

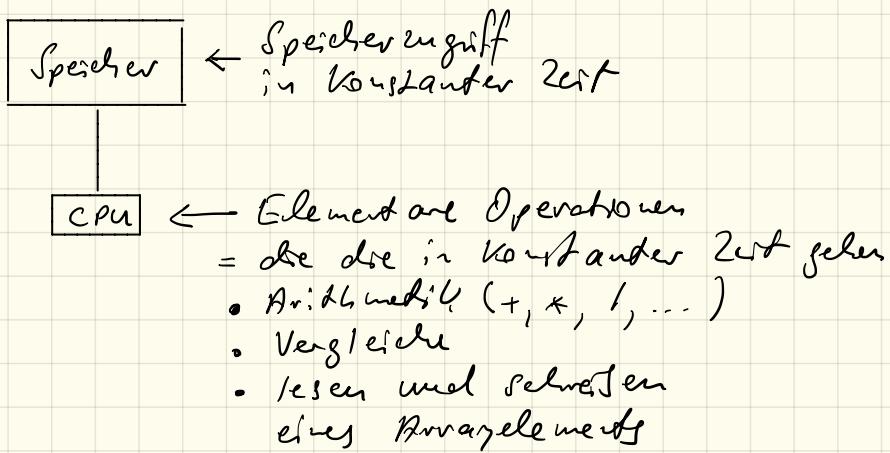
Algoritmen & Datastructuren

Herfst 2019

Vorlesing 4

Rechnermodell

Oder: Beziehung zwischen Anzahl Operationen und Laufzeit



Damit: $f(n)$ elementare Operationen

$$\Rightarrow \text{Laufzeit} \leq O(f(n))$$

$O(f(n))$ elementare Operationen

$$\Rightarrow \text{Laufzeit} \leq O(f(n))$$

Suche

Zum Beispiel: Finde einen Namen in einem Telefonbuch mit 1 Million Einträgen

Problem (Suche):

Input: Ein aufsteigend sortierter Array A :
 $A[1] \leq A[2] \leq \dots \leq A[n]$ und ein Element s

Output: k mit $A[k] = s$ oder "nicht gefunden"
falls es nicht existiert

Algorithmus 1: Binäre Suche

```

BinarySearch( $A, s$ ) //  $A$  ist sortiert
if  $A$  is empty return "nicht gefunden"
 $m = \lfloor (n+1)/2 \rfloor$  // auch  $\text{floor}(\frac{n+1}{2})$ : größte ganze
if  $s = A[m]$  return  $m$  Zahl  $\leq \frac{n+1}{2}$ 
if  $s < A[m]$ 
    BinarySearch( $A[1..m-1], s$ )
else
    BinarySearch( $A[m+1..n], s$ )

```

Laufzeit: $T(1) = c$ konstant
 $(n=2^k) \quad T(n) \leq T(n/2) + d$, d konstant

↑ wie früher schon: $n=2^k$ liefert asymptotisch das gleiche Resultat

Lösung: Teleskopieren

$$\begin{aligned} T(n) &\leq T(n/2) + c \leq T(n/4) + 2c \leq T(n/8) + 3c \leq \dots \\ &= T(n/2) + \log_2 n \cdot c \\ &= c + \log_2 n \cdot c \end{aligned}$$

(gibt Beweis mit
Induktion: Übung)

(schau wo die Konstanten
landen: sind irrelevant
für Asymptotik)

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

Der Algorithmus lässt sich auch iterativ
formulieren, ohne Rekurrenz:

Binary Search Iter(A, S) // A ist sortiert

$$\text{left} = 1$$

$$\text{right} = n$$

while left ≤ right do

$$\text{middle} = \lfloor (\text{left} + \text{right}) / 2 \rfloor$$

if A[middle] = S: return middle

if S < A[middle]: right = middle - 1

else left = middle + 1

return "nicht gefunden"

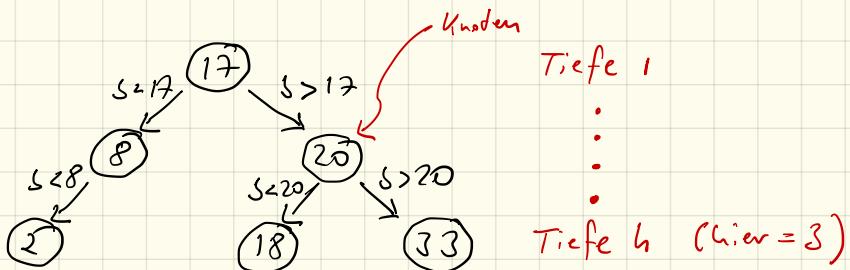
Gibt es besser als $O(\log n)$? Nein!

