

Algorithmen & Datenstrukturen

Herbst 2020

Vorlesung 4

Suche

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

Problem (Suche):

Input: Ein aufsteigend sortiertes 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 Zahl  $\leq \frac{n+1}{2}$ 
if  $s = A[m]$  return m
if  $s < A[m]$ 
    BinarySearch(A[1..m-1], s)
else
    BinarySearch(A[m+1..n], s)
```

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

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

Lösung: Teleskopieren

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

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

(geht Beweis mit
Induktion: Übung)

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

Der Algorithmus lässt sich auch iterativ
formulieren, ohne Rekursion:

Binary Search $It(A, S)$ // A ist sortiert

left = 1

right = n

while left ≤ right do

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

 if $A[middle] = S$: return middle

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

 else left = middle + 1

return "nicht gefunden"

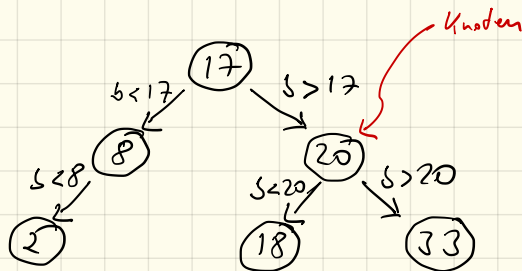
Geht es besser als $O(\log n)$? Nein!

Idee: Schreibe einen beliebigen Suchalgorithmus
als Entscheidungsbaum

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

Beispiel:

Binär: jeder
Knoten hat
 ≤ 2 Nachfolger



Tiefe 1

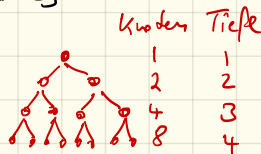
...

Tiefe h (hier = 3)

$\Rightarrow$ 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



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

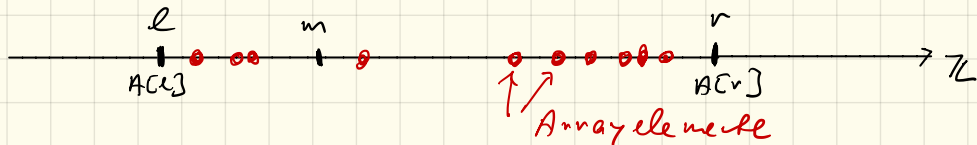
$\Rightarrow h > \log_2(n)$ Vergleiche mindestens im worst case

$\Rightarrow$ Binäre Suche ist asymptotisch optimal

Algorithmus 2: Interpolationsuche (optional)

Idee: Vergleiche b nicht mit der Zielf, sondern
Schätze den Index (Annahme: gleichmäßige
Verteilung)

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



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

ohne Beweis

Problem: Suche in unsortiertem Array

Algorithmus: Linear Search

LinearSearch (A, S)

for $i = 1 \dots n$

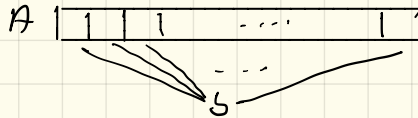
if $A[i] = S$ return i
return "nicht gefunden"

Laufzeit $O(n)$

Gibt es besser?

(Annahme ist wieder: Suche
durch Vergleiche)

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



Aber: Argument betrachtet nicht Vergleiche
innerhalb A !

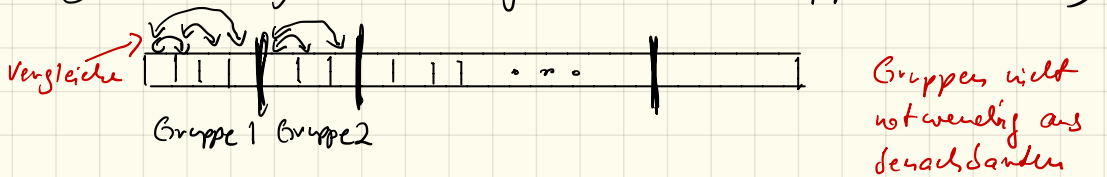
z.B. denkbar: sortiere oder teilsortiere
in $O(\log n)$ und dann Suche in $O(\log n)$

Argument 2 (verbessert):

Betrachte einen beliebigen, vergleichsfähigen, Suchalgorithmus.

n = Anzahl Vergleiche innerhalb A
 s = " " " " " " mit G

Die n Vergleiche zerlegen A in Gruppen (Partition)



- Gruppe:
- Elemente sind durch Ketten von Vergleichen verbunden
 - Kein Vergleich zwischen Elementen verschiedener Gruppen

Am Anfang: n Gruppen

Zusammenlegen zweier Gruppen: ≥ 1 Vergleich

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

Suche braucht ≥ 1 Vergleich pro Gruppe:

$$s \geq g$$

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

□

Also: lineare Suche ist asymptotisch optimal
(wenn vergleichsfähig)

Sortieren

Suche ist viel schneller auf sortierten Daten
(annotiert Sortieraufwand)

Zeitalter von Big Data: Suche ist essentiell

Problem (Sortieren):

Input: ein Array A der Länge n

Output: eine Permutation A' von A die
(Umordnung)
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.
innerhalb A (kein Extraspeicher)

Algorithmus: Prüfe Sortiertheit

isSorted(A)

for $i = 1 \dots n-1$

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

return true

Laufzeit $O(n)$

Sortieralgorithmus: Elementare Operationen
Vergleiche, Vertauschungen, etc.

