Algorithmen & Datenstrukturen

Herbst 2022

Vorlesung 5

Sortieren, Teil 2

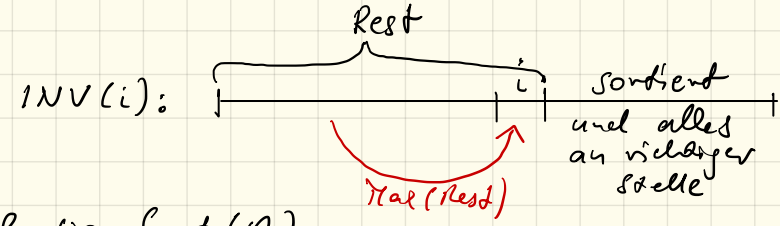
Problem: Sortieren eines Arrays $A[1], \ldots, A[n]$

Algorithmen bisher: ↑ keys (Schlüssel)

	Vergleiche	Bewegungen
Bubble Sort	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n)$
Insertion Sort	$O(n \log n)$	$O(n^2)$

Alle diese sind "inplace", d.h. brauchen keinen Extraspeicher: Resultat ist in A.

Selection Sort noch einmal, diesmal von rechts nach links:



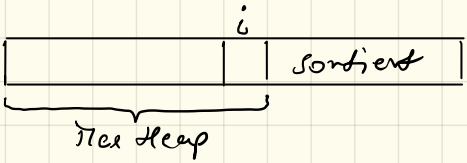
SelectionSort(A)
  for $i = n \ldots 2$
    j = Index des Maximum in $A[1..i]$
    tausche $A[i]$ und $A[j]$

das kostete $O(n^2)$ da jedes Max $O(i)$ kostet

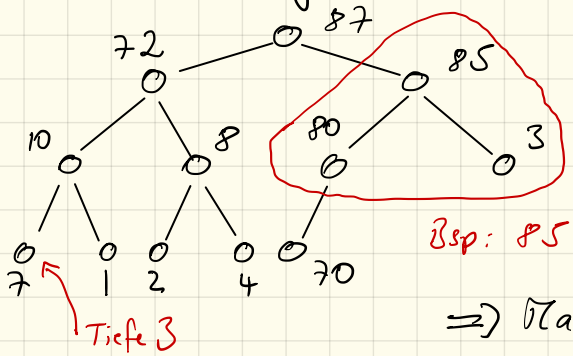
Gesucht: Datenstruktur die das Finden des Max billiger macht.

Hoffnung: Max in $O(\log i) \Rightarrow$ Algorithmus $O(n \log n)$!
$\left(\sum_{i=2}^{n} a \cdot \log i \in O(n \log n)\right)$

Lösung: Max-Heap



Visualisierung als Baum: (Beispiel $i = 12$)

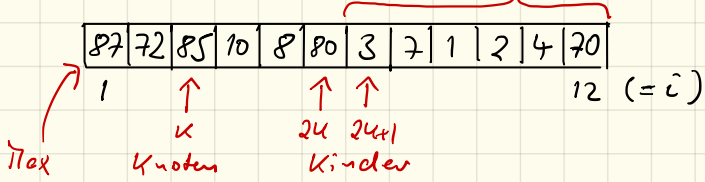


Heapbedingung:
Für alle Knoten:
Schlüssel Knoten $\geq$ Schlüssel Kinder

Bsp: $85 \geq 80, 3$

$\Rightarrow$ Maximum ist Wurzel

Gespeichert im Array:



Max-Finden: $O(1)$!

Heapbedingung (Array) Heap der Länge n

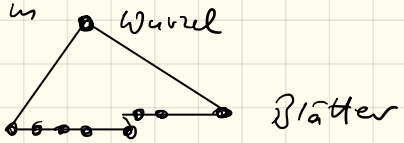
$\forall k \in \{1, \dots, n\}$:

$2k \leq n \Rightarrow A[2k] \leq A[k]$

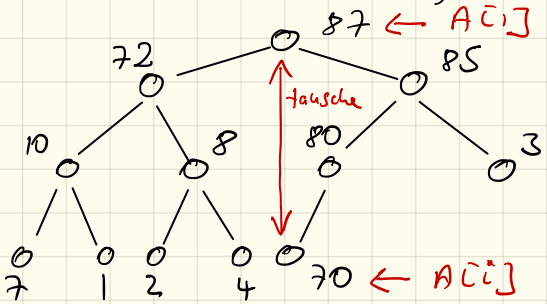
$2k+1 \leq n \Rightarrow A[2k+1] \leq A[k]$

