Algorithmen & Datenstrukturen

Herbst 2021

Vorlesung 4

Suchen & Sortieren

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 b

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

Algorithmus 1: Binäre Suche

```
BinarySearch (A, b)   // A ist sortiert
    if A is empty return "nicht gefunden"
    m = ⌊(n+1)/2⌋     // auch floor((n+1)/2): grösste ganze Zahl ≤ (n+1)/2
    if b = A[m] return m
    if b < A[m]
        BinarySearch (A[1..m-1], b)
    else
        BinarySearch (A[m+1..n], b)
```

Laufzeit: $T(1) = c$ konstant

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

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

Lösung: Teleskopieren

$$T(n) \leq T(n/2) + d \leq T(n/4) + 2d \leq T(n/8) + 3d \leq \dots$$
$$= T(n/n) + \log_2 n \cdot d$$
$$= c + \log_2 n \cdot d$$

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

(Geht Beweis mit Induktion: üsung)

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

Der Algorithmus lässt sich auch iterativ formulieren, ohne Rekursion:

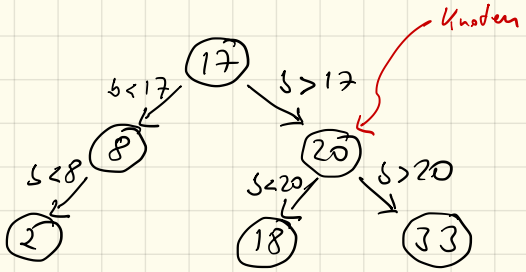
```
Binary Search It (A, s)        // A ist sortiert
    left = 1
    right = n
    while left ≤ right do
        middle = ⌊(left + right)/2⌋
        if A[middle] = s: return middle
        if s < A[middle]: right = middle - 1
        else left = middle + 1
    return "nicht gefunden"
```

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

Idee: Schraube einen beliebigen Suchalgorithmus als Entscheidungsbaum

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

Beispiel:



Tiefe 1

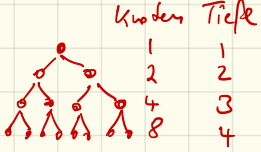
⋮

Tiefe h (hier = 3)

Sinn: jeder Knoten hat ≤ 2 Nachfolger

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

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



Knoten	Tiefe
1	1
2	2
4	3
8	4

Baum mit Tiefe h hat höchstens

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

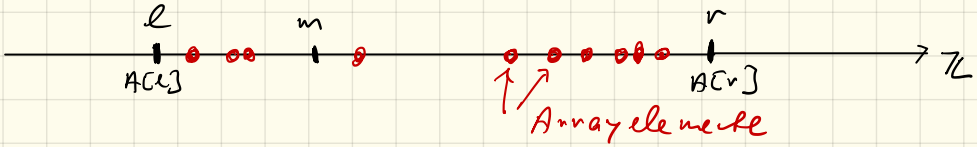
$\Rightarrow$ $h > \log_2(n)$ Vergleiche mindestens im worst case

$\Rightarrow$ binäre Suche ist asymptotisch optimal

Algorithmus 2: Interpolationssuche (optional)

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

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



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

ohne Beweis

Problem: Suche in unsortiertem Array

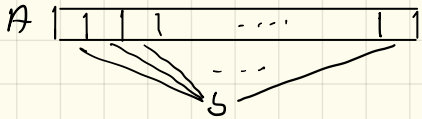
Algorithmus: Linear Search

```
LinearSearch (A, S)
  for i = 1..n
    if A[i] = S return i
  return "nicht gefunden"
```

Laufzeit $O(n)$

Geht es besser? (Annahme ist wieder: Suche durch Vergleiche)

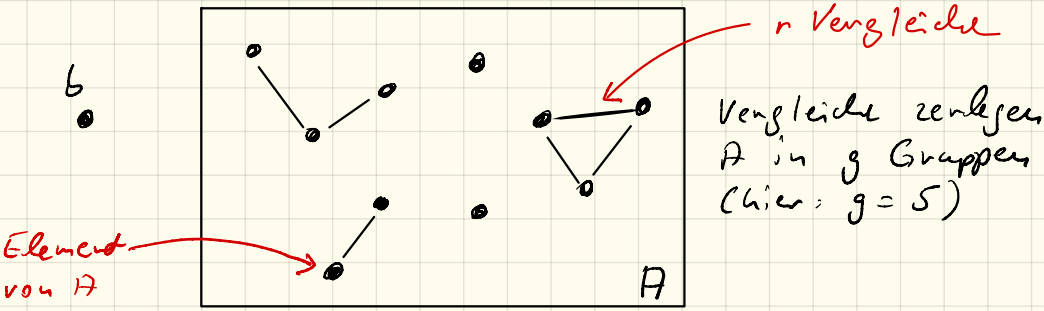
Argument 1: S muss mit allen Elementen in A verglichen werden



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 (verfeinert):
Betrachte einen beliebigen, vergleichsbasierten, Suchalgorithmus.

