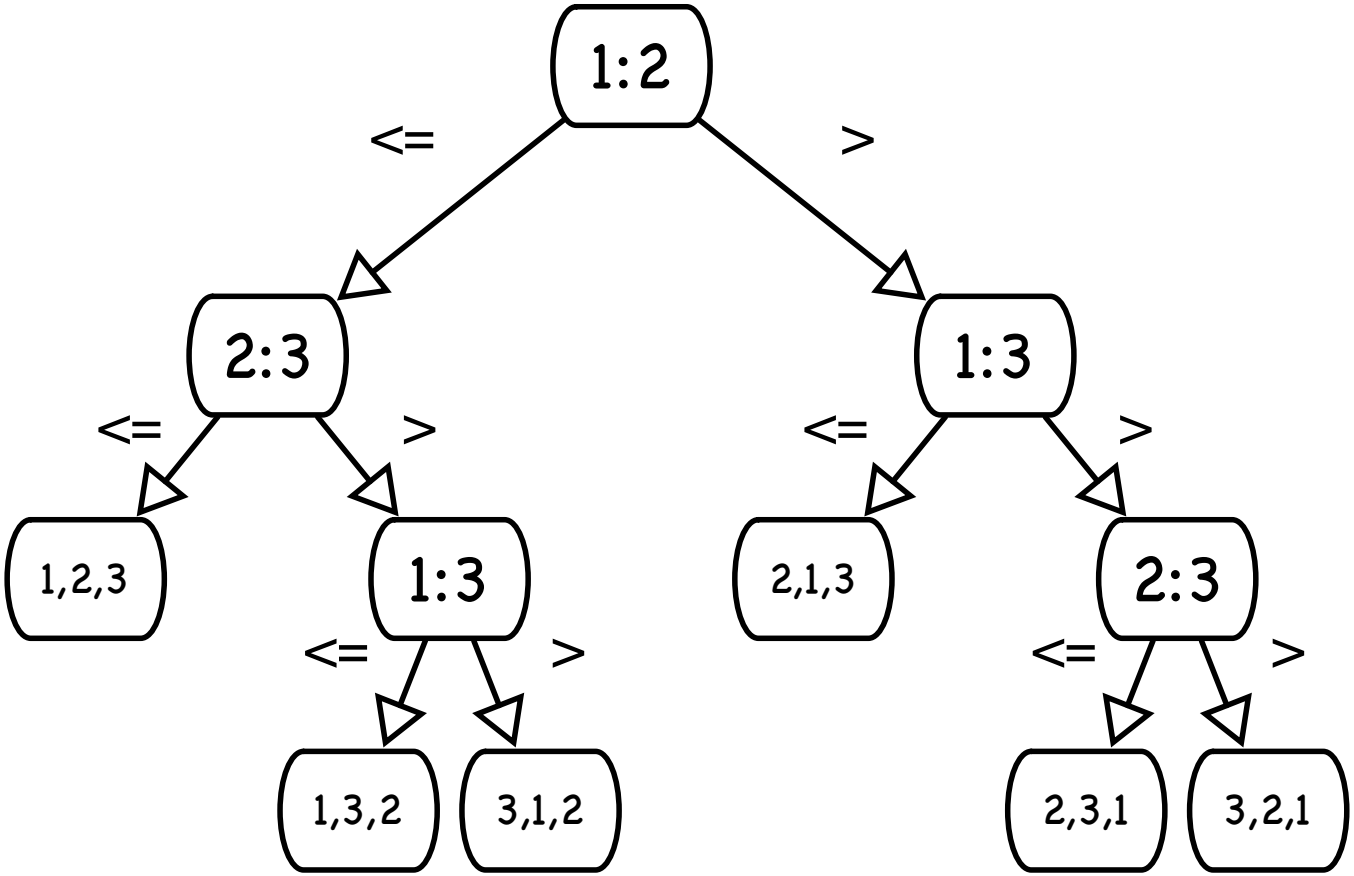
Man überlege sich nochmal, dass dies bei allen bisherigen Sortierverfahren tatsächlich der Fall ist!

Vergleichende Sortierverfahren

Wir gehen nachfolgend davon aus, dass die Elemente alle paarweise verschieden sind. Wir brauchen dann nur eine Vergleichsoperation $a_i \leq a_j$.

Wir können dann vergleichende Sortierverfahren abstrakt als *Entscheidungsbäume* betrachten...

