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):

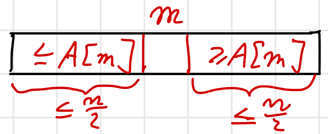
```
LinearSearch (A, b)     // A = A[1... n] Array

    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, $A[1] \leq A[2] \leq \ldots \leq A[n]$

Algorithmus 2: Binäre Suche




```
BinarySearch(A, b)   // für sortiertes A

  l ← 1, r ← n       // A[l...r] gibt den Bereich an,
                     in dem wir noch suchen müssen

  while l ≤ r do

       m ← ⌊(l+r)/2⌋   // ⌊...⌋, auch floor(...) geschrieben: abrunden

       if b = A[m]: return m

       if b < A[m]: r ← m-1

                else: l ← m+1

  return „nicht gefunden"
```

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

$T(n) \leq T(n/2) + d$, 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)

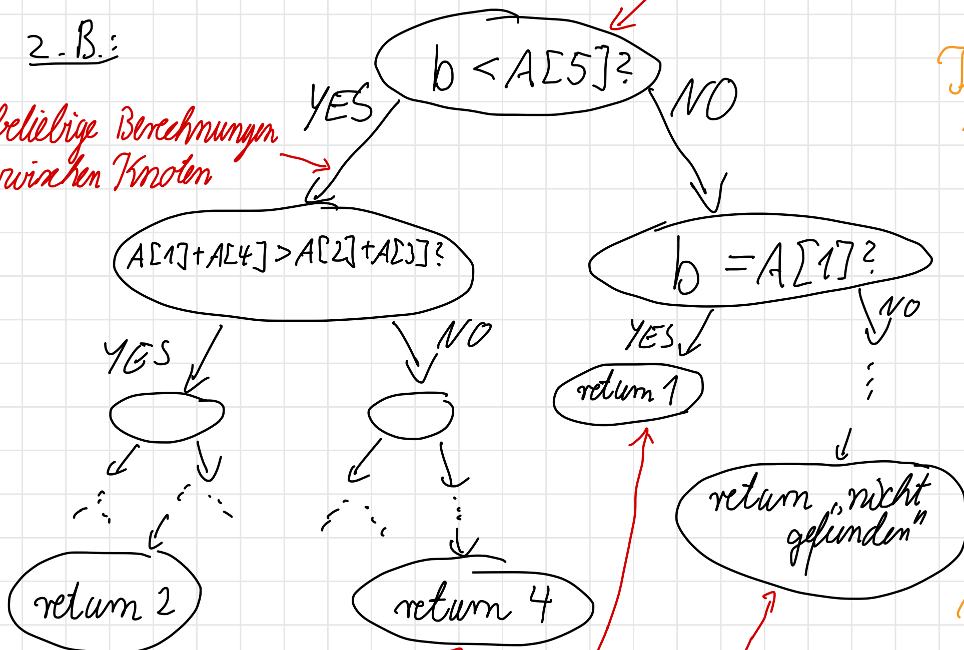
$\rightsquigarrow T(n) \leq O(\log n)$

Geht es besser? NEIN!

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

<u>Beweis</u>: Betrachte einen beliebigen Suchalgorithmus als Entscheidungsbaum

„informationstheoretischer Beweis"



Anzahl Vergleiche im worst case = Tiefe h

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

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

Daher gilt:

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

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

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:

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

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

```
BubbleSort(A)
  for j = 1...n
    for i = 1...n-1
      if A[i] > A[i+1]
        tausche A[i] und A[i+1]
```

notwendig: ein einzelner Durchgang reicht nicht (siehe Beispiel)

Beispiel:

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

Laufzeit: $\Theta(n^2)$ Vergleiche

$O(n^2)$ Vertauschungen

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

__Idee__: Finde geeignete __Invariante__ und beweise 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: $A[1 \dots n-j+1]$ | $A[n-j+2 \dots n]$ (schon korrekt) ← $j-1$ 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...n   // Alles ab A[n-j+2] schon korrekt
    k ← Index des Maximums in A[1...n-j+1]
    tausche A[k] mit A[n-j+1]
```

braucht $n-j \leq n$ Vergleiche (für den Schritt „Index des Maximums in $A[1 \dots n-j+1]$“)

