

SUCHEN UND SORTIEREN

Suche:

Beispiel: Finde die Prüfung von Maria Müller
in einem Stapel aus 600 Prüfungen.

Problem (Suche):

Input: Ein Array $A[1..n]$ von Zahlen und ein Element b .

Output: Index k mit $A[k]=b$ oder „nicht gefunden“

Fall 1: A nicht sortiert

Algorithmus 1 (Lineare Suche):

Linear Search (A, b) // $A = A[1..n]$ Array

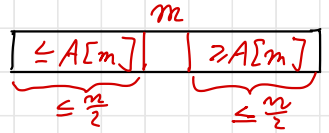
```
for  $i=1..n$   
  if  $A[i]=b$ : return  $i$   
return „nicht gefunden“
```

Laufzeit $O(n)$

Geht es besser? NEIN, denn wir müssen jedes Element
von A anschauen, falls b nicht vorkommt!

Fall 2: A sortiert, $A[1] \leq A[2] \leq \dots \leq A[n]$

Algorithmus 2: Binäre Suche



Binary Search (A, b) // für sortiertes A

$l \leftarrow 1, r \leftarrow n$ // $A[l \dots r]$ gibt den Bereich an, in dem wir noch suchen müssen

while $l \leq r$ do

$m \leftarrow \lfloor \frac{l+r}{2} \rfloor$ // $\lfloor \dots \rfloor$, auch floor (...) geschrieben: abrunden

if $b = A[m]$: return m

if $b < A[m]$: $r \leftarrow m-1$

else: $l \leftarrow m+1$

return „nicht gefunden“

Laufzeit: $T(1) = c$ konstant
(worst-case)

$T(n) \leq T(\frac{n}{2}) + d$, d konstant

↑
stillschweigend benutzt: Laufzeit wird höchstens größer für längere Arrays.

Behauptung: $T(n) \leq c + d \cdot \log_2 n$

($n = 2^k$)

Beweis: Induktion (Übung)

$\rightarrow T(n) = O(\log n)$

Geht es besser? NEIN!

(Wir nehmen hier an, dass die Suche durch Vergleiche ausgeführt wird)

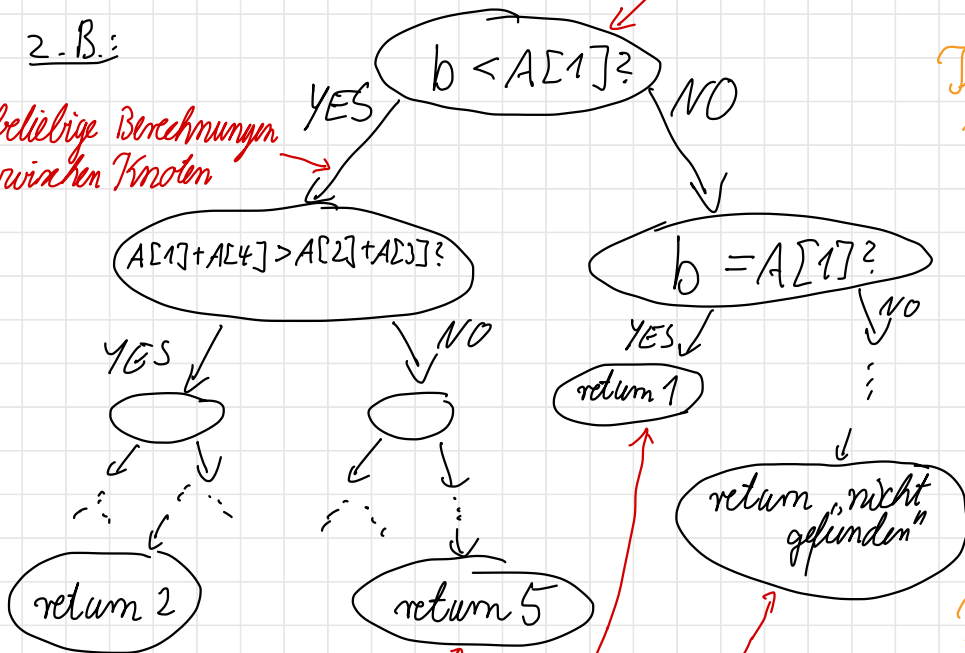
Beweis: Betrachte einen beliebigen Suchalgorithmus als Entscheidungsbaum