Entscheidungsbäume - Beispiel

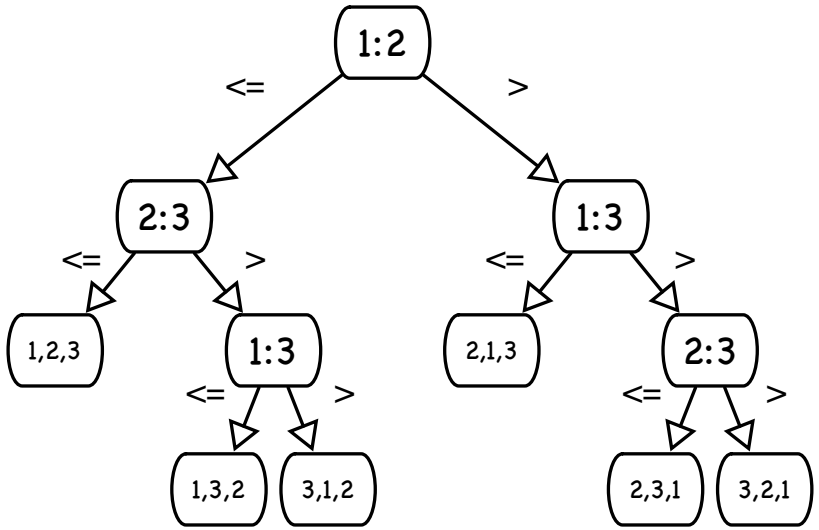


Ein Entscheidungsbaum für InsertionSort (Array.length = 3)

Entscheidungsbäume - Definition (1/2)

Definition

Ein Entscheidungsbaum ist ein binärer Baum, der die Vergleiche zwischen den Elementen darstellt, die von einem speziellen Sortieralgorithmus auf einem Array einer bestimmten Größe durchgeführt werden. Andere Aspekte des Algorithmus werden vernachlässigt.

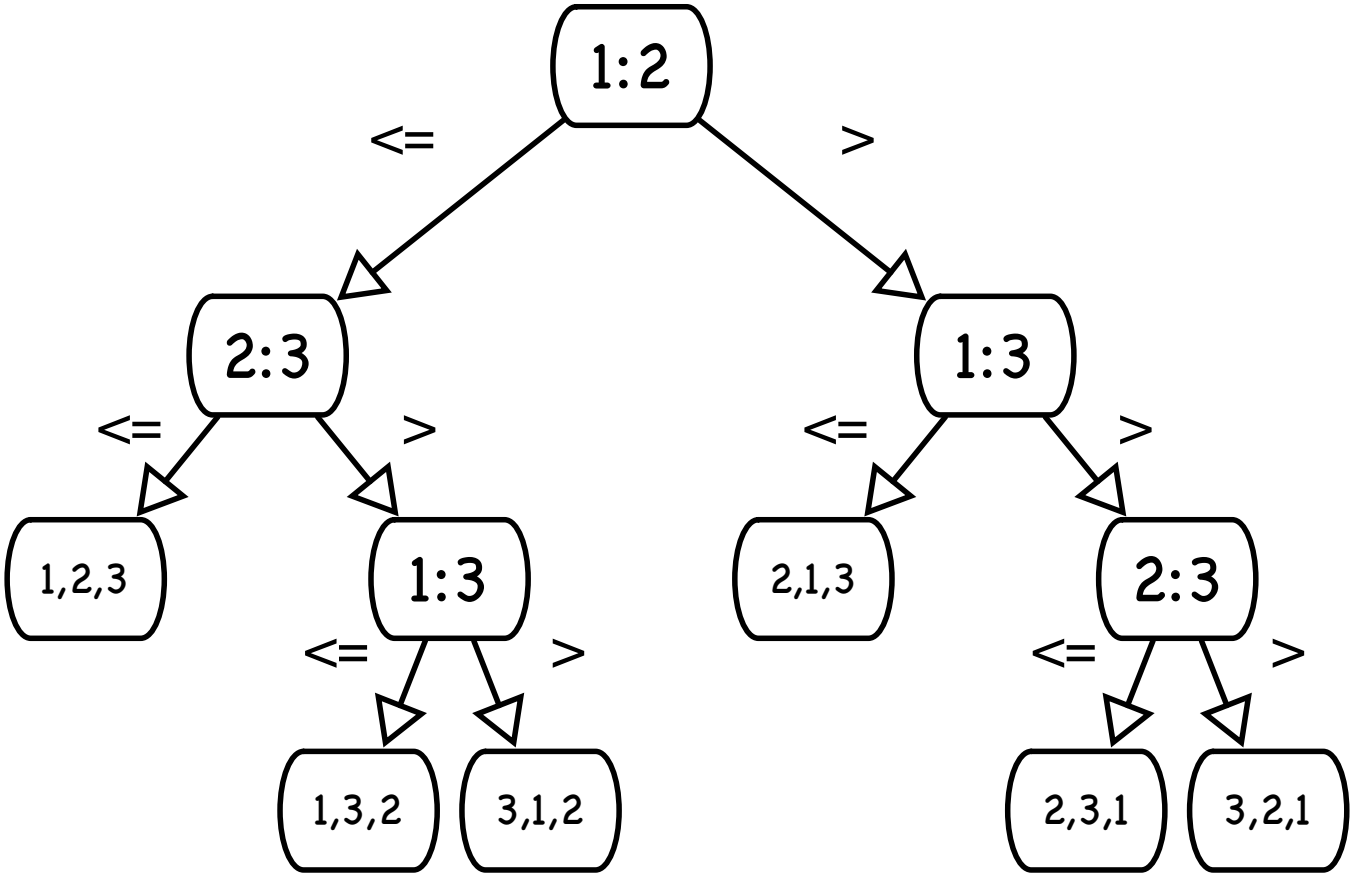


Entscheidungsbäume - Definition (2/2)