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 $I(5)$ ⇒ sortiert

Laufzeit: $O(n^2)$ Vergleiche

$O(n)$ Vertauschungen ← besser als Bubblesort

Andere Invariante:

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

Array: | sortiert ($A[1-j]$) | $A[j+1]$ | unsortierter Teil |

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

→ Setze $A[j+1]$ an korrekten Stelle in $A[1], \dots, A[j]$ ein

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

→ | 1 2 4 7 9 | 8 3 5 |

Algorithmus 3 (Insertion Sort)

InsertionSort(A)

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

Finde Stelle k, an die $A[j]$ in $A[1..j-1]$ gehört ← geht mit binärer Suche in Zeit $O(\log(j-1))$

$k \leftarrow j$, falls $A[j] = \max\{A[1],...,A[j]\}$
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) ⟹ sortiert

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

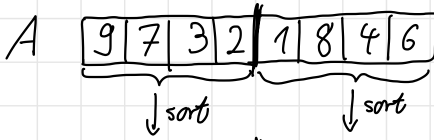
Vertauschungen $\leq \sum_{j=2}^{n} (j-1) \leq O(n^2)$

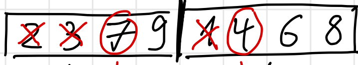
Bis jetzt: Alle Algorithmen haben Laufzeit $O(n^2)$
Selection Sort: $O(n)$ Vertauschungen
Insertion Sort: $O(n \log n)$ Vergleiche

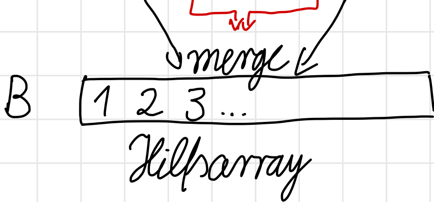
Geht es besser? Können wir das beste von beidem haben?

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 ... r]

if $\ell < r$

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

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

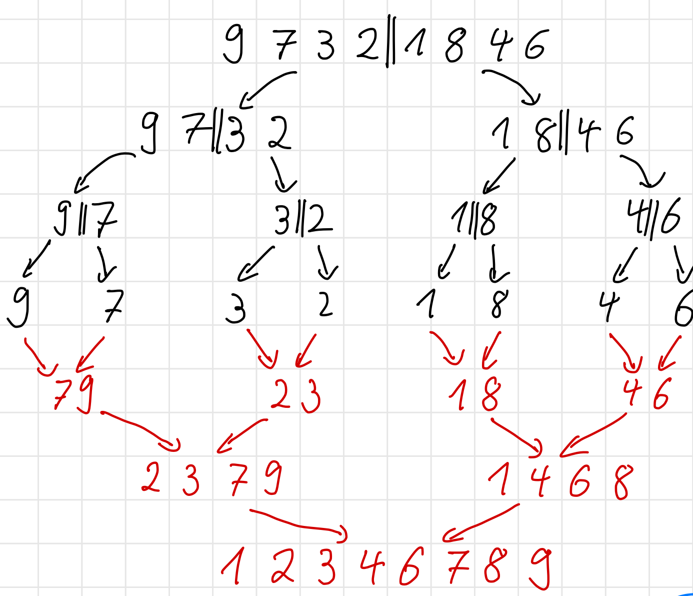
MergeSort(A, m+1, r) // sortiere rechte Hälfte

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 ← ℓ   // erstes unbenutztes Element in linker Hälfte
j ← m+1   // erstes unbenutztes Element in rechter Hälfte
k ← 1   // nächste Position in B
while i ≤ m and j ≤ r   // beide Hälften noch nicht ausgeschöpft
    if A[i] < A[j]
        B[k] ← A[i]
        i ← i+1
        k ← k+1
    else
        B[k] ← A[j]
        j ← j+1
        k ← 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:

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

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

$\leq O(n \log n)$

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

Nachteil: braucht Zusatzarray (nicht „in place")