n = Anzahl Vergleiche innerhalb A
s = " mit b



Vergleiche zerlegen A in g Gruppen (hier: $g = 5$)

Es braucht mindestens $s \geq g$ Vergleiche mit b (einen pro Gruppe)

Wie klein bekomme ich g mit n Vergleichen?

r	0	1	2	3	...	n
g	n	$n-1$	$\geq n-2$	$\geq n-3$	...	$\geq n-n$

1 Vergleich kann höchstens 2 Gruppen zusammenlegen

$\Rightarrow$ mindestens $n + s \geq n + g \geq n + n - n = n$ Vergleiche

Also: lineare Suche ist asymptotisch optimal (wenn vergleichsbasiert)

Sortieren

Suche ist viel schneller auf sortierten Daten (amortisiert Sortieraufwand)

Zeitalter von Big Data: Suche ist essentiell

Problem (Sortieren):

Input: ein Array A der Länge n

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

Elementare Operationen:
Vergleiche, Vertauschungen

Algorithmus: Prüfe Sortiertheit

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

Laufzeit $O(n)$

Algorithmus 1: Bubble Sort

Idee: modifiziere 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 → 4, 3, 5)

also: mehrere Tauschdurchgänge, wieviele?

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..n-1                      ← Verbesserung: lasse nur bis n-j laufen
        for i = 1..n-1
            if A[i] > A[i+1]
                tausche A[i] und A[i+1]
```

Beispiel:

j=1					j=2					
3	7	5	1	4	3	1	5	4	7	j=4: nichts
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	

Verbesserung: 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 (same Lösung von links nach rechts)

Bild: (Array A) | sortierter Teil | A[i] | unsortierter Rest |
und alle Elemente am richtigen Platz

INV(i) Das nennt sich Invariante, siehe später

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

```
Selection Sort (A)
    for i = 1..n-1
        j = Index des Minimums in A[i..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:

Minimum: Min 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

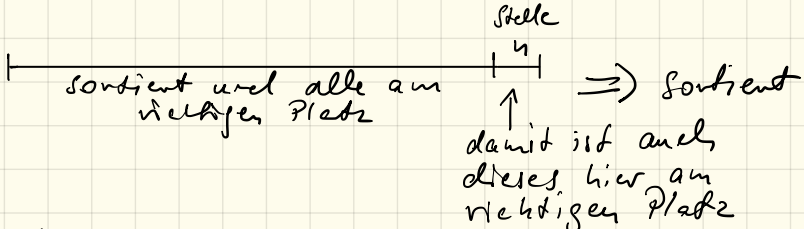
$\Rightarrow O(n^2)$

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

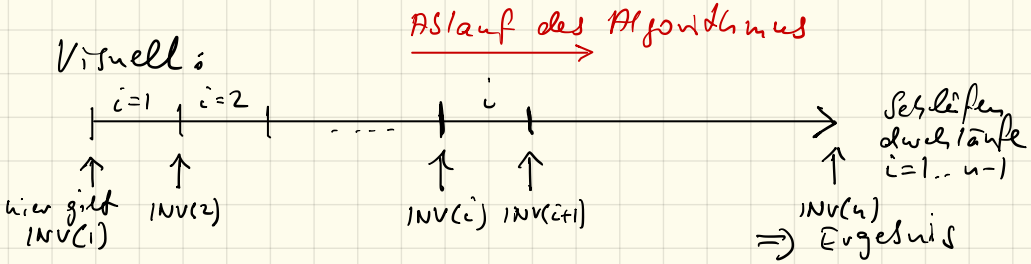
INV(i) = A[1..i-1] sind sortiert und am richtigen Platz

Sie heisst "Invariante" weil sie in jedem Schleifendurchlauf gilt, d.h., für alle i.

Damit gilt am Ende:

INV(n): 

Idee Invariante:
1.) Gilt am Anfang (i=1)
2.) Konserviert in jedem Schritt $i \to i+1$
3.) Ende + Invariante $\Rightarrow$ korrektes Ergebnis



Ermöglicht Korrektheitsbeweis durch Induktion über Schleifenvariable i: (Skizze)

Ind. anfang INV(1) ✓ → Selection Sort (A)
For i = 1..n-1
neues i ist i+1 ← Ind. Schritt
INV(i) → INV(i+1)
j = Index des Minimums in A[i..n]
tausche A[i] und A[j]

Es folgt: am Ende gilt INV(n) ⇒ 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 → i+1 ?

Setze A[i] an richtiger Stelle ein ⊗

Beispiel: | 1 2 7 9 | 4 | Rest |

→ | 1 2 4 7 9 | Rest |

Insertion Sort (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 $A[1..i-1]$ → Stelle k

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

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

$A[k] = x$

Beispiel:

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

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

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

im worst case ist k immer 1

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

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

Selection Sort: $O(n)$ Tauschops

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

Können wir das beste von beiden haben?