Definition

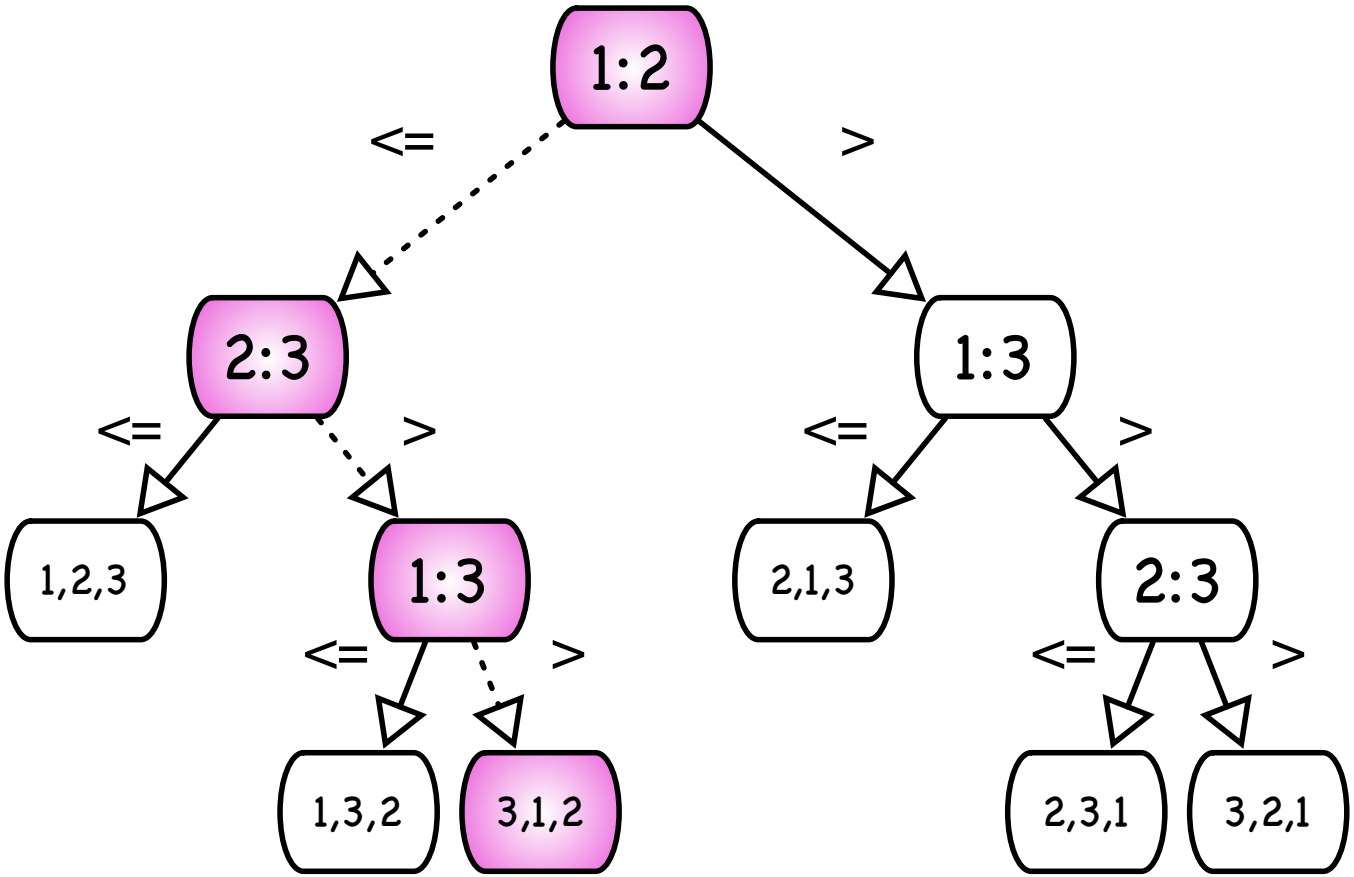
- Ein Knoten $i : j$ stellt einen Vergleich zwischen a_i und a_j da. Abhängig vom Ergebnis, wird dann im linken oder rechten Teilbaum weitergearbeitet.
- Ein Weg von der Wurzel zu einem Blatt ist eine Ausführung des Sortieralgorithmus auf einem Array der Länge n.
- Ein Blatt ist mit einer Permutation x, y, z gekennzeichnet, was bedeutet, dass der Algorithmus das Ergebnis $a_x \leq a_y \leq a_z$ liefert.

