Algorithmen & Datenstrukturen

Herbst 2018

Vorlesung 6

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/2⌋          // auch floor(n/2): grösste ganze Zahl ≤ n/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

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

↑ wie letztes Mal: $n \neq 2^k$ liefert asymptotisch das gleiche Resultat

Lösung: Teleskopieren

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

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

(geht Beweis mit Induktion)

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

Der Algorithmus lässt sich auch iterativ formulieren, ohne Rekursion:

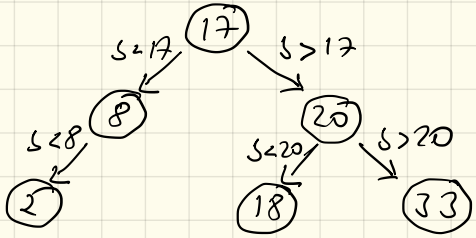
```
Binary Search It (A, S)        // A ist sortiert
    left = 1
    right = n
    while left ≤ right do
        middle = ⌊(left + right)/2⌋
        if A[middle] = S: return middle
        if S < A[middle]: right = middle - 1
        else left = middle + 1
    return "nicht gefunden"
```

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

Nein! Idee: Schraube Suche als Entscheidungsbaum

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

Beispiel:



Tiefe 0

Tiefe h (hier = 2)

$\Rightarrow$ Baum muss n Knoten haben (n = Länge A)
Laufzeit = Tiefe h

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

Baum mit Tiefe h hat

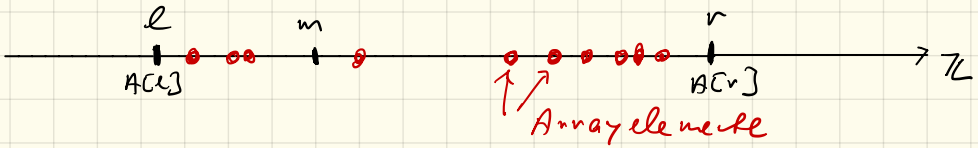
$$n \leq 1 + 2 + 2^2 + \dots + 2^h = 2^{h+1} - 1 \quad \text{Knoten}$$

$$\Rightarrow h \geq \log_2(n+1) - 1 \quad \text{d.h.} \quad h \geq \Omega(\log n) \qquad \square$$

Algorithmus 2: Interpolationssuche (optional)