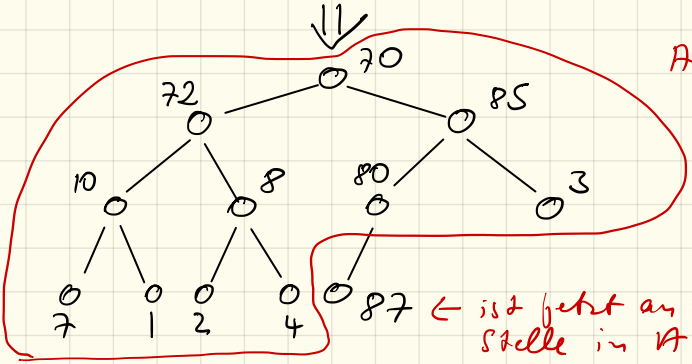
Weitere Eigenschaften des Heaps:
- grösste Tiefe: $\lfloor \log_2(n) \rfloor$ (Wurzel hat Tiefe 0)
- Anzahl Blätter: $\lceil n/2 \rceil$ (die Hälfte)

$\Rightarrow$ Voller Binärbaum



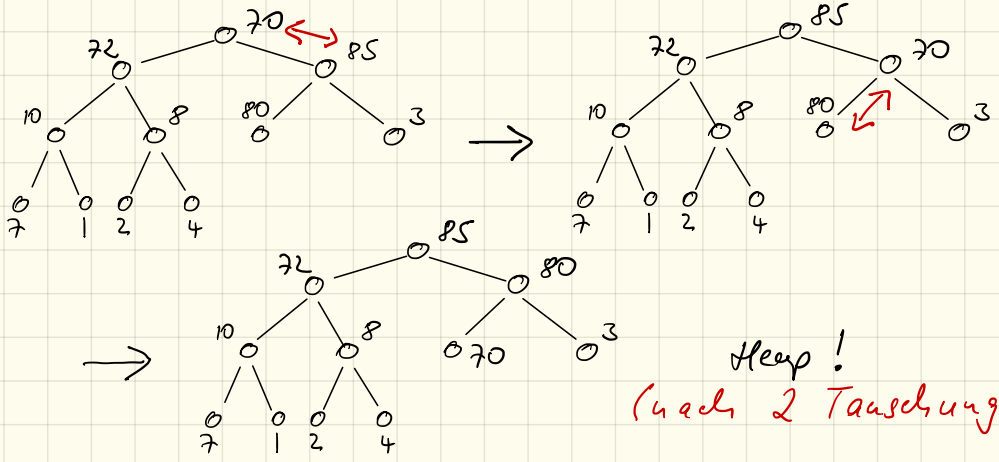
Zurück zum Sortieren, Iteration i:





A[1..i-1] ist kein Heap mehr!

Also: stelle Heap Bedingung wieder her
"versickere neue Wurzel durch Tausch mit größerem Kind, bis Kinder kleiner"



Heap!
(nach 2 Tauschungen)

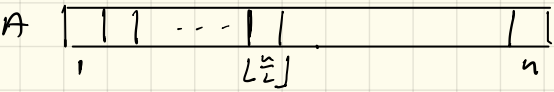
Versickern: $O(\log(i))$ Vergleiche
$O(\log(i))$ Vertauschungen

also gesamt $\sum_{i=1}^{n} c \cdot \log(i) \leq O(n \log n)$

↑ hatten wir letztes Mal schon

Pseudocode:
RestoreHeapCondition(A, k, i) // versickere Element k in $A[1..i]$
im Skript

Was noch fehlt: A am Anfang in Heap verwandeln



Diese muss man versickern von rechts nach links | $\lceil \frac{n}{2} \rceil$ Blätter ⇒ verletzen dann keine Heapbedingung