Entscheidungsbäume - Beispiel 1



Ein Entscheidungsbaum für InsertionSort (Array.length = 3)

Entscheidungsbäume - Beispiel 2



Ein Entscheidungsbaum für InsertionSort $a_1 = 7, a_2 = 9, a_3 = 4$

Entscheidungsbäume - Eigenschaften

Satz

Eine wichtige Erkenntnis: Jeder korrekte Sortieralgorithmus muss in der Lage sein, jede Permutation seiner Eingabe zu erzeugen. (Dies ist beweisbar!)
Daher ist eine notwendige Bedingung für die Korrektheit eines vergleichenden Sortierverfahrens, dass der zugehörige Entscheidungsbaum mindestens $n!$ *von der Wurzel aus erreichbare Blätter haben muss. (Eingabe hat die Länge* n*.)*

Die untere Schranke - Die Idee

Da ein Entscheidungsbaum die Vergleiche eines bestimmten Verfahrens repräsentiert, sind im schlechtesten Fall soviele Vergleiche nötig, wie Vergleiche auf dem Längsten Pfad im Entscheidungsbaum von Wurzel bis zu einem Blatt stattfinden.

Die Anzahl der Vergleichsoperationen im schlechtesten Fall entspricht also der Höhe des Entscheidungsbaumes ... und dieser muss mindestens $n!$ Blätter haben!

Die untere Schranke - Der Beweis

Sei h die Höhe des Baumes, l die Anzahl der erreichbaren Blätter und n die Länge des Arrays, dann gilt

$$n! \leq l \leq 2^h$$

Bilden des Logarithmus führt zu

$$\log n! \leq h$$

was wegen $\log n! \in \Theta(n \log n)$ zum gesuchten Ergebnis führt.

Satz

Jeder vergleichende Sortieralgorithmus muss im schlechtesten Fall mindestens $\Omega(n \log n)$ Operationen ausführen.

Satz

HeapSort und MergeSort sind asymptotisch optimal.

CountingSort - Die Idee

Um die Grenze doch noch zu drücken, gehen wir von vergleichenden Sortierverfahren weg und nutzen Eigenschaften der Elemente. (Die Sortierverfahren sind jetzt aber weniger allgemein!)

CountingSort. Sei $k \in \mathbb{N}$, wir gehen nun davon aus, dass für jedes Element a_i der Eingabe $0 \leq a_i \leq k$ gilt.

- Bestimme für jedes a_i die *Anzahl m der Elemente*, die kleiner als a_i sind.
- Es ist dann $B[m+1] = a_i$ (im Ausgabearray).

CountingSort - Pseudocode

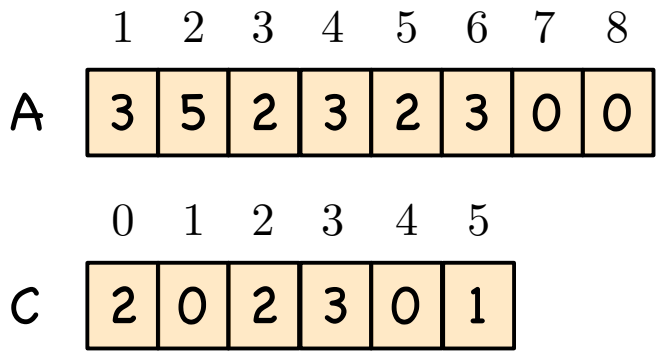
Algorithmus 11 CountingSort(A, B, k)

```
1:  for i = 0 to k do
2:      C[i] = 0
3:  end for
4:  for j = 1 to länge[A] do
5:      C[A[j]] = C[A[j]] + 1
6:  end for
7:  for i = 1 to k do
8:      C[i] = C[i] + C[i − 1]
9:  end for
10: for j = länge[A] downto 1 do
11:     B[C[A[j]]] = A[j]
12:     C[A[j]] = C[A[j]] − 1
13: end for
```

CountingSort - Beispiel

Algorithmus 12 CountingSort(A, B, k) - Teilstück

1: **for** $j = 1$ to länge[A] **do**
2: $\quad C[A[j]] = C[A[j]] + 1$
3: **end for**



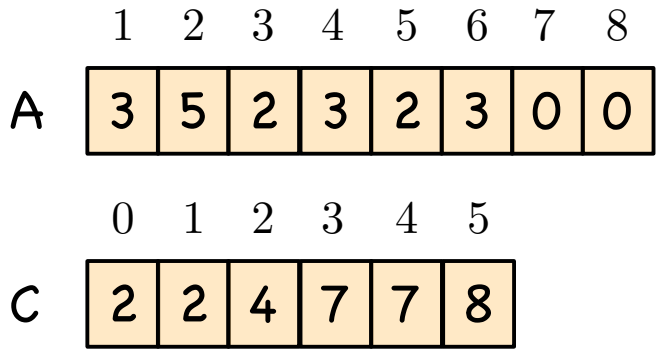
Anmerkung

In $C[k]$ ist nun die Anzahl der Elemente (aus A), die gleich k sind, gespeichert. $C[0] = 2$ bedeutet, dass es in A zweimal die 0 gibt.