„informationstheoretischer Beweis“

Knoten: Vergleiche

z.B.:

beliebige Berechnungen zwischen Knoten



Tiefe 0:
1 Knoten

Tiefe 1:
 ≤ 2 Knoten

Tiefe 2:
 ≤ 4 Knoten

Tiefe k :
 $\leq 2^k$ Knoten

Alle $n+1$ möglichen Outputs müssen vorkommen.
 $\rightarrow \# \text{ Knoten} \geq n+1$

Anzahl Vergleiche im worst case = Tiefe h

Ein Baum der Tiefe h hat höchstens $1 + 2 + 4 + \dots + 2^h = 2^{h+1} - 1$
 $\leq 2^{h+1}$ Knoten

(genauere Rechnung: $\leq 2^h$ „return“-Knoten)

Daher gilt:

$$n+1 \leq \text{Anzahl Knoten} \leq 2^{h+1}, \text{ bzw. } h \geq \log_2(n+1) - 1$$

\leadsto worst-case-Laufzeit $\geq h \geq \log_2(n+1) - 1 = \Omega(\log n)$. \square

Sortieren

Suche ist viel schneller auf sortierten Daten

Wie sortieren wir Daten effizient?

Problem (Sortieren):

Input: ein Array A von n Zahlen

Output: eine Permutation (Umordnung) von A , die
aufsteigend sortiert ist:

$$A[1] \leq A[2] \leq \dots \leq A[n]$$

Elementare Operationen: - Vergleiche
- Vertauschungen

Algorithmus: Prüfe Sortiertheit

$\text{IsSorted}(A)$

for $i = 1 \dots n-1$

if $A[i] > A[i+1]$ return false

return true

Laufzeit $O(n)$

Algorithmus 1: Bubble Sort

BubbleSort(A)

for $j = 1 \dots n$

← notwendig: ein einzelner Durchgang
reicht nicht (siehe Beispiel)

for $i = 1 \dots n-1$

if $A[i] > A[i+1]$

tausche $A[i]$ und $A[i+1]$

Beispiel:

$j=1$	$j=2$	$j=3$	$j=4$
<u>5</u> 3 7 1 4	<u>3</u> 5 1 4 7	<u>3</u> 1 4 5 7	<u>1</u> 3 4 5 7
3 <u>5</u> 7 1 4	<u>3</u> 5 1 4 7	<u>1</u> 3 4 5 7	<u>1</u> 3 4 5 7
3 5 <u>7</u> 1 4	3 <u>1</u> 5 4 7	<u>1</u> 3 4 5 7	<u>1</u> 3 4 5 7
3 5 1 <u>7</u> 4	3 1 <u>4</u> 5 7	<u>1</u> 3 4 5 7	<u>1</u> 3 4 5 7
3 5 1 4 7	3 1 4 5 7	<u>1</u> 3 4 5 7	<u>1</u> 3 4 5 7

Laufzeit: $O(n^2)$ Vergleiche

$O(n^2)$ Vertauschungen

Wie können wir beweisen / einsehen, dass BubbleSort korrekt ist?

Idee: Finde geeignete Invariante und beweis sie mit Induktion.

Invariante $I(j)$: Nach j Durchgängen befinden sich die j größten Elemente am korrekten Ort

Beweis: Induktion (Übung)

Aus $I(n)$ folgt: Nach n Durchläufen sind alle n Elemente korrekt.