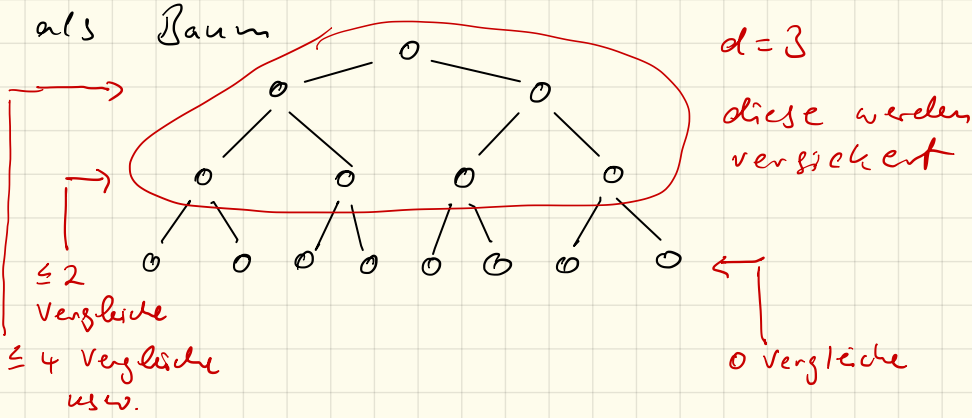
```
HeapSort(A)
  for i = ⌊n/2⌋ .. 1                          } Erzeuge Heap
    RestoreHeapCondition(A, i, n)
  for i = n .. 2                              } O(n log n) s.o.
    Vertausche A[1] und A[i]
    RestoreHeapCondition(A, 1, i-1)
```

Analyse "Erzeuge Heap":

$n/2$ Aufrufe, jeder $\leq O(\log n)$ ⇒ $O(n \log n)$

Es gilt sogar $O(n)$: Siehe Skript oder nächste Seite (nicht in Vorlesung)

Wir nehmen an $n = 2^d - 1$ und malen A als Baum



Anzahl Vergleiche für "Erzeuge Heap" höchstens

$$0 \cdot 2^d + 2 \cdot 2^{d-1} + 4 \cdot 2^{d-2} + 6 \cdot 2^{d-3} + \dots + 2d \cdot 2^{d-d}$$

$$= 2 \sum_{i=0}^{d-1} 2^i (d-i) = 2d(2^d - 1) - 2 \sum_{i=0}^{d-1} i\, 2^i$$

Bemerke: $\sum_{i=0}^{d-1} i\, x^i = \dfrac{x - d x^d + (d-1) x^{d+1}}{(1-x)^2}$, $x \neq 1$

woher kommt das?
Nimm $\sum_{i=0}^{d-1} x^i = (1 - x^d)/(1-x)$ und leite auf beiden Seiten ab

$$= 2d(2^d - 1) - 2\left(2 - d\,2^d + (d-1)\,2^{d+1}\right)$$

$$= 4 \cdot 2^d - 2d - 4 \leq O(n)$$

Pro 2 Vergleiche ≤ 1 Vertauschung $\Rightarrow O(n)$ Vertauschungen

HeapSort: + $O(n \log n)$

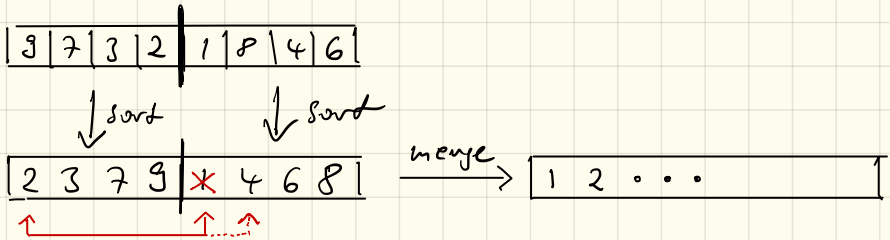
gut → + Inplace (Extraplatz $O(1)$)

schlecht → − schlechte Lokalität (= es wird viel in A hin und her geschupft)

neues Konzept kommt in späteren Vorlesungen

Algorithmus 5: MergeSort

Idee: Divide-and-Conquer

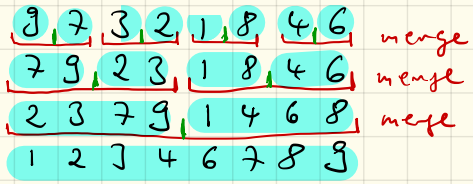


das nächstkleinste im Resultat ist immer eines der beiden ersten (lasse 2 Zeiger wandern)