CountingSort - Beispiel

Algorithmus 13 CountingSort(A, B, k) - Teilstück

1: **for** $i = 1$ to k **do**
2: $\quad C[i] = C[i] + C[i-1]$
3: **end for**



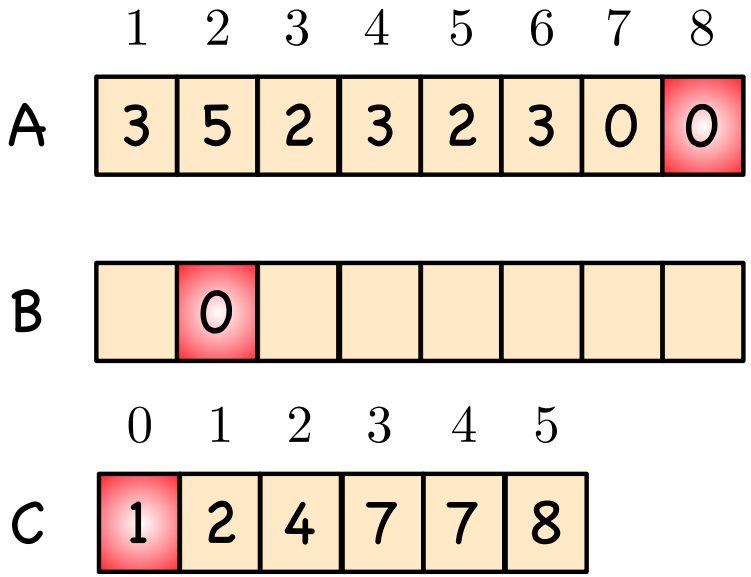
Anmerkung

In $C[k]$ ist nun die Anzahl der Elemente (aus A), die *kleiner*-gleich k sind, gespeichert. $C[2] = 4$ bedeutet, dass es in A vier Element gibt, die kleiner-gleich 2 sind.

CountingSort - Beispiel

Algorithmus 14 CountingSort(A, B, k) - Teilstück

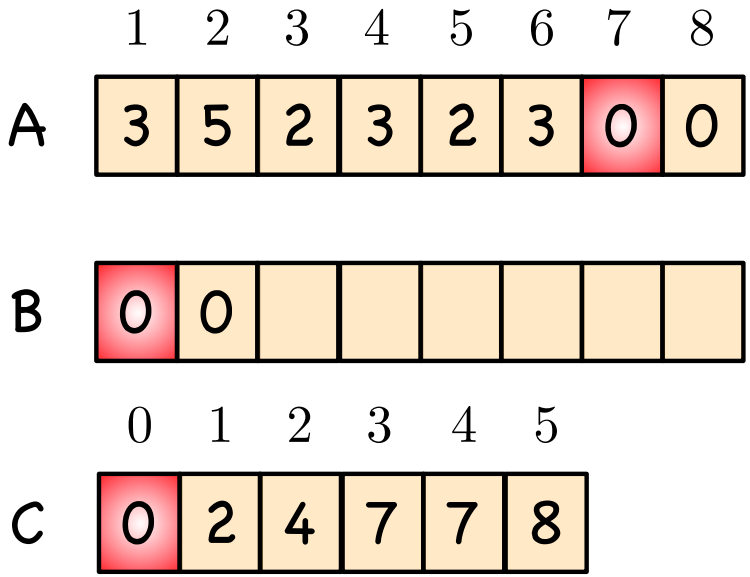
1: **for** $j =$ länge$[A]$ downto 1 **do**
2: $\quad B[C[A[j]]] = A[j]$
3: $\quad C[A[j]] = C[A[j]] - 1$
4: **end for**