Idee: Schachte einer selektiver Suchalgorithmen
als Entscheidungsbaum

(Beispiel: Wir nehmen an dass die Suche durch
Vergleiche ausgeführt wird)

Beispiel:

Sinä: jeder
Knoten hat
 ≤ 2 Nachfolger



\Rightarrow Baum muss n Knoten haben ($n = \text{Länge A}$)
Anzahl Vergleiche (worst case) = Tiefe h

Also Frage: Was ist das kleinste h, das n Knoten ermöglicht?

Baum mit Tiefe h hat höchstens

$$n \leq 1 + 2 + 2^2 + \dots + 2^{h-1} = 2^h - 1 < 2^h \text{ Knoten}$$

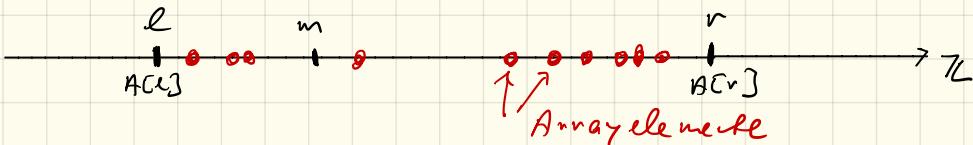
$\Rightarrow h > \log_2(n)$ Vergleiche mindestens im worst case

\Rightarrow binäre Suche ist asymptotisch optimal

Algorithmus 2: Interpolationssuche (optional)

Idee: Vergleiche s nicht mit der Tiefe, sondern schätzt den Index (Annahme: gleichmäßige Verteilung)

$$\text{also } m = \lfloor l + \frac{s - A[l]}{A[r] - A[l]} (r - l) \rfloor$$



"Gut" verteiltes Array: $O(\log \log n)$
worst case : $O(n)$

ohne Bewertung

Problem: Suche in unsortiertem Array

Algorithmus: Linear Search

LinearSearch (A, S)

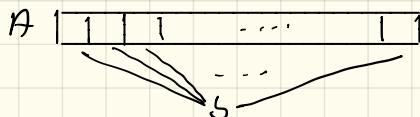
```
for i = 1..n  
  if  $A[i] = S$  return i  
return "nicht gefunden"
```

(laufzeit $O(n)$)

Gehst es besser?

(Annahme ist wieder: Suche durch Vergleiche)

Argument 1: S muss mit allen Elementen in A verglichen werden



Aber: Argument schreibt nicht Vergleiche innerhalb B !

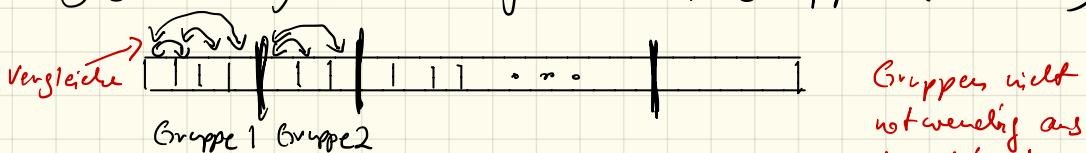
z.B. denkbar: sortiere oder teil sortiere in $O(\log n)$ und dann Suche in $O(\log n)$

Argument 2 (verdichtet):

Beachte einen selbigen, vergleichsfaenden, Suchalgorithmus.

$$n = \text{Anzahl Vergleiche innerhalb } A$$
$$s = \text{, } \quad \text{mit } b$$

Die n Vergleiche zerlegen A in Gruppen (Paartheit)