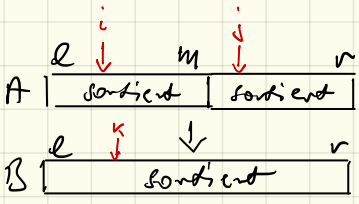
```
MergeSort(A, left, right)   // Anfang: left=1, right=n
  if left < right
    middle = ⌊(left+right)/2⌋
    MergeSort(A, left, middle)
    MergeSort(A, middle+1, right)
    Merge(A, left, middle, right)
```

Den Algorithmus kann man iterativ implementieren: Straight MergeSort im Skript

Beispiel:



```
Merge (A, l, m, r)
    i = l   // Index in A links
    j = m+1 // Index in A rechts
    k = l   // Index in B
    while i ≤ m and j ≤ r
        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  } nur ein Fall tritt ein
    "      "    rechts  }
    kopiere B zurück nach A
```



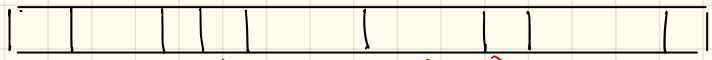
braucht Hilfsarray

Merging mit weniger Extraplatz (sogar O(1)) ist möglich aber komplizierter

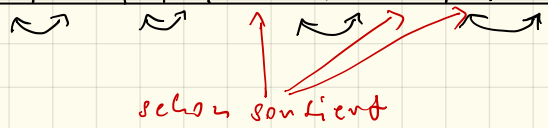
Laufzeit: (MergeSort)
- Vergleiche $T(n) = 2T(\frac{n}{2}) + cn \leq O(n \log n)$
- Bewegungen $\leq O(n \log n)$
- Extraplatz $O(n)$

Variante: Natural Mergesort (siehe Skript)

1.) Finde sortierte Teilstücke:



2.) Merge



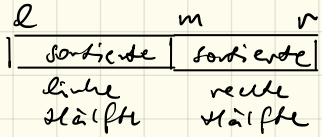
usw.

Algorithmus 6: Quicksort

Mergesort
- teile Array
- sortiere links
- sortiere rechts
- verschmelze

↳ Array + Extraplatz ist hier

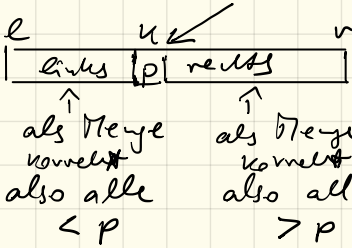
Invariante:



Idee: schiebe die Arbeit ins Aufteilen

1.) Aufteilen und Arbeit
2.) Rekursion links und rechts
3.) Verschmelzen nicht notwendig

Invariante: an richtiger Stelle

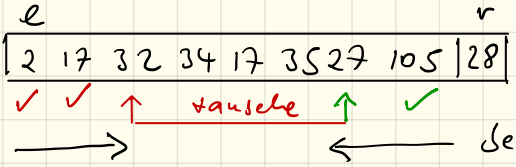



```
QuickSort(A, l, r)
  if l < r
    k = Aufteilen(A, l, r)   // wählt ein Element p und setzt es an die richtige Stelle indem es INV herstellt
    QuickSort(A, l, k-1)
    QuickSort(A, k+1, r)
```

p = Pivotelement, z. B. p = A[r] ← letztes Element

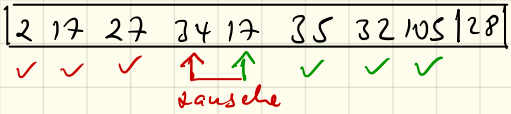
(Pivot: Angelpunkt (franz.))

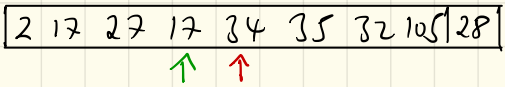
Aufteilen:



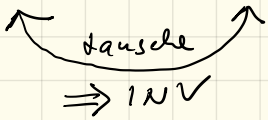
← p = z.B. letztes Element

bewege Zeiger bis linkes Element > p und rechtes Element < p dann tausche



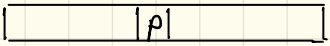


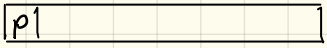
stop wenn rot ≥ grün:




```
Aufteilen(A, l, r)        // l < r
    i = l
    j = r - 1
    p = A[r]
    repeat
        while i < r and A[i] < p : i = i + 1
        while j > l and A[j] > p : j = j - 1
        if i < j : tausche A[i], A[j]
    until i ≥ j
    tausche A[i], A[r]    // jetzt ist Pivot am richtigen Platz
    return i
```

Laufzeit: Hängt davon ab wo Pivot landet

gut: 

schlecht: 

gut: $T(n) = 2T(\frac{n}{2}) + cn \leq O(n \log n)$

schlecht: $T(n) = T(n-1) + cn \leq O(n^2)$

Trotzdem wird QuickSort viel verwendet
Warum?