CountingSort - Beispiel

Algorithmus 15 CountingSort(A, B, k) - Teilstück

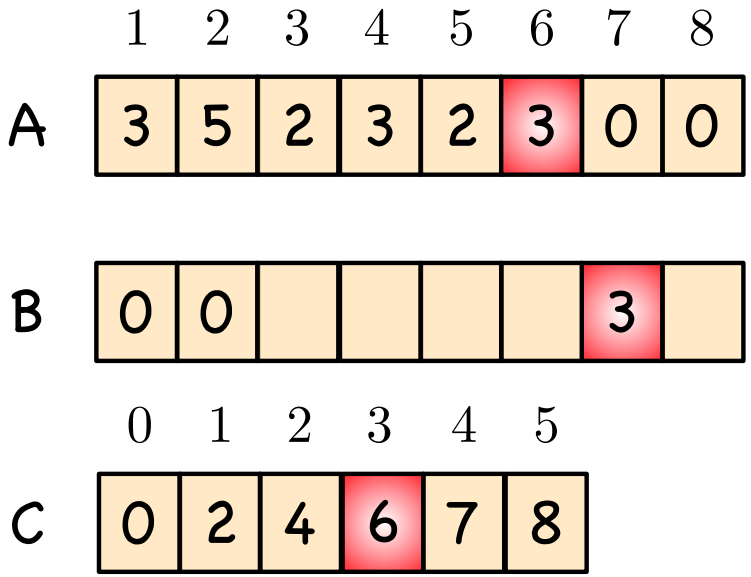
1: **for** $j =$ länge$[A]$ downto 1 **do**
2: $\quad B[C[A[j]]] = A[j]$
3: $\quad C[A[j]] = C[A[j]] - 1$
4: **end for**



CountingSort - Beispiel

Algorithmus 16 CountingSort(A, B, k) - Teilstück

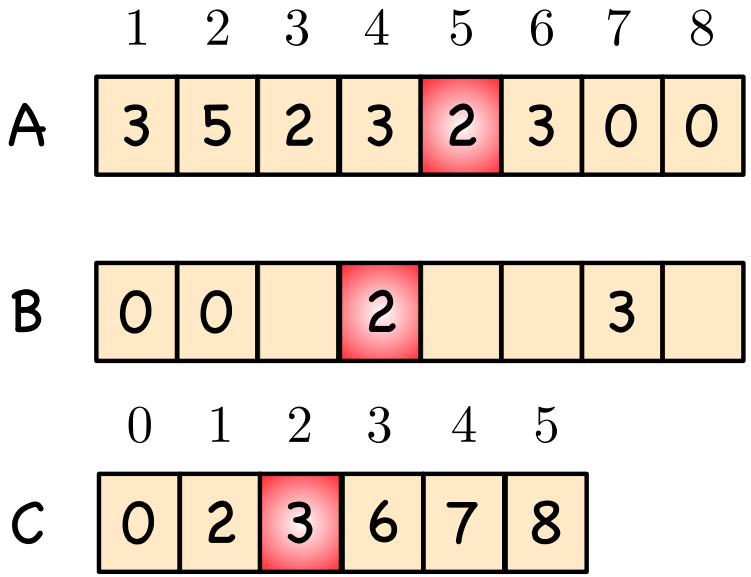
1: **for** $j =$ länge$[A]$ downto 1 **do**
2: $\quad B[C[A[j]]] = A[j]$
3: $\quad C[A[j]] = C[A[j]] - 1$
4: **end for**



CountingSort - Beispiel

Algorithmus 17 CountingSort(A, B, k) - Teilstück

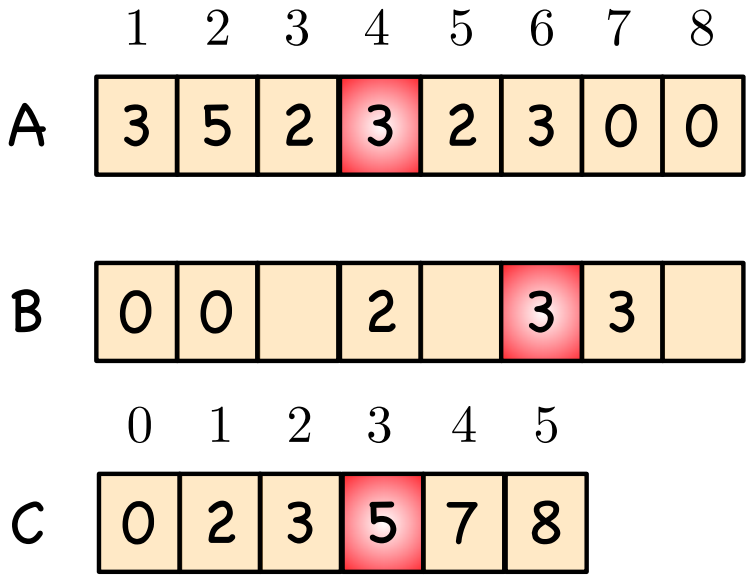
1: **for** $j =$ länge$[A]$ downto 1 **do**
2: $\quad B[C[A[j]]] = A[j]$
3: $\quad C[A[j]] = C[A[j]] - 1$
4: **end for**