- Gruppe:
- Elemente sind durch Kette von Vergleichen verbinden
 - Kein Vergleich zwischen Elementen verschiedener Gruppen
- Elementen

Am Anfang: n Gruppen

Zusammenlegen zweier Gruppen: ≥ 1 Vergleich

g Gruppen $\Rightarrow n \geq n-g$ Vergleiche

Suche braucht ≥ 1 Vergleiche pro Gruppe:

$$s \geq g$$

$$\Rightarrow n+s \geq n$$

II

Aber: lineare Suche ist asymptotisch optimal
(wegen vergleichsfaulent)

Sortieren

Suche ist viel schneller auf sortierten Daten
(anwaltigst sortierter Aufwand)

Zustalter von Big Data: Suche ist essentiell

Problem (Sortieren):

Input: ein Array A der Länge n

Output: eine Permutation A' von A die aufsteigend sortiert ist

$$i < j \Rightarrow A'[i] \leq A'[j], \quad 1 \leq i, j \leq n$$

Oft wird A' "in-place" berechnet, d.h.
in A als A (kein Extraspeicher)

Algorithmus: Prüfe Sortiertheit

isSorted (A)

```
for i = 1..n-1
    if  $A[i] > A[i+1]$  return false
return true
```

Laufzeit $O(n)$

Sortieralgorithmen: Elementare Operationen
Vergleiche, Veransetzungen, etc.

Algorithmus 1: Bubble Sort

Idee: mehrfizire Prüfalgorithmus

```

for i = 1 .. n-1
  if A[i] > A[i+1]
    Tausche A[i] und A[i+1]
  
```

reicht nicht! (z.B. 4, 5, 3 \rightarrow 4, 3, 5)

also: mehrere Tausch durchgänge, wieviel?

Belaupfung: nach $n-1$ ist Array sortiert

Rechtfertigung:

1. Durchgang: größtes Element ganz rechts
2. Durchgang: zweitgrößtes an korrekter Stelle

etc.

Bubble Sort (K)

```

for j = 1 .. n-1
  for i = 1 .. n-1
    if A[i] > A[i+1]
      Tausche A[i] und A[i+1]
  
```

Verbesserung: lasse nur von $i=j$ laufen

Beispiel:

$j=1$	$j=2$	$j=3$	$j=4 = \text{nichts}$
3 7 5 1 4	3 1 5 4 7	1 3 4 5 7	
3 5 7 1 4	3 1 4 5 7		
3 5 1 7 4		$j=3$	
3 5 1 4 7	1 3 4 5 7		

Verarbeitung: wenn sich in einem Durchgang nichts ändert, dann fertig

Laufzeit: $O(n^2)$ Vergleiche
 $O(n^2)$ Verdauschen
 $\Rightarrow O(n^2)$

Was ist der schlechteste Fall?

Algorithmus 2: Selektiver Sort

Idee: induktiv (Sau Lösung vor links nach rechts)

Ziel: 1 sortierter Teil | A[0:j] unsortierter Rest |
(Array A) und alle Elemente am richtigen Platz (INV(i)) das nennt sich Invariante, siehe später

Wie erhalten wir diesen Zustand wenn $i \rightarrow i+1$?

Selektiver Sort (A)

für $i = 1..n-1$
 $j = \text{Index des Minimums in } A[i..n]$
tausche $A[i,j]$ und $A[j,i]$

Beispiel: Auftrag 3 7 5 1 4
 $i=1:$ 1 1 7 5 3 4
 $i=2:$ 1 3 1 5 7 4
 $i=3:$ 1 3 4 1 7 5
 $i=4:$ 1 3 4 5 1 7 fertig

Laufzeit:

Konstanz: $\forall i \in \{1, \dots, n\}$ aus $n-i$ Elementen: $n-i$

insgesamt: $\sum_{i=1}^{n-1} n - (i-1) = n + n - 1 + \dots + 2 \leq O(n^2)$

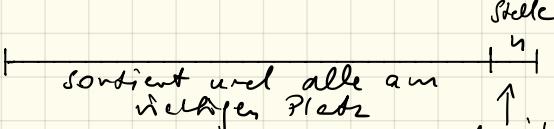
Tauschoperationen: $O(n)$ weniger als Bubble sort
 $\Rightarrow O(n^2)$

Konsistenz: Neu hier war dass wir den Algorithmus von der Invariante $INV(i)$ abgelenkt haben. $INV(i)$ ist eine Aussage die von i abhängt:

$INV(i) = A[1..i-1]$ sind sortiert und am richtigen Platz

Sie heißt "Invariante" weil sie in jedem Schleifenabsatz gilt, d.h., für alle i .