Worst-case ist selten!
z.B. best/worst case alternierend $\Rightarrow O(n \log n)$

QuickSort ist $O(n \log n)$ im <u>average case</u>
(neue Analyse, machen wir nicht)

Unglücklicherweise: "schon sortiert" ist worst case

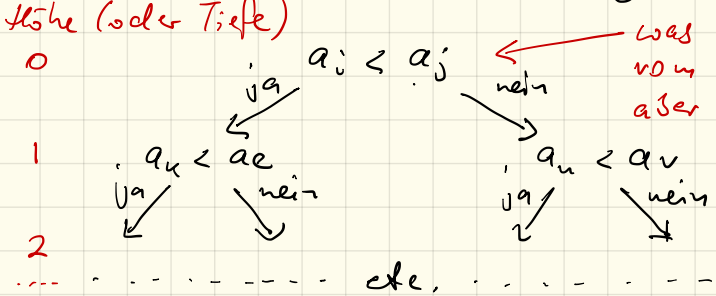
Häufig wird das Pivotelement zufällig gewählt (randomisiertes QuickSort)

Randomisierte Algorithmen: nächstes Semester

Komplexität vergleichsbasiertes Sortieren

Geht es mit weniger als $\Theta(n \log n)$ vielen Vergleichen?

Idee: Jeder Algorithmus entspricht einem Entscheidungsbaum (zähle Suche)



was i und j ist hängt vom Algorithmus ab, aber jeder fängt irgendwo an

Bsp: ADS 2018 Übungsblatt 5

Blätter $\leftrightarrow$ Ende Algorithmus für verschiedene Arrays der Länge n

$\Rightarrow$ Anzahl Blätter $\geq n!$ (mögliche Sortierungen)

Laufzeit (worst case) $\longleftrightarrow$ Höhe Baum h

Baum der Höhe h hat $\leq 2^h$ Blätter, also muss

$$2^h \geq n!$$

$$\Leftrightarrow \quad h \geq \log_2(n!) \geq \Omega(n \log n)$$

Also Komplexität vergleichsbasiertes Sortieren ist $\Theta(n \log n)$.

Bäume: Manchmal sagt man Höhe, manchmal Tiefe
Tiefe der Wurzel definiert man $=0$ oder $=1$, was gerade besser passt

O-Notation: O, Ω, Θ

wie immer $f: \mathbb{N} \to \mathbb{R}^+$ (positive reelle Zahlen $0 \notin \mathbb{R}^+$)

wir schreiben oft $f(n)$ statt f

1.) $O(f(n)) = \{ g(n) \mid$ es gibt $c > 0$ so daß $g(n) \leq c f(n)$ für alle $n \in \mathbb{N} \}$

$g(n) \in O(f(n))$, Schreibweise $g(n) \leq O(f(n))$

"$f(n)$ ist asymptotisch eine obere Schranke für $g(n)$"

2.) $\Omega(f(n)) = \{ g(n) \mid$ es gibt $c > 0$ so daß $g(n) \geq c f(n)$ für alle $n \in \mathbb{N} \}$

$g(n) \in \Omega(f(n))$, Schreibweise $g(n) \geq \Omega(f(n))$

"$f(n)$ ist asymptotisch eine untere Schranke für $g(n)$"

3.) Θ (Theta) $(f(n)) = O(f(n)) \cap \Omega(f(n))$

$g(n) \in \Theta(n)$, Schreibweise $g(n) = \Theta(f(n))$

"$g(n)$ wächst asymptotisch wie $f(n)$"

Bemerkung: Manchmal macht es Sinn sei $n_0 > 1$ anzufangen damit Funktionen > 0 werden. z.B. $\log_2(n) + 1 \leq O(\log n)$