CountingSort - Beispiel

Algorithmus 18 CountingSort(A, B, k) - Teilstück

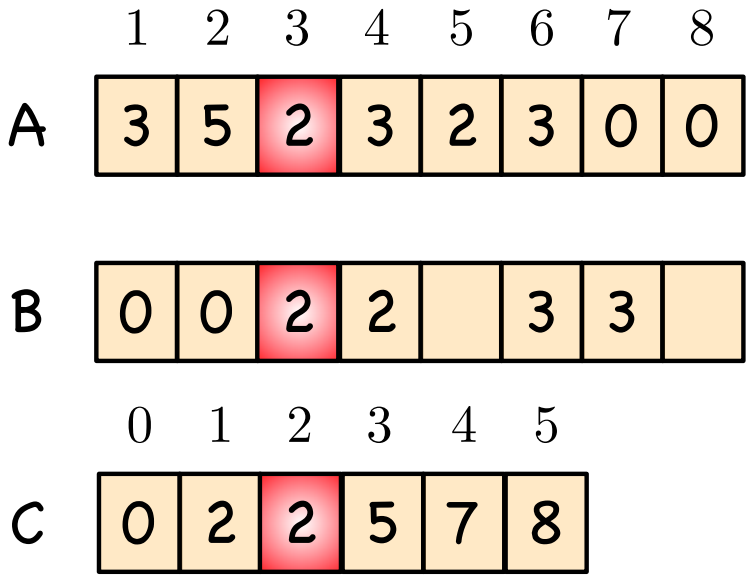
1: **for** $j =$ länge$[A]$ downto 1 **do**
2: $\quad B[C[A[j]]] = A[j]$
3: $\quad C[A[j]] = C[A[j]] - 1$
4: **end for**



CountingSort - Beispiel

Algorithmus 19 CountingSort(A, B, k) - Teilstück

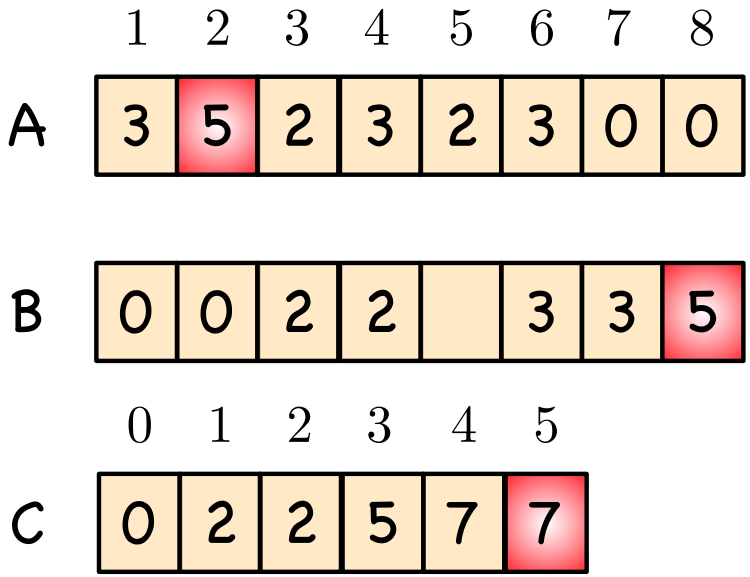
1: **for** $j =$ länge$[A]$ downto 1 **do**
2: $\quad B[C[A[j]]] = A[j]$
3: $\quad C[A[j]] = C[A[j]] - 1$
4: **end for**



CountingSort - Beispiel

Algorithmus 20 CountingSort(A, B, k) - Teilstück

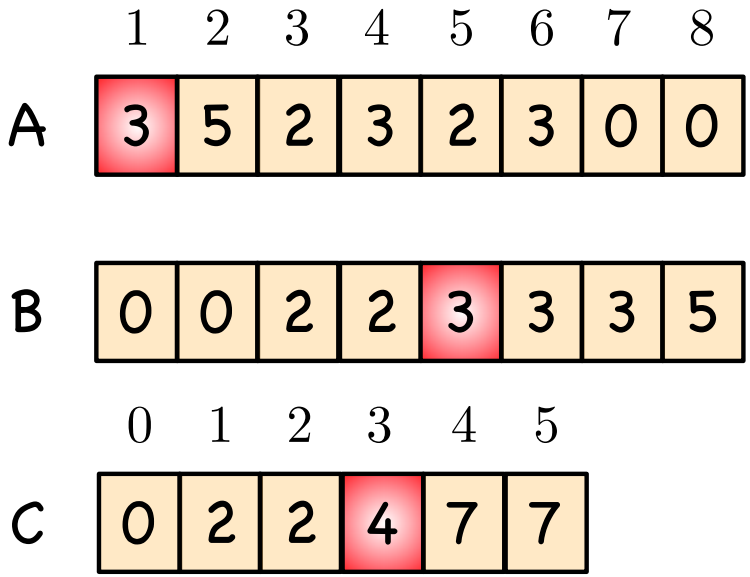
1: **for** $j =$ länge$[A]$ downto 1 **do**
2: $\quad B[C[A[j]]] = A[j]$
3: $\quad C[A[j]] = C[A[j]] - 1$
4: **end for**