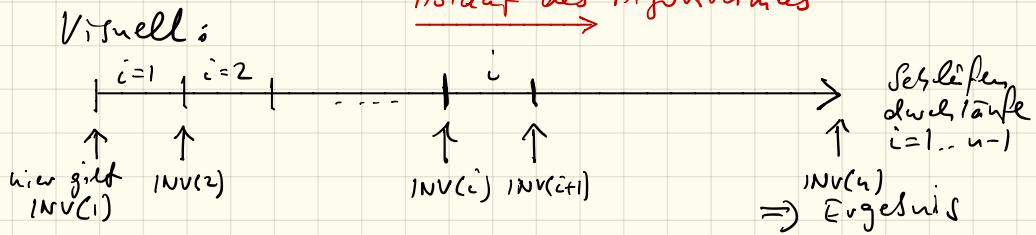
Damit gilt am Ende:

$INV(n)$:  \Rightarrow sortiert
denn ist auch dieses hier am richtigen Platz

Idee Invariante:

- 1.) Gilt am Anfang
- 2.) Konserviert in jedem Schritt
- 3.) Ende + Invariante \Rightarrow korrektes Ergebnis

Ablauf des Algorithms



Ermöglicht Korrektheitsbeweis durch Induktion über Schleifenvariable i : (Skizze)

Ind. anfang $INV(1) \checkmark$ Selectiven Sort (A)

neues i ist $\rightarrow INV(i)$ für $i = 1..n-1$ $j = \text{Index des Minimums in } A[i..n]$

$i+1 \quad \downarrow \text{Ind. Schritt}$ tausche $A[i]$ und $A[j]$

Es folgt: am Ende gilt $INV(n) \Rightarrow$ Ergebnis.

Algorithmus 3: Insertion Sort

Idee: wieder induktiv, aber andere Invariante

Array: (sortierter Teil $[A[i]]$ unsortierter Teil $)$

genauer:
 $A[0]..A[i-1]$ sortiert

Wie erhält man INV ? schre $A[i]$ an nächster Stelle ein

Beispiel: 1 1 2 \rightarrow 3 | 4 | Rest |

\rightarrow 1 1 2 4 \rightarrow 3 | 1 Rest |

InsertionsSort(A)

hier verwenden wir dass Binary Search
als Nebeneffekt die richtige Stelle findet
wenn "nicht gefunden"

for $i = 2..n$

suche binär nach $A[i]$ in $A[1..i-1]$ \rightarrow Stelle k

$x = A[i]$ // merke $A[i]$ da gleich überschrieben

verschiebe $A[k+1..i]$ nach $A[k+1..i]$

$A[k] = x$

Beispiel: Anfang

$i=2:$ 3 | 7 5 4 1

$i=3:$ 3 7 | 5 4 1

$i=4:$ 3 5 7 | 4 1

$i=5:$ 3 4 5 7 | 1

$i=5:$ 1 3 4 5 7 |

Laufzeit:

$$\text{Vergleiche} \leq \sum_{i=2}^n a \log(i) = a \log(n!) \leq O(n \log n)$$

$$[\text{Berechte: } \left(\frac{n}{2}\right)^{n/2} \leq n! \leq n^n]$$

$$\text{Tauschops} = \sum_{i=2}^n (i-n) \leq \sum_{i=2}^n (i-1) \leq O(n^2)$$

im Worst case ist
K immer 1

Es ist jetzt: alle Algorithmen sind $O(n^2)$

SelectionSort: $O(n)$ Tauschops

InsertionsSort: $O(n \log n)$ Vergleiche

Können wir das Beste von beiden haben?