Von der Invariante zum Algorithmus

Betrachte dieselbe Invariante $I(j)$ wie oben.

Gegeben $I(j-1)$, können wir $I(j)$ mit weniger Aufwand erreichen?

Array A : $\underbrace{A[1 \dots n-j+1]}_{\text{schon korrekt}}$ $A[n-j+2 \dots n]$ $\leftarrow j-1$ Elemente

Idee: Finde größtes Element in $A[1 \dots n-j+1]$ und tausche es mit $A[n-j+1]$

Algorithmus 2: Selection Sort

for $j = 1 \dots n$ // Alles ab $A[n-j+2]$ schon korrekt

$k \leftarrow$ Index des Maximums in $A[1 \dots n-j+1]$ \leftarrow braucht $n-j \leq n$ Vergleiche

tausche $A[k]$ mit $A[n-j+1]$

Beispiel: 5 3 7 1 4 | I(0)

5 3 4 | 1 7 I(1)

1 3 4 | 5 7 I(2)

1 3 | 4 5 7 I(3)

1 | 3 4 5 7 I(4)

1 3 4 5 7 I(5) \Rightarrow sortiert

Laufzeit: $O(n^2)$ Vergleiche

$O(n)$ Vertauschungen \leftarrow besser als Bubblesort

Andere Invariante:

$I(j) = A[1 \dots j]$ ist sortiert (enthält aber nicht garantiert die richtigen Elemente)

Array: sortiert $A[1 \dots j]$ | $A[j+1]$ | unsortierter Teil |

Wenn $I(j)$ gilt, wie erhalten wir $I(j+1)$?

\rightarrow Setze $A[j+1]$ an korrekter Stelle in $A[1] \dots A[j]$ ein

Beispiel: 1 2 7 9 | 4 | 8 3 5 |

\rightarrow 1 2 4 7 9 | 8 3 5 |

Algorithmus 3 (Insertion Sort)

In selectionSort(A)

geht mit binärer Suche
in Zeit $O(\log(j-1))$

for $j = 2..n$ // $A[1..j-1]$ schon sortiert

Finde Stelle k , an die $A[j]$ in $A[1..j-1]$ gehört

$k \leftarrow j$, falls $A[j] = \max\{A[1], \dots, A[j]\}$

sonst $k \leftarrow$ kleinster Index k' mit $A[j] \leq A[k']$

$x \leftarrow A[j]$ // merke $A[j]$, da es gleich überschrieben wird

verschiebe $A[k..j-1]$ nach $A[k+1..j]$

$A[k] \leftarrow x$

aufwendig, braucht $j-k$ Operationen (worst case: $j-1$)

Beispiel:

5 3 7 4 1

I (1)

3 5 7 4 1

I (2)

3 5 7 4 1

I (3)

3 4 5 7 1

I (4)

1 3 4 5 7

I (5) \Rightarrow sortiert

Laufzeit: Vergleiche $\leq \sum_{j=2}^n c \cdot \log(j-1) \leq \sum_{j=2}^n c \cdot \log n \leq O(n \cdot \log n)$

Vertauschungen $\leq \sum_{j=2}^n (j-1) = O(n^2)$

Bis jetzt: Alle Algorithmen haben Laufzeit $O(n^2)$

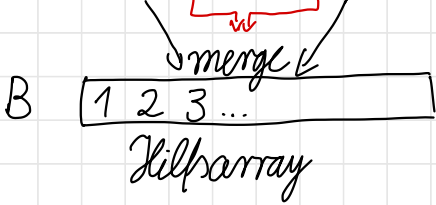
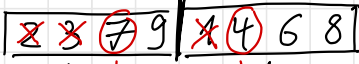
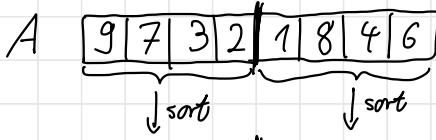
Selection Sort: $O(n)$ Vertauschungen

Insertion Sort: $O(n \log n)$ Vergleiche

Geht es besser? Können wir das Beste von beidem haben?