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

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



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

ohne Beweis

Problem: Suche in unsortiertem Array

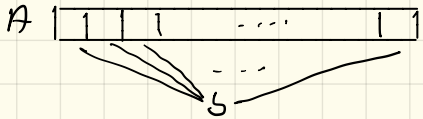
Algorithmus: Linear Search

```
LinearSearch (A, s)
    for i = 1..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



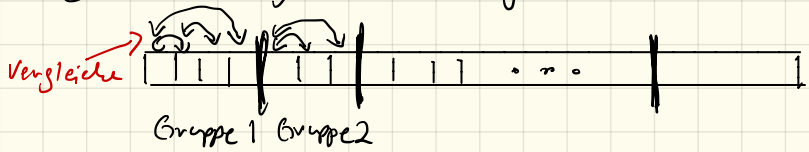
Aber: Argument betrachtet nicht Vergleiche innerhalb A!
z.B. denkbar: sortiere oder teilsortiere in $O(\log n)$ und dann Suche in $O(\log n)$

Argument 2 (verfeinert):

r = Anzahl Vergleiche innerhalb A

s = " mit b

Die r Vergleiche zerlegen A in Gruppen (Partition)



Gruppe: • Elemente sind durch Kette von Vergleichen verbunden
• Kein Vergleich zwischen Elementen verschiedener Gruppen

Am Anfang: n Gruppen

Zusammenlegen zweier Gruppen: ≥ 1 Vergleich

$$g \text{ Gruppen} \Rightarrow r \geq n - g \text{ Vergleiche}$$

Suche braucht ≥ 1 Vergleich pro Gruppe:

$$s \geq g$$

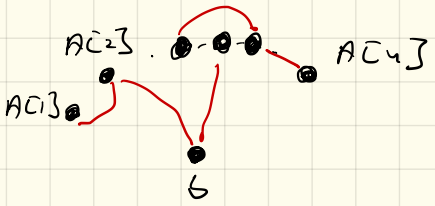
$$\Rightarrow r + s \geq n \geq \Omega(n) \qquad \square$$

War nicht in der Vorlesung:

Alternative Beweisidee:

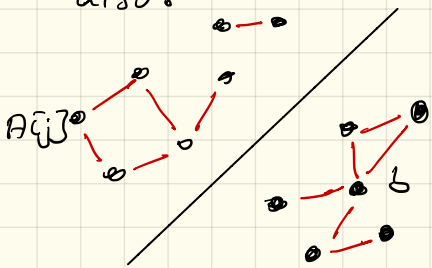
Nimm beliebigen, vergleichsbasierten Algorithmus. Nimm an er braucht k Vergleiche im schlimmsten Fall.

Baue Graph:



$n+1$ Knoten:
$A[1], \dots, A[n], b$
k Kanten für jeden Vergleich

Annahme: Graph ist nicht zusammenhängend, also:



$\Rightarrow$ Falls $b = A[j]$ kann es nicht gefunden werden

$\Rightarrow$ Algorithmus inkorrekt

Also: Graph ist zusammenhängend
$\Rightarrow k \geq (n+1) - 1 = n$ also $k \geq \Omega(n)$

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 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)

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, Vertauschungen, etc.

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 (baue Lösung von links nach rechts)

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

(INV(i)) Das nennt sich Invariante, siehe später

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

```
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 $n-i$ Elementen: $n-i$

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

Tauschoperationen: $O(n)$ weniger als Bubblesort

$\Rightarrow O(n^2)$

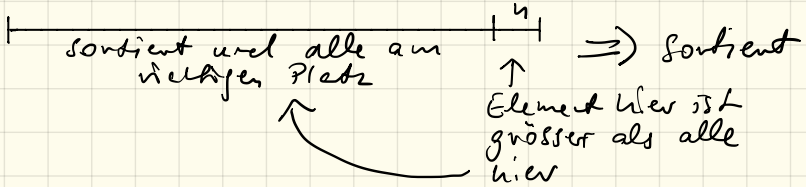
Korrektheit: Neu hier war dass wir den Algorithmus von der Invariante INV(i) abgeleitet haben. INV(i) ist eine Aussage die von i abhängt:

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

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

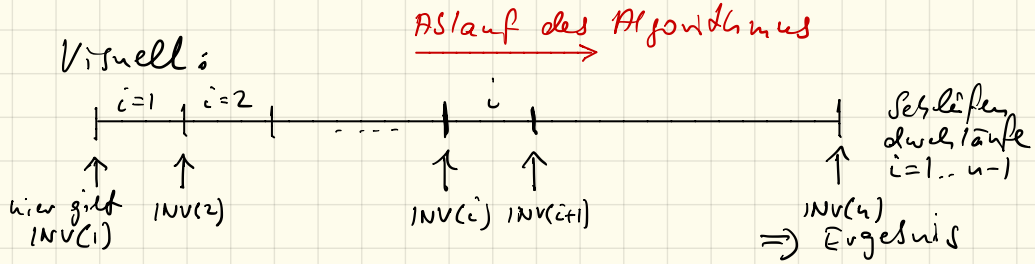
Damit gilt am Ende:

INV(n):



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

Visuell:



Ermöglicht Korrektheitsbeweis durch Induktion über Schleifenvariable i: (Skizze, $n \geq 2$)

Ind. anfang $INV(1)$ ✓ → Selection Sort (A)

For $i = 1..n-1$

$INV(i)$ → j = Index des Minimums in $A[i..n]$

↓ Ind. Schritt

neues i ist $i+1$ → $INV(i+1)$ 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 | i | unsortierter Teil |

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

Wie erhält man INV? ← setze 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

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

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

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

Können wir das beste von beiden haben?