CountingSort - Beispiel

Algorithmus 21 CountingSort(A, B, k) - Teilstück

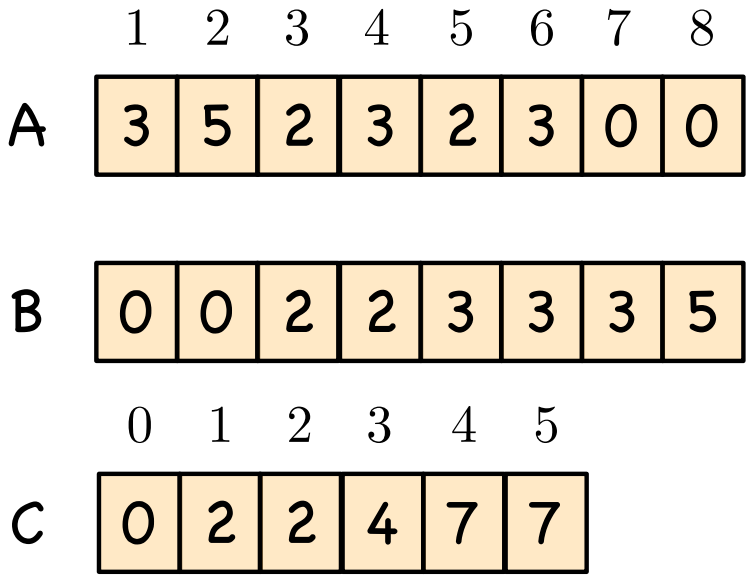
1: **for** $j =$ länge$[A]$ downto 1 **do**
2: $\quad B[C[A[j]]] = A[j]$
3: $\quad C[A[j]] = C[A[j]] - 1$
4: **end for**



CountingSort - Beispiel

Algorithmus 22 CountingSort(A, B, k) - Teilstück

1: **for** $j =$ länge$[A]$ downto 1 **do**
2: $\quad B[C[A[j]]] = A[j]$
3: $\quad C[A[j]] = C[A[j]] - 1$
4: **end for**



CountingSort - Analyse

Algorithmus 23 CountingSort(A, B, k)

```
1:  for i = 0 to k do
2:      C[i] = 0
3:  end for
4:  for j = 1 to länge[A] do
5:      C[A[j]] = C[A[j]] + 1
6:  end for
7:  for i = 1 to k do
8:      C[i] = C[i] + C[i − 1]
9:  end for
10: for j = länge[A] downto 1 do
11:     B[C[A[j]]] = A[j]
12:     C[A[j]] = C[A[j]] − 1
13: end for
```

CountingSort - Analyse

Wir haben vier for-Schleifen. Zwei in $\Theta(k)$, zwei in $\Theta(n)$, damit folgt:

Satz

CountingSort ist ein stabiles Sortierverfahren, das Eingaben der Länge n, wobei jedes Element zwischen 0 und k liegt, in $O(n+k)$ sortiert. In der Praxis nutzt man CoutingSort meist, wenn $k \in O(n)$ gilt, so dass CoutingSort dann eine Laufzeit von $\Theta(n)$ hat.

Anmerkung

Die Korrektheit kann wieder mit Schleifeninvarianten gezeigt werden. Zur Übung und bei Bedarf im [Cormen] nachlesen.

CountingSort - Anmerkung

Anmerkung

CountingSort ist kein vergleichendes Sortierverfahren. Tatsächlich findet man nirgendwo im Code Vergleiche zwischen zwei Elementen. CountingSort verwendet die Werte der Elemente als Index in ein Feld. Die untere Schranke für vergleichendes Sortieren gilt hier also nicht - und tatsächlich durchbricht CountingSort sie ja (wenn $k \in O(n)$ gilt).

Weitere Verfahren

Es gibt weitere nicht-vergleichende Sortierverfahren, die - bei bestimmten Eingaben - die Grenze von $\Omega(n \cdot \log n)$ durchbrechen.

- *RadixSort* geht von d-stelligen Zahlen aus, wobei jede Stelle k mögliche Werte hat. Die Zahlen werden dann Stelle für Stelle beginnend bei der niederwertigsten durch ein stabiles (!) Sortierverfahren (z.B. CountingSort) sortiert. Die Laufzeit ist in $\Theta(d \cdot (n + k))$.
- *BucketSort* geht von einer gleichverteilten Eingabe auf dem Interval $[0, 1)$ aus, teilt dieses Interval in n gleichgroße Buckets und verteilt die n Eingabewerte auf die Buckets. (Es sollten dann wenige Eingabewerte im gleichen Bucket liegen. Die Buckets werden dann sortiert und aneinandergefügt.)

Zusammenfassung und Ausblick

- Vergleichendes Sortieren
  - InsertionSort, Min/MaxSort (SelectionSort), BubbleSort
  - QuickSort
  - MergeSort, HeapSort
  - Untere Schranke für das vergleichende Sortieren: $\Omega(n \cdot \log n)$
- Nicht-vergleichendes Sortieren
  - CountingSort, (RadixSort, BucketSort)

Untere Schranken für andere Verfahren?