Algorithmus 1: Bubble Sort

Idee: modifizierte Prüfalgorithmus

for $i = 1 \dots n-1$

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

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

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

also: mehrere Tauschdurchgänge, wie viele?

Behauptung: nach $n-1$ ist Array sortiert

Begründung: 1. Durchgang: größtes Element ganz rechts

2. Durchgang: zweitgrößtes an korrekter Stelle etc.

Bubble Sort (A)

for $j = 1 \dots n-1$

for $i = 1 \dots n-1$

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

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

Verbesserung: lasse nur bis $n-j$ laufen

Beispiel:

$j=1$

3 7 5 1 4

3 5 7 1 4

3 5 1 7 4

3 5 1 4 7

$j=2$

3 1 5 4 7

3 1 4 5 7

$j=3$

1 3 4 5 7

$j=4$ = nicht

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

Laufzeit: $O(n^2)$ Vergleiche
 $O(n^2)$ Vertauschungen

$\Rightarrow O(n^2)$

Was ist der schlimmste Fall?

Algorithmus 2: Selection Sort

Idee: induktiv (saure Lösung von links nach rechts)

Bild: 1 sortierter Teil $A[i]$ unsortierter Rest i
(Annahme A) und alle Elemente am richtigen Platz INV(i) Das nennt sich Invariante, siehe später

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

Selection Sort (A)

für $i = 1 \dots n-1$

j = Index des Minimums in $A[i \dots n]$
tausche $A[i]$ und $A[j]$

Beispiel: Anfang 3 7 5 1 4

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

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

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

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

fertig

Laufzeit:

Vergleichen: n aus k Elementen: k Vergleiche

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

Tauschoperationen: $O(n)$ weniger als
Bubblesort

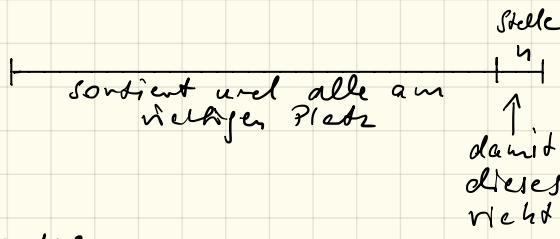
$\Rightarrow O(n^2)$

Korrektheit: Neu hier war dass wir den Algorithmus von der Invariante $INV(i)$ abgeleitet 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 Schritt durchlauf gilt, d.h., für alle i .

Damit gilt am Ende:

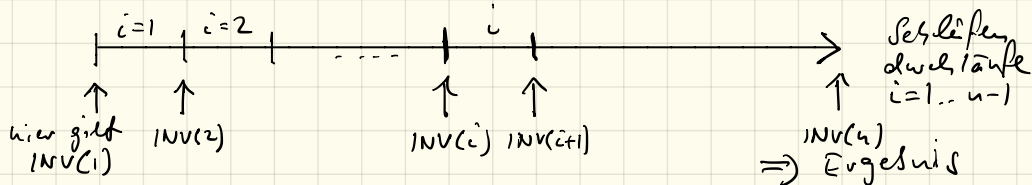
$INV(n)$:  $\Rightarrow$ sortiert

Idee Invariante:

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

Ablauf des Algorithmus

Visuell:



Ermöglicht Korrektheitsbeweis durch Induktion über Schleifenvariable i : (Skizze)

Ind. aufang $INV(1) \checkmark$ $\rightarrow$ Selection Sort (A)
for $i = 1..n-1$
 $j = \text{Index des Minimums in } A[i..n]$
 tausche $A[i]$ und $A[j]$
 $\downarrow$ Ind. schritt
 $INV(i+1)$
neues i ist $i+1$

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

Algorithmus 3: Insertion Sort

Idee: wieder induktiv, aber andere Invariante

Array: [sortierter Teil $A[i]$ | unsortierter Teil]
genauer:
 $A[1]..A[i-1]$ sortiert

Wie erhält man INV wenn $i \rightarrow i+1$?
 $\swarrow$ setze $A[i]$ an richtiger Stelle ein

Beispiel: 1 1 2 7 3 | 4 | Rest |

$\rightarrow$ 1 1 2 4 7 3 | Rest |

Insertionsort (A)

for $i = 2 \dots n$

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

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

verschiebe $A[k+1 \dots i]$ nach $A[k+1 \dots i]$

$A[k] = x$

hier verwenden wir das Binary Search
als Neben-effekt die richtige Stelle findet
wenn "nicht gefunden"

Beispiel: Anfang

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

Laufzeit:

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

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

im worst case ist k immer 1

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

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

Selectionsort: $O(n)$ Tauschops

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

Können wir das Beste von beiden haben?