Algorithmus 4: Merge Sort

Idee: Divide-and-Conquer (Teile-und-Herrsche)



nächstes Element in B ist das erste (kleinste) übrige Element in einer der beiden Hälften von A

MergeSort(A, l, r) // sortiert den Bereich $A[l \dots r]$

if $l < r$

$$m \leftarrow \lfloor \frac{l+r}{2} \rfloor$$

MergeSort(A, l, m) // sortiere linke Hälfte

MergeSort(A, m+1, r) // sortiere rechte Hälfte

Merge(A, l, m, r) // verschmelze beide Hälften

Merge (A, l, m, r)

B ← new Array mit $r-l+1$ Zellen // selbe Größe wie $A[l...r]$

$i \leftarrow l$ // erstes unbenutztes Element in linker Hälfte

$j \leftarrow m+1$ // erstes unbenutztes Element in rechter Hälfte

$k \leftarrow 1$ // nächste Position in B

while $i \leq m$ and $j \leq r$ // beide Hälften noch nicht ausgeschöpft

if $A[i] < A[j]$

$B[k] \leftarrow A[i]$

$i \leftarrow i+1$

$k \leftarrow k+1$

else $B[k] \leftarrow A[j]$

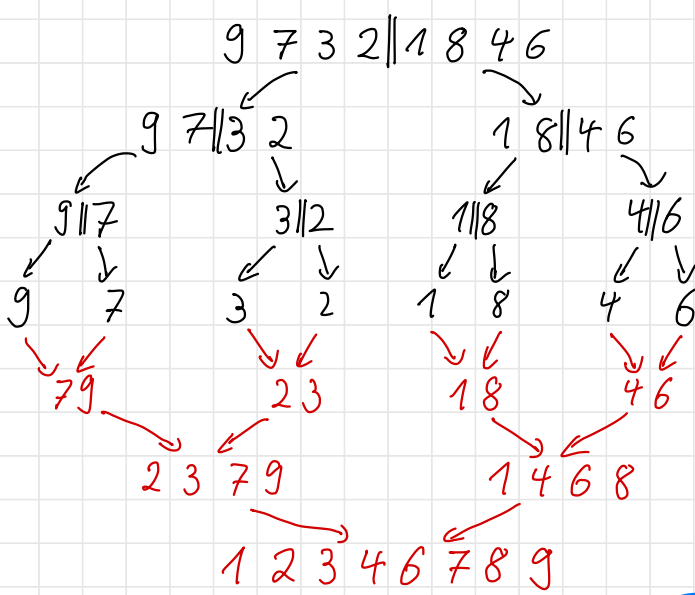
$j \leftarrow j+1$

$k \leftarrow k+1$

übernimm Rest links bzw. rechts // wenn die andere Hälfte ausgeschöpft ist

kopiere B nach $A[l...r]$

Beispiel:



Gesamtarbeit
pro Level:

- } $\leq n$ Vergleiche
- } $2n$ Kopier-Ops
- } $\leq n$ Vergleiche
- } $2n$ Kopier-Ops
- } $\leq n$ Vergleiche
- } $2n$ Kopier-Ops

jede Zahl wird
von A nach B
und anschließend
wieder zurück
kopiert

insgesamt $O(n \log n)$,
da es $O(\log n)$ Levels gibt

Alternative Bestimmung der Laufzeit:

$$T(1) \leq c, \text{ für } n = 2^k: T(n) \leq 2 \cdot T(n/2) + d \cdot n$$

Induktion $T(n) \leq d \cdot n \cdot \log_2 n + c \cdot n$ (Übung!)

$$\leq O(n \log n)$$

Beste Laufzeit bisher: $O(n \log n)$

Nachteil: braucht Zusatzarray (nicht „in place“)